# Library Reference


# Borland® C++

# Library Reference

# Borland® C++

Version 4.0

**Borland International, Inc.**
100 Borland Way, Scotts Valley, CA 95067-3249

PRINTED IN THE UNITED STATES OF AMERICA

1EOR993
9394959697-9876543
W1

# Contents

# Tables

# Figures

# Introduction

If you are developing a 16-bit DOS-only application, you can also use the routines described in the *DOS Reference*

This manual contains definitions of the Borland C++ classes, nonprivate class members, library routines, common variables, and common defined types for windows programming.

If you're new to C or C++ programming, or if you're looking for information on the contents of the Borland C++ manuals, see the introduction in the *User's Guide*.

Here is a summary of the chapters in this manual:

**Chapter 1: Library cross-reference** provides an overview of the Borland C++ library routines and header files. After describing the static and dynamic-link libraries, this chapter lists the header files, and then groups the library routines according to the tasks they commonly perform.

**Chapter 2: The main function** discusses arguments to *main* (including wildcard arguments), provides some example programs, and describes Pascal calling conventions and the value that *main* returns.

**Chapter 3: Run-time functions** is an alphabetical reference of Borland C++ library functions. Each entry gives syntax, portability information, an operative description, and return values for the function, together with a reference list of related functions.

**Chapter 4: Global variables** defines and discusses Borland C++'s global variables. You can use these to save yourself a great deal of programming time on commonly needed variables (such as dates, time, error messages, stack size, and so on).

**Chapter 5: The C++ iostream classes** describes the classes that provide support for input and output in C++ programs.

**Chapter 6: Persistent stream classes and macros** describes the persistent streams classes and macros.

**Chapter 7: The C++ container classes** describes the container classes provided by Borland C++ such as array, stack, and linked list.

**Chapter 8: The C++ mathematical classes** describes how to use *bcd* and *complex* classes.

**Chapter 9: Class diagnostic macros** describes the classes and macros that support object diagnostics.

**Chapter 10: Run-time support** describes functions and classes that let you control the way your program executes at run time in case the program runs out of memory or encounters some exception.

**Chapter 11: C++ utility classes** describes the C++ *date, string,* and *time* classes.

C H A P T E R 1

# Library cross-reference

If you are developing a 16-bit DOS-only application, you can also use the routines described in the *DOS Reference*

This chapter provides an overview of the Borland C++ library routines and header files. Library routines are composed of classes, functions, and macros that you can call from within your C and C++ programs to perform a wide variety of tasks. These tasks include low- and high-level I/O, string and file manipulation, memory allocation, process control, data conversion, mathematical calculations, and much more.

This chapter provides the following information:

- Names the static and dynamic-link libraries, files, and subdirectories found in the LIB and BIN subdirectories, and describes their uses.
- Explains why you might want to obtain the source code for the Borland C++ run-time library.
- Lists and describes the header files.
- Categorizes the library routines according to the type of tasks they perform.

## Reasons to access the run-time library source code

There are several good reasons you might want to obtain the source code for the run-time library routines:

- A particular function you want to write might be similar to, but not the same as, a Borland C++ function. With access to the run-time library source code, you can tailor the library function to suit your needs, and avoid having to write a separate function of your own.
- Sometimes, when you're debugging code, you might want to know more about the internals of a library function.
- If you want to delete the leading underscores on C symbols, access to the run-time library source code will let you do so.
- You can learn a lot from studying tight, professionally written library source code.

For all these reasons, and more, you will want to have access to the Borland C++ run-time library source code. Because Borland believes strongly in the concept of "open architecture," we have made the Borland C++ run-time library source code available for licensing. All you have to do is fill out the order form distributed with your Borland C++ package, include your payment, and we'll ship you the Borland C++ run-time library source code.

## The run-time libraries

The run-time libraries are divided into static (OBJ and LIB) and dynamic-link (DLL) versions. These different versions of the libraries are installed in separate directories. The static and dynamic libraries are described in separate tables.

See the *ObjectWindows Reference Guide* for information about the libraries and DLLs specific to ObjectWindows.

Several versions of the run-time library are available. For example, there are memory-model–specific versions, diagnostic versions, and 16- and 32-bit–specific versions. There are also optional libraries that provide mathematics, container, ObjectWindows development, and international applications.

Here are some guidelines for selecting which run-time libraries to use:

- Segmented memory-model libraries are supported only in 16-bit programs. Tiny and huge memory models are not supported.
- 16-bit DLLs are supported only in the large memory model.
- For 32-bit programs, only the flat memory model is supported.
- 32-bit console and GUI programs require different startup code.
- Multithread applications are supported only in 32-bit programs.

### The static libraries

The static (OBJ and LIB) version of the Borland C++ run-time library is contained in the LIB subdirectory of your installation. For each of the library file names, the '?' character represents one of the four (compact, small, medium, and large) distinct memory models supported by Borland. Each model has its own library file and math file, containing versions of the routines written for that particular model.

The following table identifies the default run-time libraries used with each compiler. See the *User's Guide* for discussions about compiling and linking.

Table 1.1: Default run-time libraries

| Compiler | Application | Default libraries |
|---|---|---|
| BCC.EXE | 16-bit Windows | C0WS.OBJ, CWS.LIB, MATHWS.LIB, IMPORT.LIB |
| BCC32.EXE | Win32 | C0X32.OBJ, CW32.LIB, IMPORT32.LIB |
| BCW.EXE | Same as BCC.EXE | Same as BCC.EXE |
| BCWS32.EXE | Same as BCC32.EXE | Same as BCC32.EXE |

The following table lists the names and uses of the Borland C++ static libraries; it also lists the operating system under which each library item is available. See the *User's Guide* for information on linkers, linker options, requirements, and selection of libraries.

Table 1.2: Summary of static run-time libraries

| File name | Application | Use |
|---|---|---|
| ***Directory of BC4\LIB*** | | |
| BIDSDI.LIB | Win 16 | 16-bit diagnostic, dynamic BIDS import library for BIDS40D.DLL |
| BIDSI.LIB | Win 16 | 16-bit dynamic BIDS import library for BIDS40.DLL |
| BIDSF.LIB | Win32s, Win32 | 32-bit BIDS library |
| BIDSDF.LIB | Win32s, Win32 | 32-bit diagnostic BIDS library |
| BIDSFI.LIB | Win32s, Win32 | 32-bit dynamic BIDS import library for BIDS40F.DLL |
| BIDSDFI.LIB | Win32s, Win32 | 32-bit diagnostic, dynamic BIDS import library for BIDS40DF.DLL |
| BIDSDB?.LIB | Win 16 | 16-bit diagnostic BIDS library |
| BIDS?.LIB | Win 16 | 16-bit BIDS library |
| BWCC.LIB | Win 16 | 16-bit import library for BWCC.DLL |
| BWCC32.LIB | Win32s, Win32 | 32-bit import library for BWCC32.DLL |
| C0D32.OBJ | Win32s, Win32 | 32-bit DLL startup module |
| C0D?.OBJ | Win 16 | 16-bit DLL startup module |
| C0W32.OBJ | Win32s, Win32 | 32-bit GUI EXE startup module |
| C0W?.OBJ | Win 16 | 16-bit EXE startup module |
| C0X32.OBJ | Win32 | 32-bit console-mode EXE startup module |
| CRTDLL.LIB | Win 16 | 16-bit dynamic import library for BC40RTL.DLL |
| CW32.LIB | Win32s, Win32 | 32-bit GUI single-thread library |
| CW?.LIB | Win 16 | 16-bit library |
| CW32I.LIB | Win32s, Win32 | 32-bit single-thread, GUI, dynamic RTL import library for CW32.DLL |
| CW32MT.LIB | Win32 | 32-bit GUI multithread library |

Table 1.2: Summary of static run-time libraries (continued)

| | | |
|---|---|---|
| CW32MTI.LIB | Win32 | 32-bit multithread, GUI, dynamic RTL import library for CW32MT.DLL |
| IMPORT.LIB | Win 16 | 16-bit import library for Windows 3.1 |
| IMPORT32.LIB | Win32s, Win32 | 32-bit import library; use with IMPRTW32.LIB |
| MATHW?.LIB | Win 16 | 16-bit math libraries |
| W32SUT16.LIB | Win 16 | 16-bit universal thunking library |
| W32SUT32.LIB | Win32s | 32-bit universal thunking library |
| OBSOLETE.LIB | Win 16, Win32, Win32s | Provides obsolete global variables. |
| ***Directory of BC4\LIB\16-BIT*** | | |
| FILES.C | Win 16 | Increases the number of file handles |
| FILES2.C | Win 16 | Increases the number of file handles |
| MATHERR.C | Win 16 | Sample of a user-defined floating-point math exception handler for **float** and **double** types |
| MATHERRL.C | Win 16 | Sample of a user-defined floating-point math exception handler for **long double** type |
| ***Directory of BC4\LIB\32-BIT*** | | |
| FILES.C | Win32s, Win32 | Increases the number of file handles |
| FILES2.C | Win32s, Win32 | Increases the number of file handles |
| FILEINFO.OBJ | Win32s, Win32 | Passes open file-handle information to child processes |
| GP.OBJ | Win32s, Win32 | Prints register-dump information when an exception occurs |
| MATHERR.C | Win32s, Win32 | Sample of a user-defined floating-point math exception handler for **float** and **double** types |
| MATHERRL.C | Win32s, Win32 | Sample of a user-defined floating-point math exception handler for **long double** type |
| WILDARGS.OBJ | Win32 | Transforms wild-card arguments into an array of arguments to *main* in console-mode applications |
| ***Directory of BC4\LIB\STARTUP*** | | |
| BUILD-C0.BAT | Win 16 | Batch file to build C0D?.OBJ, C0F?.OBJ, and C0W?.OBJ |
| C0D.ASM | Win 16 | Source for C0D?.OBJ |
| C0W.ASM | Win 16 | Source for C0W?.OBJ |
| RULES.ASI | Win 16 | Assembly rules for C0D.ASM and C0W.ASM |

## The dynamic-link libraries

The dynamic-link (DLL) version of the run-time library is contained in the BIN subdirectory of the installation. Several versions of the DLL libraries

are available. For example, there are diagnostic versions, 16- and 32-bit–specific versions, and versions that support multithread applications.

In the 16-bit specific version, only the large-memory model DLL is provided. No other memory-model is supported in a 16-bit DLL.

The following table lists the Borland C++ DLL names and uses, and the operating system under which the library item is available. See the *User's Guide* for information on linkers, linker options, requirements, and selection of libraries.

Table 1.3: Summary of dynamic link libraries

| File name | Application | Use |
|---|---|---|
| ***Directory of BC4\BIN*** | | |
| BC40RTL.DLL | Win 16 | 16-bit, large-memory model |
| BIDS40.DLL | Win 16 | 16-bit, BIDS |
| BIDS40D.DLL | Win 16 | 16-bit, diagnostic BIDS |
| BIDS40F.DLL | Win32s, Win32 | 32-bit BIDS |
| BIDS40DF.DLL | Win32s, Win32 | 32-bit diagnostic BIDS |
| CW32.DLL | Win32s, Win32 | 32-bit, single thread |
| CW32MT.DLL | Win32 | 32-bit, multithread |
| LOCALE.BLL | Win 16, Win32s, Win32 | Locale library |

## The Borland C++ header files

C++ header files, and header files defined by ANSI C, are marked in the margin.

Header files provide function prototype declarations for library functions. Data types and symbolic constants used with the library functions are also defined in them, along with global variables defined by Borland C++ and by the library functions. The Borland C++ library follows the ANSI C standard on header-file names and their contents.

| Header file | Description |
|---|---|
| alloc.h | Declares memory-management functions (allocation, deallocation, and so on). |
| assert.h[1] | Defines the *assert* debugging macro. |
| bcd.h[2] | Declares the C++ class *bcd* and the overloaded operators for *bcd* and *bcd* math functions. |
| bios.h | Declares various functions used in calling IBM-PC ROM BIOS routines. |
| checks.h[2] | Defines PRECONDITION, WARN, and TRACE diagnostic macros. |

| | |
|---|---|
| complex.h[2] | Declares the C++ *complex* math class. |
| conio.h | Declares various functions used in calling the operating-system console I/O routines. The functions defined in this header file cannot be used in GUI applications. |
| constrea.h[2] | Declares C++ classes and methods to support console output. |
| cstring.h[2] | Declares the ANSI C++ string class support. |
| ctype.h[1] | Contains information used by the character classification and character conversion macros (such as *isalpha* and *toascii*). |
| dir.h | Contains structures, macros, and functions for working with directories and path names. |
| direct.h | Defines structures, macros, and functions for dealing with directories and path names. |
| dirent.h | Declares functions and structures for POSIX directory operations. |
| dos.h | Defines various constants and gives declarations needed for DOS and 80*x*86-specific calls. |
| errno.h[1] | Defines constant mnemonics for the error codes. |
| except.h[2] | Declares ANSI C++ exceptions support. |
| excpt.h | Declares C exceptions support. |
| fcntl.h | Defines symbolic constants used in connection with the library routine *open*. |
| float.h[1] | Contains parameters for floating-point routines. |
| fstream.h[2] | Declares the C++ stream classes that support file input and output. |
| generic.h | Contains macros for generic class declarations. |
| io.h | Contains structures and declarations for low-level input/output routines. |
| iomanip.h[2] | Declares the C++ streams I/O manipulators and contains templates for creating parameterized manipulators. |
| iostream.h[2] | Declares the basic C++ streams (I/O) routines. |
| limits.h[1] | Contains environmental parameters, information about compile-time limitations, and ranges of integral quantities. |
| locale.h[1] | Declares functions that provide country- and language-specific information. |
| sys\locking.h | Definitions for *mode* parameter of *locking* function. |
| malloc.h | Memory-management functions and variables. |
| math.h[1] | Declares prototypes for the math functions and math error handlers. |
| mem.h | Declares the memory-manipulation functions. (Many of these are also defined in string.h.) |
| memory.h | Memory-manipulation functions. |
| new.h[2] | Access to *_new_handler* and *set_new_handler*. |
| process.h | Contains structures and declarations for the *spawn...* and *exec...* functions. |
| ref.h[2] | Provides support for reference counting. Used with the string class. |
| regexp.h[2] | Implements regular-expression searching. |

| | |
|---|---|
| search.h | Declares functions for searching and sorting. |
| setjmp.h[1] | Defines a type *jmp_buf* used by the *longjmp* and *setjmp* functions and declares the functions *longjmp* and *setjmp*. |
| share.h | Defines parameters used in functions that make use of file-sharing. |
| signal.h[1] | Defines constants and declarations for use by the *signal* and *raise* functions. |
| stdarg.h[1] | Defines macros used for reading the argument list in functions declared to accept a variable number of arguments (such as *vprintf*, *vscanf*, and so on). |
| stddef.h[1] | Defines several common data types and macros. |
| stdio.h[1] | Defines types and macros needed for the standard I/O package defined in Kernighan and Ritchie and extended under UNIX System V. Defines the standard I/O predefined streams *stdin*, *stdout*, *stdprn*, and *stderr*, and declares stream-level I/O routines. |
| stdiostr.h[2] | Declares the C++ (version 2.0) stream classes for use with stdio FILE structures. You should use iostream.h for new code. |
| stdlib.h[1] | Declares several commonly used routines: conversion routines, search/sort routines, and other miscellany. |
| string.h[1] | Declares several string-manipulation and memory-manipulation routines. |
| strstrea.h[2] | Declares the C++ stream classes for use with byte arrays in memory. |
| sys\stat.h | Defines symbolic constants used for opening and creating files. |
| time.h[1] | Defines a structure filled in by the time-conversion routines *asctime*, *localtime*, and *gmtime*, and a type used by the routines *ctime*, *difftime*, *gmtime*, *localtime*, and *stime*; also provides prototypes for these routines. |
| sys\timeb.h | Declares the function *ftime* and the structure *timeb* that *ftime* returns. |
| sys\types.h | Declares the type *time_t* used with time functions. |
| typeinfo.h[2] | Provides declarations for ANSI C++ run-time type identification (RTTI). |
| utime.h | Declares the *utime* function and the *utimbuf* struct that it returns. |
| values.h | Defines important constants, including machine dependencies; provided for UNIX System V compatibility. |
| varargs.h | Definitions for accessing parameters in functions that accept a variable number of arguments. Provided for UNIX compatibility; you should use stdarg.h for new code. |

[1] Defined by ANSI C.
[2] C++ header files.

## Library routines by category

The Borland C++ library routines perform a variety of tasks. The routines, along with the header files in which they are declared, are listed by category of task performed.

### C++ prototyped routines

Certain routines described in this book have multiple declarations. You must choose the prototype appropriate for your program. In general, the multiple prototypes are required to support the original C implementation and the stricter and sometimes different C++ function declaration syntax. For example, some string-handling routines have multiple prototypes because in addition to the ANSI-C specified prototype, Borland C++ provides prototypes consistent with the ANSI C++ draft.

| | | | |
|---|---|---|---|
| *getvect* | (dos.h) | *strchr* | (string.h) |
| *max* | (stdlib.h) | *strpbrk* | (string.h) |
| *memchr* | (string.h) | *strrchr* | (string.h) |
| *min* | (stdlib.h) | *strstr* | (string.h) |
| *setvect* | (dos.h) | | |

### Classification routines

These routines classify ASCII characters as letters, control characters, punctuation, uppercase, and so on.

| | | | |
|---|---|---|---|
| *isalnum* | (ctype.h) | *islower* | (ctype.h) |
| *isalpha* | (ctype.h) | *isprint* | (ctype.h) |
| *isascii* | (ctype.h) | *ispunct* | (ctype.h) |
| *iscntrl* | (ctype.h) | *isspace* | (ctype.h) |
| *isdigit* | (ctype.h) | *isupper* | (ctype.h) |
| *isgraph* | (ctype.h) | *isxdigit* | (ctype.h) |

### Console I/O routines

These routines output text to the screen or read from the keyboard. They cannot be used in a GUI application.

| | | | |
|---|---|---|---|
| *cgets* | (conio.h) | *movetext* | (conio.h) |
| *clreol* | (conio.h) | *normvideo* | (conio.h) |
| *clrscr* | (conio.h) | *putch* | (conio.h) |
| *cprintf* | (conio.h) | *puttext* | (conio.h) |
| *cputs* | (conio.h) | *_setcursortype* | (conio.h) |
| *delline* | (conio.h) | *textattr* | (conio.h) |
| *getpass* | (conio.h) | *textbackground* | (conio.h) |
| *gettext* | (conio.h) | *textcolor* | (conio.h) |
| *gettextinfo* | (conio.h) | *textmode* | (conio.h) |
| *gotoxy* | (conio.h) | *ungetc* | (stdio.h) |
| *highvideo* | (conio.h) | *wherex* | (conio.h) |
| *insline* | (conio.h) | *wherey* | (conio.h) |
| *lowvideo* | (conio.h) | *window* | (conio.h) |

### Conversion routines

These routines convert characters and strings from alpha to different numeric representations (floating-point, integers, longs) and vice versa, and from uppercase to lowercase and vice versa.

| | | | |
|---|---|---|---|
| *atof* | (stdlib.h) | *strtol* | (stdlib.h) |
| *atoi* | (stdlib.h) | *_strtold* | (stdlib.h) |
| *atol* | (stdlib.h) | *strtoul* | (stdlib.h) |
| *ecvt* | (stdlib.h) | *toascii* | (ctype.h) |
| *fcvt* | (stdlib.h) | *_tolower* | (ctype.h) |
| *gcvt* | (stdlib.h) | *tolower* | (ctype.h) |
| *itoa* | (stdlib.h) | *_toupper* | (ctype.h) |
| *ltoa* | (stdlib.h) | *toupper* | (ctype.h) |
| *strtod* | (stdlib.h) | *ultoa* | (stdlib.h) |

## Diagnostic routines

These routines provide built-in troubleshooting capability.

| | | | |
|---|---|---|---|
| *assert* | (assert.h) | *perror* | (errno.h) |
| *CHECK* | (checks.h) | *PRECONDITION* | (checks.h) |
| *_matherr* | (math.h) | *TRACE* | (checks.h) |
| *_matherrl* | (math.h) | *WARN* | (checks.h) |

## Directory control routines

These routines manipulate directories and path names.

| | | | |
|---|---|---|---|
| *chdir* | (dir.h) | *_getdcwd* | (direct.h) |
| *_chdrive* | (direct.h) | *getdisk* | (dir.h) |
| *closedir* | (dirent.h) | *_makepath* | (stdlib.h) |
| *_dos_findfirst* | (dos.h) | *mkdir* | (dir.h) |
| *_dos_findnext* | (dos.h) | *mktemp* | (dir.h) |
| *_dos_getdiskfree* | (dos.h) | *opendir* | (dirent.h) |
| *_dos_getdrive* | (dos.h) | *readdir* | (dirent.h) |
| *_dos_setdrive* | (dos.h) | *rewinddir* | (dirent.h) |
| *findfirst* | (dir.h) | *rmdir* | (dir.h) |
| *findnext* | (dir.h) | *_searchenv* | (stdlib.h) |
| *fnmerge* | (dir.h) | *searchpath* | (dir.h) |
| *fnsplit* | (dir.h) | *_searchstr* | (stdlib.h) |
| *_fullpath* | (stdlib.h) | *setdisk* | (dir.h) |
| *getcurdir* | (dir.h) | *_splitpath* | (stdlib.h) |
| *getcwd* | (dir.h) | | |

## EasyWin routines

These routines are portable to EasyWin programs but are not available in Windows 16-bit programs. They are provided to ease porting of existing code into a Windows 16-bit application.

| | | | |
|---|---|---|---|
| *clreol* | (conio.h) | *getche* | (stdio.h) |
| *clrscr* | (conio.h) | *gets* | (stdio.h) |
| *fgetchar* | (stdio.h) | *gotoxy* | (conio.h) |
| *getch* | (stdio.h) | *kbhit* | (conio.h) |
| *getchar* | (stdio.h) | *perror* | (errno.h) |

| | | | |
|---|---|---|---|
| *printf* | (stdio.h) | *vprintf* | (stdio.h) |
| *putch* | (conio.h) | *vscanf* | (stdio.h) |
| *putchar* | (stdio.h) | *wherex* | (conio.h) |
| *puts* | (stdio.h) | *wherey* | (conio.h) |
| *scanf* | (stdio.h) | | |

## Inline routines

These routines have inline versions. The compiler will generate code for the inline versions when you use **#pragma intrinsic** or if you specify program optimization. See the *User's Guide* for more details.

| | | | |
|---|---|---|---|
| *abs* | (math.h) | *stpcpy* | (string.h) |
| *alloca* | (malloc.h) | *strcat* | (string.h) |
| *_crotl* | (stdlib.h) | *strchr* | (string.h) |
| *_crotr* | (stdlib.h) | *strcmp* | (string.h) |
| *_lrotl* | (stdlib.h) | *strcpy* | (string.h) |
| *_lrotr* | (stdlib.h) | *strlen* | (string.h) |
| *memchr* | (mem.h) | *strncat* | (string.h) |
| *memcmp* | (mem.h) | *strncmp* | (string.h) |
| *memcpy* | (mem.h) | *strncpy* | (string.h) |
| *memset* | (mem.h) | *strnset* | (string.h) |
| *_rotl* | (stdlib.h) | *strrchr* | (string.h) |
| *_rotr* | (stdlib.h) | *strset* | (string.h) |

## Input / output routines

These routines provide stream- and operating-system level I/O capability.

| | | | |
|---|---|---|---|
| *access* | (io.h) | *_dos_write* | (dos.h) |
| *_rtl_chmod* | (io.h) | *dup* | (io.h) |
| *chmod* | (io.h) | *dup2* | (io.h) |
| *chsize* | (io.h) | *eof* | (io.h) |
| *clearerr* | (stdio.h) | *fclose* | (stdio.h) |
| *_rtl_close* | (io.h) | *fcloseall* | (stdio.h) |
| *close* | (io.h) | *fdopen* | (stdio.h) |
| *_rtl_creat* | (io.h) | *feof* | (stdio.h) |
| *creat* | (io.h) | *ferror* | (stdio.h) |
| *creatnew* | (io.h) | *fflush* | (stdio.h) |
| *creattemp* | (io.h) | *fgetc* | (stdio.h) |
| *cscanf* | (conio.h) | *fgetchar* | (stdio.h) |
| *_dos_close* | (dos.h) | *fgetpos* | (stdio.h) |
| *_dos_creat* | (dos.h) | *fgets* | (stdio.h) |
| *_dos_creatnew* | (dos.h) | *filelength* | (io.h) |
| *_dos_getfileattr* | (dos.h) | *fileno* | (stdio.h) |
| *_dos_getftime* | (dos.h) | *flushall* | (stdio.h) |
| *_dos_open* | (dos.h) | *fopen* | (stdio.h) |
| *_dos_read* | (dos.h) | *fprintf* | (stdio.h) |
| *_dos_setfileattr* | (dos.h) | *fputc* | (stdio.h) |
| *_dos_setftime* | (dos.h) | *fputchar* | (stdio.h) |

| | | | |
|---|---|---|---|
| *fputs* | (stdio.h) | *putw* | (stdio.h) |
| *fread* | (stdio.h) | *_rtl_read* | (io.h) |
| *freopen* | (stdio.h) | *read* | (io.h) |
| *fscanf* | (stdio.h) | *remove* | (stdio.h) |
| *fseek* | (stdio.h) | *rename* | (stdio.h) |
| *fsetpos* | (stdio.h) | *rewind* | (stdio.h) |
| *_fsopen* | (stdio.h) | *rmtmp* | (stdio.h) |
| *fstat* | (sys\stat.h) | *scanf* | (stdio.h) |
| *ftell* | (stdio.h) | *setbuf* | (stdio.h) |
| *fwrite* | (stdio.h) | *setftime* | (io.h) |
| *getc* | (stdio.h) | *setmode* | (io.h) |
| *getch* | (conio.h) | *setvbuf* | (stdio.h) |
| *getchar* | (stdio.h) | *sopen* | (io.h) |
| *getche* | (conio.h) | *sprintf* | (stdio.h) |
| *getftime* | (io.h) | *sscanf* | (stdio.h) |
| *gets* | (stdio.h) | *_strerror* | (string.h, stdio.h) |
| *getw* | (stdio.h) | *strerror* | (stdio.h) |
| *ioctl* | (io.h) | *tell* | (io.h) |
| *isatty* | (io.h) | *tempnam* | (stdio.h) |
| *kbhit* | (conio.h) | *TFile* | (file.h) |
| *lock* | (io.h) | *tmpfile* | (stdio.h) |
| *locking* | (io.h) | *tmpnam* | (stdio.h) |
| *lseek* | (io.h) | *umask* | (io.h) |
| *_rtl_open* | (io.h) | *unlink* | (dos.h) |
| *open* | (io.h) | *unlock* | (io.h) |
| *_pclose* | (stdio.h) | *utime* | (utime.h) |
| *perror* | (stdio.h) | *vfprintf* | (stdio.h) |
| *_pipe* | (io.h) | *vfscanf* | (stdio.h) |
| *_popen* | (stdio.h) | *vprintf* | (stdio.h) |
| *printf* | (stdio.h) | *vscanf* | (stdio.h) |
| *putc* | (stdio.h) | *vsprintf* | (stdio.h) |
| *putchar* | (stdio.h) | *vsscanf* | (io.h) |
| *puts* | (stdio.h) | *_rtl_write* | (io.h) |

## Interface routines

These routines provide operating-system BIOS and machine-specific capabilities.

| | | | |
|---|---|---|---|
| *bdos* | (dos.h) | *dosexterr* | (dos.h) |
| *bdosptr* | (dos.h) | *_dos_getvect* | (dos.h) |
| *biosequip* | (bios.h) | *_dos_setvect* | (dos.h) |
| *_bios_equiplist* | (bios.h) | *_enable* | (dos.h) |
| *biosmemory* | (bios.h) | *enable* | (dos.h) |
| *biostime* | (bios.h) | *FP_OFF* | (dos.h) |
| *_chain_intr* | (dos.h) | *FP_SEG* | (dos.h) |
| *country* | (dos.h) | *geninterrupt* | (dos.h) |
| *ctrlbrk* | (dos.h) | *getcbrk* | (dos.h) |
| *_disable* | (dos.h) | *getdfree* | (dos.h) |
| *disable* | (dos.h) | *getdta* | (dos.h) |

| | | | |
|---|---|---|---|
| *getfat* | (dos.h) | *outpw* | (conio.h) |
| *getfatd* | (dos.h) | *outport* | (dos.h) |
| *getpsp* | (dos.h) | *outportb* | (dos.h) |
| *getvect* | (dos.h) | *parsfnm* | (dos.h) |
| *getverify* | (dos.h) | *peek* | (dos.h) |
| *inp* | (conio.h | *peekb* | (dos.h) |
| *inpw* | (conio.h | *poke* | (dos.h) |
| *inport* | (dos.h) | *pokeb* | (dos.h) |
| *inportb* | (dos.h) | *segread* | (dos.h) |
| *int86* | (dos.h) | *setcbrk* | (dos.h) |
| *int86x* | (dos.h) | *_setcursortype* | (conio.h) |
| *intdos* | (dos.h) | *setdta* | (dos.h) |
| *intdosx* | (dos.h) | *setvect* | (dos.h) |
| *intr* | (dos.h) | *setverify* | (dos.h) |
| *MK_FP* | (dos.h) | *sleep* | (dos.h) |
| *outp* | (conio.h) | | |

## International locale API routines

These routines are affected by the current locale. The current locale is specified by the *setlocale* function and is enabled by defining __USELOCALES__ with **-D** command line option. When you define __USELOCALES__, only function versions of the following routines are used in the run-time library rather than macros. See online Help for a discussion of the International API.

| | | | |
|---|---|---|---|
| *cprintf* | (stdio.h) | *scanf* | (stdio.h) |
| *cscanf* | (stdio.h) | *setlocale* | (locale.h) |
| *fprintf* | (stdio.h) | *sprintf* | (stdio.h) |
| *fscanf* | (stdio.h) | *sscanf* | (stdio.h) |
| *isalnum* | (ctype.h) | *strcoll* | (string.h) |
| *isalpha* | (ctype.h) | *strftime* | (time.h) |
| *iscntrl* | (ctype.h) | *strlwr, _fstrlwr* | (string.h) |
| *isdigit* | (ctype.h) | *strupr, _fstrupr* | (string.h) |
| *isgraph* | (ctype.h) | *strxfrm* | (string.h) |
| *islower* | (ctype.h) | *tolower* | (ctype.h) |
| *isprint* | (ctype.h) | *toupper* | (ctype.h) |
| *ispunct* | (ctype.h) | *vfprintf* | (stdio.h) |
| *isspace* | (ctype.h) | *vfscanf* | (stdio.h) |
| *isupper* | (ctype.h) | *vprintf* | (stdio.h) |
| *isxdigit* | (ctype.h) | *vscanf* | (stdio.h) |
| *localeconv* | (locale.h) | *vsprintf* | (stdio.h) |
| *printf* | (stdio.h) | *vsscanf* | (stdio.h) |

## Manipulation routines

These routines handle strings and blocks of memory: copying, comparing, converting, and searching.

| | | | |
|---|---|---|---|
| *mblen* | (stdlib.h) | *mbtowc* | (stdlib.h) |
| *mbstowcs* | (stdlib.h) | *memccpy* | (mem.h, string.h) |

| | | | |
|---|---|---|---|
| *memchr* | (mem.h, string.h) | *string* | (cstring.h) |
| *memcmp* | (mem.h, string.h) | *strlen* | (string.h) |
| *memcpy* | (mem.h, string.h) | *strlwr* | (string.h) |
| *memicmp* | (mem.h, string.h) | *strncat* | (string.h) |
| *memmove* | (mem.h, string.h) | *strncmp* | (string.h) |
| *memset* | (mem.h, string.h) | *strncmpi* | (string.h) |
| *movedata* | (mem.h, string.h) | *strncpy* | (string.h) |
| *movmem* | (mem.h, string.h) | *strnicmp* | (string.h) |
| *setmem* | (mem.h) | *strnset* | (string.h) |
| *stpcpy* | (string.h) | *strpbrk* | (string.h) |
| *strcat* | (string.h) | *strrchr* | (string.h) |
| *strchr* | (string.h) | *strrev* | (string.h) |
| *strcmp* | (string.h) | *strset* | (string.h) |
| *strcmpi* | (string.h) | *strspn* | (string.h) |
| *strcoll* | (string.h) | *strstr* | (string.h) |
| *strcpy* | (string.h) | *strtok* | (string.h) |
| *strcspn* | (string.h) | *strupr* | (string.h) |
| *strdup* | (string.h) | *strxfrm* | (string.h) |
| *strerror* | (string.h) | *wcstombs* | (stdlib.h) |
| *stricmp* | (string.h) | *wctomb* | (stdlib.h) |

## Math routines

These routines perform mathematical calculations and conversions.

| | | | |
|---|---|---|---|
| *abs* | (complex.h, stdlib.h) | *coshl* | (math.h) |
| *acos* | (complex.h, math.h) | *cosl* | (math.h) |
| *acosl* | (math.h) | *div* | (math.h) |
| *arg* | (complex.h) | *ecvt* | (stdlib.h) |
| *asin* | (complex.h, math.h) | *exp* | (complex.h, math.h) |
| *asinl* | (math.h) | *expl* | (math.h) |
| *atan* | (complex.h, math.h) | *fabs* | (math.h) |
| *atan2* | (complex.h, math.h) | *fabsl* | (math.h) |
| *atan2l* | (math.h) | *fcvt* | (stdlib.h) |
| *atanl* | (math.h) | *floor* | (math.h) |
| *atof* | (stdlib.h, math.h) | *floorl* | (math.h) |
| *atoi* | (stdlib.h) | *fmod* | (math.h) |
| *atol* | (stdlib.h) | *fmodl* | (math.h) |
| *_atold* | (math.h) | *_fpreset* | (float.h) |
| *bcd* | (bcd.h) | *frexp* | (math.h) |
| *cabs* | (math.h) | *frexpl* | (math.h) |
| *cabsl* | (math.h) | *gcvt* | (stdlib.h) |
| *ceil* | (math.h) | *hypot* | (math.h) |
| *ceill* | (math.h) | *hypotl* | (math.h) |
| *_clear87* | (float.h) | *imag* | (complex.h) |
| *complex* | (complex.h) | *itoa* | (stdlib.h) |
| *conj* | (complex.h) | *labs* | (stdlib.h) |
| *_control87* | (float.h) | *ldexp* | (math.h) |
| *cos* | (complex.h, math.h) | *ldexpl* | (math.h) |
| *cosh* | (complex.h, math.h) | *ldiv* | (math.h) |

| | | | |
|---|---|---|---|
| *log* | (complex.h, math.h) | *randomize* | (stdlib.h) |
| *logl* | (math.h) | *real* | (complex.h) |
| *log10* | (complex.h, math.h) | *_rotl* | (stdlib.h) |
| *log10l* | (math.h) | *_rotr* | (stdlib.h) |
| *_lrotl* | (stdlib.h) | *sin* | (complex.h, math.h) |
| *_lrotr* | (stdlib.h) | *sinh* | (complex.h, math.h) |
| *ltoa* | (stdlib.h) | *sinhl* | (math.h) |
| *_matherr* | (math.h) | *sinl* | (math.h).h, math.h) |
| *_matherrl* | (math.h) | *sqrt* | (complex.h, math.h) |
| *modf* | (math.h) | *sqrtl* | (math.h) |
| *modfl* | (math.h) | *srand* | (stdlib.h) |
| *norm* | (complex.h) | *_status87* | (float.h) |
| *polar* | (complex.h) | *strtod* | (stdlib.h) |
| *poly* | (math.h) | *strtol* | (stdlib.h) |
| *polyl* | (math.h) | *_strtold* | (stdlib.h) |
| *pow* | (complex.h, math.h) | *strtoul* | (stdlib.h) |
| *pow10* | (math.h) | *tan* | (complex.h, math.h) |
| *pow10l* | (math.h) | *tanh* | (complex.h, math.h) |
| *powl* | (math.h) | *tanhl* | (complex.h, math.h) |
| *rand* | (stdlib.h) | *tanl* | (math.h) |
| *random* | (stdlib.h) | *ultoa* | (stdlib.h) |

## Memory routines

These routines provide dynamic memory allocation in the small-data and large-data models.

| | | | |
|---|---|---|---|
| *alloca* | (malloc.h) | *heapcheckfree* | (alloc.h) |
| *_bios_memsize* | (bios.h) | *heapchecknode* | (alloc.h) |
| *calloc* | (alloc.h, stdlib.h) | *heapwalk* | (alloc.h) |
| *farcalloc* | (alloc.h) | *malloc* | (alloc.h, stdlib.h) |
| *farfree* | (alloc.h) | *realloc* | (alloc.h, stdlib.h) |
| *farmalloc* | (alloc.h) | *set_new_handler* | (new.h) |
| *free* | (alloc.h, stdlib.h) | *stackavail* | (malloc.h) |
| *heapcheck* | (alloc.h) | | |

## Miscellaneous routines

These routines provide nonlocal goto capabilities and locale.

| | | | |
|---|---|---|---|
| *localeconv* | (locale.h) | *setjmp* | (setjmp.h) |
| *longjmp* | (setjmp.h) | *setlocale* | (locale.h) |

## Obsolete definitions

The following global variables have been renamed to comply with ANSI naming requirements. You should always use the new names. If you link with libraries that were compiled with Borland C++ 3.1 (or earlier) header files, you will get the message

```
Error: undefined external varname in module LIBNAME.LIB
```

A library module that results in such an error should be recompiled. However, if you cannot recompile the code for such libraries, you can link with OBSOLETE.LIB to resolve the external variable names.

The following global variables have been renamed:

Table 1.4 Obsolete global variables

| Old name | New name | Header file |
|---|---|---|
| *daylight* | *_daylight* | time.h |
| *directvideo* | *_directvideo* | conio.h |
| *environ* | *_environ* | stdlib.h |
| *sys_errlist* | *_sys_errlist* | errno.h |
| *sys_nerr* | *_sys_nerr* | errno.h |
| *timezone* | *_timezone* | time.h |
| *tzname* | *_tzname* | time.h |

The old names of the following functions are available. However, the compiler will generate a warning that you are using an obsolete name. Future versions of Borland C++ might not provide support for the old function names.

The following function names have been changed:

Table 1.5 Obsolete function names

| Old name | New name | Header file |
|---|---|---|
| *_chmod* | *_rtl_chmod* | io.h |
| *_close* | *_rtl_close* | io.h |
| *_creat* | *_rtl_creat* | io.h |
| *_heapwalk* | *_rtl_heapwalk* | malloc.h |
| *_open* | *_rtl_open* | io.h |
| *_read* | *_rtl_read* | io.h |
| *_write* | *_rtl_write* | io.h |

## Process control routines

These routines invoke and terminate new processes from within another routine.

| | | | |
|---|---|---|---|
| *abort* | (process.h) | *execve* | (process.h) |
| *_beginthread* | (process.h) | *execvp* | (process.h) |
| *_beginthreadNT* | (process.h) | *execvpe* | (process.h) |
| *_c_exit* | (process.h) | *_exit* | (process.h) |
| *_cexit* | (process.h) | *exit* | (process.h) |
| *cwait* | (process.h) | *_expand* | (process.h) |
| *_endthread* | (process.h) | *getpid* | (process.h) |
| *execl* | (process.h) | *_pclose* | (stdio.h) |
| *execle* | (process.h) | *_popen* | (stdio.h) |
| *execlp* | (process.h) | *raise* | (signal.h) |
| *execlpe* | (process.h) | *signal* | (signal.h) |
| *execv* | (process.h) | *spawnl* | (process.h) |

| | | | |
|---|---|---|---|
| *spawnle* | (process.h) | *spawnve* | (process.h) |
| *spawnlp* | (process.h) | *spawnvp* | (process.h) |
| *spawnlpe* | (process.h) | *spawnvpe* | (process.h) |
| *spawnv* | (process.h) | *wait* | (process.h) |

## Time and date routines

These are time conversion and time manipulation routines.

| | | | |
|---|---|---|---|
| *asctime* | (time.h) | *gmtime* | (time.h) |
| *_bios_timeofday* | (bios.h) | *localtime* | (time.h) |
| *ctime* | (time.h) | *mktime* | (time.h) |
| *difftime* | (time.h) | *stime* | (time.h) |
| *_dos_getdate* | (dos.h) | *_strdate* | (time.h) |
| *_dos_gettime* | (dos.h) | *strftime* | (time.h) |
| *_dos_setdate* | (dos.h) | *_strtime* | (time.h) |
| *_dos_settime* | (dos.h) | *TDate* | (date.h) |
| *dostounix* | (dos.h) | *time* | (time.h) |
| *ftime* | (sys\timeb.h) | *TTime* | (time.h) |
| *getdate* | (dos.h) | *tzset* | (time.h) |
| *gettime* | (dos.h) | *unixtodos* | (dos.h) |

## Variable argument list routines

These routines are for use when accessing variable argument lists (such as with *printf, vprintf, vscanf,* and so on).

| | | | |
|---|---|---|---|
| *va_arg* | (stdarg.h) | *va_start* | (stdarg.h) |
| *va_end* | (stdarg.h) | | |

CHAPTER 2

# The main function

See the *Programmer's Guide*, Chapter 8, for a discussion of Windows programming.

Every C and C++ program must have a program-startup function. Console-based programs call the *main* function at startup. Windows GUI programs call the *WinMain* function at startup. Where you place the startup function is a matter of preference. Some programmers place *main* at the beginning of the file, others at the end. Regardless of its location, the following points about *main* always apply.

## Arguments to main

Three parameters (arguments) are passed to *main* by the Borland C++ startup routine: *argc*, *argv*, and *env*.

- *argc*, an integer, is the number of command-line arguments passed to *main*, including the name of the executable itself.
- *argv* is an array of pointers to strings (**char** *[]).
  - *argv*[0] is the full path name of the program being run.
  - *argv*[1] points to the first string typed on the operating system command line after the program name.
  - *argv*[2] points to the second string typed after the program name.
  - *argv*[*argc*-1] points to the last argument passed to *main*.
  - *argv*[*argc*] contains NULL.
- *env* is also an array of pointers to strings. Each element of *env*[] holds a string of the form `ENVVAR=value`.
  - ENVVAR is the name of an environment variable, such as PATH or COMSPEC.
  - *value* is the value to which ENVVAR is set, such as C:\APPS;C:\TOOLS; (for PATH) or C:\DOS\COMMAND.COM (for COMSPEC).

If you declare any of these parameters, you *must* declare them exactly in the order given: *argc, argv, env*. For example, the following are all valid declarations of *main*'s arguments:

```
int main()
int main(int argc)                /* legal but very unlikely */
int main(int argc, char * argv[])
int main(int argc, char * argv[], char * env[])]
```

Refer to the *getenv* and *putenv* entries in Chapter 3, and the *environ* entry in Chapter 4 for more information.

The declaration `int main(int argc)` is legal, but it's very unlikely that you would use *argc* in your program without also using the elements of *argv*.

The argument *env* is also available through the global variable *_environ*.

For all platforms, *argc* and *argv* are also available via the global variables *_argc* and *_argv*.

## Examining arguments to main

Here is an example that demonstrates a simple way of using these arguments passed to *main*:

```
/* Program ARGS.C */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[], char *env[]) {
   int i;

   printf("The value of argc is %d \n\n", argc);
   printf("These are the %d command-line arguments passed to"
          " main:\n\n", argc);

   for (i = 0; i < argc; i++)
      printf("   argv[%d]: %s\n", i, argv[i]);

   printf("\nThe environment string(s) on this system are:\n\n");

   for (i = 0; env[i] != NULL; i++)
      printf("   env[%d]: %s\n", i, env[i]);
   return 0;
   }
```

Suppose you run ARGS.EXE at the command prompt with the following command line:

```
C:> args first_arg "arg with blanks" 3  4 "last but one" stop!
```

Note that you can pass arguments with embedded blanks by surrounding them with quotes, as shown by `"argument with blanks"` and `"last but one"` in this example command line.

The output of ARGS.EXE (assuming that the environment variables are set as shown here) would then be like this:

```
The value of argc is 7

These are the 7 command-line arguments passed to main:

   argv[0]: C:\BC4\ARGS.EXE
   argv[1]: first_arg
   argv[2]: arg with blanks
   argv[3]: 3
   argv[4]: 4
   argv[5]: last but one
   argv[6]: stop!

The environment string(s) on this system are

   env[0]: COMSPEC=C:\COMMAND.COM
   env[1]: PROMPT=$p $g
   env[2]: PATH=C:\SPRINT;C:\DOS;C:\BC4
```

The maximum combined length of the command-line arguments passed to *main* (including the space between adjacent arguments and the program name itself) is 255; this is a Win32 limit.

### Wildcard arguments


Command-line arguments containing wildcard characters can be expanded to all the matching file names, much the same way DOS expands wildcards when used with commands like COPY. All you have to do to get wildcard expansion is to link your program with the WILDARGS.OBJ object file, which is included with Borland C++.

Wildcard arguments are used only in console-mode applications.

Once WILDARGS.OBJ is linked into your program code, you can send wildcard arguments of the type *.* to your *main* function. The argument will be expanded (in the *argv* array) to all files matching the wildcard mask. The maximum size of the *argv* array varies, depending on the amount of memory available in your heap.

If no matching files are found, the argument is passed unchanged. (That is, a string consisting of the wildcard mask is passed to *main*.)

Arguments enclosed in quotes ("...") are not expanded.

### *Linking with WILDARGS.OBJ*

The following commands compile the file ARGS.C and link it with the wildcard expansion module WILDARGS.OBJ, then run the resulting executable file ARGS.EXE:

```
BCC  ARGS.C  WILDARGS.OBJ
ARGS  C:\BC4\INCLUDE\*.H  "*.C"
```

When you run ARGS.EXE, the first argument is expanded to the names of all the *.H files in your Borland C++ INCLUDE directory. Note that the

expanded argument strings include the entire path. The argument *.C is not expanded because it is enclosed in quotes.

In the IDE, simply specify a project file (from the project menu) that contains the following lines:

```
ARGS
WILDARGS.OBJ
```

If you prefer the wildcard expansion to be the default, modify your standard CW32?.LIB library files to have WILDARGS.OBJ linked automatically. To accomplish that, remove SETARGV and INITARGS from the libraries and add WILDARGS. The following commands invoke the Turbo librarian (TLIB) to modify all the standard library files (assuming the current directory contains the standard C and C++ libraries and WILDARGS.OBJ):

```
tlib  CW32    -setargv  +wildargs
tlib  CW32MT  -setargv  +wildargs
tlib  -setargv  +wildargs
```

## Using –p (Pascal calling conventions)


If you compile your program using Pascal calling conventions (described in the *Programmer's Guide*, Chapter 2), you must remember to explicitly declare *main* as a C type. Do this with the **__cdecl** keyword, like this:

```
int  __cdecl main(int argc, char* argv[], char* envp[])
```

## The value main returns

The value returned by *main* is the status code of the program: an **int**. However, if your program uses the routine *exit* (or *_exit*) to terminate, the value returned by *main* is the argument passed to the call to *exit* (or to *_exit*).

For example, if your program contains the call `exit(1)`, the status is 1.

## Passing file information to child processes


If your program uses the *exec* or *spawn* functions to create a new process, the new process will normally inherit all of the open file handles created by the original process. However, some information about these handles will be lost, including the access mode used to open the file. For

example, if your program opens a file for read-only access in binary mode, and then spawns a child process, the child process might corrupt the file by writing to it, or by reading from it in text mode.

To allow child processes to inherit such information about open files, you must link your program with the object file FILEINFO.OBJ. For example:

```
BCC32 TEST.C \BC4\LIB\FILEINFO.OBJ
```

The file information is passed in the environment variable _C_FILE_INFO. This variable contains encoded binary information, and your program should not attempt to read or modify its value. The child program must have been built with the C++ run-time library to inherit this information correctly. Other programs can ignore _C_FILE_INFO, and will not inherit file information.

## Multithread programs


32-bit programs can create more than one thread of execution. If your program creates multiple threads, and these threads also use the C++ run-time library, you must use the CW32MT.LIB or CW32MTI library instead.

See the online Help example for *_beginthread* to see how to use these functions and *_threadid* in a program.

The multithread libraries provide the *_beginthread* and *_beginthreadNT* functions, which you use to create threads. The multithread libraries also provide the *_endthread* function, which terminates threads, and the global variable *_threadid*. This global variable contains the current thread's identification number (also known as the *thread ID*). The header file stddef.h contains the declaration of *_threadid*.

When you compile or link a program that uses multiple threads, you must use the –WM compiler switch. For example:

```
BCC32 -WM THREAD.C
```

Special care must be taken when using the *signal* function in a multithread program. See the description of the *signal* function for more information.

C H A P T E R 3

# Run-time functions

Programming examples for each function are available in the online Help system. You can easily copy them from Help and paste them into your files.

This chapter contains a detailed description of each function in the Borland C++ library. The functions are listed in alphabetical order, although a few of the routines are grouped by "family" (the *exec...* and *spawn...* functions, for example) because they perform similar or related tasks.

Each function entry provides certain standard information. For instance, the entry for *free*

- Tells you which header file(s) contains the prototype for *free*.
- Summarizes what *free* does.
- Gives the syntax for calling *free*.
- Gives a detailed description of how *free* is implemented and how it relates to the other memory-allocation routines.
- Lists other language compilers that include similar functions.
- Refers you to related Borland C++ functions.

The following sample library entry lists each entry section and describes the information it contains. The alphabetical listings start on page 27.

## Sample function entry — header file name

The *function* is followed by the header file(s) containing the prototype for *function* or definitions of constants, enumerated types, and so on used by *function*.

**Function**

Summary of what this *function* does.

**Syntax**

```
function(modifier parameter[,...]);
```

This gives you the declaration syntax for *function;* parameter names are *italicized*. The `[,...]` indicates that other parameters and their modifiers can follow.

Portability is indicated by marks (■) in the columns of the portability table. A sample portability table is shown here:

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| | | | | | | |

Each entry in the portability table is described in the following table. Any additional restrictions are discussed in the *Remarks* section.

| | |
|---|---|
| DOS | Available for DOS. |
| UNIX | Available under UNIX and/or POSIX. |
| Win 16 | Compatible with 16-bit Windows programs running on Microsoft Windows 3.1, Windows for Workgroups 3.1, and Windows for Workgroups 3.11. EasyWin users should see the *User's Guide* for information about using certain non-Windows functions (such as *printf* and *scanf*) in programs that run under Windows. |
| Win 32 | Available to 32-bit Windows programs running on Win32s 1.0, and Windows NT 3.1 applications. |
| ANSI C | Defined by the ANSI C Standard. |
| ANSI C++ | Included in the ANSI C++ proposal. |
| OS/2 | Available for OS/2. |

If more than one function is discussed and their portability features are identical, only one row is used. Otherwise, each function is represented in a separate row.

**Remarks**

This section describes what *function* does, the parameters it takes, and any details you need to use *function* and the related routines listed.

**Return value**

The value that *function* returns (if any) is given here. If *function* sets any global variables, their values are also listed.

**See also**

Routines related to *function* that you might want to read about are listed here. If a routine name contains an *ellipsis*, it indicates that you should refer to a family of functions (for example, *exec...* refers to the entire family of *exec* functions: *execl, execle, execlp, execlpe, execv, execve, execvp,* and *execvpe*).

**Example**

The *function* examples have been moved into online Help so that you can easily cut-and-paste them to your own applications.

## abort — stdlib.h

**Function** Abnormally terminates a program.

**Syntax**

```
void abort(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *abort* causes an abnormal program termination by calling *raise*(SIGABRT). If there is no signal handler for SIGABRT, then *abort* writes a termination message ("`Abnormal program termination`") on stderr, then aborts the program by a call to *_exit* with exit code 3.

**Return value** *abort* returns the exit code 3 to the parent process or to the operating system command processor.

**See also** *assert, atexit, _exit, exit, raise, signal, spawn...*

## abs — stdlib.h

**Function** Returns the absolute value of an integer.

**Syntax**

```
int abs(int x);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *abs* returns the absolute value of the integer argument *x*. If *abs* is called when stdlib.h has been included, it's treated as a macro that expands to inline code.

If you want to use the *abs* function instead of the macro, include `#undef abs` in your program, after the `#include <stdlib.h>`.

This function can be used with *bcd* and *complex* types.

**Return value** The *abs* function returns an integer in the range of 0 to INT_MAX, with the exception that an argument with the value INT_MIN is returned as INT_MIN. The values for INT_MAX and INT_MIN are defined in header file limits.h.

**See also** *bcd, cabs, complex, fabs, labs*

## access — io.h

**Function**

Determines accessibility of a file.

**Syntax**

```
int access(const char *filename, int amode);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks**

*access* checks the file named by *filename* to determine if it exists, and whether it can be read, written to, or executed.

The list of *amode* values is as follows:

06 Check for read and write permission
04 Check for read permission
02 Check for write permission
01 Execute (ignored)
00 Check for existence of file

Under DOS, OS/2, and Windows (16- and 32-bit) all existing files have read access (*amode* equals 04), so 00 and 04 give the same result. Similarly, *amode* values of 06 and 02 are equivalent because under DOS write access implies read access.

If *filename* refers to a directory, *access* simply determines whether the directory exists.

**Return value**

If the requested access is allowed, *access* returns 0; otherwise, it returns a value of –1, and the global variable *errno* is set to one of the following values:

EACCES Permission denied
ENOENT Path or file name not found

**See also**

*chmod, fstat, stat*

## acos, acosl — math.h

**Function**

Calculates the arc cosine.

**Syntax**

```
double acos(double x);
long double acosl(long double x);
```

| | DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|---|
| *acos* | ■ | ■ | ■ | ■ | ■ | ■ | ■ |
| *acosl* | ■ | | ■ | ■ | | | ■ |

**Remarks**

*acos* returns the arc cosine of the input value. *acosl* is the **long double** version; it takes a **long double** argument and returns a **long double** result. Arguments to *acos* and *acosl* must be in the range –1 to 1, or else *acos* and *acosl* return NAN and set the global variable *errno* to

EDOM Domain error

This function can be used with *bcd* and *complex* types.

**Return value**

*acos* and *acosl* of an argument between –1 and +1 return a value in the range 0 to *pi*. Error handling for these routines can be modified through the functions *_matherr* and *_matherrl*.

**See also**

*asin, atan, atan2, bcd, complex, cos, _matherr, sin, tan*

## alloca — malloc.h

**Function**

Allocates temporary stack space.

**Syntax**

```
void *alloca(size_t size);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks**

*alloca* allocates size bytes on the stack; the allocated space is automatically freed up when the calling function exits.

Because *alloca* modifies the stack pointer, do not place calls to *alloca* in an expression that is an argument to a function.

The *alloca* function should not be used in the try-block of a C++ program. If an exception is thrown any values placed on the stack by *alloca* will be corrupted.

If the calling function does not contain any references to local variables in the stack, the stack will not be restored correctly when the function exits,

resulting in a program crash. To ensure that the stack is restored correctly, use the following code in the calling function:

```
char *p;
char dummy[5];

dummy[0] = 0;
   ⋮
p = alloca(nbytes);
```

**Return value** If enough stack space is available, *alloca* returns a pointer to the allocated stack area. Otherwise, it returns NULL.

**See also** *malloc*

## asctime — time.h

**Function** Converts date and time to ASCII.

**Syntax**

```
char *asctime(const struct tm *tblock);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *asctime* converts a time stored as a structure in **tblock* to a 26-character string of the same form as the *ctime* string:

```
Sun Sep 16 01:03:52 1973\n\0
```

**Return value** *asctime* returns a pointer to the character string containing the date and time. This string is a static variable that is overwritten with each call to *asctime*.

**See also** *ctime, difftime, ftime, gmtime, localtime, mktime, strftime, stime, time, tzset*

## asin, asinl — math.h

**Function** Calculates the arc sine.

**Syntax**

```
double asin(double x);
long double asinl(long double x);
```

| | DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|---|
| *asin* | ■ | ■ | ■ | ■ | ■ | ■ | ■ |
| *asinl* | ■ | | ■ | ■ | | | ■ |

**Remarks**

*asin* of a real argument returns the arc sine of the input value. *sinl* is the **long double** version; it takes a **long double** argument and returns a **long double** result.

Real arguments to *asin* and *asinl* must be in the range –1 to 1, or else *asin* and *asinl* return NAN and set the global variable *errno* to

EDOM Domain error

This function can be used with *bcd* and *complex* types.

**Return value**

*asin* and *asinl* of a real argument return a value in the range $-pi/2$ to $pi/2$. Error handling for these functions can be modified through the functions *_matherr* and *_matherrl*.

**See also**

*acos, atan, atan2, bcd, complex, cos, _matherr, sin, tan*

## assert — assert.h

**Function**

Tests a condition and possibly aborts.

**Syntax**

```
void assert(int test);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks**

*assert* is a macro that expands to an **if** statement; if *test* evaluates to zero, *assert* prints a message on *stderr* and aborts the program (by calling *abort*).

*assert* displays this message:

```
Assertion failed: test, file filename, line linenum
```

The *filename* and *linenum* listed in the message are the source file name and line number where the *assert* macro appears.

If you place the `#define NDEBUG` directive ("no debugging") in the source code before the `#include <assert.h>` directive, the effect is to comment out the *assert* statement.

**Return value**

None.

**See also**

*abort*

## atan, atanl — math.h

**Function**

Calculates the arc tangent.

**Syntax**

```
double atan(double x);
long double atanl(long double x);
```

| | DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|---|
| *atan* | ■ | ■ | ■ | ■ | ■ | ■ | ■ |
| *atanl* | ■ | | ■ | ■ | | | ■ |

**Remarks**

*atan* calculates the arc tangent of the input value.

*atanl* is the **long double** version; it takes a **long double** argument and returns a **long double** result. This function can be used with *bcd* and *complex* types.

**Return value**

*atan* and *atanl* of a real argument return a value in the range $-pi/2$ to $pi/2$. Error handling for these functions can be modified through the functions *_matherr* and *_matherrl*.

**See also**

*acos, asin, atan2, bcd, complex, cos, _matherr, sin, tan*

## atan2, atan2l — math.h

**Function**

Calculates the arc tangent of $y/x$.

**Syntax**

```
double atan2(double y, double x);
long double atan2l(long double y, long double x);
```

| | DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|---|
| *atan2* | ■ | ■ | ■ | ■ | ■ | ■ | ■ |
| *atan2l* | ■ | | ■ | ■ | | | ■ |

**Remarks**

*atan2* returns the arc tangent of $y/x$; it produces correct results even when the resulting angle is near $pi/2$ or $-pi/2$ ($x$ near 0). If both $x$ and $y$ are set to 0, the function sets the global variable *errno* to EDOM, indicating a domain error.

*atan2l* is the **long double** version; it takes **long double** arguments and returns a **long double** result.

**Return value**

*atan2* and *atan2l* return a value in the range $-pi$ to $pi$. Error handling for these functions can be modified through the functions *_matherr* and *_matherrl*.

**See also**

*acos, asin, atan, cos, _matherr, sin, tan*

## atexit — stdlib.h

**Function**

Registers termination function.

**Syntax**

```
int atexit(void (_USERENTRY * func)(void));
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | ■ | ■ | ■ |

**Remarks**

*atexit* registers the function pointed to by *func* as an exit function. Upon normal termination of the program, *exit* calls *func* just before returning to the operating system. *func* must be used with the _USERENTRY calling convention.

Each call to *atexit* registers another exit function. Up to 32 functions can be registered. They are executed on a last-in, first-out basis (that is, the last function registered is the first to be executed).

**Return value**

*atexit* returns 0 on success and nonzero on failure (no space left to register the function).

**See also**

*abort, _exit, exit, spawn...*

## atof, _atold — math.h

**Function**

Converts a string to a floating-point number.

**Syntax**

```
double atof(const char *s);
long double _atold(const char *s);
```

| | DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|---|
| *atof* | ■ | ■ | ■ | ■ | ■ | ■ | ■ |
| *_atold* | ■ | | ■ | ■ | | | ■ |

**Remarks**

*atof* converts a string pointed to by *s* to **double**; this function recognizes the character representation of a floating-point number, made up of the following:

- An optional string of tabs and spaces
- An optional sign
- A string of digits and an optional decimal point (the digits can be on both sides of the decimal point)
- An optional *e* or *E* followed by an optional signed integer

The characters must match this generic format:

[*whitespace*] [*sign*] [*ddd*] [.] [*ddd*] [e | E[*sign*]*ddd*]

*atof* also recognizes +INF and –INF for plus and minus infinity, and +NAN and –NAN for Not-a-Number.

In this function, the first unrecognized character ends the conversion.

*_atold* is the **long double** version; it converts the string pointed to by *s* to a **long double**.

*strtod* and *_strtold* are similar to *atof* and *_atold*; they provide better error detection, and hence are preferred in some applications.

**Return value**

*atof* and *_atold* return the converted value of the input string.

If there is an overflow, *atof* (or *_atold*) returns plus or minus HUGE_VAL (or _LHUGE_VAL), *errno* is set to ERANGE (Result out of range), and *_matherr* (or *_matherrl*) is not called.

**See also**

*atoi, atol, ecvt, fcvt, gcvt, scanf, strtod*

## atoi — stdlib.h

**Function**

Converts a string to an integer.

**Syntax**

```
int atoi(const char *s);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks**

*atoi* converts a string pointed to by *s* to **int**; *atoi* recognizes (in the following order)

- An optional string of tabs and spaces
- An optional sign
- A string of digits

The characters must match this generic format:

[*ws*] [*sn*] [*ddd*]

In this function, the first unrecognized character ends the conversion. There are no provisions for overflow in *atoi* (results are undefined).

**Return value**

*atoi* returns the converted value of the input string. If the string cannot be converted to a number of the corresponding type (**int**), atoi returns 0.

**See also** *atof, atol, ecvt, fcvt, gcvt, scanf, strtod*

## atol — stdlib.h

**Function** Converts a string to a long.

**Syntax**

```
long atol(const char *s);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *atol* converts the string pointed to by *s* to **long**. *atol* recognizes (in the following order)

- An optional string of tabs and spaces
- An optional sign
- A string of digits

The characters must match this generic format:

[*ws*] [*sn*] [*ddd*]

In this function, the first unrecognized character ends the conversion. There are no provisions for overflow in *atol* (results are undefined).

**Return value** *atol* returns the converted value of the input string. If the string cannot be converted to a number of the corresponding type (**long**), *atol* returns 0.

**See also** *atof, atoi, ecvt, fcvt, gcvt, scanf, strtod, strtol, strtoul*

## _atold

See *atof*.

## bdos — dos.h

**Function** Accesses DOS system calls.

**Syntax**

```
int bdos(int dosfun, unsigned dosdx, unsigned dosal);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks**

*bdos* provides direct access to many of the DOS system calls. See your DOS reference manuals for details on each system call.

For system calls that require an integer argument, use *bdos;* if they require a pointer argument, use *bdosptr*. In the large data models (compact, large, and huge), it is important to use *bdosptr* instead of *bdos* for system calls that require a pointer as the call argument.

*dosfun* is defined in your DOS reference manuals.

*dosdx* is the value of register DX.

*dosal* is the value of register AL.

**Return value**

The return value of *bdos* is the value of AX set by the system call.

**See also**

*bdosptr, geninterrupt, int86, int86x, intdos, intdosx*

## bdosptr — dos.h

**Function**

Accesses DOS system calls.

**Syntax**

```
int bdosptr(int dosfun, void *argument, unsigned dosal);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks**

*bdosptr* provides direct access to many of the DOS system calls. See your DOS reference manuals for details of each system call.

For system calls that require an integer argument, use *bdos;* if calls require a pointer argument, use *bdosptr*. In the large data models (compact, large, and huge), it is important to use *bdosptr* for system calls that require a pointer as the call argument. In the small data models, the *argument* parameter to *bdosptr* specifies DX; in the large data models, it gives the DS:DX values to be used by the system call.

*dosfun* is defined in your DOS reference manuals. *dosal* is the value of register AL.

**Return value**

The return value of *bdosptr* is the value of AX on success or –1 on failure. On failure, the global variables *errno* and *_doserrno* are set.

**See also**

*bdos, geninterrupt, int86, int86x, intdos, intdosx*

## _beginthread process.h

**Function** Starts execution of a new thread.

**Syntax**

```
unsigned long _beginthread(_USERENTRY (*start_address)(void *),
                           unsigned stack_size, void *arglist)
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| | | | ■ | | | ■ |

**Remarks** The *_beginthread* function creates and starts a new thread. The thread starts execution at *start_address*. (Note that *start_address* must be declared to be *_USERENTRY*. )The size of its stack in bytes is *stack_size*; the stack is allocated by the operating system after the stack size is rounded up to the next multiple of 4096. The thread is passed *arglist* as its only parameter; it can be NULL, but must be present. The thread terminates by simply returning, or by calling *_endthread*.

Either this function or *_beginthreadNT* must be used instead of the operating system thread-creation API function because *_beginthread* and *_beginthreadNT* perform initialization required for correct operation of the run-time library functions.

This function is available only in the multithread libraries.

The function is also available for OS/2. However, under OS/2 the function returns and **int** and does not require *_USERENTRY*.

**Return value** *_beginthread* returns the handle of the new thread. In the event of an error, the function returns –1, and the global variable *errno* is set to one of the following values:

| | |
|---|---|
| EAGAIN | Too many threads |
| EINVAL | Invalid request |

See also the Win32 description of *GetLastError*.

**See also** *_beginthreadNT*, *_endthread*

## _beginthreadNT process.h

**Function** Starts execution of a new thread under Windows NT.

**Syntax**

```
unsigned long _beginthreadNT(void (_USERENTRY *start_address)(void *),
                             unsigned stack_size, void *arglist,
                             void *security_attrib, unsigned long create_flags,
                             unsigned long *thread_id);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| | | | ■ | | | |

**Remarks**

All multithread Windows NT programs must use *_beginthreadNT* or the *_beginthread* function instead of the operating system thread-creation API function because *_beginthread* and *_beginthreadNT* perform initialization required for correct operation of the run-time library functions. The *_beginthreadNT* function provides support for the operating system security. These functions are available only in the multithread libraries.

The *_beginthreadNT* function creates and starts a new thread. The thread starts execution at *start_address*. (Note that *start_address* must be declared to be *_USERENTRY*.) The size of its stack in bytes is *stack_size;* the stack is allocated by the operating system after the stack size is rounded up to the next multiple of 4096. The thread *arglist* can be NULL, but must be present. The thread terminates by simply returning, or by calling *_endthread*.

The function uses the *security_attr* pointer to access the SECURITY_ATTRIBUTES structure. The structure contains the security attributes for the thread. If *security_attr* is NULL, the thread is created with default security attributes. The thread handle is not inherited if *security_attr* is NULL.

The function reads the *create_flags* variable for flags that provide additional information about the thread creation. This variable can be zero, specifying that the thread will run immediately upon creation. The variable can also be CREATE_SUSPENDED, in which case the thread will not run until the *ResumeThread* function is called. *ResumeThread* is provided by the Win32 API. See the Win32 description of *ResumeThread* for additional information.

The function initializes the *thread_id* variable with the thread identifier.

**Return value**

*_beginthreadNT* returns the handle of the new thread. In the event of an error, the function returns –1, and the global variable *errno* is set to one of the following values:

| | |
|---|---|
| EAGAIN | Too many threads |
| EINVAL | Invalid request |

**See also**

*_beginthread*, *_endthread*

# biosequip bios.h

**Function** Checks equipment.

**Syntax**

```
int biosequip(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks** *biosequip* uses BIOS interrupt 0x11 to return an integer describing the equipment connected to the system.

**Return value** The return value is interpreted as a collection of bit-sized fields. The IBM PC values follow:

**Bits 14-15** Number of parallel printers installed
00 = 0 printers
01 = 1 printer
10 = 2 printers
11 = 3 printers

**Bit 13** Serial printer attached
**Bit 12** Game I/O attached

DOS only sees two ports but can be pushed to see four; the IBM PS/2 can see up to eight.

**Bits 9-11** Number of COM ports
000 = 0 ports
001 = 1 port
010 = 2 ports
011 = 3 ports
100 = 4 ports
101 = 5 ports
110 = 6 ports
111 = 7 ports

**Bit 8** Direct memory access (DMA)
0 = Machine has DMA
1 = Machine does not have DMA; for example, PC Jr.

**Bits 6-7** Number of disk drives
00 = 1 drive
01 = 2 drives
10 = 3 drives
11 = 4 drives, only if bit 0 is 1

**Bits 4-5** Initial video mode
00 = Unused
01 = 40x25 BW with color card
10 = 80x25 BW with color card
11 = 80x25 BW with mono card

**Bits 2-3** Motherboard RAM size
00 = 16K
01 = 32K
10 = 48K
11 = 64K

**Bit 1** Floating-point coprocessor
**Bit 0** Boot from disk

## _bios_equiplist — bios.h

**Function** Checks equipment.

**Syntax**

```
unsigned _bios_equiplist(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks** *_bios_equiplist* uses BIOS interrupt 0x11 to return an integer describing the equipment connected to the system.

**Return value** The return value is interpreted as a collection of bit-sized fields. The IBM PC values follow:

**Bits 14-15** Number of parallel printers installed
00 = 0 printers
01 = 1 printer
10 = 2 printers
11 = 3 printers

**Bit 13** Serial printer attached
**Bit 12** Game I/O attached

**Bits 9-11** Number of COM ports
000 = 0 ports
001 = 1 port
010 = 2 ports
011 = 3 ports
100 = 4 ports
101 = 5 ports

110 = 6 ports
111 = 7 ports

**Bit 8** Direct memory access (DMA)
0 = Machine has DMA
1 = Machine does not have DMA; for example, PC Jr.

**Bits 6-7** Number of disk drives
00 = 1 drive
01 = 2 drives
10 = 3 drives
11 = 4 drives, only if bit 0 is 1

**Bit 4-5** Initial video mode
00 = Unused
01 = 40x25 BW with color card
10 = 80x25 BW with color card
11 = 80x25 BW with mono card

**Bits 2-3** Motherboard RAM size
00 = 16K
01 = 32K
10 = 48K
11 = 64K

**Bit 1** Floating-point coprocessor
**Bit 0** Boot from disk

## bioskey — bios.h

**Function** Keyboard interface, using BIOS services directly.

**Syntax**

```
int bioskey(int cmd);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | | | | | |

**Remarks** *bioskey* performs various keyboard operations using BIOS interrupt 0x16. The parameter *cmd* determines the exact operation.

**Return value** The value returned by *bioskey* depends on the task it performs, determined by the value of *cmd*:

0 If the lower 8 bits are nonzero, *bioskey* returns the ASCII character for the next keystroke waiting in the queue or the next key pressed at the keyboard. If the lower 8 bits are zero, the upper 8

bits are the extended keyboard codes defined in the IBM PC *Technical Reference Manual.*

1 This tests whether a keystroke is available to be read. A return value of zero means no key is available. The return value is 0xFFFFF (-1) if *Ctrl-Brk* has been pressed. Otherwise, the value of the next keystroke is returned. The keystroke itself is kept to be returned by the next call to *bioskey* that has a *cmd* value of zero.

2 Requests the current shift key status. The value is obtained by ORing the following values together:

| | | |
|---|---|---|
| Bit 7 | 0x80 | *Insert* on |
| Bit 6 | 0x40 | *Caps* on |
| Bit 5 | 0x20 | *Num Lock* on |
| Bit 4 | 0x10 | *Scroll Lock* on |
| Bit 3 | 0x08 | *Alt* pressed |
| Bit 2 | 0x04 | *Ctrl* pressed |
| Bit 1 | 0x02 | ← *Shift* pressed |
| Bit 0 | 0x01 | → *Shift* pressed |

## biosmemory — bios.h

**Function** Returns memory size.

**Syntax**

```
int biosmemory(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks** *biosmemory* returns the size of RAM memory using BIOS interrupt 0x12. This does not include display adapter memory, extended memory, or expanded memory.

**Return value** *biosmemory* returns the size of RAM memory in 1K blocks.

## _bios_memsize — bios.h

**Function** Returns memory size.

**Syntax**

```
unsigned _bios_memsize(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks** *_bios_memsize* returns the size of RAM memory using BIOS interrupt 0x12. This does not include display adapter memory, extended memory, or expanded memory.

**Return value** *_bios_memsize* returns the size of RAM memory in 1K blocks.

## biostime — bios.h

**Function** Reads or sets the BIOS timer.

**Syntax**

```
long biostime(int cmd, long newtime);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks** *biostime* either reads or sets the BIOS timer. This is a timer counting ticks since midnight at a rate of roughly 18.2 ticks per second. *biostime* uses BIOS interrupt 0x1A.

If *cmd* equals 0, *biostime* returns the current value of the timer. If *cmd* equals 1, the timer is set to the **long** value in *newtime*.

**Return value** When *biostime* reads the BIOS timer (*cmd* = 0), it returns the timer's current value.

## _bios_timeofday — bios.h

**Function** Reads or sets the BIOS timer.

**Syntax**

```
unsigned _bios_timeofday(int cmd, long *timep);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks** *_bios_timeofday* either reads or sets the BIOS timer. This is a timer counting ticks since midnight at a rate of roughly 18.2 ticks per second. *_bios_timeofday* uses BIOS interrupt 0x1A.

The *cmd* parameter can be either of the following values:

| | |
|---|---|
| _TIME_GETCLOCK | The function stores the current BIOS timer value into the location pointed to by *timep*. If the timer has not been read or written since midnight, the function returns 1. Otherwise, the function returns 0. |
| _TIME_SETCLOCK | The function sets the BIOS timer to the long value pointed to by *timep*. The function does not return a value. |

**Return value**

The *_bios_timeofday* returns the value in AX that was set by the BIOS timer call.

## bsearch — stdlib.h

**Function**

Binary search of an array.

**Syntax**

```
void *bsearch(const void *key, const void *base, size_t nelem, size_t width,
              int (_USERENTRY *fcmp)(const void *, const void *));
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks**

*bsearch* searches a table (array) of *nelem* elements in memory, and returns the address of the first entry in the table that matches the search key. The array must be in order. If no match is found, *bsearch* returns 0. Note that because this is a binary search, the first matching entry is not necessarily the first entry in the table.

The type *size_t* is defined in stddef.h header file.

- *nelem* gives the number of elements in the table.
- *width* specifies the number of bytes in each table entry.

The comparison routine *fcmp* must be used with the _USERENTRY calling convention.

*fcmp* is called with two arguments: *elem1* and *elem2*. Each argument points to an item to be compared. The comparison function compares each of the pointed-to items (**elem1* and **elem2*), and returns an integer based on the results of the comparison.

For *bsearch*, the *fcmp* return value is

- < 0 if **elem1* < **elem2*
- == 0 if **elem1* == **elem2*
- > 0 if **elem1* > **elem2*

**Return value**

*bsearch* returns the address of the first entry in the table that matches the search key. If no match is found, *bsearch* returns 0.

**See also**

*lfind, lsearch, qsort*

## cabs, cabsl — math.h

**Function**

Calculates the absolute value of complex number.

**Syntax**

```
double cabs(struct complex z);
long double cabsl(struct _complexl z);
```

| | DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|---|
| *cabs* | ■ | ■ | ■ | ■ | | | ■ |
| *cabsl* | ■ | | ■ | ■ | | | ■ |

**Remarks**

*cabs* is a macro that calculates the absolute value of *z*, a complex number. *z* is a structure with type *complex*. The structure is defined in math.h as

```
struct complex {
   double x, y;
   };


struct _complexl {
   long double x, y;
};
```

where *x* is the real part, and *y* is the imaginary part.

Calling *cabs* is equivalent to calling *sqrt* with the real and imaginary components of *z*, as shown here:

```
sqrt(z.x * z.x + z.y * z.y)
```

*cabsl* is the **long double** version; it takes a structure with type *_complexl* as an argument, and returns a **long double** result.

If you're using C++, you may also use the *complex* class defined in complex.h, and use the function *abs* to get the absolute value of a *complex* number.

**Return value**

*cabs* (or *cabsl*) returns the absolute value of z, a double. On overflow, *cabs* (or *cabsl*) returns HUGE_VAL (or _LHUGE_VAL) and sets the global variable *errno* to

ERANGE Result out of range

Error handling for these functions can be modified through the functions *_matherr* and *_matherrl*.

**See also**

*abs, complex, errno* (global variable), *fabs, labs, _matherr*

## calloc — stdlib.h

**Function**

Allocates main memory.

**Syntax**

```
void *calloc(size_t nitems, size_t size);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks**

*calloc* provides access to the C memory heap. The heap is available for dynamic allocation of variable-sized blocks of memory. Many data structures, such as trees and lists, naturally employ heap memory allocation.

Memory models are available only for 16-bit applications.

All the space between the end of the data segment and the top of the program stack is available for use in the small data models (small, and medium), except for a small margin immediately before the top of the stack. This margin allows room for the application to grow on the stack, and provides a small amount of room needed by the operating system.

In the large data models (compact, large, and huge), all space beyond the program stack to the end of physical memory is available for the heap.

*calloc* allocates a block of size *nitems* × *size*. The block is cleared to 0. If you want to allocate a block larger than 64K, you must use *farcalloc*.

**Return value**

*calloc* returns a pointer to the newly allocated block. If not enough space exists for the new block or if *nitems* or *size* is 0, *calloc* returns NULL.

**See also**

*farcalloc, free, malloc, realloc*

## ceil, ceill — math.h

**Function**

Rounds up.

**Syntax**

```
double ceil(double x);
long double ceill(long double x);
```

| | DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|---|
| *ceil* | ■ | ■ | ■ | ■ | ■ | ■ | ■ |
| *ceill* | ■ | | ■ | ■ | | | ■ |

**Remarks**

*ceil* finds the smallest integer not less than *x*. *ceill* is the **long double** version; it takes a **long double** argument and returns a **long double** result.

**Return value**

These functions return the integer found as a **double** (*ceil*) or a **long double** (*ceill*).

**See also**

*floor, fmod*

## _c_exit — process.h

**Function**

Performs _exit cleanup without terminating the program.

**Syntax**

```
void _c_exit(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks**

*_c_exit* performs the same cleanup as *_exit*, except that it does not terminate the calling process.

**Return value**

None.

**See also**

*abort, atexit, _cexit, exec..., _exit, exit, signal, spawn...*

## _cexit — process.h

**Function**

Performs exit cleanup without terminating the program.

**Syntax**

```
void _cexit(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks**

*_cexit* performs the same cleanup as *exit*, except that it does not close files or terminate the calling process. Buffered output (waiting to be output) is written, and any registered "exit functions" (posted with *atexit*) are called.

**Return value** None.

**See also** *abort, atexit, _c_exit,exec..., _exit, exit, signal, spawn...*

## cgets — conio.h

**Function** Reads a string from the console.

**Syntax**

```
char *cgets(char *str);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | | ■ | | | ■ |

**Remarks** *cgets* reads a string of characters from the console, storing the string (and the string length) in the location pointed to by *str*.

*cgets* reads characters until it encounters a carriage-return/linefeed (CR/LF) combination, or until the maximum allowable number of characters have been read. If *cgets* reads a CR/LF combination, it replaces the combination with a \0 (null character) before storing the string.

Before *cgets* is called, set *str*[0] to the maximum length of the string to be read. On return, *str*[1] is set to the number of characters actually read. The characters read start at *str*[2] and end with a null character. Thus, *str* must be at least *str*[0] plus 2 bytes long.

This function should not be used in Win32s or Win32 GUI applications.

**Return value** On success, *cgets* returns a pointer to *str*[2].

**See also** *cputs, fgets, getch, getche, gets*

## _chain_intr — dos.h

**Function** Chains to another interrupt handler.

**Syntax**

```
void _chain_intr(void (interrupt far *newhandler)());
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks** The *_chain_intr* function passes control from the currently executing interrupt handler to the new interrupt handler whose address is *newhandler*. The current register set is *not* passed to the new handler. Instead, the new handler receives the registers that were stacked (and

possibly modified in the stack) by the old handler. The new handler can simply return, as if it were the original handler. The old handler is not entered again.

The *_chain_intr* function can be called only by C interrupt functions. It is useful when writing a TSR that needs to insert itself in a chain of interrupt handlers (such as the keyboard interrupt).

**Return value** None.

**See also** *_dos_getvect, _dos_setvect,*

## chdir — dir.h

**Function** Changes current directory.

**Syntax**

```
int chdir(const char *path);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks** *chdir* causes the directory specified by *path* to become the current working directory. *path* must specify an existing directory.

A drive can also be specified in the *path* argument, such as

```
chdir("a:\\BC")
```

but this changes only the current directory on that drive; it doesn't change the active drive.

Only the current process is affected.

**Return value** Upon successful completion, *chdir* returns a value of 0. Otherwise, it returns a value of –1, and the global variable *errno* is set to

ENOENT Path or file name not found

**See also** *getcurdir, getcwd, getdisk, mkdir, rmdir, setdisk, system*

## _chdrive — direct.h

**Function** Sets current disk drive.

**Syntax**

```
int _chdrive(int drive);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks**

*_chdrive* sets the current drive to the one associated with *drive*: 1 for A, 2 for B, 3 for C, and so on.

This function changes the current drive of the parent process.

**Return value**

*_chdrive* returns 0 if the current drive was changed successfully; otherwise, it returns –1.

**See also**

*_dos_setdrive*

## _chmod — dos.h, io.h

Obsolete function. See *_rtl_chmod*.

## chmod — sys\stat.h

**Function**

Changes file access mode.

**Syntax**

```
int chmod(const char *path, int amode);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks**

*chmod* sets the file-access permissions of the file given by *path* according to the mask given by *amode*. *path* points to a string.

*amode* can contain one or both of the symbolic constants S_IWRITE and S_IREAD (defined in sys\stat.h).

| Value of *amode* | Access permission |
|---|---|
| S_IWRITE | Permission to write |
| S_IREAD | Permission to read |
| S_IREADIS_IWRITE | Permission to read and write |

Write permission implies read permission.

**Return value**

Upon successfully changing the file access mode, *chmod* returns 0. Otherwise, *chmod* returns a value of –1.

In the event of an error, the global variable *errno* is set to one of the following values:

| | |
|---|---|
| EACCES | Permission denied |
| ENOENT | Path or file name not found |

**See also** *access, _rtl_chmod, fstat, open, sopen, stat*

## chsize — io.h

**Function** Changes the file size.

**Syntax**

```
int chsize(int handle, long size);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *chsize* changes the size of the file associated with *handle*. It can truncate or extend the file, depending on the value of *size* compared to the file's original size.

The mode in which you open the file must allow writing.

If *chsize* extends the file, it will append null characters (\0). If it truncates the file, all data beyond the new end-of-file indicator is lost.

**Return value** On success, *chsize* returns 0. On failure, it returns –1 and the global variable *errno* is set to one of the following values:

| | |
|---|---|
| EACCES | Permission denied |
| EBADF | Bad file number |
| ENOSPC | No space left on device |

**See also** *close, _rtl_creat, creat, open*

## _clear87 — float.h

**Function** Clears floating-point status word.

**Syntax**

```
unsigned int _clear87 (void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *_clear87* clears the floating-point status word, which is a combination of the 80x87 status word and other conditions detected by the 80x87 exception handler.

**Return value** The bits in the value returned indicate the floating-point status before it was cleared. For information on the status word, refer to the constants defined in float.h.

**See also** *_control87, _fpreset, _status87*

## clearerr — stdio.h

**Function** Resets error indication.

**Syntax**

```
void clearerr(FILE *stream);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *clearerr* resets the named stream's error and end-of-file indicators to 0. Once the error indicator is set, stream operations continue to return error status until a call is made to *clearerr* or *rewind*. The end-of-file indicator is reset with each input operation.

**Return value** None.

**See also** *eof, feof, ferror, perror, rewind*

## clock — time.h

**Function** Determines processor time.

**Syntax**

```
clock_t clock(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | ■ | ■ | ■ |

**Remarks** *clock* can be used to determine the time interval between two events. To determine the time in seconds, the value returned by *clock* should be divided by the value of the macro *CLK_TCK*.

**Return value** The *clock* function returns the processor time elapsed since the beginning of the program invocation. If the processor time is not available, or its value cannot be represented, the function returns the value –1.

**See also** *time*



## _close io.h

Obsolete function. See *_rtl_close*.

## close io.h

**Function** Closes a file.

**Syntax**

```
int close(int handle);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks** *close* closes the file associated with *handle*, a file handle obtained from a *_rtl_creat*, *creat*, *creatnew*, *creattemp*, *dup*, *dup2*, *_rtl_open*, or *open* call.

The function does not write a *Ctrl-Z* character at the end of the file. If you want to terminate the file with a *Ctrl-Z*, you must explicitly output one.

**Return value** Upon successful completion, *close* returns 0. Otherwise, the function returns a value of –1.

*close* fais if *handle* is not the handle of a valid, open file, and the global variable *errno* is set to

EBADF Bad file number

**See also** *chsize*, *creat*, *creatnew*, *dup*, *fclose*, *open*, *_rtl_close*, *sopen*

## closedir dirent.h

**Function** Closes a directory stream.

**Syntax**

```
int closedir(DIR *dirp);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks** On UNIX platforms, *closedir* is available on POSIX-compliant systems.

The *closedir* function closes the directory stream *dirp*, which must have been opened by a previous call to *opendir*. After the stream is closed, *dirp* no longer points to a valid directory stream.

**Return value** If *closedir* is successful, it returns 0. Otherwise, *closedir* returns –1 and sets the global variable *errno* to

| | |
|---|---|
| EBADF | The *dirp* argument does not point to a valid open directory stream |

**See also** *errno* (global variable), *opendir*, *readdir*, *rewinddir*

## clreol — conio.h

**Function** Clears to end of line in text window.

**Syntax**

```
void clreol(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *clreol* clears all characters from the cursor position to the end of the line within the current text window, without moving the cursor.

This function should not be used in Win32s or Win32 GUI applications.

**Return value** None.

**See also** *clrscr*, *delline*, *window*

## clrscr — conio.h

**Function** Clears the text-mode window.

**Syntax**

```
void clrscr(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *clrscr* clears the current text window and places the cursor in the upper left-hand corner (at position 1,1).

This function should not be used in Win32s or Win32 GUI applications.

**Return value** None.

**See also** *clreol*, *delline*, *window*

## _control87 — float.h



**Function** Manipulates the floating-point control word.

**Syntax**

```
unsigned int _control87(unsigned int newcw, unsigned int mask);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks**

*_control87* retrieves or changes the floating-point control word.

The floating-point control word is an **unsigned int** that, bit by bit, specifies certain modes in the floating-point package; namely, the precision, infinity, and rounding modes. Changing these modes lets you mask or unmask floating-point exceptions.

*_control87* matches the bits in *mask* to the bits in *newcw*. If a *mask* bit equals 1, the corresponding bit in *newcw* contains the new value for the same bit in the floating-point control word, and *_control87* sets that bit in the control word to the new value.

Here's a simple illustration:

| | | | | |
|---|---|---|---|---|
| Original control word: | 0100 | 0011 | 0110 | 0011 |
| *mask*: | 1000 | 0001 | 0100 | 1111 |
| *newcw*: | 1110 | 1001 | 0000 | 0101 |
| Changing bits: | 1*xxx* | *xxx*1 | *x*0*xx* | 0101 |

If *mask* equals 0, *_control87* returns the floating-point control word without altering it.

**Return value** The bits in the value returned reflect the new floating-point control word. For a complete definition of the bits returned by *_control87*, see the header file float.h.

**See also** *_clear87*, *_fpreset*, *signal*, *_status87*

## cos, cosl — math.h

**Function** Calculates the cosine of a value.

**Syntax**

```
double cos(double x);
long double cosl(long double x);
```

| | DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|---|
| *cos* | ■ | ■ | ■ | ■ | ■ | ■ | ■ |
| *cosl* | ■ | | ■ | ■ | | | ■ |

**Remarks**

*cos* computes the cosine of the input value. The angle is specified in radians.

*cosl* is the **long double** version; it takes a **long double** argument and returns a **long double** result.

This function can be used with *bcd* and *complex* types.

**Return value**

*cos* of a real argument returns a value in the range –1 to 1. Error handling for these functions can be modified through *_matherr* (or *_matherrl*).

**See also**

*acos, asin, atan, atan2, bcd, complex, _matherr, sin, tan*

## cosh, coshl — math.h

**Function**

Calculates the hyperbolic cosine of a value.

**Syntax**

```
double cosh(double x);
long double coshl(long double x);
```

| | DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|---|
| *cosh* | ■ | ■ | ■ | ■ | ■ | ■ | ■ |
| *coshl* | ■ | | ■ | ■ | | | ■ |

**Remarks**

*cosh* computes the hyperbolic cosine, $(e^x + e^{-x})/2$. *coshl* is the **long double** version; it takes a **long double** argument and returns a **long double** result.

This function can be used with *bcd* and *complex* types.

**Return value**

*cosh* returns the hyperbolic cosine of the argument.

When the correct value would create an overflow, these functions return the value HUGE_VAL (*cosh*) or _LHUGE_VAL (*coshl*) with the appropriate sign, and the global variable *errno* is set to ERANGE. Error handling for these functions can be modified through the functions *_matherr* and *_matherrl*.

**See also**

*acos, asin, atan, atan2, bcd, complex, cos, _matherr, sin, sinh, tan, tanh*

# country

dos.h

C

## Function

Returns country-dependent information.

## Syntax

```
struct COUNTRY *country(int xcode, struct COUNTRY *cp);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | ■ |

## Remarks

The *country* function is not affected by *setlocale*.

*country* specifies how certain country-dependent data (such as dates, times, and currency) will be formatted. The values set by this function depend on the operating system version being used.

If *cp* has a value of –1, the current country is set to the value of *xcode*, which must be nonzero. The *COUNTRY* structure pointed to by *cp* is filled with the country-dependent information of the current country (if *xcode* is set to zero), or the country given by *xcode*.

The structure *COUNTRY* is defined as follows:

```
struct COUNTRY{
   int co_date;              /* date format */
   char co_curr[5];          /* currency symbol */
   char co_thsep[2];         /* thousands separator */
   char co_desep[2];         /* decimal separator */
   char co_dtsep[2];         /* date separator */
   char co_tmsep[2];         /* time separator */
   char co_currstyle;        /* currency style */
   char co_digits;           /* significant digits in currency */
   char co_time;             /* time format */
   long co_case;             /* case map */
   char co_dasep[2];         /* data separator */
   char co_fill[10];         /* filler */
};
```

The date format in *co_date* is

- 0 for the U.S. style of month, day, year.
- 1 for the European style of day, month, year.
- 2 for the Japanese style of year, month, day.

Currency display style is given by *co_currstyle* as follows:

- 0 for the currency symbol to precede the value with no spaces between the symbol and the number.
- 1 for the currency symbol to follow the value with no spaces between the number and the symbol.
- 2 for the currency symbol to precede the value with a space after the symbol.
- 3 for the currency symbol to follow the number with a space before the symbol.

**Return value**

On success, *country* returns the pointer argument *cp*. On error, it returns NULL.

## cprintf — conio.h

**Function**

Writes formatted output to the screen.

**Syntax**

```
int cprintf(const char *format[, argument, ...]);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | | ■ | | | ■ |

**Remarks**

See *printf* for details on format specifiers.

*cprintf* accepts a series of arguments, applies to each a format specifier contained in the format string pointed to by *format*, and outputs the formatted data directly to the current text window on the screen. There must be the same number of format specifiers as arguments.

The string is written either directly to screen memory or by way of a BIOS call, depending on the value of the global variable *directvideo*.

Unlike *fprintf* and *printf*, *cprintf* does not translate linefeed characters (\n) into carriage-return/linefeed character pairs (\r\n). Tab characters (specified by \t) are not expanded into spaces.

This function should not be used in Win32s or Win32 GUI applications.

**Return value**

*cprintf* returns the number of characters output.

**See also**

*directvideo* (global variable), *fprintf*, *printf*, *putch*, *sprintf*, *vprintf*

## cputs — conio.h

**Function** Writes a string to the screen.

**Syntax**

```
int cputs(const char *str);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | | ■ | | | ■ |

**Remarks** *cputs* writes the null-terminated string *str* to the current text window. It does not append a newline character.

The string is written either directly to screen memory or by way of a BIOS call, depending on the value of the global variable *directvideo*. Unlike *puts*, *cputs* does not translate linefeed characters (\n) into carriage-return/linefeed character pairs (\r\n).

➥ This function should not be used in Win32s or Win32 GUI applications.

**Return value** *cputs* returns the last character printed.

**See also** *cgets*, *_directvideo* (global variable), *fputs*, *putch*, *puts*

## _creat — io.h

Obsolete function. See *_rtl_creat*.

## creat — io.h

**Function** Creates a new file or overwrites an existing one.

**Syntax**

```
int creat(const char *path, int amode);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/ |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks** *creat* creates a new file or prepares to rewrite an existing file given by *path*. *amode* applies only to newly created files.

A file created with *creat* is always created in the translation mode specified by the global variable *_fmode* (O_TEXT or O_BINARY).

If the file exists and the write attribute is set, *creat* truncates the file to a length of 0 bytes, leaving the file attributes unchanged. If the existing file has the read-only attribute set, the *creat* call fails and the file remains unchanged.

The *creat* call examines only the S_IWRITE bit of the access-mode word *amode*. If that bit is 1, the file can be written to. If the bit is 0, the file is marked as read-only. All other operating system attributes are set to 0.

*amode* can be one of the following (defined in sys\stat.h):

| Value of *amode* | Access permission |
|---|---|
| S_IWRITE | Permission to write |
| S_IREAD | Permission to read |
| S_IREAD\|S_IWRITE | Permission to read and write |

➥ Write permission implies read permission.

**Return value**

Upon successful completion, *creat* returns the new file handle, a non-negative integer; otherwise, it returns –1.

In the event of error, the global variable *errno* is set to one of the following:

| | |
|---|---|
| EACCES | Permission denied |
| EMFILE | Too many open files |
| ENOENT | Path or file name not found |

**See also**

*chmod, chsize, close, _rtl_creat, creatnew, creattemp, dup, dup2, _fmode* (global variable), *fopen, open, sopen, write*

## creatnew — io.h

**Function**

Creates a new file.

**Syntax**

```
int creatnew(const char *path, int mode);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks**

*creatnew* is identical to *_rtl_creat* with one exception: If the file exists, *creatnew* returns an error and leaves the file untouched.

The *mode* argument to *creatnew* can be zero or an OR-combination of any one of the following constants (defined in dos.h):

| | |
|---|---|
| FA_HIDDEN | Hidden file |
| FA_RDONLY | Read-only attribute |
| FA_SYSTEM | System file |

**Return value**

Upon successful completion, *creat* returns the new file handle, a non-negative integer; otherwise, it returns –1.

In the event of error, the global variable *errno* is set to one of the following values:

| | |
|---|---|
| EACCES | Permission denied |
| EEXIST | File already exists |
| EMFILE | Too many open files |
| ENOENT | Path or file name not found |

**See also**

*close, _rtl_creat, creat, creattemp, _dos_creatnew, dup, _fmode* (global variable), *open*

## creattemp io.h

**Function**

Creates a unique file in the directory associated with the path name.

**Syntax**

```
int creattemp(char *path, int attrib);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks**

A file created with *creattemp* is always created in the translation mode specified by the global variable *_fmode* (O_TEXT or O_BINARY).

Remember that a backslash in *path* requires '\\'.

*path* is a path name ending with a backslash (\). A unique file name is selected in the directory given by *path*. The newly created file name is stored in the *path* string supplied. *path* should be long enough to hold the resulting file name. The file is not automatically deleted when the program terminates.

*creattemp* accepts *attrib*, a DOS attribute word. Upon successful file creation, the file pointer is set to the beginning of the file. The file is opened for both reading and writing.

The *attrib* argument to *creattemp* can be zero or an OR-combination of any one of the following constants (defined in dos.h):

| | |
|---|---|
| FA_HIDDEN | Hidden file |
| FA_RDONLY | Read-only attribute |
| FA_SYSTEM | System file |

**Return value**

Upon successful completion, the new file handle, a nonnegative integer, is returned; otherwise, –1 is returned.

In the event of error, the global variable *errno* is set to one of the following values:

| | |
|---|---|
| EACCES | Permission denied |
| EMFILE | Too many open files |
| ENOENT | Path or file name not found |

**See also**

*close, _rtl_creat, creat, creatnew, dup, _fmode* (global variable), *open*

## _crotl, _crotr — stdlib.h

**Function**

Rotates an **unsigned char** left or right.

**Syntax**

```
unsigned char _crotl(unsigned char val, int count);
unsigned char _crotr(unsigned char val, int count);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks**

*_crotl* rotates the given *val* to the left *count* bits. *_crotr* rotates the given *val* to the right *count* bits.

The argument *val* is an **unsigned char**, or its equivalent in decimal or hexadecimal form.

**Return value**

The functions return the rotated *val*.

- *_crotl* returns the value of *val* left-rotated *count* bits.
- *_crotr* returns the value of *val* right-rotated *count* bits.

**See also**

*_lrotl, _lrotr, _rotl, _rotr*

## cscanf — conio.h

C

**Function**

Scans and formats input from the console.

**Syntax**

```
int cscanf(char *format[, address, ...]);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | | ■ | | | ■ |

**Remarks**

See *scanf* for details on format specifiers.

*cscanf* scans a series of input fields one character at a time, reading directly from the console. Then each field is formatted according to a format specifier passed to *cscanf* in the format string pointed to by *format*. Finally, *cscanf* stores the formatted input at an address passed to it as an argument following *format*, and echoes the input directly to the screen. There must be the same number of format specifiers and addresses as there are input fields.

*cscanf* might stop scanning a particular field before it reaches the normal end-of-field (whitespace) character, or it might terminate entirely for a number of reasons. See *scanf* for a discussion of possible causes.

➥ This function should not be used in Win32s or Win32 GUI applications.

**Return value**

*cscanf* returns the number of input fields successfully scanned, converted, and stored; the return value does not include scanned fields that were not stored. If no fields were stored, the return value is 0.

If *cscanf* attempts to read at end-of-file , the return value is EOF.

**See also**

*fscanf, getche, scanf, sscanf*

## ctime — time.h

**Function**

Converts date and time to a string.

**Syntax**

```
char *ctime(const time_t *time);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks**

*ctime* converts a time value pointed to by *time* (the value returned by the function *time*) into a 26-character string in the following form, terminating with a newline character and a null character:

```
Mon Nov 21 11:31:54 1983\n\0
```

All the fields have constant width.

The global long variable *timezone* contains the difference in seconds between GMT and local standard time (in PST, *timezone* is 8×60×60). The global variable *daylight* is nonzero *if and only if* the standard U.S. daylight saving time conversion should be applied. These variables are set by the *tzset* function, not by the user program directly.

**Return value**

*ctime* returns a pointer to the character string containing the date and time. The return value points to static data that is overwritten with each call to *ctime*.

**See also**

*asctime, _daylight* (global variable), *difftime, ftime, getdate, gmtime, localtime, settime, time, _timezone* (global variable), *tzset*

## ctrlbrk — dos.h

**Function**

Sets control-break handler.

**Syntax**

```
void ctrlbrk(int (*handler)(void));
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks**

*ctrlbrk* sets a new control-break handler function pointed to by *handler*. The interrupt vector 0x23 is modified to call the named function.

*ctrlbrk* establishes a DOS interrupt handler that calls the named function; the named function is not called directly.

The handler function can perform any number of operations and system calls. The handler does not have to return; it can use *longjmp* to return to an arbitrary point in the program. The handler function returns 0 to abort the current program; any other value causes the program to resume execution.

**Return value**

*ctrlbrk* returns nothing.

**See also**

*getcbrk, signal*

# cwait



**Function** Waits for child process to terminate.

**Syntax**

```
int cwait(int *statloc, int pid, int action);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| | | | ■ | | | ■ |

**Remarks** The *cwait* function waits for a child process to terminate. The process ID of the child to wait for is *pid*. If *statloc* is not NULL, it points to the location where *cwait* will store the termination status. The *action* specifies whether to wait for the process alone, or for the process and all of its children.

If the child process terminated normally (by calling *exit*, or returning from *main*), the termination status word is defined as follows:

**Bits 0-7** Zero.

**Bits 8-15** The least significant byte of the return code from the child process. This is the value that is passed to *exit*, or is returned from *main*. If the child process simply exited from *main* without returning a value, this value will be unpredictable.

If the child process terminated abnormally, the termination status word is defined as follows:

**Bits 0-7** Termination information about the child:

1 Critical error abort.
2 Execution fault, protection exception.
3 External termination signal.

**Bits 8-15** Zero.

If *pid* is 0, *cwait* waits for any child process to terminate. Otherwise, *pid* specifies the process ID of the process to wait for; this value must have been obtained by an earlier call to an asynchronous *spawn* function.

The acceptable values for *action* are WAIT_CHILD, which waits for the specified child only, and WAIT_GRANDCHILD, which waits for the specified child *and* all of its children. These two values are defined in process.h.

**Return value**

When *cwait* returns after a normal child process termination, it returns the process ID of the child.

When *cwait* returns after an abnormal child termination, it returns –1 to the parent and sets *errno* to EINTR (the child process terminated abnormally).

If *cwait* returns without a child process completion, it returns a –1 value and sets *errno* to one of the following values:

| | |
|---|---|
| ECHILD | No child exists or the *pid* value is bad |
| EINVAL | A bad *action* value was specified |

**See also**

*spawn, wait*

## delline — conio.h

**Function**

Deletes line in text window.

**Syntax**

```
void delline(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks**

*delline* deletes the line containing the cursor and moves all lines below it one line up. *delline* operates within the currently active text window.

This function should not be used in Win32s or Win32 GUI applications.

**Return value**

None.

**See also**

*clreol, clrscr, insline, window*

## difftime — time.h

**Function**

Computes the difference between two times.

**Syntax**

```
double difftime(time_t time2, time_t time1);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks**

*difftime* calculates the elapsed time in seconds, from *time1* to *time2*.

**Return value** *difftime* returns the result of its calculation as a **double**.

**See also** *asctime, ctime, _daylight* (global variable), *gmtime, localtime, time, _timezone* (global variable)



## disable, _disable, enable, _enable — dos.h

**Function** Disables and enables interrupts.

**Syntax**

```
void disable(void);
void _disable(void);
void enable(void);
void _enable(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | |

**Remarks** These macros are designed to provide a programmer with flexible hardware interrupt control.

The *disable* and *_disable* macros disable interrupts. Only the NMI (non-maskable interrupt) is allowed from any external device.

The *enable* and *_enable* macros enable interrupts, allowing any device interrupts to occur.

**Return value** None.

**See also** *getvect*

## div — stdlib.h

**Function** Divides two integers, returning quotient and remainder.

**Syntax**

```
div_t  div(int numer, int denom);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | ■ | ■ | ■ |

**Remarks** *div* divides two integers and returns both the quotient and the remainder as a *div_t* type. *numer* and *denom* are the numerator and denominator, respectively. The *div_t* type is a structure of integers defined (with **typedef**) in stdlib.h as follows:

```
typedef struct {
   int  quot;      /* quotient */
   int  rem;       /* remainder */
} div_t;
```

**Return value** *div* returns a structure whose elements are *quot* (the quotient) and *rem* (the remainder).

**See also** *ldiv*

## _dos_close — dos.h

**Function** Closes a file.

**Syntax**

```
unsigned _dos_close(int handle);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | ■ |

**Remarks** *_dos_close* closes the file associated with *handle*. *handle* is a file handle obtained from a *_dos_creat*, *_dos_creatnew*, or *_dos_open* call.

**Return value** Upon successful completion, *_dos_close* returns 0. Otherwise, it returns the operating system error code and the global variable *errno* is set to

EBADF Bad file number

**See also** *_dos_creat*, *_dos_open*, *_dos_read*, *_dos_write*

## _dos_commit — dos.h

**Function** Output a file to the disk.

**Syntax**

```
unsigned _dos_commit(int handle);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks** This function makes DOS flush any output that it has buffered for a specific handle to the disk.

**Return value** The function returns zero on success. On failure the function returns the DOS error code and sets *errno* to EBADF.

**See also** *_rtl_close*, *_rtl_creat*, *_dos_creat*, *_dos_write*

## _dos_creat — dos.h, io.h



**Function**

Creates a new file or overwrites an existing one.

**Syntax**

```
unsigned _dos_creat(const char *path,int attrib,int *handlep);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | ■ |

**Remarks**

*_dos_creat* opens the file specified by *path*. The file is always opened in binary mode. Upon successful file creation, the file pointer is set to the beginning of the file. *_dos_creat* stores the file handle in the location pointed to by *handlep*. The file is opened for both reading and writing.

If the file already exists, its size is reset to 0. (This is essentially the same as deleting the file and creating a new file with the same name.)

| | |
|---|---|
| FA_RDONLY | Read-only attribute |
| FA_HIDDEN | Hidden file |
| FA_SYSTEM | System file |

The *attrib* argument is an ORed combination of one or more of the following constants (defined in dos.h):

| | |
|---|---|
| _A_NORMAL | Normal file |
| _A_RDONLY | Read-only file |
| _A_HIDDEN | Hidden file |
| _A_SYSTEM | System file |

**Return value**

Upon successful completion, *_dos_creat* returns 0. If an error occurs, *_dos_creat* returns the operating system error code.

In the event of error, the global variable *errno* is set to one of the following values:

| | |
|---|---|
| EACCES | Permission denied |
| EMFILE | Too many open files |
| ENOENT | Path or file name not found |

**See also**

*chsize, close, creat, creatnew, creattemp, _rtl_chmod, _rtl_close*

## _dos_creatnew — dos.h

**Function**

Creates a new file.

**Syntax**

```
unsigned _dos_creatnew(const char *path, int attrib, int *handlep);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | ■ |

**Remarks**

*_dos_creatnew* creates and opens the new file *path*. The file is given the access permission *attrib*, an operating-system attribute word. The file is always opened in binary mode. Upon successful file creation, the file handle is stored in the location pointed to by *handlep*, and the file pointer is set to the beginning of the file. The file is opened for both reading and writing.

If the file already exists, *_dos_creatnew* returns an error and leaves the file untouched.

The *attrib* argument to *_dos_creatnew* is an OR combination of one or more of the following constants (defined in dos.h):

| | |
|---|---|
| _A_NORMAL | Normal file |
| _A_RDONLY | Read-only file |
| _A_HIDDEN | Hidden file |
| _A_SYSTEM | System file |

**Return value**

Upon successful completion, *_dos_creatnew* returns 0. Otherwise, it returns the operating system error code, and the global variable *errno* is set to one of the following:

| | |
|---|---|
| EACCES | Permission denied |
| EEXIST | File already exists |
| EMFILE | Too many open files |
| ENOENT | Path or file name not found |

**See also**

*creatnew, _dos_close, _dos_creat, _dos_getfileattr, _dos_setfileattr*

## dosexterr — dos.h

**Function**

Gets extended DOS error information.

**Syntax**

```
int dosexterr(struct DOSERROR *eblkp);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks**

This function fills in the *DOSERROR* structure pointed to by *eblkp* with extended error information after a DOS call has failed. The structure is defined as follows:

```
struct  DOSERROR {
   int de_exterror;     /* extended error */
   char de_class;       /* error class */
   char de_action;      /* action */
   char de_locus;       /* error locus */
};
```

The values in this structure are obtained by way of DOS call 0x59. A *de_exterror* value of 0 indicates that the prior DOS call did not result in an error.

**Return value**

*dosexterr* returns the value *de_exterror*.

## _dos_findfirst — dos.h

**Function**

Searches a disk directory.

**Syntax**

```
unsigned _dos_findfirst(const char *pathname, int attrib,
                        struct find_t *ffblk);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | ■ |

**Remarks**

*_dos_findfirst* begins a search of a disk directory.

*pathname* is a string with an optional drive specifier, path, and file name of the file to be found. The file name portion can contain wildcard match characters (such as ? or *). If a matching file is found, the *find_t* structure pointed to by *ffblk* is filled with the file-directory information.

The format of the *find_t* structure is as follows:

```
struct find_t {
   char reserved[21];     /* reserved by the operating system */
   char attrib;           /* attribute found */
   int wr_time;           /* file time */
   int wr_date;           /* file date */
   long size;             /* file size */
   char name[13];         /* found file name */
};
```

*attrib* is an operating system file-attribute word used in selecting eligible files for the search. *attrib* is an OR combination of one or more of the following constants (defined in dos.h):

| | |
|---|---|
| _A_NORMAL | Normal file |
| _A_RDONLY | Read-only attribute |
| _A_HIDDEN | Hidden file |
| _A_SYSTEM | System file |
| _A_VOLID | Volume label |
| _A_SUBDIR | Directory |
| _A_ARCH | Archive |

For more detailed information about these attributes, refer to your operating system reference manuals.

Note that *wr_time* and *wr_date* contain bit fields for referring to the file's date and time. The structure of these fields was established by the operating system.

| ***wr_time:*** | |
|---|---|
| Bits 0-4 | The result of seconds divided by 2 (for example, 10 here means 20 seconds) |
| Bits 5-10 | Minutes |
| Bits 11-15 | Hours |

| ***wr_date:*** | |
|---|---|
| Bits 0-4 | Day |
| Bits 5-8 | Month |
| Bits 9-15 | Years since 1980 (for example, 9 here means 1989) |

**Return value**

*_dos_findfirst* returns 0 on successfully finding a file matching the search *pathname*. When no more files can be found, or if there is some error in the file name, the operating system error code is returned, and the global variable *errno* is set to

| | |
|---|---|
| ENOENT | Path or file name not found |

**See also**

*_dos_findnext*

## _dos_findnext — dos.h

**Function**

Continues *_dos_findfirst* search.

**Syntax**

```
unsigned _dos_findnext(struct find_t *ffblk);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | ■ |

**Remarks**

*_dos_findnext* is used to fetch subsequent files that match the *pathname* given in *_dos_findfirst*. *ffblk* is the same block filled in by the *_dos_findfirst* call. This

block contains necessary information for continuing the search. One file name for each call to *_dos_findnext* is returned until no more files are found in the directory matching the *pathname*.

**Return value**

*_dos_findnext* returns 0 on successfully finding a file matching the search *pathname*. When no more files can be found, or if there is some error in the file name, the operating system error code is returned, and the global variable *errno* is set to

ENOENT Path or file name not found

**See also**

*_dos_findfirst*

## _dos_getdate, _dos_setdate, getdate, setdate — dos.h

**Function**

Gets and sets system date.

**Syntax**

```
void _dos_getdate(struct dosdate_t *datep);
unsigned _dos_setdate(struct dosdate_t *datep);
void getdate(struct date *datep);
void setdate(struct date *datep);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | ■ |

**Remarks**

*getdate* fills in the *date* structure (pointed to by *datep*) with the system's current date.

*setdate* sets the system date (month, day, and year) to that in the *date* structure pointed to by *datep*.

The *date* structure is defined as follows:

```
struct date {
   int da_year;     /* current year */
   char da_day;     /* day of the month */
   char da_mon;     /* month (1 = Jan) */
};
```

*_dos_getdate* fills in the *dosdate_t* structure (pointed to by *datep*) with the system's current date.

The *dosdate_t* structure is defined as follows:

```
struct dosdate_t {
   unsigned char day;       /* 1-31 */
   unsigned char month;     /* 1-12 */
   unsigned int  year;      /* 1980 - 2099 */
   unsigned char dayofweek; /* 0 - 6 (0=Sunday) */
};
```

**Return value**

*_dos_getdate, getdate,* and *setdate* do not return a value.

If the date is set successfully, *_dos_setdate* returns 0. Otherwise, it returns a nonzero value and the global variable *errno* is set to

EINVAL Invalid date

**See also**

*ctime, gettime, settime*

## _dos_getdiskfree — dos.h

**Function**

Gets disk free space.

**Syntax**

```
unsigned _dos_getdiskfree(unsigned char drive, struct diskfree_t *dtable);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | ■ |

**Remarks**

*_dos_getdiskfree* accepts a drive specifier in *drive* (0 for default, 1 for A, 2 for B, and so on) and fills in the *diskfree_t* structure pointed to by *dtable* with disk characteristics.

The *diskfree_t* structure is defined as follows:

```
struct diskfree_t {
   unsigned avail_clusters;       /* available clusters */
   unsigned total_clusters;       /* total clusters */
   unsigned bytes_per_sector;     /* bytes per sector */
   unsigned sectors_per_cluster;  /* sectors per cluster */
};
```

**Return value**

*_dos_getdiskfree* returns 0 if successful. Otherwise, it returns a nonzero value and the global variable *errno* is set to

EINVAL Invalid drive specified

**See also**

*getfat, getfatd*

## _dos_getdrive, _dos_setdrive — dos.h



**Function** Gets and sets the current drive number.

**Syntax**

```
void _dos_getdrive(unsigned *drivep);
void _dos_setdrive(unsigned drivep, unsigned *ndrives);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | ■ |

**Remarks** *_dos_getdrive* gets the current drive number.

*_dos_setdrive* sets the current drive and stores the total number of drives at the location pointed to by *ndrives*.

The drive numbers at the location pointed to by *drivep* are as follows: 1 for A, 2 for B, 3 for C, and so on.

This function changes the current drive of the parent process.

**Return value** None. Use *_dos_getdrive* to verify that the current drive was changed successfully.

**See also** *getcwd*

## _dos_getfileattr, _dos_setfileattr — dos.h

**Function** Changes file access mode.

**Syntax**

```
int _dos_getfileattr(const char *path, unsigned *attribp);
int _dos_setfileattr(const char *path, unsigned attrib);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | ■ |

**Remarks** *_dos_getfileattr* fetches the file attributes for the file *path*. The attributes are stored at the location pointed to by *attribp*.

*_dos_setfileattr* sets the file attributes for the file *path* to the value *attrib*. The file attributes can be an OR combination of the following symbolic constants (defined in dos.h):

| | |
|---|---|
| _A_RDONLY | Read-only attribute |
| _A_HIDDEN | Hidden file |

| | |
|---|---|
| _A_SYSTEM | System file |
| _A_VOLID | Volume label |
| _A_SUBDIR | Directory |
| _A_ARCH | Archive |
| _A_NORMAL | Normal file (no attribute bits set) |

**Return value**

Upon successful completion, *_dos_getfileattr* and *_dos_setfileattr* return 0. Otherwise, these functions return the operating system error code, and the global variable *errno* is set to

ENOENT Path or file name not found

**See also**

*chmod, stat*

## _dos_getftime, _dos_setftime — dos.h

**Function**

Gets and sets file date and time.

**Syntax**

```
unsigned _dos_getftime(int handle, unsigned *datep, unsigned *timep);
unsigned _dos_setftime(int handle, unsigned date, unsigned time);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | ■ |

**Remarks**

*_dos_getftime* retrieves the file time and date for the disk file associated with the open *handle*. The file must have been previously opened using *_dos_open*, *_dos_creat*, or *_dos_creatnew*. *_dos_getftime* stores the date and time at the locations pointed to by *datep* and *timep*.

*_dos_setftime* sets the file's new date and time values as specified by *date* and *time*.

Note that the date and time values contain bit fields for referring to the file's date and time. The structure of these fields was established by the operating system.

**Date:**

| | |
|---|---|
| Bits 0-4 | Day |
| Bits 5-8 | Month |
| Bits 9-15 | Years since 1980 (for example, 9 here means 1989) |

**Time:**

| | |
|---|---|
| Bits 0-4 | The result of seconds divided by 2 (for example, 10 here means 20 seconds) |
| Bits 5-10 | Minutes |
| Bits 11-15 | Hours |

**Return value**

*_dos_getftime* and *_dos_setftime* return 0 on success.

In the event of an error return, the operating system error code is returned and the global variable *errno* is set to one of the following values:

| | |
|---|---|
| EACCES | Permission denied |
| EBADF | Bad file number |

**See also**

*fstat, stat*

## _dos_gettime, _dos_settime — dos.h

**Function**

Gets and sets system time.

**Syntax**

```
void _dos_gettime(struct dostime_t *timep);
unsigned _dos_settime(struct dostime_t *timep);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | ■ |

**Remarks**

*_dos_gettime* fills in the *dostime_t* structure pointed to by *timep* with the system's current time.

_dos_settime sets the system time to the values in the *dostime_t* structure pointed to by *timep*.

The *dostime_t* structure is defined as follows:

```
struct dostime_t {
   unsigned char hour;     /* hours 0-23 */
   unsigned char minute;   /* minutes 0-59 */
   unsigned char second;   /* seconds 0-59 */
   unsigned char hsecond;  /* hundredths of seconds 0-99 */
};
```

**Return value**

*_dos_gettime* does not return a value.

If *_dos_settime* is successful, it returns 0. Otherwise, it returns the operating system error code, and the global variable *errno* is set to:

| | |
|---|---|
| EINVAL | Invalid time |

**See also**

*_dos_getdate, _dos_setdate, _dos_settime, stime, time*

## _dos_getvect — dos.h

**Function** Gets interrupt vector.

**Syntax**

```
void interrupt(*_dos_getvect(unsigned interruptno)) ();
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks** Every processor of the 8086 family includes a set of interrupt vectors, numbered 0 to 255. The 4-byte value in each vector is actually an address, which is the location of an interrupt function.

*_dos_getvect* reads the value of the interrupt vector given by *interruptno* and returns that value as a (far) pointer to an interrupt function. The value of *interruptno* can be from 0 to 255.

**Return value** *_dos_getvect* returns the current 4-byte value stored in the interrupt vector named by *interruptno.*

**See also** *_disable, _enable, _dos_setvect*

## _dos_open — fcntl.h, share.h, dos.h

**Function** Opens a file for reading or writing.

**Syntax**

```
unsigned _dos_open(const char *filename, unsigned oflags, int *handlep);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | ■ |

**Remarks** *_dos_open* opens the file specified by *filename,* then prepares it for reading or writing, as determined by the value of *oflags.* The file is always opened in binary mode. *_dos_open* stores the file handle at the location pointed to by *handlep.*

*oflags* uses the flags from the following two lists. Only one flag from the first list can be used (and one *must* be used); the remaining flags can be used in any logical combination.

**List 1: Read/write flags**

| | |
|---|---|
| O_RDONLY | Open for reading. |
| O_WRONLY | Open for writing. |
| O_RDWR | Open for reading and writing. |

The following additional values can be included in *oflags* (using an OR operation):

These symbolic constants are defined in fcntl.h and share.h.

**List 2: Other access flags**

| | |
|---|---|
| O_NOINHERIT | The file is not passed to child programs. |
| SH_COMPAT | Allow other opens with SH_COMPAT. The call will fail if the file has already been opened in any other shared mode. |
| SH_DENYRW | Only the current handle can have access to the file. |
| SH_DENWR | Allow only reads from any other open to the file. |
| SH_DENYRD | Allow only writes from any other open to the file. |
| SH_DENYNO | Allow other shared opens to the file, but not other SH_COMPAT opens. |

Only one of the SH_DENY*xx* values can be included in a single *_dos_open*. These file-sharing attributes are in addition to any locking performed on the files.

The maximum number of simultaneously open files is defined by HANDLE_MAX.

**Return value**

On successful completion, *_dos_open* returns 0, and stores the file handle at the location pointed to by *handlep*. The file pointer, which marks the current position in the file, is set to the beginning of the file.

On error, *_dos_open* returns the operating system error code. The global variable *errno* is set to one of the following:

| | |
|---|---|
| EACCES | Permission denied |
| EINVACC | Invalid access code |
| EMFILE | Too many open files |
| ENOENT | Path or file not found |

**See also**

*open, _rtl_read, sopen*

## _dos_read — io.h, dos.h

**Function**

Reads from file.

**Syntax**

```
unsigned _dos_read(int handle, void far *buf, unsigned *nread);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | ■ |

**Remarks**

*_dos_read* reads *len* bytes from the file associated with *handle* into *buf*. The actual number of bytes read is stored at the location pointed to by *nread*; when an error occurs, or the end-of-file is encountered, this number might be less than *len*.

*_dos_read* does not remove carriage returns because it treats all files as binary files.

*handle* is a file handle obtained from a *_dos_creat*, *_dos_creatnew*, or *_dos_open* call.

On disk files, *_dos_read* begins reading at the current file pointer. When the reading is complete, the function increments the file pointer by the number of bytes read. On devices, the bytes are read directly from the device.

The maximum number of bytes that *_dos_read* can read is UINT_MAX –1, because UINT_MAX is the same as –1, the error return indicator. UINT_MAX is defined in limits.h.

**Return value**

On successful completion, *_dos_read* returns 0. Otherwise, the function returns the DOS error code and sets the global variable *errno*.

| | |
|---|---|
| EACCES | Permission denied |
| EBADF | Bad file number |

**See also**

*_rtl_open*, *read*, *_rtl_write*

## _dos_setdate

See *_dos_getdate*.

## _dos_setdrive

See *_dos_getdrive*.

## _dos_setfileattr

See *_dos_getfileattr*.

## _dos_setftime

See *_dos_getftime.*



## _dos_settime

See *_dos_gettime.*

## _dos_setvect dos.h

**Function** Sets interrupt vector entry.

**Syntax**

```
void _dos_setvect(unsigned interruptno, void interrupt (*isr) ());
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks** Every processor of the 8086 family includes a set of interrupt vectors, numbered 0 to 255. The 4-byte value in each vector is actually an address, which is the location of an interrupt function.

*_dos_setvect* sets the value of the interrupt vector named by *interruptno* to a new value, *isr*, which is a far pointer containing the address of a new interrupt function. The address of a C routine can be passed to *isr* only if that routine is declared to be an interrupt routine.

If you use the prototypes declared in dos.h, pass the address of an interrupt function to *_dos_setvect* in any memory model.

**Return value** None.

**See also** *_dos_getvect*

## dostounix dos.h

**Function** Converts date and time to UNIX time format.

**Syntax**

```
long dostounix(struct date *d, struct time *t);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *dostounix* converts a date and time as returned from *getdate* and *gettime* into UNIX time format. *d* points to a *date* structure, and *t* points to a *time* structure containing valid date and time information.

The date and time must not be earlier than or equal to Jan 1 1980 00:00:00.

**Return value** UNIX version of current date and time parameters: number of seconds since 00:00:00 on January 1, 1970 (GMT).

**See also** *getdate, gettime, unixtodos*

## _dos_write — dos.h

**Function** Writes to a file.

**Syntax**

```
unsigned _dos_write(int handle, const void _ _far *buf, unsigned len,
                    unsigned *nwritten);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | ■ |

**Remarks** *_dos_write* writes *len* bytes from the buffer pointed to by the **_ _far** pointer *buf* to the file associated with *handle*. *_dos_write* does not translate a linefeed character (LF) to a CR/LF pair because it treats all files as binary data.

The actual number of bytes written is stored at the location pointed to by *nwritten*. If the number of bytes actually written is less than that requested, the condition should be considered an error and probably indicates a full disk. For disk files, writing always proceeds from the current file pointer. On devices, bytes are directly sent to the device.

**Return value** On successful completion, *_dos_write* returns 0. Otherwise, it returns the operating system error code and the global variable *errno* is set to one of the following values:

| | |
|---|---|
| EACCES | Permission denied |
| EBADF | Bad file number |

**See also** *_dos_open, _dos_creat, _dos_read*

## dup — io.h

**Function** Duplicates a file handle.

**Syntax**

```
int dup(int handle);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |



**Remarks**

*dup* creates a new file handle that has the following in common with the original file handle:

- Same open file or device
- Same file pointer (that is, changing the file pointer of one changes the other)
- Same access mode (read, write, read/write)

*handle* is a file handle obtained from a *_rtl_creat*, *creat*, *_rtl_open*, *open*, *dup*, or *dup2* call.

**Return value**

Upon successful completion, *dup* returns the new file handle, a nonnegative integer; otherwise, *dup* returns –1.

In the event of error, the global variable *errno* is set to one of the following values:

| | |
|---|---|
| EBADF | Bad file number |
| EMFILE | Too many open files |

**See also**

*_rtl_close*, *close*, *_rtl_creat*, *creat*, *creatnew*, *creattemp*, *dup2*, *fopen*, *_rtl_open*, *open*

## dup2 — io.h

**Function**

Duplicates a file handle (*oldhandle*) onto an existing file handle (*newhandle*).

**Syntax**

```
int dup2(int oldhandle, int newhandle);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks**

*dup2* creates a new file handle that has the following in common with the original file handle:

- Same open file or device
- Same file pointer (that is, changing the file pointer of one changes the other)
- Same access mode (read, write, read/write)

*dup2* creates a new handle with the value of *newhandle*. If the file associated with *newhandle* is open when *dup2* is called, the file is closed.

*newhandle* and *oldhandle* are file handles obtained from a *creat*, *open*, *dup*, or *dup2* call.

**Return value**

*dup2* returns 0 on successful completion, –1 otherwise.

In the event of error, the global variable *errno* is set to one of the following values:

| | |
|---|---|
| EBADF | Bad file number |
| EMFILE | Too many open files |

**See also**

*_rtl_close*, *close*, *_rtl_creat*, *creat*, *creatnew*, *creattemp*, *dup*, *fopen*, *_rtl_open*, *open*

## ecvt — stdlib.h

**Function**

Converts a floating-point number to a string.

**Syntax**

```
char *ecvt(double value, int ndig, int *dec, int *sign);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks**

*ecvt* converts *value* to a null-terminated string of *ndig* digits, starting with the leftmost significant digit, and returns a pointer to the string. The position of the decimal point relative to the beginning of the string is stored indirectly through *dec* (a negative value for *dec* means that the decimal lies to the left of the returned digits). There is no decimal point in the string itself. If the sign of *value* is negative, the word pointed to by *sign* is nonzero; otherwise, it's 0. The low-order digit is rounded.

**Return value**

The return value of *ecvt* points to static data for the string of digits whose content is overwritten by each call to *ecvt* and *fcvt*.

**See also**

*fcvt*, *gcvt*, *sprintf*

## __emit__ — dos.h

**Function**

Inserts literal values directly into code.

**Syntax**

```
void __emit__(argument, ...);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Description**

*__emit__* is an inline function that lets you insert literal values directly into object code as it is compiling. It is used to generate machine language instructions without using inline assembly language or an assembler.

Generally the arguments of an *__emit__* call are single-byte machine instructions. However, because of the capabilities of this function, more complex instructions, complete with references to C variables, can be constructed.



You should use this function only if you are familiar with the machine language of the 80x86 processor family. You can use this function to place arbitrary bytes in the instruction code of a function; if any of these bytes is incorrect, the program misbehaves and can easily crash your machine. Borland C++ does not attempt to analyze your calls for correctness in any way. If you encode instructions that change machine registers or memory, Borland C++ will not be aware of it and might not properly preserve registers, as it would in many cases with inline assembly language (for example, it recognizes the usage of SI and DI registers in inline instructions). You are completely on your own with this function.

You must pass at least one argument to *__emit__*; any number can be given. The arguments to this function are not treated like any other function call arguments in the language. An argument passed to *__emit__* will not be converted in any way.

There are special restrictions on the form of the arguments to *__emit__*. Arguments must be in the form of expressions that can be used to initialize a static object. This means that integer and floating-point constants and the addresses of static objects can be used. The values of such expressions are written to the object code at the point of the call, exactly as if they were being used to initialize data. The address of a parameter or auto variable, plus or minus a constant offset, can also be used. For these arguments, the offset of the variable from BP is stored.

The number of bytes placed in the object code is determined from the type of the argument, except in the following cases:

- If a signed integer constant (that is 0x90) appears that fits within the range of 0 to 255, it is treated as if it were a character.
- If the address of an auto or parameter variable is used, a byte is written if the offset of the variable from BP is between –128 and 127; otherwise, a word is written.

Simple bytes are written as follows:

```
__emit__(0x90);
```

If you want a word written, but the value you are passing is under 255, simply cast it to **unsigned** using one of these methods:

```
__emit__(0xB8, (unsigned)17);
__emit__(0xB8, 17u);
```

Two- or four-byte address values can be forced by casting an address to **void near** * or **void far** *, respectively.

**Return value** None.

## enable, _enable

See *disable*.

## _endthread — process.h

**Function** Terminates execution of a thread.

**Syntax**

```
void _endthread(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| | | | ■ | | | ■ |

**Remarks** The *_endthread* function terminates the currently executing thread. The thread must have been started by an earlier call to *_beginthread*.

This function is available in the multithread libraries; it is not in the single-thread libraries.

**Return value** The function does not return a value.

**See also** *_beginthread*

## eof — io.h

**Function** Checks for end-of-file.

**Syntax**

```
int eof(int handle);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *eof* determines whether the file associated with *handle* has reached end-of-file.

**Return value** If the current position is end-of-file, *eof* returns the value 1; otherwise, it returns 0. A return value of –1 indicates an error; the global variable *errno* is set to

EBADF Bad file number

**See also** *clearerr, feof, ferror, perror*



## execl, execle, execlp, execlpe, execv, execve, execvp, execvpe process.h

**Function** Loads and runs other programs.

**Syntax**

```
int execl(char *path, char *arg0 *arg1, ..., *argn, NULL);
int execle(char *path, char *arg0, *arg1, ..., *argn, NULL, char **env);

int execlp(char *path, char *arg0,*arg1, ..., *argn, NULL);
int execlpe(char *path, char *arg0, *arg1, ..., *argn, NULL, char **env);

int execv(char *path, char *argv[]);
int execve(char *path, char *argv[], char **env);

int execvp(char *path, char *argv[]);
int execvpe(char *path, char *argv[], char **env);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | | ■ | | | ■ |

**Remarks** The functions in the *exec...* family load and run (execute) other programs, known as *child processes*. When an *exec...* call succeeds, the child process overlays the *parent process*. There must be sufficient memory available for loading and executing the child process.

*path* is the file name of the called child process. The *exec...* functions search for *path* using the standard search algorithm:

- If no explicit extension is given, the functions search for the file as given. If the file is not found, they add .EXE and search again. If not found, they add .COM and search again. If still not found, they add .BAT and search once more. The command processor COMSPEC is used to run the executable file.
- If an explicit extension or a period is given, the functions search for the file exactly as given.

The suffixes *l*, *v*, *p*, and *e* added to the *exec...* "family name" specify that the named function operates with certain capabilities.

- *l* specifies that the argument pointers (*arg0, arg1, ..., argn*) are passed as separate arguments. Typically, the *l* suffix is used when you know in advance the number of arguments to be passed.
- *v* specifies that the argument pointers (*argv[0] ..., arg[n]*) are passed as an array of pointers. Typically, the *v* suffix is used when a variable number of arguments is to be passed.
- *p* specifies that the function searches for the file in those directories specified by the PATH environment variable (without the *p* suffix, the function searches only the current working directory). If the *path* parameter does not contain an explicit directory, the function searches first the current directory, then the directories set with the PATH environment variable.
- *e* specifies that the argument *env* can be passed to the child process, letting you alter the environment for the child process. Without the *e* suffix, child processes inherit the environment of the parent process.

Each function in the *exec...* family *must* have one of the two argument-specifying suffixes (either *l* or *v*). The path search and environment inheritance suffixes (*p* and *e*) are optional; for example,

- *execl* is an *exec...* function that takes separate arguments, searches only the root or current directory for the child, and passes on the parent's environment to the child.
- *execvpe* is an *exec...* function that takes an array of argument pointers, incorporates PATH in its search for the child process, and accepts the *env* argument for altering the child's environment.

The *exec...* functions must pass at least one argument to the child process (*arg0* or *argv[0]*); this argument is, by convention, a copy of *path*. (Using a different value for this 0th argument won't produce an error.)

*path* is available for the child process.

When the *l* suffix is used, *arg0* usually points to *path*, and *arg1, ..., argn* point to character strings that form the new list of arguments. A mandatory null following *argn* marks the end of the list.

When the *e* suffix is used, you pass a list of new environment settings through the argument *env*. This environment argument is an array of character pointers. Each element points to a null-terminated character string of the form

*envvar = value*

where *envvar* is the name of an environment variable, and *value* is the string value to which *envvar* is set. The last element in *env* is null. When *env* is null, the child inherits the parents' environment settings.

The combined length of *arg0* + *arg1* + ... + *argn* (or of *argv[0]* + *argv[1]* + ... + *argn[n]*), including space characters that separate the arguments, must be less than 128 bytes for a 16-bit application, or 260 bytes for Win32 application. Null characters are not counted.

When an *exec...* function call is made, any open files remain open in the child process.

**Return value**

If successful, the *exec...* functions do not return. On error, the *exec...* functions return –1, and the global variable *errno* is set to one of the following values:

| | |
|---|---|
| EACCES | Permission denied |
| EMFILE | Too many open files |
| ENOENT | Path or file name not found |
| ENOEXEC | Exec format error |
| ENOMEM | Not enough memory |

**See also**

*abort, atexit, _exit, exit, _fpreset, searchpath, spawn..., system*

## _exit — stdlib.h

**Function**

Terminates program.

**Syntax**

```
void _exit(int status);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks**

*_exit* terminates execution without closing any files, flushing any output, or calling any exit functions.

The calling process uses *status* as the exit status of the process. Typically a value of 0 is used to indicate a normal exit, and a nonzero value indicates some error.

**Return value**

None.

**See also**

*abort, atexit, exec..., exit, spawn...*

## exit — stdlib.h

**Function**

Terminates program.

**Syntax**

```
void exit(int status);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks**

*exit* terminates the calling process. Before termination, all files are closed, buffered output (waiting to be output) is written, and any registered "exit functions" (posted with *atexit*) are called.

*status* is provided for the calling process as the exit status of the process. Typically a value of 0 is used to indicate a normal exit, and a nonzero value indicates some error. It can be, but is not required, to be set with one of the following:

| | |
|---|---|
| EXIT_FAILURE | Abnormal program termination; signal to operating system that program has terminated with an error |
| EXIT_SUCCESS | Normal program termination |

**Return value**

None.

**See also**

*abort, atexit, exec..., _exit, keep, signal, spawn...*

## exp, expl — math.h

**Function**

Calculates the exponential $e$ to the $x$.

**Syntax**

```
double exp(double x);
long double expl(long double x);
```

| | DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|---|
| *exp* | ■ | ■ | ■ | ■ | ■ | ■ | ■ |
| *expl* | ■ | | ■ | ■ | | | ■ |

**Remarks**

*exp* calculates the exponential function $e^x$.

*expl* is the **long double** version; it takes a **long double** argument and returns a **long double** result.

This function can be used with *bcd* and *complex* types.

**Return value**

*exp* returns $e^x$.

Sometimes the arguments passed to these functions produce results that overflow or are incalculable. When the correct value overflows, *exp* returns the value HUGE_VAL and *expl* returns _LHUGE_VAL. Results of excessively large magnitude cause the global variable *errno* to be set to

ERANGE Result out of range

On underflow, these functions return 0.0, and the global variable *errno* is not changed. Error handling for these functions can be modified through the functions *_matherr* and *_matherrl*.

**See also** *frexp, ldexp, log, log10, _matherr, pow, pow10, sqrt*

## _expand — malloc.h

**Function** Grows or shrinks a heap block in place.

**Syntax**

```
void *_expand(void *block, size_t size);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| | | | ■ | | | ■ |

**Remarks** This function attempts to change the size of an allocated memory *block* without moving the block's location in the heap. The data in the *block* are not changed, up to the smaller of the old and new sizes of the block. The block must have been allocated earlier with *malloc*, *calloc*, or *realloc*, and must *not* have been freed.

**Return value** If *_expand* is able to resize the block without moving it, *_expand* returns a pointer to the block, whose address is unchanged. If *_expand* is unsuccessful, it returns a NULL pointer and does not modify or resize the block.

**See also** *calloc, malloc, realloc*

## fabs, fabsl — math.h

**Function** Returns the absolute value of a floating-point number.

**Syntax**

```
double fabs(double x);
long double fabsl(long double x);
```

| | DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|---|
| *fabs* | ■ | ■ | ■ | ■ | ■ | ■ | ■ |
| *fabsl* | ■ | | ■ | ■ | | | ■ |

**Remarks** *fabs* calculates the absolute value of *x*, a double. *fabsl* is the **long double** version; it takes a **long double** argument and returns a **long double** result.

**Return value** *fabs* and *fabsl* return the absolute value of *x*.

**See also** *abs, cabs, labs*

## farcalloc — alloc.h

**Function** Allocates memory from the far heap.

**Syntax**

```
void far *farcalloc(unsigned long nunits, unsigned long unitsz);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks** *farcalloc* allocates memory from the far heap for an array containing *nunits* elements, each *unitsz* bytes long.

For allocating from the far heap, note that

- All available RAM can be allocated.
- Blocks larger than 64K can be allocated.
- Far pointers (or huge pointers if blocks are larger than 64K) are used to access the allocated blocks.

In the compact and large memory models, *farcalloc* is similar, though not identical, to *calloc*. It takes **unsigned long** parameters, while *calloc* takes **unsigned** parameters.

**Return value** *farcalloc* returns a pointer to the newly allocated block, or NULL if not enough space exists for the new block.

**See also** *calloc, farfree, farmalloc, malloc*

## farfree — alloc.h

**Function** Frees a block from far heap.

**Syntax**

```
void farfree(void far * block);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks** *farfree* releases a block of memory previously allocated from the far heap.

In the small and medium memory models, blocks allocated by *farmalloc* cannot be freed with normal *free*, and blocks allocated with *malloc* cannot be freed with *farfree*. In these models, the two heaps are completely distinct.

**Return value** None.

**See also** *farcalloc, farmalloc*

## farmalloc — alloc.h

**Function** Allocates from far heap.

**Syntax**

```
void far *farmalloc(unsigned long nbytes);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks** *farmalloc* allocates a block of memory *nbytes* bytes long from the far heap.

For allocating from the far heap, note that

- All available RAM can be allocated.
- Blocks larger than 64K can be allocated.
- Far pointers are used to access the allocated blocks.

In the compact and large memory models, *farmalloc* is similar though not identical to *malloc*. It takes **unsigned long** parameters, while *malloc* takes **unsigned** parameters.

**Return value** *farmalloc* returns a pointer to the newly allocated block, or NULL if not enough space exists for the new block.

**See also** *farcalloc, farfree, farrealloc, malloc*

## farrealloc — alloc.h

**Function** Adjusts allocated block in far heap.

**Syntax**

```
void far *farrealloc(void far *oldblock, unsigned long nbytes);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks** *farrealloc* adjusts the size of the allocated block to *nbytes*, copying the contents to a new location, if necessary.

For allocating from the far heap, note that

- All available RAM can be allocated.
- Blocks larger than 64K can be allocated.
- Far pointers are used to access the allocated blocks.

**Return value** *farrealloc* returns the address of the reallocated block, which might be different than the address of the original block. If the block cannot be reallocated, *farrealloc* returns NULL.

**See also** *farmalloc, realloc*

## fclose — stdio.h

**Function** Closes a stream.

**Syntax**

```
int fclose(FILE *stream);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *fclose* closes the named stream. All buffers associated with the stream are flushed before closing. System-allocated buffers are freed upon closing. Buffers assigned with *setbuf* or *setvbuf* are not automatically freed. (But if *setvbuf* is passed null for the buffer pointer, it *will* free it upon close.)

**Return value** *fclose* returns 0 on success. It returns EOF if any errors were detected.

**See also** *close, fcloseall, fdopen, fflush, flushall, fopen, freopen*

## fcloseall — stdio.h

**Function** Closes open streams.

**Syntax**

```
int fcloseall(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks** *fcloseall* closes all open streams except stdin, stdout, stdprn, stderr, and stdaux. stdprn and stdaux streams are not available on OS/2 and Win32.

**Return value** *fcloseall* returns the total number of streams it closed. It returns EOF if any errors were detected.

**See also** *fclose, fdopen, flushall, fopen, freopen*

## fcvt — stdlib.h



**Function** Converts a floating-point number to a string.

**Syntax**

```
char *fcvt(double value, int ndig, int *dec, int *sign);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks** *fcvt* converts *value* to a null-terminated string digit, starting with the leftmost significant digit, with *ndig* digits to the right of the decimal point. *fcvt* then returns a pointer to the string. The position of the decimal point relative to the beginning of the string is stored indirectly through *dec* (a negative value for *dec* means to the left of the returned digits). There is no decimal point in the string itself. If the sign of *value* is negative, the word pointed to by *sign* is nonzero; otherwise, it is 0.

The correct digit has been rounded for the number of digits to the right of the decimal point specified by *ndig*.

**Return value** The return value of *fcvt* points to static data whose content is overwritten by each call to *fcvt* and *ecvt*.

**See also** *ecvt, gcvt, sprintf*

## fdopen — stdio.h

**Function** Associates a stream with a file handle.

**Syntax**

```
FILE *fdopen(int handle, char *type);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks** *fdopen* associates a stream with a file handle obtained from *creat, dup, dup2,* or *open*. The type of stream must match the mode of the open *handle*.

The *type* string used in a call to *fdopen* is one of the following values:

| Value | Description |
|---|---|
| *r* | Open for reading only. |
| *w* | Create for writing. |
| *a* | Append; open for writing at end-of-file, or create for writing if the file does not exist. |
| *r+* | Open an existing file for update (reading and writing). |
| *w+* | Create a new file for update. |
| *a+* | Open for append; open (or create if the file does not exist) for update at the end of the file. |

To specify that a given file is being opened or created in text mode, append a *t* to the value of the *type* string (*rt*, *w+t*, and so on); similarly, to specify binary mode, append a *b* to the *type* string (*wb*, *a+b*, and so on).

If a *t* or *b* is not given in the *type* string, the mode is governed by the global variable *_fmode*. If *_fmode* is set to O_BINARY, files will be opened in binary mode. If *_fmode* is set to O_TEXT, they will be opened in text mode. These O_... constants are defined in fcntl.h.

When a file is opened for update, both input and output can be done on the resulting stream. However, output cannot be directly followed by input without an intervening *fseek* or *rewind*, and input cannot be directly followed by output without an intervening *fseek*, *rewind*, or an input that encounters end-of-file.

**Return value**

On successful completion, *fdopen* returns a pointer to the newly opened stream. In the event of error, it returns NULL.

**See also**

*fclose*, *fopen*, *freopen*, *open*

## feof — stdio.h

**Function**

Detects end-of-file on a stream.

**Syntax**

```
int feof(FILE *stream);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks**

*feof* is a macro that tests the given stream for an end-of-file indicator. Once the indicator is set, read operations on the file return the indicator until *rewind* is called, or the file is closed.

The end-of-file indicator is reset with each input operation.

**Return value** *feof* returns nonzero if an end-of-file indicator was detected on the last input operation on the named stream, and 0 if end-of-file has not been reached.

**See also** *clearerr, eof, ferror, perror*

## ferror — stdio.h



**Function** Detects errors on stream.

**Syntax**

```
int ferror(FILE *stream);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *ferror* is a macro that tests the given stream for a read or write error. If the stream's error indicator has been set, it remains set until *clearerr* or *rewind* is called, or until the stream is closed.

**Return value** *ferror* returns nonzero if an error was detected on the named stream.

**See also** *clearerr, eof, feof, fopen, gets, perror*

## fflush — stdio.h

**Function** Flushes a stream.

**Syntax**

```
int fflush(FILE *stream);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** If the given stream has buffered output, *fflush* writes the output for *stream* to the associated file.

The stream remains open after *fflush* has executed. *fflush* has no effect on an unbuffered stream.

**Return value** *fflush* returns 0 on success. It returns EOF if any errors were detected.

**See also** *fclose, flushall, setbuf, setvbuf*

## fgetc — stdio.h

**Function** Gets character from stream.

**Syntax**

```
int fgetc(FILE *stream);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *fgetc* returns the next character on the named input stream.

**Return value** On success, *fgetc* returns the character read, after converting it to an **int** without sign extension. On end-of-file or error, it returns EOF.

**See also** *fgetchar, fputc, getc, getch, getchar, getche, ungetc, ungetch*

## fgetchar — stdio.h

**Function** Gets character from stdin.

**Syntax**

```
int fgetchar(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks** *fgetchar* returns the next character from stdin. It is defined as *fgetc(stdin)*.

For Win32s or Win32 GUI applications, stdin must be redirected.

**Return value** On success, *fgetchar* returns the character read, after converting it to an **int** without sign extension. On end-of-file or error, it returns EOF.

**See also** *fgetc, fputchar, freopen, getchar*

## fgetpos — stdio.h

**Function** Gets the current file pointer.

**Syntax**

```
int fgetpos(FILE *stream, fpos_t *pos);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | ■ | ■ | ■ |

**Remarks** *fgetpos* stores the position of the file pointer associated with the given stream in the location pointed to by *pos*. The exact value is unimportant; its value is opaque except as a parameter to subsequent *fsetpos* calls.

**Return value** On success, *fgetpos* returns 0. On failure, it returns a nonzero value and sets the global variable *errno* to

| | |
|---|---|
| EBADF | Bad file number |
| EINVAL | Invalid number |

**See also** *fseek, fsetpos, ftell, tell*



## fgets — stdio.h

**Function** Gets a string from a stream.

**Syntax**

```
char *fgets(char *s, int n, FILE *stream);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *fgets* reads characters from *stream* into the string *s*. The function stops reading when it reads either *n* – 1 characters or a newline character, whichever comes first. *fgets* retains the newline character at the end of *s*. A null byte is appended to *s* to mark the end of the string.

**Return value** On success, *fgets* returns the string pointed to by *s*; it returns NULL on end-of-file or error.

**See also** *cgets, fputs, gets*

## filelength — io.h

**Function** Gets file size in bytes.

**Syntax**

```
long filelength(int handle);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *filelength* returns the length (in bytes) of the file associated with *handle*.

**Return value** On success, *filelength* returns a **long** value, the file length in bytes. On error, it returns –1 and the global variable *errno* is set to

EBADF Bad file number

**See also** *fopen, lseek, open*

## fileno — stdio.h

**Function** Gets file handle.

**Syntax**

```
int fileno(FILE *stream);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks** *fileno* is a macro that returns the file handle for the given stream. If *stream* has more than one handle, *fileno* returns the handle assigned to the stream when it was first opened.

**Return value** *fileno* returns the integer file handle associated with *stream*.

**See also** *fdopen, fopen, freopen*

## findfirst — dir.h

**Function** Searches a disk directory.

**Syntax**

```
int findfirst(const char *pathname, struct ffblk *ffblk, int attrib);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *findfirst* begins a search of a disk directory for files specified by attributes or wildcards.

*pathname* is a string with an optional drive specifier, path, and file name of the file to be found. Only the file name portion can contain wildcard match characters (such as ? or *). If a matching file is found, the *ffblk* structure is filled with the file-directory information.

The format of the structure *ffblk* is as follows:

```
struct ffblk {
   char ff_reserved[21];     /* reserved by DOS */
   char ff_attrib;           /* attribute found */
   int ff_ftime;             /* file time */
   int ff_fdate;             /* file date */
   long ff_fsize;            /* file size */
   char ff_name[13];         /* found file name */
   };
```

```
struct ffblk {
   long           ff_reserved;
   long           ff_fsize;       /* file size */
   unsigned long  ff_attrib;      /* attribute found */
   unsigned short ff_ftime;       /* file time */
   unsigned short ff_fdate;       /* file date */
   char           ff_name[256];   /* found file name */
   };
```

*attrib* is a file-attribute byte used in selecting eligible files for the search. *attrib* should be selected from the following constants defined in dos.h:

| | |
|---|---|
| FA_RDONLY | Read-only attribute |
| FA_HIDDEN | Hidden file |
| FA_SYSTEM | System file |
| FA_LABEL | Volume label |
| FA_DIREC | Directory |
| FA_ARCH | Archive |

A combination of constants can be ORed together.

For more detailed information about these attributes, refer to your operating system reference manuals.

Note that *ff_ftime* and *ff_fdate* contain bit fields for referring to the current date and time. The structure of these fields was established by the operating system. Both are 16-bit structures divided into three fields.

***ff_ftime*:**

| | |
|---|---|
| Bits 0 to 4 | The result of seconds divided by 2 (for example, 10 here means 20 seconds) |
| Bits 5 to 10 | Minutes |
| Bits 11 to 15 | Hours |

***ff_fdate*:**

| | |
|---|---|
| Bits 0-4 | Day |
| Bits 5-8 | Month |
| Bits 9-15 | Years since 1980 (for example, 9 here means 1989) |

The structure *ftime* declared in io.h uses time and date bit fields similar in structure to *ff_ftime*, and *ff_fdate*.

**Return value**

*findfirst* returns 0 on successfully finding a file matching the search *pathname*. When no more files can be found, or if there is some error in the file name, –1 is returned, and the global variable *errno* is set to

ENOENT Path or file name not found

and *_doserrno* is set to one of the following values:

ENMFILE No more files
ENOENT Path or file name not found

**See also**

*findnext, getftime, setftime*

## findnext dir.h

**Function**

Continues *findfirst* search.

**Syntax**

```
int findnext(struct ffblk *ffblk);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks**

*findnext* is used to fetch subsequent files that match the *pathname* given in *findfirst*. *ffblk* is the same block filled in by the *findfirst* call. This block contains necessary information for continuing the search. One file name for each call to *findnext* will be returned until no more files are found in the directory matching the *pathname*.

**Return value**

*findnext* returns 0 on successfully finding a file matching the search *pathname*. When no more files can be found, or if there is some error in the file name, –1 is returned, and the global variable *errno* is set to

ENOENT Path or file name not found

and *_doserrno* is set to one of the following values:

ENMFILE No more files
ENOENT Path or file name not found

**See also**

*findfirst*

## floor, floorl — math.h

**Function** Rounds down.

**Syntax**

```
double floor(double x);
long double floorl(long double x);
```

| | DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|---|
| *floor* | ■ | ■ | ■ | ■ | ■ | ■ | ■ |
| *floorl* | ■ | | ■ | ■ | | | ■ |

**Remarks** *floor* finds the largest integer not greater than *x*. *floorl* is the **long double** version; it takes a **long double** argument and returns a **long double** result.

**Return value** *floor* returns the integer found as a **double**. *floorl* returns the integer found as a **long double**.

**See also** *ceil, fmod*

## flushall — stdio.h

**Function** Flushes all streams.

**Syntax**

```
int flushall(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks** *flushall* clears all buffers associated with open input streams, and writes all buffers associated with open output streams to their respective files. Any read operation following *flushall* reads new data into the buffers from the input files. Streams stay open after *flushall* executes.

**Return value** *flushall* returns an integer, the number of open input and output streams.

**See also** *fclose, fcloseall, fflush*

## _fmemccpy

*See memccpy.*

## _fmemchr

*See memchr.*

## _fmemcmp

*See memcmp.*

## _fmemcpy

*See memcpy.*

## _fmemicmp

*See memicmp.*

## _fmemmove

*See memmove.*

## _fmemset

*See memset.*

## fmod, fmodl — math.h

**Function**

Calculates $x$ modulo $y$, the remainder of $x/y$.

**Syntax**

```
double fmod(double x, double y);
long double fmodl(long double x, long double y);
```

| | DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|---|
| *fmod* | ■ | ■ | ■ | ■ | ■ | ■ | ■ |
| *fmodl* | ■ | | ■ | ■ | | | ■ |

**Remarks** *fmod* calculates $x$ modulo $y$ (the remainder $f$, where $x = ay + f$ for some integer $a$ and $0 \leq f < y$). *fmodl* is the **long double** version; it takes **long double** arguments and returns a **long double** result.

**Return value** *fmod* and *fmodl* return the remainder $f$, where $x = ay + f$ (as described). Where y = 0, *fmod* and *fmodl* return 0.

**See also** *ceil, floor, modf*



## _fmovmem

*See movmem.*

## fnmerge dir.h

**Function** Builds a path from component parts.

**Syntax**

```
void fnmerge(char *path, const char *drive, const char *dir, const char *name,
             const char *ext);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *fnmerge* makes a path name from its components. The new path name is

```
X:\DIR\SUBDIR\NAME.EXT
```

where

*drive* = X:
*dir* = \DIR\SUBDIR\
*name* = NAME
*ext* = .EXT

*fnmerge* assumes there is enough space in *path* for the constructed path name. The maximum constructed length is MAXPATH. MAXPATH is defined in dir.h.

*fnmerge* and *fnsplit* are invertible; if you split a given *path* with *fnsplit*, then merge the resultant components with *fnmerge*, you end up with *path*.

**Return value** None.

**See also** *fnsplit*

# fnsplit dir.h

**Function**

Splits a full path name into its components.

**Syntax**

```
int fnsplit(const char *path, char *drive, char *dir, char *name, char *ext);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks**

*fnsplit* takes a file's full path name (*path*) as a string in the form

*X:\DIR\SUBDIR\NAME.EXT*

and splits *path* into its four components. It then stores those components in the strings pointed to by *drive*, *dir*, *name*, and *ext*. (All five components must be passed, but any of them can be a null, which means the corresponding component will be parsed but not stored.)

The maximum sizes for these strings are given by the constants MAXDRIVE, MAXDIR, MAXPATH, MAXFILE, and MAXEXT (defined in dir.h), and each size includes space for the null character.

| Constant | Max 16-bit | Max 32-bit | String |
|---|---|---|---|
| MAXPATH | 80 | 260 | *path* |
| MAXDRIVE | 3 | 3 | *drive*; includes colon (:) |
| MAXDIR | 66 | 260 | *dir*; includes leading and trailing backslashes (\) |
| MAXFILE | 9 | 260 | *name* |
| MAXEXT | 5 | 260 | *ext*; includes leading dot (.) |

*fnsplit* assumes that there is enough space to store each non-null component.

When *fnsplit* splits *path*, it treats the punctuation as follows:

- *drive* includes the colon (C:, A:, and so on).
- *dir* includes the leading and trailing backslashes (\BC\include\, \source\, and so on).
- *name* includes the file name.
- *ext* includes the dot preceding the extension (.C, .EXE, and so on).

*fnmerge* and *fnsplit* are invertible; if you split a given *path* with *fnsplit*, then merge the resultant components with *fnmerge*, you end up with *path*.

**Return value**

*fnsplit* returns an integer (composed of five flags, defined in dir.h) indicating which of the full path name components were present in *path*. These flags and the components they represent are

| | |
|---|---|
| EXTENSION | An extension |
| FILENAME | A file name |
| DIRECTORY | A directory (and possibly subdirectories) |
| DRIVE | A drive specification (see dir.h) |
| WILDCARDS | Wildcards (* or ?) |

**See also**

*fnmerge*



## fopen — stdio.h

**Function**

Opens a stream.

**Syntax**

```
FILE *fopen(const char *filename, const char *mode);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks**

*fopen* opens the file named by *filename* and associates a stream with it. *fopen* returns a pointer to be used to identify the stream in subsequent operations.

The *mode* string used in calls to *fopen* is one of the following values:

| Value | Description |
|---|---|
| *r* | Open for reading only. |
| *w* | Create for writing. If a file by that name already exists, it will be overwritten. |
| *a* | Append; open for writing at end of file, or create for writing if the file does not exist. |
| *r+* | Open an existing file for update (reading and writing). |
| *w+* | Create a new file for update (reading and writing). If a file by that name already exists, it will be overwritten. |
| *a+* | Open for append; open for update at the end of the file, or create if the file does not exist. |

To specify that a given file is being opened or created in text mode, append a *t* to the *mode* string (*rt*, *w+t*, and so on). Similarly, to specify binary mode, append a *b* to the *mode* string (*wb*, *a+b*, and so on). *fopen* also allows the *t* or

*b* to be inserted between the letter and the + character in the mode string; for example, *rt+* is equivalent to *r+t*.

If a *t* or *b* is not given in the *mode* string, the mode is governed by the global variable *_fmode*. If *_fmode* is set to O_BINARY, files are opened in binary mode. If *_fmode* is set to O_TEXT, they are opened in text mode. These O_... constants are defined in fcntl.h.

When a file is opened for update, both input and output can be done on the resulting stream. However, output cannot be followed directly by input without an intervening *fseek* or *rewind*, and input cannot be directly followed by output without an intervening *fseek*, *rewind*, or an input that encounters end-of-file.

**Return value** On successful completion, *fopen* returns a pointer to the newly opened stream. In the event of error, it returns NULL.

**See also** *creat, dup, fclose, fdopen ferror, _fmode* (global variable), *fread, freopen, fseek, fwrite, open, rewind, setbuf, setmode*

## FP_OFF, FP_SEG — dos.h

**Function** Gets a far address offset or segment.

**Syntax**

```
unsigned FP_OFF(void far *p);
unsigned FP_SEG(void far *p);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks** The *FP_OFF* macro can be used to get or set the offset of the **far** pointer *p*.

*FP_SEG* is a macro that gets or sets the segment value of the **far** pointer *p*.

**Return value** *FP_OFF* returns an **unsigned** integer value representing an offset value.

*FP_SEG* returns an **unsigned** integer representing a segment value.

**See also** *MK_FP, movedata, segread*

## _fpreset — float.h

**Function** Reinitializes floating-point math package.

**Syntax**

```
void _fpreset(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks**

*_fpreset* reinitializes the floating-point math package. This function is usually used in conjunction with *system* or the *exec...* or *spawn...* functions. It is also used to recover from floating-point errors before calling *longjmp*.

If an 80x87 coprocessor is used in a program, a child process (executed by *system* or by an *exec...* or *spawn...* function) might alter the parent process' floating-point state.

If you use an 80x87, take the following precautions:

- Do not call *system* or an *exec...* or *spawn...* function while a floating-point expression is being evaluated.
- Call *_fpreset* to reset the floating-point state after using *system*, *exec...*, or *spawn...* if there is *any* chance that the child process performed a floating-point operation with the 80x87.

**Return value**

None.

**See also**

*_clear87, _control87, _status87*

## fprintf — stdio.h

**Function**

Writes formatted output to a stream.

**Syntax**

```
int fprintf(FILE *stream, const char *format[, argument, ...]);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks**

See *printf* for details on format specifiers.

*fprintf* accepts a series of arguments, applies to each a format specifier contained in the format string pointed to by *format*, and outputs the formatted data to a stream. There must be the same number of format specifiers as arguments.

**Return value**

*fprintf* returns the number of bytes output. In the event of error, it returns EOF.

**See also**

*cprintf, fscanf, printf, putc, sprintf*

## fputc — stdio.h

**Function** Puts a character on a stream.

**Syntax**

```
int fputc(int c, FILE *stream);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks** *fputc* outputs character *c* to the named stream.

For Win32s or Win32 GUI applications, stdout must be redirected.

**Return value** On success, *fputc* returns the character *c*. On error, it returns EOF.

**See also** *fgetc, putc*

## fputchar — stdio.h

**Function** Outputs a character on stdout.

**Syntax**

```
int fputchar(int c);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | | ■ | ■ | | ■ |

**Remarks** *fputchar* outputs character *c* to stdout. *fputchar*(*c*) is the same as *fputc*(*c, stdout*).

For Win32s or Win32 GUI applications, stdout must be redirected.

**Return value** On success, *fputchar* returns the character *c*. On error, it returns EOF.

**See also** *fgetchar, freopen, putchar*

## fputs — stdio.h

**Function** Outputs a string on a stream.

**Syntax**

```
int fputs(const char *s, FILE *stream);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *fputs* copies the null-terminated string *s* to the given output stream; it does not append a newline character, and the terminating null character is not copied.

**Return value** On successful completion, *fputs* returns a non-negative value. Otherwise, it returns a value of EOF.

**See also** *fgets, gets, puts*



## fread — stdio.h

**Function** Reads data from a stream.

**Syntax**

```
size_t fread(void *ptr, size_t size, size_t n, FILE *stream);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *fread* reads *n* items of data, each of length *size* bytes, from the given input stream into a block pointed to by *ptr*.

The total number of bytes read is ($n \times size$).

**Return value** On successful completion, *fread* returns the number of items (not bytes) actually read. It returns a short count (possibly 0) on end-of-file or error.

**See also** *fopen, fwrite, printf, read*

## free — stdlib.h

**Function** Frees allocated block.

**Syntax**

```
void free(void *block);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *free* deallocates a memory block allocated by a previous call to *calloc, malloc,* or *realloc.*

**Return value** None.

**See also** *calloc, malloc, realloc, strdup*

# freopen stdio.h

**Function**

Associates a new file with an open stream.

**Syntax**

```
FILE *freopen(const char *filename, const char *mode, FILE *stream);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks**

*freopen* substitutes the named file in place of the open stream. It closes *stream*, regardless of whether the open succeeds. *freopen* is useful for changing the file attached to stdin, stdout, or stderr.

The *mode* string used in calls to *fopen* is one of the following values:

| Value | Description |
|---|---|
| *r* | Open for reading only. |
| *w* | Create for writing. |
| *a* | Append; open for writing at end-of-file, or create for writing if the file does not exist. |
| *r+* | Open an existing file for update (reading and writing). |
| *w+* | Create a new file for update. |
| *a+* | Open for append; open (or create if the file does not exist) for update at the end of the file. |

To specify that a given file is being opened or created in text mode, append a *t* to the *mode* string (*rt*, *w+t*, and so on); similarly, to specify binary mode, append a *b* to the *mode* string (*wb*, *a+b*, and so on).

If a *t* or *b* is not given in the *mode* string, the mode is governed by the global variable *_fmode*. If *_fmode* is set to O_BINARY, files are opened in binary mode. If *_fmode* is set to O_TEXT, they are opened in text mode. These O_... constants are defined in fcntl.h.

When a file is opened for update, both input and output can be done on the resulting stream. However, output cannot be directly followed by input without an intervening *fseek* or *rewind*, and input cannot be directly followed by output without an intervening *fseek*, *rewind*, or an input that encounters end-of-file.

**Return value**

On successful completion, *freopen* returns the argument *stream*. In the event of error, it returns NULL.

**See also**

*fclose*, *fdopen*, *fopen*, *open*, *setmode*

## frexp, frexpl — math.h

**Function** Splits a number into mantissa and exponent.

**Syntax**

```
double frexp(double x, int *exponent);
long double frexpl(long double x, int *exponent);
```

| | DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|---|
| *frexp* | ■ | ■ | ■ | ■ | ■ | ■ | ■ |
| *frexpl* | ■ | | ■ | ■ | | | ■ |

**Remarks** *frexp* calculates the mantissa *m* (a **double** greater than or equal to 0.5 and less than 1) and the integer value *n*, such that *x* (the original **double** value) equals $m \times 2^n$. *frexp* stores *n* in the integer that *exponent* points to.

*frexpl* is the **long double** version; it takes a **long double** argument for *x* and returns a **long double** result.

**Return value** *frexp* and *frexpl* return the mantissa *m*. Error handling for these routines can be modified through the functions *_matherr* and *_matherrl*.

**See also** *exp*, *ldexp*, *_matherr*,

## fscanf — stdio.h

**Function** Scans and formats input from a stream.

**Syntax**

```
int fscanf(FILE *stream, const char *format[, address, ...]);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *fscanf* scans a series of input fields, one character at a time, reading from a stream. Then each field is formatted according to a format specifier passed to *fscanf* in the format string pointed to by *format*. Finally, *fscanf* stores the formatted input at an address passed to it as an argument following *format*. The number of format specifiers and addresses must be the same as the number of input fields.

See *scanf* for details on format specifiers.

*fscanf* can stop scanning a particular field before it reaches the normal end-of-field character (whitespace), or it can terminate entirely for a number of reasons. See *scanf* for a discussion of possible causes.

**Return value** *fscanf* returns the number of input fields successfully scanned, converted, and stored; the return value does not include scanned fields that were not stored.

If *fscanf* attempts to read at end-of-file, the return value is EOF. If no fields were stored, the return value is 0.

**See also** *atof, cscanf, fprintf, printf, scanf, sscanf, vfscanf, vscanf, vsscanf*

## fseek — stdio.h

**Function** Repositions a file pointer on a stream.

**Syntax**

```
int fseek(FILE *stream, long offset, int whence);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *fseek* sets the file pointer associated with *stream* to a new position that is *offset* bytes from the file location given by *whence*. For text mode streams, *offset* should be 0 or a value returned by *ftell*.

*whence* must be one of the values 0, 1, or 2, which represent three symbolic constants (defined in stdio.h) as follows:

| Constant | *whence* | File location |
|---|---|---|
| SEEK_SET | 0 | File beginning |
| SEEK_CUR | 1 | Current file pointer position |
| SEEK_END | 2 | End-of-file |

*fseek* discards any character pushed back using *ungetc*. *fseek* is used with stream I/O; for file handle I/O, use *lseek*.

After *fseek*, the next operation on an update file can be either input or output.

**Return value** *fseek* returns 0 if the pointer is successfully moved and nonzero on failure.

*fseek* might return a 0, indicating that the pointer has been moved successfully, when in fact it has not been. This is because DOS, which actually resets the pointer, does not verify the setting. *fseek* returns an error code only on an unopened file or device.

In the event of an error return, the global variable *errno* is set to one of the following values:

| | |
|---|---|
| EBADF | Bad file pointer |
| EINVAL | Invalid argument |
| ESPIPE | Illegal seek on device |

**See also** *fgetpos, fopen, fsetpos, ftell, lseek, rewind, setbuf, tell*

## fsetpos — stdio.h

**Function** Positions the file pointer of a stream.

**Syntax**

```
int fsetpos(FILE *stream, const fpos_t *pos);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | ■ | ■ | ■ |

**Remarks** *fsetpos* sets the file pointer associated with *stream* to a new position. The new position is the value obtained by a previous call to *fgetpos* on that stream. It also clears the end-of-file indicator on the file that *stream* points to and undoes any effects of *ungetc* on that file. After a call to *fsetpos*, the next operation on the file can be input or output.

**Return value** On success, *fsetpos* returns 0. On failure, it returns a nonzero value and also sets the global variable *errno* to a nonzero value.

**See also** *fgetpos, fseek, ftell*

## _fsopen — stdio.h, share.h

**Function** Opens a stream with file sharing.

**Syntax**

```
FILE *_fsopen(const char *filename, const char *mode, int shflag);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *_fsopen* opens the file named by *filename* and associates a stream with it. *_fsopen* returns a pointer that is used to identify the stream in subsequent operations.

The *mode* string used in calls to *_fsopen* is one of the following values:

| Mode | Description |
|---|---|
| *r* | Open for reading only. |
| *w* | Create for writing. If a file by that name already exists, it will be overwritten. |
| *a* | Append; open for writing at end of file, or create for writing if the file does not exist. |

| | |
|---|---|
| *r+* | Open an existing file for update (reading and writing). |
| *w+* | Create a new file for update (reading and writing). If a file by that name already exists, it will be overwritten. |
| *a+* | Open for append; open for update at the end of the file, or create if the file does not exist. |

To specify that a given file is being opened or created in text mode, append a *t* to the *mode* string (*rt*, *w+t*, and so on). Similarly, to specify binary mode, append a *b* to the *mode* string (*wb*, *a+b*, and so on). *_fsopen* also allows the *t* or *b* to be inserted between the letter and the + character in the mode string; for example, *rt+* is equivalent to *r+t*.

If a *t* or *b* is not given in the *mode* string, the mode is governed by the global variable *_fmode*. If *_fmode* is set to O_BINARY, files are opened in binary mode. If *_fmode* is set to O_TEXT, they are opened in text mode. These O_... constants are defined in fcntl.h.

When a file is opened for update, both input and output can be done on the resulting stream. However, output cannot be followed directly by input without an intervening *fseek* or *rewind*, and input cannot be directly followed by output without an intervening *fseek*, *rewind*, or an input that encounters end-of-file.

*shflag* specifies the type of file-sharing allowed on the file *filename*. Symbolic constants for *shflag* are defined in share.h.

| Value of *shflag* | Description |
|---|---|
| SH_COMPAT | Sets compatibility mode |
| SH_DENYRW | Denies read/write access |
| SH_DENYWR | Denies write access |
| SH_DENYRD | Denies read access |
| SH_DENYNONE | Permits read/write access |
| SH_DENYNO | Permits read/write access |

**Return value**

On successful completion, *_fsopen* returns a pointer to the newly opened stream. In the event of error, it returns NULL.

**See also**

*creat*, *_dos_open*, *dup*, *fclose*, *fdopen*, *ferror*, *_fmode* (global variable), *fopen*, *fread*, *freopen*, *fseek*, *fwrite*, *open*, *rewind*, *setbuf*, *setmode*, *sopen*

## fstat, stat — sys\stat.h

**Function**

Gets open file information.

**Syntax**

```
int fstat(int handle, struct stat *statbuf);
int stat(char *path, struct stat *statbuf);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks**

*fstat* stores information in the *stat* structure about the file or directory associated with *handle*.

*stat* stores information about a given file or directory in the *stat* structure. The name of the file is *path*.

*statbuf* points to the *stat* structure (defined in sys\stat.h). That structure contains the following fields:

| | |
|---|---|
| *st_mode* | Bit mask giving information about the file's mode |
| *st_dev* | Drive number of disk containing the file, or file handle if the file is on a device |
| *st_rdev* | Same as *st_dev* |
| *st_nlink* | Set to the integer constant 1 |
| *st_size* | Size of the file in bytes |
| *st_atime* | Most recent access |
| *st_mtime* | Same as *st_atime* |
| *st_ctime* | Same as *st_atime* |

The *stat* structure contains three more fields not mentioned here. They contain values that are meaningful only in UNIX.

The *st_mode* bit mask that gives information about the mode of the open file includes the following bits:

One of the following bits will be set:

S_IFCHR If *handle* refers to a device.
S_IFREG If an ordinary file is referred to by *handle*.

One or both of the following bits will be set:

S_IWRITE If user has permission to write to file.
S_IREAD If user has permission to read to file.

The HPFS and NTFS file-management systems make the following distinctions:

*st_atime* Most recent access.
*st_mtime* Most recent modify.
*st_ctime* Creation time.

**Return value**

*fstat* and *stat* return 0 if they successfully retrieved the information about the open file. On error (failure to get the information), these functions return –1 and set the global variable *errno* to

EBADF Bad file handle

**See also**

*access, chmod*

## _fstr* — string.h

**Function**

Provides string operations in a large-code model.

**Syntax**

```
far string functions
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks**

Note that when a far string function returns an **int** or *size_t*, the return is never modified by the **far** keyword.

The **See also** section below provides a list of string functions that have a **far** version. The **far** version of a string function is prefixed with *_fstr*. The behavior of a **far** string function is identical to the behavior of the standard function to which it corresponds. The only difference is that for a **far** string function, the arguments and return value (only when the return value is of type '**char far** *' ) are each modified by the **far** keyword. The entry for each of the functions provides a description that applies to the **far** version.

**Return value**

When an *_fstr*-type function returns a **char** pointer, the return is a **far** type.

**See also**

*strcat, strchr, strcmp, strcpy, strcspn, strdup, stricmp, strlen, strlwr, strncat, strncmp, strncpy, strnicmp, strnset, strpbrk, strrchr, strrev, strset, strspn, strstr, strtok, strupr*

## ftell — stdio.h

**Function**

Returns the current file pointer.

**Syntax**

```
long int ftell(FILE *stream);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks**

*ftell* returns the current file pointer for *stream*. The offset is measured in bytes from the beginning of the file (if the file is binary). The value returned by *ftell* can be used in a subsequent call to *fseek*.

**Return value**

*ftell* returns the current file pointer position on success. It returns –1L on error and sets the global variable *errno* to a positive value.

In the event of an error return, the global variable *errno* is set to one of the following values:

| | |
|---|---|
| EBADF | Bad file pointer |
| ESPIPE | Illegal seek on device |

**See also**

*fgetpos, fseek, fsetpos, lseek, rewind, tell*

## ftime — sys\timeb.h

**Function**

Stores current time in *timeb* structure.

**Syntax**

```
void ftime(struct timeb *buf)
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks**

On UNIX platforms, *ftime* is available only on System V systems.

*ftime* determines the current time and fills in the fields in the *timeb* structure pointed to by *buf*. The *timeb* structure contains four fields: *time*, *millitm*, *_timezone*, and *dstflag*:

```
struct timeb {
   long time ;
   short millitm ;
   short _timezone ;
   short dstflag ;
};
```

- *time* provides the time in seconds since 00:00:00 Greenwich mean time (GMT), January 1, 1970.
- *millitm* is the fractional part of a second in milliseconds.
- *_timezone* is the difference in minutes between GMT and the local time. This value is computed going west from GMT. *ftime* gets this field from the global variable *_timezone*, which is set by *tzset*.
- *dstflag* is used to indicate whether daylight saving time will be taken into account during time calculations.

*ftime* calls *tzset*. Therefore, it isn't necessary to call *tzset* explicitly when you use *ftime*.

**Return value** None.

**See also** *asctime, ctime, gmtime, localtime, stime, time, tzset*

## _fullpath — stdlib.h

**Function** Converts a path name from relative to absolute.

**Syntax**

```
char * _fullpath(char *buffer, const char *path, int buflen);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *_fullpath* converts the relative path name in *path* to an absolute path name that is stored in the array of characters pointed to by *buffer*. The maximum number of characters that can be stored at *buffer* is *buflen*. The function returns NULL if the buffer isn't big enough to store the absolute path name, or if the path contains an invalid drive letter.

If *buffer* is NULL, *_fullpath* allocates a buffer of up to _MAX_PATH characters. This buffer should be freed using *free* when it is no longer needed. _MAX_PATH is defined in stdlib.h

**Return value** If successful, the *_fullpath* function returns a pointer to the buffer containing the absolute path name. Otherwise, it returns NULL.

**See also** *_makepath, _splitpath*

## fwrite — stdio.h

**Function** Writes to a stream.

**Syntax**

```
size_t fwrite(const void *ptr, size_t size, size_t n, FILE *stream);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *fwrite* appends *n* items of data, each of length *size* bytes, to the given output file. The data written begins at *ptr*. The total number of bytes written is ($n \times size$). *ptr* in the declarations is a pointer to any object.

**Return value** On successful completion, *fwrite* returns the number of items (not bytes) actually written. It returns a short count on error.

**See also** *fopen, fread*

## gcvt — stdlib.h



**Function** Converts floating-point number to a string.

**Syntax**

```
char *gcvt(double value, int ndec, char *buf);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks** *gcvt* converts *value* to a null-terminated ASCII string and stores the string in *buf*. It produces *ndec* significant digits in FORTRAN F format, if possible; otherwise, it returns the value in the *printf* E format (ready for printing). It might suppress trailing zeros.

**Return value** *gcvt* returns the address of the string pointed to by *buf*.

**See also** *ecvt, fcvt, sprintf*

## geninterrupt — dos.h

**Function** Generates a software interrupt.

**Syntax**

```
void geninterrupt(int intr_num);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | |

**Remarks** The *geninterrupt* macro triggers a software trap for the interrupt given by *intr_num*. The state of all registers after the call depends on the interrupt called.

➥ Interrupts can leave registers in unpredictable states.

**Return value** None.

**See also** *bdos, bdosptr, disable, enable, getvect, int86, int86x, intdos, intdosx, intr*

## getc — stdio.h

**Function** Gets character from stream.

**Syntax**

```
int getc(FILE *stream);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *getc* is a macro that returns the next character on the given input stream and increments the stream's file pointer to point to the next character.

**Return value** On success, *getc* returns the character read, after converting it to an **int** without sign extension. On end-of-file or error, it returns EOF.

**See also** *fgetc, getch, getchar, getche, gets, putc, putchar, ungetc*

## getcbrk — dos.h

**Function** Gets control-break setting.

**Syntax**

```
int getcbrk(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks** *getcbrk* uses the DOS system call 0x33 to return the current setting of control-break checking.

**Return value** *getcbrk* returns 0 if control-break checking is off, or 1 if checking is on.

**See also** *ctrlbrk, setcbrk*

## getch — conio.h

**Function** Gets character from keyboard, does not echo to screen.

**Syntax**

```
int getch(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | | ■ | | | ■ |

**Remarks** *getch* reads a single character directly from the keyboard, without echoing to the screen.

This function should not be used in Win32s or Win32 GUI applications.

**Return value** *getch* returns the character read from the keyboard.

**See also** *cgets, cscanf, fgetc, getc, getchar, getche, getpass, kbhit, putch, ungetch*

## getchar — stdio.h

**Function** Gets character from stdin.

**Syntax**

```
int getchar(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | | ■ | ■ | ■ | ■ |

**Remarks** *getchar* is a macro that returns the next character on the named input stream stdin. It is defined to be *getc(stdin)*.

For Win32s or Win32 GUI applications, stdin must be redirected.

**Return value** On success, *getchar* returns the character read, after converting it to an **int** without sign extension. On end-of-file or error, it returns EOF.

**See also** *fgetc, fgetchar, freopen, getc, getch, getche, gets, putc, putchar, scanf, ungetc*

## getche — conio.h

**Function** Gets character from the keyboard, echoes to screen.

**Syntax**

```
int getche(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | | ■ | | | ■ |

**Remarks** *getche* reads a single character from the keyboard and echoes it to the current text window using direct video or BIOS.

This function should not be used in Win32s or Win32 GUI applications.

**Return value** *getche* returns the character read from the keyboard.

**See also** *cgets, cscanf, fgetc, getc, getch, getchar, kbhit, putch, ungetch*

## getcurdir — dir.h

**Function** Gets current directory for specified drive.

**Syntax**

```
int getcurdir(int drive, char *directory);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *getcurdir* gets the name of the current working directory for the drive indicated by *drive*. *drive* specifies a drive number (0 for default, 1 for A, and so on). *directory* points to an area of memory of length MAXDIR where the null-terminated directory name will be placed. The name does not contain the drive specification and does not begin with a backslash.

**Return value** *getcurdir* returns 0 on success or –1 in the event of error.

**See also** *chdir, getcwd, getdisk, mkdir, rmdir*

## getcwd — dir.h

**Function** Gets current working directory.

**Syntax**

```
char *getcwd(char *buf, int buflen);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *getcwd* gets the full path name (including the drive) of the current working directory, up to *buflen* bytes long and stores it in *buf*. If the full path name length (including the null character) is longer than *buflen* bytes, an error occurs.

If *buf* is NULL, a buffer *buflen* bytes long is allocated for you with *malloc*. You can later free the allocated buffer by passing the return value of *getcwd* to the function *free*.

**Return value** *getcwd* returns the following values:

- If *buf* is not NULL on input, *getcwd* returns *buf* on success, NULL on error.
- If *buf* is NULL on input, *getcwd* returns a pointer to the allocated buffer.

In the event of an error return, the global variable *errno* is set to one of the following values:

| | |
|---|---|
| ENODEV | No such device |
| ENOMEM | Not enough memory to allocate a buffer (*buf* is NULL) |
| ERANGE | Directory name longer than *buflen* (*buf* is not NULL) |

**See also** *chdir, getcurdir, _getdcwd, getdisk, mkdir, rmdir*



## getdate

See *_dos_getdate.*

## _getdcwd direct.h

**Function** Gets current directory for specified drive.

**Syntax**

```
char * _getdcwd(int drive, char *buffer, int buflen);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *_getdcwd* gets the full path name of the working directory of the specified drive (including the drive name), up to *buflen* bytes long, and stores it in *buffer*. If the full path name length (including the null character) is longer than *buflen*, an error occurs. The *drive* is 0 for the default drive, 1=A, 2=B, and so on.

If *buffer* is NULL, *_getdcwd* allocates a buffer at least *buflen* bytes long. You can later free the allocated buffer by passing the *_getdcwd* return value to the *free* function.

**Return value** If successful, *_getdcwd* returns a pointer to the buffer containing the current directory for the specified drive. Otherwise it returns NULL, and sets the global variable *errno* to one of the following values:

| | |
|---|---|
| ENOMEM | Not enough memory to allocate a buffer (*buffer* is NULL) |
| ERANGE | Directory name longer than *buflen* (*buffer* is not NULL) |

**See also** *chdir, getcwd, mkdir, rmdir*

## getdfree — dos.h

**Function** Gets disk free space.

**Syntax**

```
void getdfree(unsigned char drive, struct dfree *dtable);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *getdfree* accepts a drive specifier in *drive* (0 for default, 1 for A, and so on) and fills the *dfree* structure pointed to by *dtable* with disk attributes.

The *dfree* structure is defined as follows:

```
struct dfree {
   unsigned df_avail;     /* available clusters */
   unsigned df_total;     /* total clusters */
   unsigned df_bsec;      /* bytes per sector */
   unsigned df_sclus;     /* sectors per cluster */
};
```

**Return value** *getdfree* returns no value. In the event of an error, *df_sclus* in the *dfree* structure is set to **(unsigned)** –1.

**See also** *getfat, getfatd*

## getdisk, setdisk — dir.h

**Function** Gets or sets the current drive number.

**Syntax**

```
int getdisk(void);
int setdisk(int drive);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *getdisk* gets the current drive number. It returns an integer: 0 for A, 1 for B, 2 for C, and so on. *setdisk* sets the current drive to the one associated with *drive*: 0 for A, 1 for B, 2 for C, and so on.

The *setdisk* function changes the current drive of the parent process.

**Return value** *getdisk* returns the current drive number. *setdisk* returns the total number of drives available.

**See also** *getcurdir, getcwd*

## getdta — dos.h

**Function** Gets disk transfer address.

**Syntax**

```
char far *getdta(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks** *getdta* returns the current setting of the disk transfer address (DTA).

In the small and medium memory models, it's assumed the segment is the current data segment. If you use C or C++ exclusively, this will be the case, but assembly routines can set the DTA to any hardware address.

In the compact or large models, the address returned by *getdta* is the correct hardware address and can be located outside the program.

**Return value** *getdta* returns a far pointer to the current DTA.

**See also** *setdta*

## getenv — stdlib.h

**Function** Gets a string from environment.

**Syntax**

```
char *getenv(const char *name);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *getenv* returns the value of a specified variable. On DOS and OS/2, *name* must be uppercase. On other systems, *name* can be either uppercase or lowercase. *name* must not include the equal sign (=). If the specified environment variable does not exist, *getenv* returns a NULL pointer.

To delete the variable from the environment, use `getenv("name=")`.

➥ Environment entries must not be changed directly. If you want to change an environment value, you must use *putenv*.

**Return value** On success, *getenv* returns the value associated with *name*. If the specified *name* is not defined in the environment, *getenv* returns a NULL pointer.

**See also** *_environ* (global variable), *getpsp*, *putenv*

## getfat — dos.h

**Function**

Gets file allocation table information for given drive.

**Syntax**

```
void getfat(unsigned char drive, struct fatinfo *dtable);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks**

*getfat* gets information from the file allocation table (FAT) for the drive specified by *drive* (0 for default, 1 for A, 2 for B, and so on). *dtable* points to the *fatinfo* structure to be filled in. The *fatinfo* structure filled in by *getfat* is defined as follows:

```
struct fatinfo {
   char fi_sclus;          /* sectors per cluster */
   char fi_fatid;          /* the FAT id byte */
   unsigned fi_nclus;      /* number of clusters */
   int fi_bysec;           /* bytes per sector */
};
```

**Return value**

None.

**See also**

*getdfree, getfatd*

## getfatd — dos.h

**Function**

Gets file allocation table information.

**Syntax**

```
void getfatd(struct fatinfo *dtable);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks**

*getfatd* gets information from the file allocation table (FAT) of the default drive. *dtable* points to the *fatinfo* structure to be filled in.

The *fatinfo* structure filled in by *getfatd* is defined as follows:

```
struct fatinfo {
   char fi_sclus;      /* sectors per cluster */
   char fi_fatid;      /* the FAT id byte */
   int fi_nclus;       /* number of clusters */
   int fi_bysec;       /* bytes per sector */
};
```

**Return value**

None.

**See also** *getdfree, getfat*

## getftime, setftime — io.h

**Function** Gets and sets the file date and time.

**Syntax**

```
int getftime(int handle, struct ftime *ftimep);
int setftime(int handle, struct ftime *ftimep);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *getftime* retrieves the file time and date for the disk file associated with the open *handle*. The *ftime* structure pointed to by *ftimep* is filled in with the file's time and date.

*setftime* sets the file date and time of the disk file associated with the open *handle* to the date and time in the *ftime* structure pointed to by *ftimep*. The file must not be written to after the *setftime* call or the changed information will be lost. The file must be open for writing; an EACCES error will occur if the file is open for read-only access.

The *ftime* structure is defined as follows:

```
struct ftime {
   unsigned ft_tsec: 5;      /* two seconds */
   unsigned ft_min: 6;       /* minutes */
   unsigned ft_hour: 5;      /* hours */
   unsigned ft_day: 5;       /* days */
   unsigned ft_month: 4;     /* months */
   unsigned ft_year: 7;      /* year - 1980*/
   };
```

**Return value** *getftime* and *setftime* return 0 on success.

In the event of an error return, –1 is returned and the global variable *errno* is set to one of the following values:

| | |
|---|---|
| EACCES | Permission denied |
| EBADF | Bad file number |
| EINVFNC | Invalid function number |

**See also** *fflush, open*

## getpass — conio.h

**Function** Reads a password.

**Syntax**

```
char *getpass(const char *prompt);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | | ■ | | | ■ |

**Remarks** *getpass* reads a password from the system console, after prompting with the null-terminated string *prompt* and disabling the echo. A pointer is returned to a null-terminated string of up to eight characters (not counting the null character).

➥ This function should not be used in Win32s or Win32 GUI applications.

**Return value** The return value is a pointer to a static string, which is overwritten with each call.

**See also** *getch*

## getpid — process.h

**Function** Gets the process ID of a program.

**Syntax**

```
unsigned getpid(void)
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks** This function returns the current process ID—an integer that uniquely identifies the process.

**Return value** *getpid* returns the identification number of the current process.

**See also** *getpsp*, *_psp* (global variable)

## getpsp — dos.h

**Function** Gets the program segment prefix (PSP).

**Syntax**

```
unsigned getpsp(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks** *getpsp* gets the segment address of the PSP using DOS call 0x62.

**Return value** *getpsp* returns the address of the PSP.

**See also** *getenv*, *_psp* (global variable)



## gets — stdio.h

**Function** Gets a string from stdin.

**Syntax**

```
char *gets(char *s);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | | ■ | ■ | ■ | ■ |

**Remarks** *gets* collects a string of characters terminated by a new line from the standard input stream stdin and puts it into *s*. The new line is replaced by a null character ('\0') in *s*.

*gets* allows input strings to contain certain whitespace characters (spaces, tabs). *gets* returns when it encounters a new line; everything up to the new line is copied into *s*.

➥ For Win32s or Win32 GUI applications, stdin must be redirected.

**Return value** On success, *gets* returns the string argument *s*; it returns NULL on end-of-file or error.

**See also** *cgets*, *ferror*, *fgets*, *fopen*, *fputs*, *fread*, *freopen*, *getc*, *puts*, *scanf*

## gettext — conio.h

**Function** Copies text from text mode screen to memory.

**Syntax**

```
int gettext(int left, int top, int right, int bottom, void *destin);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | | ■ | | | ■ |

**Remarks**

*gettext* stores the contents of an onscreen text rectangle defined by *left*, *top*, *right*, and *bottom* into the area of memory pointed to by *destin*.

All coordinates are absolute screen coordinates, not window-relative. The upper left corner is (1,1).

*gettext* reads the contents of the rectangle into memory sequentially from left to right and top to bottom.

Each position onscreen takes 2 bytes of memory: The first byte is the character in the cell, and the second is the cell's video attribute. The space required for a rectangle *w* columns wide by *h* rows high is defined as

*bytes* = (*h* rows) × (*w* columns) × 2

This function should not be used in Win32s or Win32 GUI applications.

**Return value**

*gettext* returns 1 if the operation succeeds. It returns 0 if it fails (for example, if you gave coordinates outside the range of the current screen mode).

**See also**

*movetext*, *puttext*

## gettextinfo — conio.h

**Function**

Gets text mode video information.

**Syntax**

```
void gettextinfo(struct text_info *r);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | | ■ | | | ■ |

**Remarks**

*gettextinfo* fills in the *text_info* structure pointed to by *r* with the current text video information.

The *text_info* structure is defined in conio.h as follows:

```
struct text_info {
   unsigned char winleft;       /* left window coordinate */
   unsigned char wintop;        /* top window coordinate */
   unsigned char winright;      /* right window coordinate */
   unsigned char winbottom;     /* bottom window coordinate */
   unsigned char attribute;     /* text attribute */
```

```
    unsigned char normattr;       /* normal attribute */
    unsigned char currmode;       /* BW40, BW80, C40, C80, or C4350 */
    unsigned char screenheight;   /* text screen's height */
    unsigned char screenwidth;    /* text screen's width */
    unsigned char curx;           /* x-coordinate in current window */
    unsigned char cury;           /* y-coordinate in current window */
};
```

➥ This function should not be used in Win32s or Win32 GUI applications.

**Return value** *gettextinfo* returns nothing; the results are returned in the structure pointed to by *r*.

**See also** *textattr, textbackground, textcolor, textmode, wherex, wherey, window*

G

## gettime, settime — dos.h

**Function** Gets and sets the system time.

**Syntax**

```
void gettime(struct time *timep);
void settime(struct time *timep);
```

| | DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|---|
| *gettime* | ■ | | ■ | ■ | | | ■ |
| *settime* | ■ | | ■ | | | | ■ |

**Remarks** *gettime* fills in the *time* structure pointed to by *timep* with the system's current time.

*settime* sets the system time to the values in the *time* structure pointed to by *timep*.

The *time* structure is defined as follows:

```
struct time {
    unsigned char ti_min;      /* minutes */
    unsigned char ti_hour;     /* hours */
    unsigned char ti_hund;     /* hundredths of seconds */
    unsigned char ti_sec;      /* seconds */
};
```

**Return value** None.

**See also** *_dos_gettime, _dos_settime, getdate, setdate, stime, time*

## getvect, setvect — dos.h

**Function**

Gets and sets interrupt vector.

**Syntax**

```
void interrupt(*getvect(int interruptno)) ();                    /* C version */
void interrupt(*getvect(int interruptno)) ( ... );                // C++ version
void setvect(int interruptno, void interrupt (*isr) ());         /* C version */
void setvect(int interruptno, void interrupt (*isr) ( ... ));     // C++ version
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks**

Every processor of the 8086 family includes a set of interrupt vectors, numbered 0 to 255. The 4-byte value in each vector is actually an address, which is the location of an interrupt function.

*getvect* reads the value of the interrupt vector given by *interruptno* and returns that value as a (far) pointer to an interrupt function. The value of *interruptno* can be from 0 to 255.

*setvect* sets the value of the interrupt vector named by *interruptno* to a new value, *isr*, which is a far pointer containing the address of a new interrupt function. The address of a C routine can be passed to *isr* only if that routine is declared to be an interrupt routine.

In C++ only static member functions or non-member functions can be declared to be an interrupt routine.

If you use the prototypes declared in dos.h, simply pass the address of an interrupt function to *setvect* in any memory model.

**Return value**

*getvect* returns the current 4-byte value stored in the interrupt vector named by *interruptno*.

*setvect* does not return a value.

**See also**

*disable, _dos_getvect, _dos_setvect, enable, geninterrupt*

## getverify — dos.h

**Function**

Returns the state of the operating system verify flag.

**Syntax**

```
int getverify(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | ■ |

**Remarks** *getverify* gets the current state of the verify flag.

The verify flag controls output to the disk. When verify is off, writes are not verified; when verify is on, all disk writes are verified to ensure proper writing of the data.

**Return value** *getverify* returns the current state of the verify flag, either 0 (off) or 1 (on).

**See also** *setverify*

## getw — stdio.h



**Function** Gets integer from stream.

**Syntax**

```
int getw(FILE *stream);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks** *getw* returns the next integer in the named input stream. It assumes no special alignment in the file.

*getw* should not be used when the stream is opened in text mode.

**Return value** *getw* returns the next integer on the input stream. On end-of-file or error, *getw* returns EOF. Because EOF is a legitimate value for *getw* to return, *feof* or *ferror* should be used to detect end-of-file or error.

**See also** *putw*

## gmtime — time.h

**Function** Converts date and time to Greenwich mean time (GMT).

**Syntax**

```
struct tm *gmtime(const time_t *timer);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *gmtime* accepts the address of a value returned by *time* and returns a pointer to the structure of type *tm* containing the time elements. *gmtime* converts directly to GMT.

The global long variable *_timezone* should be set to the difference in seconds between GMT and local standard time (in PST, *_timezone* is 8×60×60). The

global variable *_daylight* should be set to nonzero *only if* the standard U.S. daylight saving time conversion should be applied.

This is the *tm* structure declaration from the time.h header file:

```
struct tm {
   int tm_sec;          /* Seconds */
   int tm_min;          /* Minutes */
   int tm_hour;         /* Hour (0 - 23) */
   int tm_mday;         /* Day of month (1 - 31) */
   int tm_mon;          /* Month (0 - 11) */
   int tm_year;         /* Year (calendar year minus 1900) */
   int tm_wday;         /* Weekday (0 - 6; Sunday is 0) */
   int tm_yday;         /* Day of year (0 -365) */
   int tm_isdst;        /* Nonzero if daylight saving time is in effect. */
};
```

These quantities give the time on a 24-hour clock, day of month (1 to 31), month (0 to 11), weekday (Sunday equals 0), year – 1900, day of year (0 to 365), and a flag that is nonzero if daylight saving time is in effect.

**Return value**

*gmtime* returns a pointer to the structure containing the time elements. This structure is a static that is overwritten with each call.

**See also**

*asctime, ctime, ftime, localtime, stime, time, tzset*

## gotoxy — conio.h

**Function**

Positions cursor in text window.

**Syntax**

```
void gotoxy(int x, int y);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks**

*gotoxy* moves the cursor to the given position in the current text window. If the coordinates are in any way invalid, the call to *gotoxy* is ignored. An example of this is a call to *gotoxy*(40,30), when (35,25) is the bottom right position in the window.

Neither argument to *gotoxy* can be zero.

This function should not be used in Win32s or Win32 GUI applications.

**Return value**

None.

**See also**

*wherex, wherey, window*

## _heapadd — malloc.h

**Function** Add a block to the heap.

**Syntax**

```
int _heapadd(void *block, size_t size);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| | | | ■ | | | ■ |

**Remarks** This function adds a new block of memory to the heap. The block must not have been previously allocated from the heap. *_heapadd* is typically used to add large static data areas to the heap.

**Return value** *_heapadd* returns 0 if it is successful, and –1 if it is unsuccessful.

**See also** *free, malloc*



## heapcheck — alloc.h

**Function** Checks and verifies the heap.

**Syntax**

```
int heapcheck(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | | ■ | | | ■ |

**Remarks** *heapcheck* walks through the heap and examines each block, checking its pointers, size, and other critical attributes.

**Return value** The return value is less than 0 for an error and greater than 0 for success. The return values and their meaning are as follows:

| | |
|---|---|
| _HEAPCORRUPT | Heap has been corrupted |
| _HEAPEMPTY | No heap |
| _HEAPOK | Heap is verified |

## heapcheckfree — alloc.h

**Function** Checks the free blocks on the heap for a constant value.

**Syntax**

```
int heapcheckfree(unsigned int fillvalue);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | | ■ | | | ■ |

**Return value**

The return value is less then 0 for an error and greater than 0 for success. The return values and their meaning are as follows:

| | |
|---|---|
| _BADVALUE | A value other than the fill value was found |
| _HEAPCORRUPT | Heap has been corrupted |
| _HEAPEMPTY | No heap |
| _HEAPOK | Heap is accurate |

## heapchecknode — alloc.h

**Function**

Checks and verifies a single node on the heap.

**Syntax**

```
int heapchecknode(void *node);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | | ■ | | | ■ |

**Remarks**

If a node has been freed and *heapchecknode* is called with a pointer to the freed block, *heapchecknode* can return _BADNODE rather than the expected _FREEENTRY. This is because adjacent free blocks on the heap are merged, and the block in question no longer exists.

**Return value**

One of the following values:

| | |
|---|---|
| _BADNODE | Node could not be found |
| _FREEENTRY | Node is a free block |
| _HEAPCORRUPT | Heap has been corrupted |
| _HEAPEMPTY | No heap |
| _USEDENTRY | Node is a used block |

## _heapchk — malloc.h

**Function**

Checks and verifies the heap.

**Syntax**

```
int _heapchk(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| | | | ■ | | | ■ |

**Remarks** *_heapchk* walks through the heap and examines each block, checking its pointers, size, and other critical attributes.

**Return value** One of the following values:

| | |
|---|---|
| _HEAPBADNODE | A corrupted heap block has been found |
| _HEAPEMPTY | No heap exists |
| _HEAPOK | The heap appears to be uncorrupted |

**See also** *_heapset, _rtl_heapwalk*

## heapfillfree — alloc.h



**Function** Fills the free blocks on the heap with a constant value.

**Syntax**

```
int heapfillfree(unsigned int fillvalue);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | | ■ | | | ■ |

**Return value** One of the following values:

| | |
|---|---|
| _HEAPCORRUPT | Heap has been corrupted |
| _HEAPEMPTY | No heap |
| _HEAPOK | Heap is accurate |

## _heapmin — malloc.h

**Function** Release unused heap areas.

**Syntax**

```
int _heapmin(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** The *_heapmin* function returns unused areas of the heap to the operating system. This allows blocks that have been allocated and then freed to be used by other processes. Due to fragmentation of the heap, *_heapmin* might not always be able to return unused memory to the operating system; this is not an error.

**Return value** *_heapmin* returns 0 if it is successful, or –1 if an error occurs.

**See also** *free, malloc*

## _heapset — malloc.h

**Function** Fills the free blocks on the heap with a constant value.

**Syntax**

```
int _heapset(unsigned int fillvalue);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| | | | ■ | | | ■ |

**Remarks** *_heapset* checks the heap for consistency using the same methods as *_heapchk*. It then fills each free block in the heap with the value contained in the least significant byte of *fillvalue*. This function can be used to find heap-related problems. It does *not* guarantee that subsequently allocated blocks will be filled with the specified value.

**Return value** One of the following values:

| | |
|---|---|
| _HEAPOK | The heap appears to be uncorrupted |
| _HEAPEMPTY | No heap exists |
| _HEAPBADNODE | A corrupted heap block has been found |

**See also** *_heapchk, _rtl_heapwalk*

## heapwalk — alloc.h

**Function** *heapwalk* is used to "walk" through the heap, node by node.

**Syntax**

```
int heapwalk(struct heapinfo *hi);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | | ■ | | | ■ |

**Remarks** *heapwalk* assumes the heap is correct. Use *heapcheck* to verify the heap before using *heapwalk*. _HEAPOK is returned with the last block on the heap. _HEAPEND will be returned on the next call to *heapwalk*.

*heapwalk* receives a pointer to a structure of type *heapinfo* (declared in alloc.h). For the first call to *heapwalk*, set the hi.ptr field to null. *heapwalk*

returns with hi.ptr containing the address of the first block. hi.size holds the size of the block in bytes. hi.in_use is a flag that's set if the block is currently in use.

**Return value** One of the following values:

| | |
|---|---|
| _HEAPEMPTY | No heap |
| _HEAPEND | End of the heap has been reached |
| _HEAPOK | *Heapinfo* block contains valid data |

**See also** *farheapwalk, _rtl_heapwalk*



## _heapwalk — malloc.h

**Remarks** Obsolete function. See *_rtl_heapwalk*.

## highvideo — conio.h

**Function** Selects high-intensity characters.

**Syntax**

```
void highvideo(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | | ■ | | | ■ |

**Remarks** *highvideo* selects high-intensity characters by setting the high-intensity bit of the currently selected foreground color.

This function does not affect any characters currently onscreen, but does affect those displayed by functions (such as *cprintf*) that perform direct video, text mode output *after highvideo* is called.

➥ This function should not be used in Win32s or Win32 GUI applications.

**Return value** None.

**See also** *cprintf, cputs, gettextinfo, lowvideo, normvideo, textattr, textcolor*

## hypot, hypotl — math.h

**Function** Calculates hypotenuse of a right triangle.

**Syntax**

```
double hypot(double x, double y);
long double hypotl(long double x, long double y);
```

| | DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|---|
| *hypot* | ■ | ■ | ■ | ■ | | | ■ |
| *hypotl* | ■ | | ■ | ■ | | | ■ |

**Remarks**

*hypot* calculates the value z where

$z^2 = x^2 + y^2$ and $z >= 0$

This is equivalent to the length of the hypotenuse of a right triangle, if the lengths of the two sides are $x$ and $y$.

*hypotl* is the **long double** version; it takes **long double** arguments and returns a **long double** result.

**Return value**

On success, these functions return z, a **double** (*hypot*) or a **long double**) (*hypotl*). On error (such as an overflow), they set the global variable *errno* to

ERANGE Result out of range

and return the value HUGE_VAL (*hypot*) or _LHUGE_VAL (*hypotl*). Error handling for these routines can be modified through the functions *_matherr* and *_matherrl*.

## _InitEasyWin — io.h

**Function**

Initializes Easy Windows.

**Syntax**

```
void _InitEasyWin(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| | | ■ | | | | |

**Remarks**

*_InitEasyWin* allows programs to use functions which perform console I/O within 16-bit Windows.

**Return value**

None.

## inp — conio.h

**Function**

Reads a byte from a hardware port.

**Syntax**

```
int inp(unsigned portid);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks** *inp* is a macro that reads a byte from the input port specified by *portid*. If *inp* is called when conio.h has been included, it will be treated as a macro that expands to inline code. If you don't include conio.h, or if you do include conio.h and undefine the macro *inp*, you get the *inp* function.

**Return value** *inp* returns the value read.

**See also** *inpw, outp, outpw*

## inport — dos.h

**Function** Reads a word from a hardware port.

**Syntax**

```
int inport(int portid);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks** *inport* works just like the 80x86 instruction *IN*. It reads the low byte of a word from the input port specified by *portid*; it reads the high byte from *portid* + 1.

**Return value** *inport* returns the value read.

**See also** *inportb, outport, outportb*

## inportb — dos.h

**Function** Reads a byte from a hardware port.

**Syntax**

```
unsigned char inportb(int portid);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks** *inportb* is a macro that reads a byte from the input port specified by *portid*.

If *inportb* is called when dos.h has been included, it will be treated as a macro that expands to inline code. If you don't include dos.h, or if you do include dos.h and **#undef** the macro *inportb*, you get the *inportb* function.

**Return value** *inportb* returns the value read.

**See also** *inport, outport, outportb*

## inpw — conio.h

**Function** Reads a word from a hardware port.

**Syntax**

```
unsigned inpw(unsigned portid);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks** *inpw* is a macro that reads a 16-bit word from the inport port specified by *portid*. It reads the low byte of the word from *portid*, and the high byte from *portid* + 1.

If *inpw* is called when conio.h has been included, it will be treated as a macro that expands to inline code. If you don't include conio.h, or if you do include conio.h and **#undef** the macro *inpw*, you get the *inpw* function.

**Return value** *inpw* returns the value read.

**See also** *inp, outp, outpw*

## insline — conio.h

**Function** Inserts a blank line in the text window.

**Syntax**

```
void insline(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | | ■ | | | ■ |

**Remarks** *insline* inserts an empty line in the text window at the cursor position using the current text background color. All lines below the empty one move down one line, and the bottom line scrolls off the bottom of the window.

This function should not be used in Win32s or Win32 GUI applications.

**Return value** None.

**See also** *clreol, delline, window*

## int86 — dos.h

**Function**

General 8086 software interrupt.

**Syntax**

```
int int86(int intno, union REGS *inregs, union REGS *outregs);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks**

*int86* executes an 8086 software interrupt specified by the argument *intno*. Before executing the software interrupt, it copies register values from *inregs* into the registers.

After the software interrupt returns, *int86* copies the current register values to *outregs*, copies the status of the carry flag to the *x.cflag* field in *outregs*, and copies the value of the 8086 flags register to the *x.flags* field in *outregs*. If the carry flag is set, it usually indicates that an error has occurred.

Note that *inregs* can point to the same structure that *outregs* points to.

**Return value**

*int86* returns the value of AX after completion of the software interrupt. If the carry flag is set (`outregs -> x.cflag != 0`), indicating an error, this function sets the global variable *_doserrno* to the error code. Note that when the carry flag is *not* set (`outregs -> x.cflag = 0`), you may or may not have an error. To be certain, always check *_doserrno*.

**See also**

*bdos, bdosptr, geninterrupt, int86x, intdos, intdosx, intr*

## int86x — dos.h

**Function**

General 8086 software interrupt interface.

**Syntax**

```
int int86x(int intno, union REGS *inregs, union REGS *outregs,
           struct SREGS *segregs);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks**

*int86x* executes an 8086 software interrupt specified by the argument *intno*. Before executing the software interrupt, it copies register values from *inregs* into the registers.

In addition, *int86x* copies the *segregs ->ds* and *segregs ->es* values into the corresponding registers before executing the software interrupt. This

feature allows programs that use far pointers or a large data memory model to specify which segment is to be used for the software interrupt.

After the software interrupt returns, *int86x* copies the current register values to *outregs*, the status of the carry flag to the x.cflag field in *outregs*, and the value of the 8086 flags register to the x.flags field in *outregs*. In addition, *int86x* restores DS and sets the *segregs ->es* and *segregs ->ds* fields to the values of the corresponding segment registers. If the carry flag is set, it usually indicates that an error has occurred.

*int86x* lets you invoke an 8086 software interrupt that takes a value of DS different from the default data segment, and/or takes an argument in ES.

Note that *inregs* can point to the same structure that *outregs* points to.

**Return value**

*int86x* returns the value of AX after completion of the software interrupt. If the carry flag is set (`outregs -> x.cflag != 0`), indicating an error, this function sets the global variable *_doserrno* to the error code. Note that when the carry flag is *not* set (`outregs -> x.cflag = 0`), you may or may not have an error. To be certain, always check *_doserrno*.

**See also**

*bdos, bdosptr, geninterrupt, intdos, intdosx, int86, intr, segread*

## intdos — dos.h

**Function**

General DOS interrupt interface.

**Syntax**

```
int intdos(union REGS *inregs, union REGS *outregs);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks**

*intdos* executes DOS interrupt 0x21 to invoke a specified DOS function. The value of *inregs -> h.ah* specifies the DOS function to be invoked.

After the interrupt 0x21 returns, *intdos* copies the current register values to *outregs*, copies the status of the carry flag to the *x.cflag* field in *outregs*, and copies the value of the 8086 flags register to the *x.flags* field in *outregs*. If the carry flag is set, it indicates that an error has occurred.

Note that *inregs* can point to the same structure that *outregs* points to.

**Return value**

*intdos* returns the value of AX after completion of the DOS function call. If the carry flag is set (`outregs -> x.cflag != 0`), indicating an error, it sets the global variable *_doserrno* to the error code. Note that when the carry flag is *not* set (`outregs -> x.cflag = 0`), you may or may not have an error. To be certain, always check *_doserrno*.

**See also** *bdos, bdosptr, geninterrupt, int86, int86x, intdosx, intr*

## intdosx — dos.h

**Function** General DOS interrupt interface.

**Syntax**

```
int intdosx(union REGS *inregs, union REGS *outregs, struct SREGS *segregs);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks** *intdosx* executes DOS interrupt 0x21 to invoke a specified DOS function. The value of *inregs -> h.ah* specifies the DOS function to be invoked.

In addition, *intdosx* copies the *segregs ->ds* and *segregs ->es* values into the corresponding registers before invoking the DOS function. This feature allows programs that use far pointers or a large data memory model to specify which segment is to be used for the function execution.

After the interrupt 0x21 returns, *intdosx* copies the current register values to *outregs*, copies the status of the carry flag to the x.cflag field in *outregs*, and copies the value of the 8086 flags register to the x.flags field in *outregs*. In addition, *intdosx* sets the *segregs ->es* and *segregs ->ds* fields to the values of the corresponding segment registers and then restores DS. If the carry flag is set, it indicates that an error occurred.

*intdosx* lets you invoke a DOS function that takes a value of DS different from the default data segment and/or takes an argument in ES.

Note that *inregs* can point to the same structure that *outregs* points to.

**Return value** *intdosx* returns the value of AX after completion of the DOS function call. If the carry flag is set (`outregs -> x.cflag != 0`), indicating an error, it sets the global variable *_doserrno* to the error code. Note that when the carry flag is *not* set (`outregs -> x.cflag = 0`), you may or may not have an error. To be certain, always check *_doserrno*.

**See also** *bdos, bdosptr, geninterrupt, int86, int86x, intdos, intr, segread*

## intr — dos.h

**Function** Alternate 8086 software interrupt interface.

**Syntax**

```
void intr(int intno, struct REGPACK *preg);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks**

The *intr* function is an alternate interface for executing software interrupts. It generates an 8086 software interrupt specified by the argument *intno*.

*intr* copies register values from the *REGPACK* structure **preg* into the registers before executing the software interrupt. After the software interrupt completes, *intr* copies the current register values into **preg*, including the flags.

The arguments passed to *intr* are as follows:

*intno* Interrupt number to be executed

*preg* Address of a structure containing

(a) the input registers before the interrupt call
(b) the value of the registers after the interrupt call

The *REGPACK* structure (defined in dos.h) has the following format:

```
struct  REGPACK {
   unsigned  r_ax, r_bx, r_cx, r_dx;
   unsigned  r_bp, r_si, r_di, r_ds, r_es, r_flags;
};
```

**Return value**

No value is returned. The *REGPACK* structure **preg* contains the value of the registers after the interrupt call.

**See also**

*geninterrupt, int86, int86x, intdos, intdosx*

## ioctl — io.h

**Function**

Controls I/0 device.

**Syntax**

```
int ioctl(int handle, int func[, void *argdx, int argcx]);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | | | | |

**Remarks**

*ioctl* is available on UNIX systems, but not with these parameters or functionality. UNIX version 7 and System III differ from each other in their use of *ioctl*. *ioctl* calls are not portable to UNIX and are rarely portable across DOS machines.

DOS 3.0 extends *ioctl* with *func* values of 8 and 11.

This is a direct interface to the DOS call 0x44 (IOCTL).

The exact function depends on the value of *func* as follows:

| | |
|---|---|
| 0 | Get device information. |
| 1 | Set device information (in *argdx*). |
| 2 | Read *argcx* bytes into the address pointed to by *argdx*. |
| 3 | Write *argcx* bytes from the address pointed to by *argdx*. |
| 4 | Same as 2 except *handle* is treated as a drive number (0 equals default, 1 equals A, and so on). |
| 5 | Same as 3 except *handle* is a drive number (0 equals default, 1 equals A, and so on). |
| 6 | Get input status. |
| 7 | Get output status. |
| 8 | Test removability; DOS 3.0 only. |
| 11 | Set sharing conflict retry count; DOS 3.0 only. |

*ioctl* can be used to get information about device channels. Regular files can also be used, but only *func* values 0, 6, and 7 are defined for them. All other calls return an EINVAL error for files.

See the documentation for system call 0x44 in your DOS reference manuals for detailed information on argument or return values.

The arguments *argdx* and *argcx* are optional.

*ioctl* provides a direct interface to DOS device drivers for special functions. As a result, the exact behavior of this function varies across different vendors' hardware and in different devices. Also, several vendors do not follow the interfaces described here. Refer to the vendor BIOS documentation for exact use of *ioctl*.

**Return value**

For *func* 0 or 1, the return value is the device information (DX of the *ioctl* call). For *func* values of 2 through 5, the return value is the number of bytes actually transferred. For *func* values of 6 or 7, the return value is the device status.

In any event, if an error is detected, a value of –1 is returned, and the global variable *errno* is set to one of the following:

| | |
|---|---|
| EBADF | Bad file number |
| EINVAL | Invalid argument |
| EINVDAT | Invalid data |

## isalnum — ctype.h

**Function** Tests for an alphanumeric character.

**Syntax**

```
int isalnum(int c);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *isalnum* is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, *c* is a letter (*A* to Z or *a* to z) or a digit (0 to 9).

You can make this macro available as a function by undefining (**#undef**) it.

**Return value** It is a predicate returning nonzero for true and 0 for false. *isalnum* returns nonzero if *c* is a letter or a digit.

## isalpha — ctype.h

**Function** Classifies an alphabetical character.

**Syntax**

```
int isalpha(int c);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *isalpha* is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, *c* is a letter (*A* to Z or *a* to z).

You can make this macro available as a function by undefining (**#undef**) it.

**Return value** *isalpha* returns nonzero if *c* is a letter.

## isascii — ctype.h

**Function** Character classification macro.

**Syntax**

```
int isascii(int c);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks**

*isascii* is a macro that classifies ASCII-coded integer values by table lookup. It is a predicate returning nonzero for true and 0 for false.

*isascii* is defined on all integer values.

**Return value**

*isascii* returns nonzero if the low order byte of *c* is in the range 0 to 127 (0x00-0x7F).

## isatty — io.h

**Function**

Checks for device type.

**Syntax**

```
int isatty(int handle);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks**

*isatty* determines whether *handle* is associated with any one of the following character devices:

- A terminal
- A console
- A printer
- A serial port

**Return value**

If the device is one of the four character devices listed above, *isatty* returns a nonzero integer. If it is not such a device, *isatty* returns 0.

## iscntrl — ctype.h

**Function**

Tests for a control character.

**Syntax**

```
int iscntrl(int c);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks**

*iscntrl* is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, *c* is a delete character or control character (0x7F or 0x00 to 0x1F).

You can make this macro available as a function by undefining (**#undef**) it.

**Return value** *iscntrl* returns nonzero if *c* is a delete character or ordinary control character.

## isdigit — ctype.h

**Function** Tests for decimal-digit character.

**Syntax**

```
int isdigit(int c);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *isdigit* is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, *c* is a digit (0 to 9).

You can make this macro available as a function by undefining (**#undef**) it.

**Return value** *isdigit* returns nonzero if *c* is a digit.

## isgraph — ctype.h

**Function** Tests for printing character.

**Syntax**

```
int isgraph(int c);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *isgraph* is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, *c* is a printing character except blank space (' ').

You can make this macro available as a function by undefining (**#undef**) it.

**Return value** *isgraph* returns nonzero if *c* is a printing character.

## islower — ctype.h

**Function** Tests for lowercase character.

**Syntax**

```
int islower(int c);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *islower* is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, *c* is a lowercase letter (*a* to *z*).

You can make this macro available as a function by undefining (**#undef**) it.

**Return value** *islower* returns nonzero if *c* is a lowercase letter.

## isprint — ctype.h

**Function** Tests for printing character.

**Syntax**

```
int isprint(int c);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *isprint* is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, *c* is a printing character including the blank space (' ').

You can make this macro available as a function by undefining (**#undef**) it.

**Return value** *isprint* returns nonzero if *c* is a printing character.

## ispunct — ctype.h

**Function** Tests for punctuation character.

**Syntax**

```
int ispunct(int c);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *ispunct* is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, *c* is any printing character that is neither an alphanumeric nor a blank space (' ').

You can make this macro available as a function by undefining (**#undef**) it.

**Return value** *ispunct* returns nonzero if *c* is a punctuation character.

## isspace — ctype.h

**Function** Tests for space character.

**Syntax**

```
int isspace(int c);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *isspace* is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category.

You can make this macro available as a function by undefining (**#undef**) it.

**Return value** *isspace* returns nonzero if *c* is a space, tab, carriage return, new line, vertical tab, formfeed (0x09 to 0x0D, 0x20), or any other locale-defined space character.

## isupper — ctype.h

**Function** Tests for uppercase character.

**Syntax**

```
int isupper(int c);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *isupper* is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, *c* is an uppercase letter (*A* to *Z*).

You can make this macro available as a function by undefining (**#undef**) it.

**Return value** *isupper* returns nonzero if *c* is an uppercase letter.

## isxdigit — ctype.h

**Function** Tests for hexadecimal character.

**Syntax**

```
int isxdigit(int c);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *isxdigit* is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category.

You can make this macro available as a function by undefining (**#undef**) it.

**Return value** *isxdigit* returns nonzero if *c* is a hexadecimal digit (0 to 9, *A to F, a to f*) or any other hexadecimal digit defined by the locale.

## itoa — stdlib.h

**Function** Converts an integer to a string.

**Syntax**

```
char *itoa(int value, char *string, int radix);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *itoa* converts *value* to a null-terminated string and stores the result in *string*. With *itoa*, *value* is an integer.

*radix* specifies the base to be used in converting *value*; it must be between 2 and 36, inclusive. If *value* is negative and *radix* is 10, the first character of *string* is the minus sign (–).

➥ The space allocated for *string* must be large enough to hold the returned string, including the terminating null character (\0). *itoa* can return up to 17 bytes.

**Return value** *itoa* returns a pointer to *string*.

**See also** *ltoa, ultoa*

## kbhit — conio.h

**Function** Checks for currently available keystrokes.

**Syntax**

```
int kbhit(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *kbhit* checks to see if a keystroke is currently available. Any available keystrokes can be retrieved with *getch* or *getche*.

This function should not be used in Win32s or Win32 GUI applications.

**Return value** If a keystroke is available, *kbhit* returns a nonzero value. Otherwise, it returns 0.

**See also** *getch, getche*

## labs — math.h

**Function** Gives long absolute value.

**Syntax**

```
long labs(long int x);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *labs* computes the absolute value of the parameter $x$.

**Return value** *labs* returns the absolute value of $x$.

**See also** *abs, cabs, fabs*

## ldexp, ldexpl — math.h

**Function** Calculates $x \times 2^{exp}$.

**Syntax**

```
double ldexp(double x, int exp);
long double ldexpl(long double x, int exp);
```

| | DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|---|
| *ldexp* | ■ | ■ | ■ | ■ | ■ | ■ | ■ |
| *ldexpl* | ■ | | ■ | ■ | | | ■ |

**Remarks** *ldexp* calculates the **double** value $x \times 2^{exp}$.

*expl* is the **long double** version; it takes a **long double** argument for $x$ and returns a **long double** result.

**Return value** On success, *ldexp* (or *ldexpl*) returns the value it calculated, $x \times 2^{exp}$. Error handling for these routines can be modified through the functions *_matherr* and *_matherrl*.

**See also** *exp, frexp, modf*

## ldiv — stdlib.h

**Function** Divides two **long**s, returning quotient and remainder.

**Syntax**

```
ldiv_t ldiv(long int numer, long int denom);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | ■ | ■ | ■ |

**Remarks** *ldiv* divides two **long**s and returns both the quotient and the remainder as an *ldiv_t* type. *numer* and *denom* are the numerator and denominator, respectively. The *ldiv_t* type is a structure of **long**s defined in stdlib.h as follows:

```
typedef struct {
   long int quot;      /* quotient */
   long int rem;       /* remainder */
   } ldiv_t;
```

**Return value** *ldiv* returns a structure whose elements are *quot* (the quotient) and *rem* (the remainder).

**See also** *div*

## lfind — stdlib.h

**Function** Performs a linear search.

**Syntax**

```
void  *lfind(const void *key, const void *base, size_t *num, size_t width,
             int (_USERENTRY *fcmp)(const void *, const void *));
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks** *lfind* makes a linear search for the value of *key* in an array of sequential records. It uses a user-defined comparison routine *fcmp*. The *fcmp* function must be used with the _USERENTRY calling convention.

The array is described as having **num* records that are *width* bytes wide, and begins at the memory location pointed to by *base*.

**Return value** *lfind* returns the address of the first entry in the table that matches the search key. If no match is found, *lfind* returns NULL. The comparison

routine must return 0 if **elem1* == **elem2*, and nonzero otherwise (*elem1* and *elem2* are its two parameters).

**See also** *bsearch, lsearch, qsort*

## localeconv — locale.h

**Function** Queries the locale for numeric format.

**Syntax**

```
struct lconv *localeconv(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | ■ | ■ | ■ |

**Remarks** This function provides information about the monetary and other numeric formats for the current locale. The information is stored in a **struct** *lconv* type. The structure can only be modified by the *setlocale*. Subsequent calls to *localeconv* will update the *lconv* structure.

The *lconv* structure is defined in locale.h. It contains the following fields:

Table 3.1: Locale monetary and numeric settings

| Field | Application |
|---|---|
| **char** **decimal_point;* | Decimal point used in nonmonetary formats. This can never be an empty string. |
| **char** **thousands_sep;* | Separator used to group digits to the left of the decimal point. Not used with monetary quantities. |
| **char** **grouping;* | Size of each group of digits. Not used with monetary quantities. See the value listing table below. |
| **char** **int_curr_symbol;* | International monetary symbol in the current locale. The symbol format is specified in the *ISO 4217 Codes for the Representation of Currency and Funds.* |
| **char** **currency_symbol;* | Local monetary symbol for the current locale. |
| **char** **mon_decimal_point;* | Decimal point used to format monetary quantities. |
| **char** **mon_thousands_sep;* | Separator used to group digits to the left of the decimal point for monetary quantities. |
| **char** **mon_grouping;* | Size of each group of digits used in monetary quantities. See the value listing table below. |
| **char** **positive_sign;* | String indicating nonnegative monetary quantities. |
| **char** **negative_sign;* | String indicating negative monetary quantities. |
| **char** *int_frac_digits;* | Number of digits after the decimal point that are to be displayed in an internationally formatted monetary quantity. |

Table 3.1: Locale monetary and numeric settings (continued)

| | |
|---|---|
| **char** *frac_digits;* | Number of digits after the decimal point that are to be displayed in a formatted monetary quantity. |
| **char** *p_cs_precedes;* | Set to 1 if *currency_symbol* precedes a nonnegative formatted monetary quantity. If *currency_symbol* is after the quantity, it is set to 0. |
| **char** *p_sep_by_space;* | Set to 1 if *currency_symbol* is to be separated from the nonnegative formatted monetary quantity by a space. Set to 0 if there is no space separation. |
| **char** *n_cs_precedes;* | Set to 1 if *currency_symbol* precedes a negative formatted monetary quantity. If *currency_symbol* is after the quantity, set to 0. |
| **char** *n_sep_by_space;* | Set to 1 if *currency_symbol* is to be separated from the negative formatted monetary quantity by a space. Set to 0 if there is no space separation. |
| **char** *p_sign_posn;* | Indicate where to position the positive sign in a nonnegative formatted monetary quantity. |
| **char** *n_sign_posn;* | Indicate where to position the positive sign in a negative formatted monetary quantity. |



Any of the above strings (except *decimal_point*) that is empty "" is not supported in the current locale. The nonstring **char** elements are nonnegative numbers. Any nonstring **char** element that is set to *CHAR_MAX* indicates that the element is not supported in the current locale.

The *grouping* and *mon_grouping* elements are set and interpreted as follows:

| Value | Meaning |
|---|---|
| *CHAR_MAX* | No further grouping is to be performed. |
| 0 | The previous element is to be used repeatedly for the remainder of the digits. |
| *any other integer* | Indicates how many digits make up the current group. The next element is read to determine the size of the next group of digits before the current group. |

The *p_sign_posn* and *n_sign_posn* elements are set and interpreted as follows:

| Value | Meaning |
|---|---|
| 0 | Use parentheses to surround the quantity and *currency_symbol* |
| 1 | Sign string precedes the quantity and *currency_symbol.* |
| 2 | Sign string succeeds the quantity and *currency_symbol.* |

| Value | Meaning |
|---|---|
| 3 | Sign string immediately precedes the quantity and *currency_symbol*. |
| 4 | Sign string immediately succeeds the quantity and *currency_symbol*. |

**Return value**

Returns a pointer to the filled-in structure of type **struct** *lconv*. The values in the structure will change whenever *setlocale* modifies the LC_MONETARY or LC_NUMERIC categories.

**See also**

*setlocale*

## localtime — time.h

**Function**

Converts date and time to a structure.

**Syntax**

```
struct tm *localtime(const time_t *timer);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks**

*localtime* accepts the address of a value returned by *time* and returns a pointer to the structure of type *tm* containing the time elements. It corrects for the time zone and possible daylight saving time.

The global long variable *timezone* contains the difference in seconds between GMT and local standard time (in PST, *timezone* is 8×60×60). The global variable *daylight* contains nonzero *only if* the standard U.S. daylight saving time conversion should be applied. These values are set by *tzset*, not by the user program directly.

This is the **tm** structure declaration from the time.h header file:

```
struct tm {
    int tm_sec;
    int tm_min;
    int tm_hour;
    int tm_mday;
    int tm_mon;
    int tm_year;
    int tm_wday;
    int tm_yday;
    int tm_isdst;
};
```

These quantities give the time on a 24-hour clock, day of month (1 to 31), month (0 to 11), weekday (Sunday equals 0), year – 1900, day of year (0 to 365), and a flag that is nonzero if daylight saving time is in effect.

**Return value** *localtime* returns a pointer to the structure containing the time elements. This structure is a static that is overwritten with each call.

**See also** *asctime, ctime, ftime, gmtime, stime, time, tzset*

## lock — io.h

**Function** Sets file-sharing locks.

**Syntax**

```
int lock(int handle, long offset, long length);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |



**Remarks** *lock* provides an interface to the operating system file-sharing mechanism.

A lock can be placed on arbitrary, nonoverlapping regions of any file. A program attempting to read or write into a locked region will retry the operation three times. If all three retries fail, the call fails with an error.

**Return value** *lock* returns 0 on success. On error, *lock* returns –1 and sets the global variable *errno* to

EACCES Locking violation

**See also** *locking, open, sopen, unlock*

## locking — io.h, sys\locking.h

**Function** Sets or resets file-sharing locks.

**Syntax**

```
int locking(int handle, int cmd, long length);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *locking* provides an interface to the operating system file-sharing mechanism. The file to be locked or unlocked is the open file specified by *handle*. The region to be locked or unlocked starts at the current file position, and is *length* bytes long.

Locks can be placed on arbitrary, nonoverlapping regions of any file. A program attempting to read or write into a locked region will retry the operation three times. If all three retries fail, the call fails with an error.

The *cmd* specifies the action to be taken (the values are defined in sys\locking.h):

| | |
|---|---|
| LK_LOCK | Lock the region. If the lock is unsuccessful, try once a second for 10 seconds before giving up. |
| LK_RLCK | Same as LK_LOCK. |
| LK_NBLCK | Lock the region. If the lock if unsuccessful, give up immediately. |
| LK_NBRLCK | Same as LK_NBLCK. |
| LK_UNLCK | Unlock the region, which must have been previously locked. |

**Return value**

On successful operations, *locking* returns 0. Otherwise, it returns –1, and the global variable *errno* is set to one of the following values:

| | |
|---|---|
| EACCES | File already locked or unlocked |
| EBADF | Bad file number |
| EDEADLOCK | File cannot be locked after 10 retries (*cmd* is LK_LOCK or LK_RLCK) |
| EINVAL | Invalid *cmd*, or SHARE.EXE not loaded |

**See also**

*_fsopen, lock, open, sopen, unlock*

## log, logl — math.h

**Function**

Calculates the natural logarithm of *x*.

**Syntax**

```
double log(double x);
long double logl(long double x);
```

| | DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|---|
| *log* | ■ | ■ | ■ | ■ | ■ | ■ | ■ |
| *logl* | ■ | | ■ | ■ | | | ■ |

**Remarks**

*log* calculates the natural logarithm of *x*.

*logl* is the **long double** version; it takes a **long double** argument and returns a **long double** result.

This function can be used with *bcd* and *complex* types.

**Return value**

On success, *log* and *logl* return the value calculated, $ln(x)$.

If the argument $x$ passed to these functions is real and less than 0, the global variable *errno* is set to

EDOM Domain error

If $x$ is 0, the functions return the value negative HUGE_VAL (*log*) or negative _LHUGE_VAL (*logl*), and set *errno* to ERANGE. Error handling for these routines can be modified through the functions *_matherr* and *_matherrl*.

**See also**

*bcd, complex, exp, log10, sqrt*

## log10, log10l — math.h



**Function**

Calculates $\log_{10}(x)$.

**Syntax**

```
double log10(double x);
long double log10l(long double x);
```

| | DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|---|
| *log10* | ■ | ■ | ■ | ■ | ■ | ■ | ■ |
| *log10l* | ■ | | ■ | ■ | | | ■ |

**Remarks**

*log10* calculates the base 10 logarithm of $x$.

*log10l* is the **long double** version; it takes a **long double** argument and returns a **long double** result.

This function can be used with *bcd* and *complex* types.

**Return value**

On success, *log10* (or *log10l*) returns the value calculated, $log_{10}(x)$.

If the argument $x$ passed to these functions is real and less than 0, the global variable *errno* is set to

EDOM Domain error

If $x$ is 0, these functions return the value negative HUGE_VAL (*log10*) or _LHUGE_VAL (*log10l*). Error handling for these routines can be modified through the functions *_matherr* and *_matherrl*.

**See also**

*bcd, complex, exp, log*

# longjmp — setjmp.h

**Function**

Performs nonlocal goto.

**Syntax**

```
void longjmp(jmp_buf jmpb, int retval);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks**

A call to *longjmp* restores the task state captured by the last call to *setjmp* with the argument *jmpb*. It then returns in such a way that *setjmp* appears to have returned with the value *retval*.

A task state includes:

| Win 16 | Win 32 |
|---|---|
| All segment registers CS, DS, ES, SS | No segment registers are saved |
| Register variables DI and SI | Register variables EBX, EDI, ESI |
| Stack pointer SP | Stack pointer ESP |
| Frame pointer BP | Frame pointer EBP |
| Flags | Flags are not saved |

A task state is complete enough that *setjmp* and *longjmp* can be used to implement co-routines.

*setjmp* must be called before *longjmp*. The routine that called *setjmp* and set up *jmpb* must still be active and cannot have returned before the *longjmp* is called. If this happens, the results are unpredictable.

*longjmp* cannot pass the value 0; if 0 is passed in *retval*, *longjmp* will substitute 1.

You can not use *longjmp* to switch between different threads in a multithread process. That is, do not jump to a *jmp_buf* that was saved by a *setjmp* call in a different thread.

**Return value**

None.

**See also**

*ctrlbrk*, *setjmp*, *signal*

## lowvideo — conio.h

**Function** Selects low-intensity characters.

**Syntax**

```
void lowvideo(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | | ■ | | | ■ |

**Remarks** *lowvideo* selects low-intensity characters by clearing the high-intensity bit of the currently selected foreground color.

This function does not affect any characters currently onscreen. It affects only those characters displayed by functions that perform text mode, direct console output *after* this function is called.

➥ This function should not be used in Win32s or Win32 GUI applications.

**Return value** None.

**See also** *highvideo, normvideo, textattr, textcolor*

K-M

## _lrotl, _lrotr — stdlib.h

**Function** Rotates an **unsigned long** integer value to the left or right.

**Syntax**

```
unsigned long _lrotl(unsigned long val, int count);
unsigned long _lrotr(unsigned long val, int count);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *_lrotl* rotates the given *val* to the left *count* bits. *_lrotr* rotates the given *val* to the right *count* bits.

**Return value** The functions return the rotated integer:

- *_lrotl* returns the value of *val* left-rotated *count* bits.
- *_lrotr* returns the value of *val* right-rotated *count* bits.

**See also** *_crotr, _crotl, _rotl, _rotr*

## lsearch — stdlib.h

**Function**

Performs a linear search.

**Syntax**

```
void *lsearch(const void *key, void *base, size_t *num, size_t width,
              int (_USERENTRY *fcmp)(const void *, const void *));
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks**

*lsearch* searches a table for information. Because this is a linear search, the table entries do not need to be sorted before a call to *lsearch*. If the item that *key* points to is not in the table, *lsearch* appends that item to the table.

- *base* points to the base (0th element) of the search table.
- *num* points to an integer containing the number of entries in the table.
- *width* contains the number of bytes in each entry.
- *key* points to the item to be searched for (the *search key*).

The function *fcmp* must be used with the _USERENTRY calling convention.

The argument *fcmp* points to a user-written comparison routine, that compares two items and returns a value based on the comparison.

To search the table, *lsearch* makes repeated calls to the routine whose address is passed in *fcmp*.

On each call to the comparison routine, *lsearch* passes two arguments: *key*, a pointer to the item being searched for, and *elem*, a pointer to the element of *base* being compared.

*fcmp* is free to interpret the search key and the table entries in any way.

**Return value**

*lsearch* returns the address of the first entry in the table that matches the search key.

If the search key is not identical to **elem*, *fcmp* returns a nonzero integer. If the search key is identical to **elem*, *fcmp* returns 0.

**See also**

*bsearch*, *lfind*, *qsort*

## lseek — io.h

**Function**

Moves file pointer.

**Syntax**

```
long lseek(int handle, long offset, int fromwhere);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks**

*lseek* sets the file pointer associated with *handle* to a new position *offset* bytes beyond the file location given by *fromwhere*. *fromwhere* must be one of the following symbolic constants (defined in io.h):

| *fromwhere* | File location |
|---|---|
| SEEK_CUR | Current file pointer position |
| SEEK_END | End-of-file |
| SEEK_SET | File beginning |

**Return value**

*lseek* returns the offset of the pointer's new position measured in bytes from the file beginning. *lseek* returns –1L on error, and the global variable *errno* is set to one of the following values:

| | |
|---|---|
| EBADF | Bad file handle |
| EINVAL | Invalid argument |
| ESPIPE | Illegal seek on device |

On devices incapable of seeking (such as terminals and printers), the return value is undefined.

**See also**

*filelength, fseek, ftell, getc, open, sopen, ungetc, _rtl_write, write*



## ltoa — stdlib.h

**Function**

Converts a **long** to a string.

**Syntax**

```
char *ltoa(long value, char *string, int radix);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks**

*ltoa* converts *value* to a null-terminated string and stores the result in *string*. *value* is a long integer.

*radix* specifies the base to be used in converting *value*; it must be between 2 and 36, inclusive. If *value* is negative and *radix* is 10, the first character of *string* is the minus sign (–).

The space allocated for *string* must be large enough to hold the returned string, including the terminating null character (\0). *ltoa* can return up to 33 bytes.

**Return value** *ltoa* returns a pointer to *string*.

**See also** *itoa, ultoa*

## _makepath — stdlib.h

**Function** Builds a path from component parts.

**Syntax**

```
void _makepath(char *path, const char *drive, const char *dir,
               const char *name, const char *ext);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *_makepath* makes a path name from its components. The new path name is

```
X:\DIR\SUBDIR\NAME.EXT
```

where

*drive* = X:
*dir* = \DIR\SUBDIR\
*name* = NAME
*ext* = .EXT

If *drive* is empty or NULL, no drive is inserted in the path name. If it is missing a trailing colon (:), a colon is inserted in the path name.

If *dir* is empty or NULL, no directory is inserted in the path name. If it is missing a trailing slash (\ or /), a backslash is inserted in the path name.

If *name* is empty or NULL, no file name is inserted in the path name.

If *ext* is empty or NULL, no extension is inserted in the path name. If it is missing a leading period (.), a period is inserted in the path name.

*_makepath* assumes there is enough space in *path* for the constructed path name. The maximum constructed length is _MAX_PATH. _MAX_PATH is defined in stdlib.h.

*_makepath* and *_splitpath* are invertible; if you split a given *path* with *_splitpath*, then merge the resultant components with *_makepath*, you end up with *path*.

**Return value** None.

**See also** *_fullpath, _splitpath*

## malloc — stdlib.h

**Function** Allocates main memory.

**Syntax**

```
void *malloc(size_t size);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *malloc* allocates a block of *size* bytes from the memory heap. It allows a program to allocate memory explicitly as it's needed, and in the exact amounts needed.

The heap is used for dynamic allocation of variable-sized blocks of memory. Many data structures, for example, trees and lists, naturally employ heap memory allocation.


All the space between the end of the data segment and the top of the program stack is available for use in the small data models, except for a small margin immediately before the top of the stack. This margin is intended to allow the application some room to make the stack larger, in addition to a small amount needed by DOS.

In the large data models, all the space beyond the program stack to the end of available memory is available for the heap.

**Return value** On success, *malloc* returns a pointer to the newly allocated block of memory. If not enough space exists for the new block, it returns NULL. The contents of the block are left unchanged. If the argument *size* == 0, *malloc* returns NULL.

**See also** *calloc, farcalloc, farmalloc, free, realloc*

## _matherr, _matherrl — math.h

**Function** User-modifiable math error handler.

**Syntax**

```
int _matherr(struct _exception *e);
int _matherrl(struct _exceptionl *e);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks**

*_matherr* is called when an error is generated by the math library.

*_matherrl* is the **long double** version; it is called when an error is generated by the **long double** math functions.

*_matherr* and *_matherrl* each serve as a user hook (a function that can be customized by the user) that you can replace by writing your own math error handling routine. The example shows a user-defined *_matherr* implementation.

*_matherr* and *_matherrl* are useful for trapping domain and range errors caused by the math functions. They do not trap floating-point exceptions, such as division by zero. See *signal* for information on trapping such errors.

You can define your own *_matherr* or *_matherrl* routine to be a custom error handler (such as one that catches and resolves certain types of errors); this customized function overrides the default version in the C library. The customized *_matherr* or *_matherrl* should return 0 if it fails to resolve the error, or nonzero if the error is resolved. If nonzero is returned, no error message is printed and the global variable *errno* is not changed.

Here are the *_exception* and *_exceptionl* structures (defined in math.h):

```
struct _exception {
   int   type;
   char  *name;
   double arg1, arg2, retval;
};

struct _exceptionl {
   int   type;
   char  *name;
   long double arg1, arg2, retval;
};
```

The members of the *_exception* and *_exceptionl* structures are shown in the following table:

| Member | What it is (or represents) |
|---|---|
| *type* | The type of mathematical error that occurred; an **enum** type defined in the **typedef** *_mexcep* (see definition after this list). |
| *name* | A pointer to a null-terminated string holding the *name* of the math library function that resulted in an error. |
| *arg1*, *arg2* | The arguments (passed to the function that *name* points to) caused the error; if only one argument was passed to the function, it is stored in *arg1*. |
| *retval* | The default return value for *_matherr* (or *_matherrl*); you can modify this value. |

The **typedef** *_mexcep*, also defined in math.h, enumerates the following symbolic constants representing possible mathematical errors:

| Symbolic constant | Mathematical error |
|---|---|
| DOMAIN | Argument was not in domain of function, such as *log*(-1). |
| SING | Argument would result in a singularity, such as *pow*(0, –2). |
| OVERFLOW | Argument would produce a function result greater than DBL_MAX (or LDBL_MAX), such as *exp*(1000). |
| UNDERFLOW | Argument would produce a function result less than DBL_MIN (or LDBL_MIN), such as *exp*(-1000). |
| TLOSS | Argument would produce function result with total loss of significant digits, such as *sin*(10e70). |

The macros DBL_MAX, DBL_MIN, LDBL_MAX, and LDBL_MIN are defined in float.h.



The source code to the default *_matherr* and *_matherrl* is on the Borland C++ distribution disks.

The UNIX-style *_matherr* and *_matherrl* default behavior (printing a message and terminating) is not ANSI compatible. If you want a UNIX-style version of these routines, use MATHERR.C and MATHERRL.C provided on the Borland C++ distribution disks.

**Return value**

The default return value for *_matherr* and *_matherrl* is 1 if the error is UNDERFLOW or TLOSS, 0 otherwise. *_matherr* and *_matherrl* can also modify *e -> retval*, which propagates back to the original caller.

When *_matherr* and *_matherrl* return 0 (indicating that they were not able to resolve the error), the global variable *errno* is set to 0 and an error message is printed.

When *_matherr* and *_matherrl* return nonzero (indicating that they were able to resolve the error), the global variable *errno* is not set and no messages are printed.

## max — stdlib.h

**Function**

Returns the larger of two values.

**Syntax**

```
(type) max(a, b);
template <class T> T max( T t1, T t2 );
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** The C macro and the C++ template function compare two values and return the larger of the two. Both arguments and the routine declaration must be of the same type.

**Return value** *max* returns the larger of two values.

**See also** *min*

## mblen — stdlib.h

**Function** Determines the length of a multibyte character.

**Syntax**

```
int mblen(const char *s, size_t n);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | ■ | ■ | ■ |

**Remarks** If *s* is not null, *mblen* determines the number of bytes in the multibyte character pointed to by *s*. The maximum number of bytes examined is specified by *n*.

The behavior of *mblen* is affected by the setting of LC_CTYPE category of the current locale.

**Return value** If *s* is null, *mblen* returns a nonzero value if multibyte characters have state-dependent encodings. Otherwise, *mblen* returns 0.

If *s* is not null, *mblen* returns 0 if *s* points to the null character, and –1 if the next *n* bytes do not comprise a valid multibyte character; the number of bytes that comprise a valid multibyte character.

**See also** *mbstowcs, mbtowc, setlocale*

## mbstowcs — stdlib.h

**Function** Converts a multibyte string to a *wchar_t* array.

**Syntax**

```
size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | ■ | ■ | ■ |

**Remarks**

The function converts the multibyte string *s* into the array pointed to by *pwcs*. No more than *n* values are stored in the array. If an invalid multibyte sequence is encountered, *mbstowcs* returns (*size_t*) –1.

The *pwcs* array will not be terminated with a zero value if *mbstowcs* returns *n*.

The behavior of *mbstowcs* is affected by the setting of LC_CTYPE category of the current locale.

**Return value**

If an invalid multibyte sequence is encountered, *mbstowcs* returns (*size_t*) –1. Otherwise, the function returns the number of array elements modified, not including the terminating code, if any.

**See also**

*mblen, mbtowc, setlocale*

## mbtowc — stdlib.h



**Function**

Converts a multibyte character to *wchar_t* code.

**Syntax**

```
int mbtowc(wchar_t *pwc, const char *s, size_t n);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | ■ | ■ | ■ |

**Remarks**

If *s* is not null, *mbtowc* determines the number of bytes that comprise the multibyte character pointed to by *s*. Next, *mbtowc* determines the value of the type wchar_t that corresponds to that multibyte character. If there is a successful match between wchar_t and the multibyte character, and *pwc* is not null, the wchar_t value is stored in the array pointed to by *pwc*. At most *n* characters are examined.

**Return value**

When *s* points to an invalid multibyte character, –1 is returned. When *s* points to the null character, 0 is returned. Otherwise, *mbtowc* returns the number of bytes that comprise the converted multibyte character.

The return value never exceeds MB_CUR_MAX or the value of *n*.

The behavior of *mbtowc* is affected by the setting of LC_CTYPE category of the current locale.

**See also**

*mblen, mbstowcs, setlocale*

## memccpy, _fmemccpy — mem.h

**Function**

Copies a block of *n* bytes.

**Syntax**

```
void *memccpy(void *dest, const void *src, int c, size_t n);
void far * far _fmemccpy(void far *dest, const void far *src, int c, size_t n)
```

| | DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|---|
| *memccpy* | ■ | ■ | ■ | ■ | | | ■ |
| *_fmemccpy* | ■ | | ■ | | | | |

**Remarks**

*memccpy* is available on UNIX System V systems.

*memccpy* copies a block of *n* bytes from *src* to *dest*. The copying stops as soon as either of the following occurs:

- The character *c* is first copied into *dest*.
- *n* bytes have been copied into *dest*.

**Return value**

*memccpy* returns a pointer to the byte in *dest* immediately following *c*, if *c* was copied; otherwise, *memccpy* returns NULL.

**See also**

*memcpy, memmove, memset*

## memchr, _fmemchr — mem.h

**Function**

Searches *n* bytes for character *c*.

**Syntax**

```
void *memchr(const void *s, int c, size_t n);                    /* C only */
void far * far _fmemchr(const void far *s, int c, size_t n);     /* C and C++ */
const void *memchr(const void *s, int c, size_t n);               // C++ only
void *memchr(void *s, int c, size_t n);                           // C++ only
```

| | DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|---|
| *memchr* | ■ | ■ | ■ | ■ | ■ | ■ | ■ |
| *_fmemchr* | ■ | | ■ | | | | |

**Remarks**

*memchr* is available on UNIX System V systems.

*memchr* searches the first *n* bytes of the block pointed to by *s* for character *c*.

**Return value** On success, *memchr* returns a pointer to the first occurrence of *c* in *s*; otherwise, it returns NULL.

➥ If you are using the intrinsic version of these functions, the case of *n*=0 will return NULL.

## memcmp, _fmemcmp — mem.h

**Function** Compares two blocks for a length of exactly *n* bytes.

**Syntax**

```
int memcmp(const void *s1, const void *s2, size_t n);
int far _fmemcmp(const void far *s1, const void far *s2, size_t n)
```

| | DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|---|
| *memcmp* | ■ | ■ | ■ | ■ | ■ | ■ | ■ |
| *_fmemcmp* | ■ | | ■ | | | | |

**Remarks** *memcmp* is available on UNIX System V systems.

*memcmp* compares the first *n* bytes of the blocks *s1* and *s2* as **unsigned chars**.

**Return value** Because it compares bytes as **unsigned chars**, *memcmp* returns a value that is

- < 0 if *s1* is less than *s2*
- = 0 if *s1* is the same as *s2*
- > 0 if *s1* is greater than *s2*

For example,

```
memcmp("\xFF", "\x7F", 1)
```

returns a value greater than 0.

➥ If you are using the intrinsic version of these functions, the case of *n*=0 will return NULL.

**See also** *memicmp*

## memcpy, _fmemcpy — mem.h

**Function** Copies a block of *n* bytes.

**Syntax**

```
void *memcpy(void *dest, const void *src, size_t n);
void far *far _fmemcpy(void far *dest, const void far *src, size_t n);
```

| | DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|---|
| *memcpy* | ■ | ■ | ■ | ■ | ■ | ■ | ■ |
| *_fmemcpy* | ■ | | ■ | | | | |

**Remarks**

*memcpy* is available on UNIX System V systems.

*memcpy* copies a block of *n* bytes from *src* to *dest*. If *src* and *dest* overlap, the behavior of *memcpy* is undefined.

**Return value**

*memcpy* returns *dest*.

**See also**

*memccpy, memmove, memset, movedata, movmem*

## memicmp, _fmemicmp — mem.h

**Function**

Compares *n* bytes of two character arrays, ignoring case.

**Syntax**

```
int memicmp(const void *s1, const void *s2, size_t n);
int far _fmemicmp(const void far *s1, const void far *s2, size_t n)
```

| | DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|---|
| *memicmp* | ■ | ■ | ■ | ■ | | | ■ |
| *_fmemicmp* | ■ | | ■ | | | | |

**Remarks**

*memicmp* is available on UNIX System V systems.

*memicmp* compares the first *n* bytes of the blocks *s1* and *s2*, ignoring character case (upper or lower).

**Return value**

*memicmp* returns a value that is

- < 0 if *s1* is less than *s2*
- = 0 if *s1* is the same as *s2*
- > 0 if *s1* is greater than *s2*

**See also**

*memcmp*

## memmove, _fmemmove — mem.h

**Function**

Copies a block of *n* bytes.

**Syntax**

```
void *memmove(void *dest, const void *src, size_t n);
void far * far _fmemmove (void far *dest, const void far *src, size_t n)
```

| | DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|---|
| *memmove* | ■ | ■ | ■ | ■ | ■ | ■ | ■ |
| *_fmemmove* | ■ | | ■ | | | | |

**Remarks** *memmove* copies a block of *n* bytes from *src* to *dest*. Even when the source and destination blocks overlap, bytes in the overlapping locations are copied correctly.

**Return value** *memmove* returns *dest*.

**See also** *memccpy, memcpy, movmem*

## memset, _fmemset — mem.h



**Function** Sets *n* bytes of a block of memory to byte *c*.

**Syntax**

```
void *memset(void *s, int c, size_t n);
void far * far _fmemset (void far *s, int c, size_t n)
```

| | DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|---|
| *memset* | ■ | ■ | ■ | ■ | ■ | ■ | ■ |
| *_fmemset* | ■ | | ■ | | | | |

**Remarks** *memset* sets the first *n* bytes of the array *s* to the character *c*.

**Return value** *memset* returns *s*.

**See also** *memccpy, memcpy, setmem*

## min — stdlib.h

**Function** Returns the smaller of two values.

**Syntax**

```
(type) min(a, b);
template <class T> T min( T t1, T t2 );
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** The C macro and the C++ template function compare two values and return the smaller of the two. Both arguments and the routine declaration must be of the same type.

**Return value** *min* returns the smaller of two values.

**See also** *max*

## mkdir — dir.h

**Function** Creates a directory.

**Syntax**

```
int mkdir(const char *path);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks** *mkdir* is available on UNIX, though it then takes an additional parameter.

*mkdir* creates a new directory from the given path name *path*.

**Return value** *mkdir* returns the value 0 if the new directory was created.

A return value of –1 indicates an error, and the global variable *errno* is set to one of the following values:

| | |
|---|---|
| EACCES | Permission denied |
| ENOENT | No such file or directory |

**See also** *chdir*, *getcurdir*, *getcwd*, *rmdir*

## MK_FP — dos.h

**Function** Makes a **far** pointer.

**Syntax**

```
void far * MK_FP(unsigned seg, unsigned ofs);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks** *MK_FP* is a macro that makes a **far** pointer from its component segment (*seg*) and offset (*ofs*) parts.

**Return value** *MK_FP* returns a **far** pointer.

**See also** *FP_OFF, FP_SEG, movedata, segread*

## mktemp — dir.h

**Function** Makes a unique file name.

**Syntax**

```
char *mktemp(char *template);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks** *mktemp* replaces the string pointed to by *template* with a unique file name and returns *template*.

*template* should be a null-terminated string with six trailing *X*s. These *X*s are replaced with a unique collection of letters plus a period, so that there are two letters, a period, and three suffix letters in the new file name.

Starting with AA.AAA, the new file name is assigned by looking up the name on the disk and avoiding pre-existing names of the same format.

**Return value** If *template* is well-formed, *mktemp* returns the address of the *template* string. Otherwise, it returns null.

## mktime — time.h

**Function** Converts time to calendar format.

**Syntax**

```
time_t mktime(struct tm *t);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | ■ | ■ | ■ |

**Remarks** Converts the time in the structure pointed to by *t* into a calendar time with the same format used by the *time* function. The original values of the fields *tm_sec, tm_min, tm_hour, tm_mday,* and *tm_mon* are not restricted to the ranges described in the *tm* structure. If the fields are not in their proper ranges, they are adjusted. Values for fields *tm_wday* and *tm_yday* are computed after the other fields have been adjusted. If the calendar time cannot be represented, *mktime* returns –1.

The allowable range of calendar times is Jan 1 1970 00:00:00 to Jan 19 2038 03:14:07.

**Return value** See Remarks.

**See also** *localtime, strftime, time*

## modf, modfl — math.h

**Function** Splits a **double** or **long double** into integer and fractional parts.

**Syntax**

```
double modf(double x, double *ipart);
long double modfl(long double x, long double *ipart);
```

| | DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|---|
| *modf* | ■ | ■ | ■ | ■ | ■ | ■ | ■ |
| *modfl* | ■ | | ■ | ■ | | | ■ |

**Remarks** *modf* breaks the **double** *x* into two parts: the integer and the fraction. *modf* stores the integer in *ipart* and returns the fraction.

*modfl* is the **long double** version; it takes **long double** arguments and returns a **long double** result.

**Return value** *modf* and *modfl* return the fractional part of *x*.

**See also** *fmod, ldexp*

## movedata — mem.h

**Function** Copies *n* bytes.

**Syntax**

```
void movedata(unsigned srcseg, unsigned srcoff, unsigned dstseg, unsigned dstoff,
              size_t n);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks** *movedata* copies *n* bytes from the source address (*srcseg:srcoff*) to the destination address (*dstseg:dstoff*). *movedata* provides a memory-model independent means for moving blocks of data.

**Return value** None.

**See also** *FP_OFF, memcpy, MK_FP,movmem, segread*

## movmem, _fmovmem — mem.h

**Function** Moves a block of *length* bytes.

**Syntax**

```
void movmem(const void *src, void *dest, unsigned length);
void _fmovmem(const void far *src, void far *dest, unsigned length);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks** *movmem* moves a block of *length* bytes from *src* to *dest*. Even if the source and destination blocks overlap, the move direction is chosen so that the data is always moved correctly. *_fmovmem* provides the same functionality in a large memory model as *movmem* does in small memory model.

**Return value** None.

**See also** *memcpy, memmove, movedata*



## movetext — conio.h

**Function** Copies text onscreen from one rectangle to another.

**Syntax**

```
int movetext(int left, int top, int right, int bottom, int destleft, int desttop);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | | ■ | | | ■ |

**Remarks** *movetext* copies the contents of the onscreen rectangle defined by *left, top, right,* and *bottom* to a new rectangle of the same dimensions. The new rectangle's upper left corner is position (*destleft, desttop*).

All coordinates are absolute screen coordinates. Rectangles that overlap are moved correctly.

*movetext* is a text mode function performing direct video output.

➥ This function should not be used in Win32s or Win32 GUI applications.

**Return value** *movetext* returns nonzero if the operation succeeded. If the operation failed (for example, if you gave coordinates outside the range of the current screen mode), *movetext* returns 0.

**See also** *gettext, puttext*

## _msize — malloc.h

**Function** Returns the size of a heap block.

**Syntax**

```
size_t _msize(void *block);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| | | | ■ | | | ■ |

**Remarks** *_msize* returns the size of the allocated heap block whose address is *block*. The block must have been allocated with *malloc*, *calloc*, or *realloc*. The returned size can be larger than the number of bytes originally requested when the block was allocated.

**Return value** *_msize* returns the size of the block in bytes.

**See also** *malloc, free, realloc*

## normvideo — conio.h

**Function** Selects normal-intensity characters.

**Syntax**

```
void normvideo(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | | ■ | | | ■ |

**Remarks** *normvideo* selects normal characters by returning the text attribute (foreground and background) to the value it had when the program started.

This function does not affect any characters currently on the screen, only those displayed by functions (such as *cprintf*) performing direct console output functions after *normvideo* is called.

➥ This function should not be used in Win32s or Win32 GUI applications.

**Return value** None.

**See also** *highvideo, lowvideo, textattr, textcolor*

## offsetof — stddef.h

**Function** Gets the byte offset to a structure member.

**Syntax**

```
size_t offsetof(struct_type, struct_member);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks**

*offsetof* is available only as a macro. The argument *struct_type* is a **struct** type. *struct_member* is any element of the **struct** that can be accessed through the member selection operators or pointers.

If *struct_member* is a bit field, the result is undefined.

See also Chapter 2 in the *Programmer's Guide* for a discussion of the **sizeof** operator, memory allocation, and alignment of structures.

**Return value**

*offsetof* returns the number of bytes from the start of the structure to the start of the named structure member.

## _open — fcntl.h, share.h, dos.h

**Remarks**

Obsolete function. See *_rtl_open*.

## open — fcntl.h, io.h



**Function**

Opens a file for reading or writing.

**Syntax**

```
int open(const char *path, int access [, unsigned mode]);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks**

*open* opens the file specified by *path*, then prepares it for reading and/or writing as determined by the value of *access*.

To create a file in a particular mode, you can either assign to the global variable *_fmode* or call *open* with the O_CREAT and O_TRUNC options ORed with the translation mode desired. For example, the call

```
open("XMP",O_CREAT|O_TRUNC|O_BINARY,S_IREAD)
```

creates a binary-mode, read-only file named XMP, truncating its length to 0 bytes if it already existed.

For *open*, *access* is constructed by bitwise ORing flags from the following two lists. Only one flag from the first list can be used (and one *must* be used); the remaining flags can be used in any logical combination.

These symbolic constants are defined in fcntl.h.

**List 1: Read/write flags**

| | |
|---|---|
| O_RDONLY | Open for reading only. |
| O_WRONLY | Open for writing only. |
| O_RDWR | Open for reading and writing. |

**List 2: Other access flags**

| | |
|---|---|
| O_NDELAY | Not used; for UNIX compatibility. |
| O_APPEND | If set, the file pointer will be set to the end of the file prior to each write. |
| O_CREAT | If the file exists, this flag has no effect. If the file does not exist, the file is created, and the bits of *mode* are used to set the file attribute bits as in *chmod*. |
| O_TRUNC | If the file exists, its length is truncated to 0. The file attributes remain unchanged. |
| O_EXCL | Used only with O_CREAT. If the file already exists, an error is returned. |
| O_BINARY | Can be given to explicitly open the file in binary mode. |
| O_TEXT | Can be given to explicitly open the file in text mode. |

If neither O_BINARY nor O_TEXT is given, the file is opened in the translation mode set by the global variable *_fmode*.

If the O_CREAT flag is used in constructing *access*, you need to supply the *mode* argument to *open* from the following symbolic constants defined in sys\stat.h.

| Value of *mode* | Access permission |
|---|---|
| S_IWRITE | Permission to write |
| S_IREAD | Permission to read |
| S_IREADIS_IWRITE | Permission to read and write |

**Return value**

On successful completion, *open* returns a nonnegative integer (the file handle). The file pointer, which marks the current position in the file, is set to the beginning of the file. On error, *open* returns –1 and the global variable *errno* is set to one of the following values:

| | |
|---|---|
| EACCES | Permission denied |
| EINVACC | Invalid access code |
| EMFILE | Too many open files |
| ENOENT | No such file or directory |

**See also** *chmod, chsize, close, _rtl_creat, creat, creatnew, creattemp, dup, dup2, fdopen, filelength, fopen, freopen, getftime, lseek, lock, _rtl_open, read, sopen, _rtl_write, write*

## opendir — dirent.h

**Function** Opens a directory stream for reading.

**Syntax**

```
DIR *opendir(char *dirname);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks** *opendir* is available on POSIX-compliant UNIX systems.

The *opendir* function opens a directory stream for reading. The name of the directory to read is *dirname*. The stream is set to read the first entry in the directory.

A directory stream is represented by the *DIR* structure, defined in dirent.h. This structure contains no user-accessible fields. Multiple directory streams can be opened and read simultaneously. Directory entries can be created or deleted while a directory stream is being read.

Use the *readdir* function to read successive entries from a directory stream. Use the *closedir* function to remove a directory stream when it is no longer needed.

**Return value** If successful, *opendir* returns a pointer to a directory stream that can be used in calls to *readdir, rewinddir,* and *closedir*. If the directory cannot be opened, *opendir* returns NULL and sets the global variable *errno* to

| | |
|---|---|
| ENOENT | The directory does not exist |
| ENOMEM | Not enough memory to allocate a DIR object |

**See also** *closedir, readdir, rewinddir*

## outp — conio.h

**Function** Outputs a byte to a hardware port.

**Syntax**

```
int outp(unsigned portid, int value);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

N-P

**Remarks** *outp* is a macro that writes the low byte of *value* to the output port specified by *portid*.

If *outp* is called when conio.h has been included, it will be treated as a macro that expands to inline code. If you don't include conio.h, or if you do include conio.h and **#undef** the macro *outp*, you'll get the *outp* function.

**Return value** *outp* returns *value*.

**See also** *inp, inpw, outpw*

## outport, outportb — dos.h

**Function** Outputs a word or byte to a hardware port.

**Syntax**

```
void outport(int portid, int value);
void outportb(int portid, unsigned char value);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks** *outport* works just like the 80x86 instruction *OUT*. It writes the low byte of the word given by *value* to the output port specified by *portid* and writes the high byte of the word to *portid* +1.

*outportb* is a macro that writes the byte given by *value* to the output port specified by *portid*.

If *outportb* is called when dos.h has been included, it will be treated as a macro that expands to inline code. If you don't include dos.h, or if you do include dos.h and **#undef** the macro *outportb*, you'll get the *outportb* function.

**Return value** None.

**See also** *inport, inportb*

## outpw — conio.h

**Function** Outputs a word to a hardware port.

**Syntax**

```
unsigned outpw(unsigned portid, unsigned value);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks** *outpw* is a macro that writes the 16-bit word given by *value* to the output port specified by *portid*. It writes the low byte of *value* to *portid*, and the high byte of the word to *portid* +1, using a single 16-bit *OUT* instruction.

If *outpw* is called when conio.h has been included, it will be treated as a macro that expands to inline code. If you don't include conio.h, or if you do include conio.h and **#undef** the macro *outpw*, you'll get the *outpw* function.

**Return value** *outpw* returns *value*.

**See also** *inp, inpw, outp*

## parsfnm — dos.h

**Function** Parses file name.

**Syntax**

```
char *parsfnm(const char *cmdline, struct fcb *fcb, int opt);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks** *parsfnm* parses a string pointed to by *cmdline* for a file name. The string is normally a command line. The file name is placed in a file control block (FCB) as a drive, file name, and extension. The FCB is pointed to by *fcb*.

The *opt* parameter is the value documented for AL in the DOS parse system call. See your DOS reference manuals under system call 0x29 for a description of the parsing operations performed on the file name.

**Return value** On success, *parsfnm* returns a pointer to the next byte after the end of the file name. If there is any error in parsing the file name, *parsfnm* returns null.

## _pclose — stdio.h

**Function** Waits for piped command to complete.

**Syntax**

```
int _pclose(FILE * stream);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| | | | ■ | | | ■ |

**Remarks** This function is not available in Win32s programs.

*_pclose* closes a pipe stream created by a previous call to *_popen*, and then waits for the associated child command to complete.

**Return value** If it is successful, *_pclose* returns the termination status of the child command. This is the same value as the termination status returned by *cwait*, except that the high and low order bytes of the low word are swapped. If *_pclose* is unsuccessful, it returns –1.

**See also** *_pipe, _popen*

## peek — dos.h

**Function** Returns the word at memory location specified by *segment:offset*.

**Syntax**

```
int peek(unsigned segment, unsigned offset);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks** *peek* returns the word at the memory location *segment:offset*.

If *peek* is called when dos.h has been included, it is treated as a macro that expands to inline code. If you don't include dos.h, or if you do include it and **#undef** *peek*, you'll get the function rather than the macro.

**Return value** *peek* returns the word of data stored at the memory location *segment:offset*.

**See also** *peekb, poke*

## peekb — dos.h

**Function** Returns the byte of memory specified by *segment:offset*.

**Syntax**

```
char peekb(unsigned segment, unsigned offset);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks** *peekb* returns the byte at the memory location addressed by *segment:offset*.

If *peekb* is called when dos.h has been included, it is treated as a macro that expands to inline code. If you don't include dos.h, or if you do include it and **#undef** peekb, you'll get the function rather than the macro.

**Return value** *peekb* returns the byte of information stored at the memory location *segment:offset*.

**See also** *peek, pokeb*

# perror stdio.h

**Function**

Prints a system error message.

**Syntax**

```
void perror(const char *s);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | | ■ | ■ | ■ | ■ |

**Remarks**

*perror* prints to the *stderr* stream (normally the console) the system error message for the last library routine that set *errno*.

First the argument *s* is printed, then a colon, then the message corresponding to the current value of the global variable *errno*, and finally a newline. The convention is to pass the file name of the program as the argument string.

The array of error message strings is accessed through the global variable *_sys_errlist*. The global variable *errno* can be used as an index into the array to find the string corresponding to the error number. None of the strings include a newline character.

The global variable *_sys_nerr* contains the number of entries in the array.

Refer to *errno*, *_sys_errlist*, and *_sys_nerr* in Chapter 4 for more information.

The following messages are generated by *perror*:

Table 3.2 These messages are generated in both Win 16 and Win 32.

| Win 16 and Win 32 messages | |
|---|---|
| Arg list too big | Is a directory |
| Attempted to remove current directory | Math argument |
| Bad address | Memory arena trashed |
| Bad file number | Name too long |
| Block device required | No child processes |
| Broken pipe | No more files |
| Cross-device link | No space left on device |
| Error 0 | No such device |
| Exec format error | No such device or address |
| Executable file in use | No such file or directory |
| File already exists | No such process |
| File too large | Not a directory |
| Illegal seek | Not enough memory |
| Inappropriate I/O control operation | Not same device |
| Input/output error | Operation not permitted |
| Interrupted function call | Path not found |
| | Permission denied |
| | Possible deadlock |

Table 3.2: These messages are generated in both Win 16 and Win 32. (continued)

| | |
|---|---|
| Invalid access code | Read-only file system |
| Invalid argument | Resource busy |
| Invalid data | Resource temporarily unavailable |
| Invalid environment | Result too large |
| Invalid format | Too many links |
| Invalid function number | Too many open files |
| Invalid memory block address | |

Table 3.3 These messages are generated only in Win 32.

| **Win 32-only messages** | |
|---|---|
| Bad address | No child processes |
| Block device required | No space left on device |
| Broken pipe | No such device or address |
| Executable file in use | No such process |
| File too large | Not a directory |
| Illegal seek | Operation not permitted |
| Inappropriate I/O control operation | Possible deadlock |
| Input/output error | Read-only file system |
| Interrupted function call | Resource busy |
| Is a directory | Resource temporarily unavailable |
| Name too long | Too many links |

➥ For Win32s or Win32 GUI applications, stderr must be redirected.

**Return value** None.

**See also** *clearerr, eof, freopen, _strerror, strerror*

## _pipe — fcntl.h, io.h

**Function** Creates a read/write pipe.

**Syntax**

```
int _pipe(int *handles, unsigned int size, int mode);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| | | | ■ | | | ■ |

**Remarks** This function is not available in Win32s programs.

The *_pipe* function creates an anonymous pipe that can be used to pass information between processes. The pipe is opened for both reading and writing. Like a disk file, a pipe can be read from and written to, but it does not have a name or permanent storage associated with it; data written to

and from the pipe exist only in a memory buffer managed by the operating system.

The read handle is returned to *handles*[0], and the write handle is returned to *handles*[1]. The program can use these handles in subsequent calls to *read, write, dup, dup2,* or *close.* When all pipe handles are closed, the pipe is destroyed.

The size of the internal pipe buffer is *size.* A recommended minimum value is 512 bytes.

The translation mode is specified by *mode,* as follows:

| | |
|---|---|
| O_BINARY | The pipe is opened in binary mode |
| O_TEXT | The pipe is opened in text mode |

If *mode* is zero, the translation mode is determined by the external variable *_fmode.*

**Return value**

On successful completion, *_pipe* returns 0 and returns the pipe handles to *handles*[0] and *handles*[1]. Otherwise it returns –1 and sets *errno* to one of the following values:

| | |
|---|---|
| EMFILE | Too many open files |
| ENOMEM | Out of memory |

**See also**

*_pclose, _popen*



## poke — dos.h

**Function**

Stores an integer value at a memory location given by *segment:offset.*

**Syntax**

```
void poke(unsigned segment, unsigned offset, int value);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks**

*poke* stores the integer *value* at the memory location *segment:offset.*

If this routine is called when dos.h has been included, it will be treated as a macro that expands to inline code. If you don't include dos.h, or if you do include it and **#undef** *poke,* you'll get the function rather than the macro.

**Return value**

None.

**See also**

*peek, pokeb*

## pokeb — dos.h

**Function** Stores a byte value at memory location *segment:offset*.

**Syntax**

```
void pokeb(unsigned segment, unsigned offset, char value);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks** *pokeb* stores the byte *value* at the memory location *segment:offset*.

If this routine is called when dos.h has been included, it will be treated as a macro that expands to inline code. If you don't include dos.h, or if you do include it and **#undef** *pokeb*, you'll get the function rather than the macro.

**Return value** None.

**See also** *peekb, poke*

## poly, polyl — math.h

**Function** Generates a polynomial from arguments.

**Syntax**

```
double poly(double x, int degree, double coeffs[]);
long double polyl(long double x, int degree, long double coeffs[]);
```

| | DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|---|
| *poly* | ■ | ■ | ■ | ■ | | | ■ |
| *polyl* | ■ | | ■ | ■ | | | ■ |

**Remarks** *poly* generates a polynomial in $x$, of degree *degree*, with coefficients *coeffs[0], coeffs[1], ..., coeffs[degree]*. For example, if $n = 4$, the generated polynomial is

$$coeffs[4]x^4 + coeffs[3]x^3 + coeffs[2]x^2 + coeffs[1]x + coeffs[0]$$

*polyl* is the **long double** version; it takes **long double** arguments and returns a **long double** result.

**Return value** *poly* and *polyl* return the value of the polynomial as evaluated for the given $x$.

## _popen — stdio.h

**Function** Creates a command processor pipe.

**Syntax**

```
FILE *_popen (const char *command, const char *mode);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| | | | ■ | | | ■ |

**Remarks**

This function is not available in Win32s programs.

The *_popen* function creates a pipe to the command processor. The command processor is executed asynchronously, and is passed the command line in *command*. The *mode* string specifies whether the pipe is connected to the command processor's standard input or output, and whether the pipe is to be opened in binary or text mode.

The *mode* string can take one of the following values:

| Value | Description |
|---|---|
| *rt* | Read child command's standard output (text). |
| *rb* | Read child command's standard output (binary). |
| *wt* | Write to child command's standard input (text). |
| *wb* | Write to child command's standard input (binary). |

The terminating *t* or *b* is optional; if missing, the translation mode is determined by the external variable *_fmode*.

Use the *_pclose* function to close the pipe and obtain the return code of the command.

**Return value**

If *_popen* is successful it returns a FILE pointer that can be used to read the standard output of the command, or to write to the standard input of the command, depending on the *mode* string. If *_popen* is unsuccessful, it returns NULL.

**See also**

*_pclose*, *_pipe*

## pow, powl — math.h

**Function**

Calculates $x$ to the power of $y$.

**Syntax**

```
double pow(double x, double y);
long double powl(long double x, double y);
```

| | DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|---|
| *pow* | ■ | ■ | ■ | ■ | ■ | ■ | ■ |
| *powl* | ■ | | ■ | ■ | | | ■ |

**Remarks**

*pow* calculates $x^y$.

*powl* is the **long double** version; it takes **long double** arguments and returns a **long double** result.

This function can be used with *bcd* and *complex* types.

**Return value**

On success, *pow* and *powl* return the value calculated, $x^y$.

Sometimes the arguments passed to these functions produce results that overflow or are incalculable. When the correct value would overflow, the functions return the value HUGE_VAL (*pow*) or _LHUGE_VAL (*powl*). Results of excessively large magnitude can cause the global variable *errno* to be set to

ERANGE Result out of range

If the argument $x$ passed to *pow* or *powl* is real and less than 0, and $y$ is not a whole number, or you call *pow( 0, 0 )*, the global variable *errno* is set to

EDOM Domain error

Error handling for these functions can be modified through the functions *_matherr* and *_matherrl*.

**See also**

*bcd, complex, exp, pow10, sqrt*

## pow10, pow10l — math.h

**Function**

Calculates 10 to the power of $p$.

**Syntax**

```
double pow10(int p);
long double pow10l(int p);
```

| | DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|---|
| *pow10* | ■ | ■ | ■ | ■ | | | ■ |
| *pow10l* | ■ | | ■ | ■ | | | ■ |

**Remarks**

*pow10* computes $10^p$.

**Return value**

On success, *pow10* returns the value calculated, $10^p$.

The result is actually calculated to **long double** accuracy. All arguments are valid, although some can cause an underflow or overflow.

*powl* is the **long double** version; it returns a **long double** result.

**See also**

*exp*, *pow*

## printf — stdio.h

**Function**

Writes formatted output to stdout.

**Syntax**

```
int printf(const char *format[, argument, ...]);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | | ■ | ■ | ■ | ■ |

**Remarks**

*printf* accepts a series of arguments, applies to each a format specifier contained in the format string given by *format*, and outputs the formatted data to *stdout*. There must be the same number of format specifiers as arguments.

➥ For Win32s or Win32 GUI applications, stdout must be redirected.

**The format string**

The format string, present in each of the *...printf* function calls, controls how each function will convert, format, and print its arguments. *There must be enough arguments for the format; if not, the results will be unpredictable and possibly disastrous.* Excess arguments (more than required by the format) are ignored.

The format string is a character string that contains two types of objects—*plain characters* and *conversion specifications*:

- Plain characters are copied verbatim to the output stream.
- Conversion specifications fetch arguments from the argument list and apply formatting to them.

***Format specifiers***

*...printf* format specifiers have the following form:

```
% [flags] [width] [.prec] [F|N|h|l|L] type
```

Each format specifier begins with the percent character (%). After the % come the following, in this order:

- An optional sequence of flag characters, `[flags]`
- An optional width specifier, `[width]`
- An optional precision specifier, `[.prec]`
- An optional input-size modifier, `[F|N|h|l|L]`
- The conversion-type character, `[type]`

**Optional format string components**

These are the general aspects of output formatting controlled by the optional characters, specifiers, and modifiers in the format string:

| Character or specifier | What it controls or specifies |
|---|---|
| Flags | Output justification, numeric signs, decimal points, trailing zeros, octal and hex prefixes |
| Width | Minimum number of characters to print, padding with blanks or zeros |
| Precision | Maximum number of characters to print; for integers, minimum number of digits to print |
| Size | Override default size of argument:<br>**N** = **near** pointer<br>**F** = **far** pointer<br>**h** = **short int**<br>**l** = **long**<br>**L** = **long double** |

**...printf conversion-type characters**

The following table lists the *...printf* conversion-type characters, the type of input argument accepted by each, and in what format the output appears.

The information in this table of type characters is based on the assumption that no flag characters, width specifiers, precision specifiers, or input-size modifiers were included in the format specifiers. To see how the addition of the optional characters and specifiers affects the *...printf* output, refer to the tables following this one.

| Type character | Input argument | Format of output |
|---|---|---|
| ***Numerics*** | | |
| **d** | Integer | **signed** decimal **int**. |
| **i** | Integer | **signed** decimal **int**. |
| **o** | Integer | **unsigned** octal **int**. |
| **u** | Integer | **unsigned** decimal **int**. |

| | | |
|---|---|---|
| **x** | Integer | **unsigned** hexadecimal **int** (with **a**, **b**, **c**, **d**, **e**, **f**). |
| **X** | Integer | **unsigned** hexadecimal **int** (with **A**, **B**, **C**, **D**, **E**, **F**). |
| **f** | Floating-point | **signed** value of the form [-]*dddd.dddd*. |
| **e** | Floating-point | **signed** value of the form [-]*d.dddd* or **e** [*+/-*]*ddd*. |
| **g** | Floating-point | **signed** value in either **e** or **f** form, based on given value and precision. Trailing zeros and the decimal point are printed only if necessary. |
| **E** | Floating-point | Same as **e**, but with **E** for exponent. |
| **G** | Floating-point | Same as **g**, but with **E** for exponent if **e** format used. |
| ***Characters*** | | |
| **c** | Character | Single character. |
| **s** | String pointer | Prints characters until a null-terminator is pressed or precision is reached. |
| **%** | None | The **%** character is printed. |
| ***Pointers*** | | |
| **n** | Pointer to **int** | Stores (in the location pointed to by the input argument) a count of the characters written so far. |
| **p** | Pointer | Prints the input argument as a pointer; format depends on which memory model was used. It will be either XXXX:YYYY or YYYY (offset only). |



**Conventions** Certain conventions accompany some of these specifications. The decimal-point character used in the output is determined by the current locale's LC_NUMERIC category. The conventions are summarized in the following table:

| Characters | Conventions |
|---|---|
| **e** or **E** | The argument is converted to match the style [-] *d.ddd...***e**[*+/-*]*ddd*, where<br>■ One digit precedes the decimal point.<br>■ The number of digits after the decimal point is equal to the precision.<br>■ The exponent always contains at least two digits. |
| **f** | The argument is converted to decimal notation in the style [-] *ddd.ddd...*, where the number of digits after the decimal point is equal to the precision (if a nonzero precision was given). |
| **g** or **G** | The argument is printed in style **e**, **E** or **f**, with the precision specifying the number of significant digits. Trailing zeros are removed from the result, and a decimal point appears only if necessary. |

| Characters | Conventions |
|---|---|
| | The argument is printed in style **e** or **f** (with some restraints) if **g** is the conversion character, and in style **E** if the character is **G**. Style **e** is used only if the exponent that results from the conversion is either greater than the precision or less than –4. |
| **x** or **X** | For **x** conversions, the letters **a**, **b**, **c**, **d**, **e**, and **f** appear in the output; for **X** conversions, the letters **A**, **B**, **C**, **D**, **E**, and **F** appear. |

➥ Infinite floating-point numbers are printed as +INF and –INF. An IEEE Not-a-Number is printed as +NAN or –NAN.

**Flag characters**

The flag characters are minus (-), plus (+), sharp (#), and blank (). They can appear in any order and combination.

| Flag | What it specifies |
|---|---|
| **–** | Left-justifies the result, pads on the right with blanks. If not given, it right-justifies the result, pads on the left with zeros or blanks. |
| **+** | Signed conversion results always begin with a plus (+) or minus (-) sign. |
| **blank** | If value is nonnegative, the output begins with a blank instead of a plus; negative values still begin with a minus. |
| # | Specifies that *arg* is to be converted using an "alternate form." See the following table. |

➥ Plus (+) takes precedence over blank () if both are given.

**Alternate forms**

If the # flag is used with a conversion character, it has the following effect on the argument (*arg*) being converted:

| Conversion character | How # affects *arg* |
|---|---|
| **c**,**s**,**d**,**i**,**u** | No effect. |
| **0** | 0 is prepended to a nonzero *arg*. |
| **x** or **X** | 0x (or 0X) is prepended to *arg*. |
| **e**, **E**, or **f** | The result always contains a decimal point even if no digits follow the point. Normally, a decimal point appears in these results only if a digit follows it. |
| **g** or **G** | Same as **e** and **E**, with the addition that trailing zeros are not removed. |

**Width specifiers**

The width specifier sets the minimum field width for an output value.

Width is specified in one of two ways: directly, through a decimal digit string, or indirectly, through an asterisk (*). If you use an asterisk for the width specifier, the next argument in the call (which must be an **int**) specifies the minimum output field width.

In no case does a nonexistent or small field width cause truncation of a field. If the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result.

| Width specifier | How output width is affected |
|---|---|
| *n* | At least *n* characters are printed. If the output value has less than *n* characters, the output is padded with blanks (right-padded if – flag given, left-padded otherwise). |
| 0*n* | At least *n* characters are printed. If the output value has less than *n* characters, it is filled on the left with zeros. |
| * | The argument list supplies the width specifier, which must precede the actual argument being formatted. |

**Precision specifiers**

A precision specification always begins with a period (.) to separate it from any preceding width specifier. Then, like width, precision is specified either directly through a decimal digit string, or indirectly through an asterisk (*). If you use an asterisk for the precision specifier, the next argument in the call (treated as an **int**) specifies the precision.

If you use asterisks for the width or the precision, or for both, the width argument must immediately follow the specifiers, followed by the precision argument, then the argument for the data to be converted.



| Precision specifier | How output precision is affected |
|---|---|
| (none given) | Precision set to default:<br>default = 1 for *d, i, o, u, x, X* types<br>default = 6 for *e, E, f* types<br>default = all significant digits for *g, G* types<br>default = print to first null character for *s* types; no effect on *c* types |
| .0 | For *d, i, o, u, x* types, precision set to default; for *e, E, f* types, no decimal point is printed. |
| .*n* | *n* characters or *n* decimal places are printed. If the output value has more than *n* characters, the output might be truncated or rounded. (Whether this happens depends on the type character.) |
| .* | The argument list supplies the precision specifier, which must precede the actual argument being formatted. |

If an explicit precision of zero is specified, *and* the format specifier for the field is one of the integer formats (that is, *d, i, o, u, x*), *and* the value to be printed is 0, no numeric characters will be output for that field (that is, the field will be blank).

| Conversion character | How precision specification (.n) affects conversion |
|---|---|
| **d**<br>**i**<br>**o**<br>**u**<br>**x**<br>**X** | *.n* specifies that at least *n* digits are printed. If the input argument has less than *n* digits, the output value is left-padded with zeros. If the input argument has more than *n* digits, the output value is not truncated. |
| **e**<br>**E**<br>**f** | *.n* specifies that *n* characters are printed after the decimal point, and the last digit printed is rounded. |
| **g**<br>**G** | *.n* specifies that at most *n* significant digits are printed. |
| **c** | *.n* has no effect on the output. |
| **s** | *.n* specifies that no more than *n* characters are printed. |

**Input-size modifier**

The input-size modifier character (*F, N, h, l,* or *L*) gives the size of the subsequent input argument:

*F* = **far** pointer
*N* = **near** pointer
*h* = **short int**
*l* = **long**
*L* = **long double**

The input-size modifiers (*F, N, h, l,* and *L*) affect how the *...printf* functions interpret the data type of the corresponding input argument *arg*. *F* and *N* apply only to input *args* that are pointers (*%p*, *%s*, and *%n*). *h, L,* and *L* apply to input *args* that are numeric (integers and floating-point).

Both *F* and *N* reinterpret the input *arg*. Normally, the *arg* for a *%p*, *%s*, or *%n* conversion is a pointer of the default size for the memory model. *F* means "interpret *arg* as a **far** pointer." *N* means "interpret *arg* as a **near** pointer."

*h, l,* and *L* override the default size of the numeric data input arguments: *l* and *L* apply to integer (*d, i, o, u, x, X*) and floating-point (*e, E, f, g,* and *G*) types, while *h* applies to integer types only. Neither *h* nor *l* affect character (*c, s*) or pointer (*p, n*) types.

| Input-size modifier | How *arg* is interpreted |
|---|---|
| *F* | *arg* is read as a **far** pointer. |
| *N* | *arg* is read as a **near** pointer. *N* cannot be used with any conversion in **huge** model. |
| *h* | *arg* is interpreted as a **short int** for *d, i, o, u, x,* or *X.* |
| *l* | *arg* is interpreted as a **long int** for *d, i, o, u, x,* or *X*; *arg* is interpreted as a **double** for *e, E, f, g,* or *G.* |
| *L* | *arg* is interpreted as a **long double** for *e, E, f, g,* or *G.* |

**Return value** *printf* returns the number of bytes output. In the event of error, *printf* returns EOF.

**See also** *cprintf, ecvt, fprintf, fread, freopen, fscanf, putc, puts, putw, scanf, sprintf, vprintf, vsprintf*

## putc — stdio.h

**Function** Outputs a character to a stream.

**Syntax**

```
int putc(int c, FILE *stream);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *putc* is a macro that outputs the character *c* to the stream given by *stream.*

**Return value** On success, *putc* returns the character printed, *c*. On error, *putc* returns EOF.

**See also** *fprintf, fputc, fputchar, fputs, fwrite, getc, getchar, printf, putch, putchar, putw, vprintf*

## putch — conio.h

**Function** Outputs character to screen.

**Syntax**

```
int putch(int c);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks**

*putch* outputs the character *c* to the current text window. It is a text mode function performing direct video output to the console. *putch* does not translate linefeed characters (\n) into carriage-return/linefeed pairs.

The string is written either directly to screen memory or by way of a BIOS call, depending on the value of the global variable *directvideo*.

➥ This function should not be used in Win32s or Win32 GUI applications.

**Return value**

On success, *putch* returns the character printed, *c*. On error, it returns EOF.

**See also**

*cprintf, cputs, getch, getche, putc, putchar*

## putchar — stdio.h

**Function**

Outputs character on stdout.

**Syntax**

```
int putchar(int c);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks**

*putchar(c)* is a macro defined to be *putc(c, stdout)*.

➥ For Win32s or Win32 GUI applications, stdout must be redirected.

**Return value**

On success, *putchar* returns the character *c*. On error, *putchar* returns EOF.

**See also**

*fputchar, getc, getchar, printf, putc, putch, puts, putw, freopen, vprintf*

## putenv — stdlib.h

**Function**

Adds string to current environment.

**Syntax**

```
int putenv(const char *name);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks**

*putenv* accepts the string *name* and adds it to the environment of the *current* process. For example,

```
putenv("PATH=C:\\BC");
```

*putenv* can also be used to modify an existing *name*. On DOS and OS/2, *name* must be uppercase. On other systems, *name* can be either uppercase or

lowercase. *name* must not include the equal sign (=). You can set a variable to an empty value by specifying an empty string on the right side of the '=' sign. This effectively removes the environment variable. Environment variables created by *putenv* can be lower or upper case.

*putenv* can be used only to modify the current program's environment. Once the program ends, the old environment is restored. The environment of the current process is passed to child processes, including any changes made by *putenv*.

Note that the string given to *putenv* must be static or global. Unpredictable results will occur if a local or dynamic string given to *putenv* is used after the string memory is released.

**Return value** On success, *putenv* returns 0; on failure, –1.

**See also** *getenv*

## puts — stdio.h

**Function** Outputs a string to stdout.

**Syntax**

```
int puts(const char *s);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | | ■ |

**Remarks** *puts* copies the null-terminated string *s* to the standard output stream stdout and appends a newline character.

➥ For Win32s or Win32 GUI applications, stdout must be redirected.

**Return value** On successful completion, *puts* returns a nonnegative value. Otherwise, it returns a value of EOF.

**See also** *cputs, fputs, gets, printf, putchar, freopen*

## puttext — conio.h

**Function** Copies text from memory to the text mode screen.

**Syntax**

```
int puttext(int left, int top, int right, int bottom, void *source);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | | ■ | | | ■ |

**Remarks**

*puttext* writes the contents of the memory area pointed to by *source* out to the onscreen rectangle defined by *left*, *top*, *right*, and *bottom*.

All coordinates are absolute screen coordinates, not window-relative. The upper left corner is (1,1).

*puttext* places the contents of a memory area into the defined rectangle sequentially from left to right and top to bottom.

Each position onscreen takes 2 bytes of memory: The first byte is the character in the cell, and the second is the cell's video attribute. The space required for a rectangle *w* columns wide by *h* rows high is defined as

$$bytes = (h \text{ rows}) \times (w \text{ columns}) \times 2$$

*puttext* is a text mode function performing direct video output.

This function should not be used in Win32s or Win32 GUI applications.

**Return value**

*puttext* returns a nonzero value if the operation succeeds; it returns 0 if it fails (for example, if you gave coordinates outside the range of the current screen mode).

**See also**

*gettext*, *movetext*, *window*

## putw — stdio.h

**Function**

Puts an integer on a stream.

**Syntax**

```
int putw(int w, FILE *stream);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks**

*putw* outputs the integer *w* to the given stream. *putw* neither expects nor causes special alignment in the file.

**Return value**

On success, *putw* returns the integer *w*. On error, *putw* returns EOF. Because EOF is a legitimate integer, use *ferror* to detect errors with *putw*.

**See also**

*getw*, *printf*

## qsort — stdlib.h

**Function**

Sorts using the quicksort algorithm.

**Syntax**

```
void qsort(void *base, size_t nelem, size_t width,
           int (_USERENTRY *fcmp)(const void *, const void *));
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks**

*qsort* is an implementation of the "median of three" variant of the quicksort algorithm. *qsort* sorts the entries in a table by repeatedly calling the user-defined comparison function pointed to by *fcmp*.

- *base* points to the base (0th element) of the table to be sorted.
- *nelem* is the number of entries in the table.
- *width* is the size of each entry in the table, in bytes.

*fcmp*, the comparison function, must be used with the _USERENTRY calling convention.

*fcmp* accepts two arguments, *elem1* and *elem2*, each a pointer to an entry in the table. The comparison function compares each of the pointed-to items (**elem1* and **elem2*), and returns an integer based on the result of the comparison.

- **elem1* < **elem2* — *fcmp* returns an integer < 0
- **elem1* == **elem2* — *fcmp* returns 0
- **elem1* > **elem2* — *fcmp* returns an integer > 0

In the comparison, the less-than symbol (<) means the left element should appear before the right element in the final, sorted sequence. Similarly, the greater-than (>) symbol means the left element should appear after the right element in the final, sorted sequence.

**Return value**

None.

**See also**

*bsearch, lsearch*



## raise — signal.h

**Function**

Sends a software signal to the executing program.

**Syntax**

```
int raise(int sig);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks**

*raise* sends a signal of type *sig* to the program. If the program has installed a signal handler for the signal type specified by *sig*, that handler will be executed. If no handler has been installed, the default action for that signal type will be taken.

The signal types currently defined in signal.h are noted here:

| Signal | Description |
|---|---|
| SIGABRT | Abnormal termination |
| SIGFPE | Bad floating-point operation |
| SIGILL | Illegal instruction |
| SIGINT | *Ctrl-C* interrupt |
| SIGSEGV | Invalid access to storage |
| SIGTERM | Request for program termination |
| SIGUSR1 | User-defined signal |
| SIGUSR2 | User-defined signal |
| SIGUSR3 | User-defined signal |
| SIGBREAK | *Ctrl-Break* interrupt |

SIGABRT isn't generated by Borland C++ during normal operation. However, it can be generated by *abort*, *raise*, or unhandled exceptions.

**Return value**

*raise* returns 0 if successful, nonzero otherwise.

**See also**

*abort*, *signal*

## rand — stdlib.h

**Function**

Random number generator.

**Syntax**

```
int rand(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks**

*rand* uses a multiplicative congruential random number generator with period $2^{32}$ to return successive pseudorandom numbers in the range from 0 to RAND_MAX. The symbolic constant RAND_MAX is defined in stdlib.h.

**Return value**

*rand* returns the generated pseudorandom number.

**See also**

*random*, *randomize*, *srand*

## random — stdlib.h

**Function** Random number generator.

**Syntax**

```
int random(int num);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *random* returns a random number between 0 and (*num*-1). *random*(*num*) is a macro defined in stdlib.h. Both *num* and the random number returned are integers.

**Return value** *random* returns a number between 0 and (*num*-1).

**See also** *rand, randomize, srand*

## randomize — stdlib.h, time.h

**Function** Initializes random number generator.

**Syntax**

```
void randomize(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *randomize* initializes the random number generator with a random value.

**Return value** None.

**See also** *rand, random, srand*



## _read — io.h, dos.h

**Remarks** Obsolete function. See *_rtl_read*.

## read — io.h

**Function** Reads from file.

**Syntax**

```
int read(int handle, void *buf, unsigned len);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks**

*read* attempts to read *len* bytes from the file associated with *handle* into the buffer pointed to by *buf*.

For a file opened in text mode, *read* removes carriage returns and reports end-of-file when it reaches a *Ctrl-Z*.

The file handle *handle* is obtained from a *creat*, *open*, *dup*, or *dup2* call.

On disk files, *read* begins reading at the current file pointer. When the reading is complete, it increments the file pointer by the number of bytes read. On devices, the bytes are read directly from the device.

The maximum number of bytes that *read* can read is UINT_MAX –1, because UINT_MAX is the same as –1, the error return indicator. UINT_MAX is defined in limits.h.

**Return value**

On successful completion, *read* returns an integer indicating the number of bytes placed in the buffer. If the file was opened in text mode, *read* does not count carriage returns or *Ctrl-Z* characters in the number of bytes read.

On end-of-file, *read* returns 0. On error, *read* returns –1 and sets the global variable *errno* to one of the following values:

| | |
|---|---|
| EACCES | Permission denied |
| EBADF | Bad file number |

**See also**

*open*, *_rtl_read*, *write*

## readdir — dirent.h

**Function**

Reads the current entry from a directory stream.

**Syntax**

```
struct dirent *readdir(DIR *dirp);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks**

*readdir* is available on POSIX-compliant UNIX systems.

The *readdir* function reads the current directory entry in the directory stream pointed to by *dirp*. The directory stream is advanced to the next entry.

The *readdir* function returns a pointer to a **dirent** structure that is overwritten by each call to the function on the same directory stream. The structure is not overwritten by a *readdir* call on a different directory stream.

The **dirent** structure corresponds to a single directory entry. It is defined in dirent.h, and contains (in addition to other non-accessible members) the following member:

```
char  d_name[];
```

where *d_name* is an array of characters containing the null-terminated file name for the current directory entry. The size of the array is indeterminate; use *strlen* to determine the length of the file name.

All valid directory entries are returned, including subdirectories, "." and ".." entries, system files, hidden files, and volume labels. Unused or deleted directory entries are skipped.

A directory entry can be created or deleted while a directory stream is being read, but *readdir* might or might not return the affected directory entry. Rewinding the directory with *rewinddir* or reopening it with *opendir* ensures that *readdir* will reflect the current state of the directory.

**Return value**

If successful, *readdir* returns a pointer to the current directory entry for the directory stream. If the end of the directory has been reached, or *dirp* does not refer to an open directory stream, *readdir* returns NULL.

**See also**

*closedir*, *opendir*, *rewinddir*

## realloc — stdlib.h



**Function**

Reallocates main memory.

**Syntax**

```
void *realloc(void *block, size_t size);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks**

*realloc* attempts to shrink or expand the previously allocated block to *size* bytes. If *size* is zero, the memory block is freed and NULL is returned. The *block* argument points to a memory block previously obtained by calling *malloc*, *calloc*, or *realloc*. If *block* is a NULL pointer, *realloc* works just like *malloc*.

*realloc* adjusts the size of the allocated block to *size*, copying the contents to a new location if necessary.

**Return value**

*realloc* returns the address of the reallocated block, which can be different than the address of the original block. If the block cannot be reallocated, *realloc* returns NULL.

If the value of *size* is 0, the memory block is freed and *realloc* returns NULL.

**See also**

*calloc, farrealloc, free, malloc*

## remove — stdio.h

**Function**

Removes a file.

**Syntax**

```
int remove(const char *filename);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks**

*remove* deletes the file specified by *filename*. It is a macro that simply translates its call to a call to *unlink*. If your file is open, be sure to close it before removing it.

The *filename* string can include a full path.

**Return value**

On successful completion, *remove* returns 0. On error, it returns –1, and the global variable *errno* is set to one of the following values:

| | |
|---|---|
| EACCES | Permission denied |
| ENOENT | No such file or directory |

**See also**

*unlink*

## rename — stdio.h

**Function**

Renames a file.

**Syntax**

```
int rename(const char *oldname, const char *newname);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | ■ | ■ | ■ |

**Remarks**

*rename* changes the name of a file from *oldname* to *newname*. If a drive specifier is given in *newname*, the specifier must be the same as that given in *oldname*.

Directories in *oldname* and *newname* need not be the same, so *rename* can be used to move a file from one directory to another. Wildcards are not allowed.

This function will fail (EACCES) if either file is currently open in any process.

**Return value**

On successfully renaming the file, *rename* returns 0. In the event of error, –1 is returned, and the global variable *errno* is set to one of the following values:

| | |
|---|---|
| EACCES | Permission denied: filename already exists or has an invalid path |
| ENOENT | No such file or directory |
| ENOTSAM | Not same device |

## rewind — stdio.h

**Function**

Repositions a file pointer to the beginning of a stream.

**Syntax**

```
void rewind(FILE *stream);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks**

*rewind*(*stream*) is equivalent to *fseek*(*stream*, 0L, SEEK_SET), except that *rewind* clears the end-of-file and error indicators, while *fseek* clears the end-of-file indicator only.

After *rewind*, the next operation on an update file can be either input or output.

**Return value**

None.

**See also**

*fopen*, *fseek*, *ftell*

## rewinddir — dirent.h

**Function**

Resets a directory stream to the first entry.

**Syntax**

```
void rewinddir(DIR *dirp);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks** *rewinddir* is available on POSIX-compliant UNIX systems.

The *rewinddir* function repositions the directory stream *dirp* at the first entry in the directory. It also ensures that the directory stream accurately reflects any directory entries that might have been created or deleted since the last *opendir* or *rewinddir* on that directory stream.

**Return value** None.

**See also** *closedir, opendir, readdir*

## rmdir — dir.h

**Function** Removes a directory.

**Syntax**

```
int rmdir(const char *path);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks** *rmdir* deletes the directory whose path is given by *path*. The directory named by *path*

- Must be empty
- Must not be the current working directory
- Must not be the root directory

**Return value** *rmdir* returns 0 if the directory is successfully deleted. A return value of –1 indicates an error, and the global variable *errno* is set to one of the following values:

| | |
|---|---|
| EACCES | Permission denied |
| ENOENT | Path or file function not found |

**See also** *chdir, getcurdir, getcwd, mkdir*

## rmtmp — stdio.h

**Function** Removes temporary files.

**Syntax**

```
int rmtmp(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** The *rmtmp* function closes and deletes all open temporary file streams, which were previously created with *tmpfile*. The current directory must the same as when the files were created, or the files will not be deleted.

**Return value** *rmtmp* returns the total number of temporary files it closed and deleted.

**See also** *tmpfile*

## _rotl, _rotr — stdlib.h

**Function** Bit-rotates an **unsigned** short integer value to the left or right.

**Syntax**

```
unsigned short _rotl(unsigned short value, int count);
unsigned short _rotr(unsigned short value, int count);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *_rotl* rotates the given *value* to the left *count* bits.

*_rotr* rotates the given *value* to the right *count* bits.

**Return value** The functions return the rotated integer:

- *_rotl* returns the value of *value* left-rotated *count* bits.
- *_rotr* returns the value of *value* right-rotated *count* bits.

**See also** *_crotl, _crotr, _lrotl, _lrotr*

## _rtl_chmod — dos.h, io.h



**Function** Gets or sets file attributes.

**Syntax**

```
int _rtl_chmod(const char *path, int func [, int attrib]);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | |

**Remarks** *_rtl_chmod* can either fetch or set file attributes. If *func* is 0, *_rtl_chmod* returns the current attributes for the file. If *func* is 1, the attribute is set to *attrib*.

*attrib* can be one of the following symbolic constants (defined in dos.h):

| | |
|---|---|
| FA_RDONLY | Read-only attribute |
| FA_HIDDEN | Hidden file |

| | |
|---|---|
| FA_SYSTEM | System file |
| FA_LABEL | Volume label |
| FA_DIREC | Directory |
| FA_ARCH | Archive |

**Return value**

Upon successful completion, *_rtl_chmod* returns the file attribute word; otherwise, it returns a value of –1.

In the event of an error, the global variable *errno* is set to one of the following:

| | |
|---|---|
| EACCES | Permission denied |
| ENOENT | Path or file name not found |

**See also**

*chmod, _rtl_creat*

## _rtl_close — io.h

**Function**

Closes a file.

**Syntax**

```
int _rtl_close(int handle);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | |

**Remarks**

*_rtl_close* closes the file associated with *handle*, a file handle obtained from a *_rtl_creat, creat, creatnew, creattemp, dup, dup2, _rtl_open,* or *open* call.

The function does not write a *Ctrl-Z* character at the end of the file. If you want to terminate the file with a *Ctrl-Z*, you must explicitly output one.

**Return value**

Upon successful completion, *_rtl_close* returns 0. Otherwise, the function returns a value of –1.

*_rtl_close* fails if *handle* is not the handle of a valid, open file, and the global variable *errno* is set to

| | |
|---|---|
| EBADF | Bad file number |

**See also**

*chsize, close, creatnew, dup, fclose, _rtl_creat, _rtl_open, sopen*

## _rtl_creat — dos.h, io.h

**Function**

Creates a new file or overwrites an existing one.

**Syntax**

```
int _rtl_creat(const char *path, int attrib);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | |

**Remarks**

*_rtl_creat* opens the file specified by *path*. The file is always opened in binary mode. Upon successful file creation, the file pointer is set to the beginning of the file. The file is opened for both reading and writing.

If the file already exists, its size is reset to 0. (This is essentially the same as deleting the file and creating a new file with the same name.)

The *attrib* argument is an ORed combination of one or more of the following constants (defined in dos.h):

| | |
|---|---|
| FA_RDONLY | Read-only attribute |
| FA_HIDDEN | Hidden file |
| FA_SYSTEM | System file |

**Return value**

Upon successful completion, *_rtl_creat* returns the new file handle, a non-negative integer; otherwise, it returns –1.

In the event of error, the global variable *errno* is set to one of the following values:

| | |
|---|---|
| EACCES | Permission denied |
| EMFILE | Too many open files |
| ENOENT | Path or file name not found |

**See also**

*chsize, close, creat, creatnew, creattemp, _rtl_chmod, _rtl_close*

## _rtl_heapwalk — malloc.h



**Function**

Inspects the heap, node by node.

**Syntax**

```
int _rtl_heapwalk(_HEAPINFO *hi);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| | | | ■ | | | |

**Remarks**

*_rtl_heapwalk* assumes the heap is correct. Use *_heapchk* to verify the heap before using *_rtl_heapwalk*. _HEAPOK is returned with the last block on the heap. _HEAPEND will be returned on the next call to *_rtl_heapwalk*.

*_rtl_heapwalk* receives a pointer to a structure of type *_HEAPINFO* (declared in malloc.h).

For the first call to *_rtl_heapwalk*, set the *hi._pentry* field to NULL. *_rtl_heapwalk* returns with *hi._pentry* containing the address of the first block.

*hi._size* holds the size of the block in bytes.

*hi._useflag* is a flag that is set to _USEDENTRY if the block is currently in use. If the block is free, *hi._useflag* is set to _FREEENTRY.

**Return value**

One of the following values:

| | |
|---|---|
| _HEAPBADNODE | A corrupted heap block has been found |
| _HEAPBADPTR | The _pentry field does not point to a valid heap block |
| _HEAPEMPTY | No heap exists |
| _HEAPEND | The end of the heap has been reached |
| _HEAPOK | The _heapinfo block contains valid information about the next heap block |

**See also**

*_heapchk, _heapset*

## _rtl_open — fcntl.h, share.h, io.h

**Function**

Opens an existing file for reading or writing.

**Syntax**

```
int _rtl_open(const char *filename, int oflags);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | |

**Remarks**

*_rtl_open* opens the file specified by *filename*, then prepares it for reading or writing, as determined by the value of *oflags*. The file is always opened in binary mode.

*oflags* uses the flags from the following two lists. Only one flag from the first list can be used (and one *must* be used); the remaining flags can be used in any logical combination.

**List 1: Read/write flags**

| | |
|---|---|
| O_RDONLY | Open for reading. |
| O_WRONLY | Open for writing. |
| O_RDWR | Open for reading and writing. |

The following additional values can be included in *oflags* (using an OR operation):

These symbolic constants are defined in fcntl.h and share.h.

**List 2: Other access flags**

| | |
|---|---|
| O_NOINHERIT | The file is not passed to child programs. |
| SH_COMPAT | Allow other opens with SH_COMPAT. The call will fail if the file has already been opened in any other shared mode. |
| SH_DENYRW | Only the current handle can have access to the file. |
| SH_DENWR | Allow only reads from any other open to the file. |
| SH_DENYRD | Allow only writes from any other open to the file. |
| SH_DENYNO | Allow other shared opens to the file, but not other SH_COMPAT opens. |

Only one of the SH_DENY*xx* values can be included in a single *_rtl_open*. These file-sharing attributes are in addition to any locking performed on the files.

The maximum number of simultaneously open files is defined by HANDLE_MAX.

**Return value**

On successful completion, *_rtl_open* returns a nonnegative integer (the file handle). The file pointer, which marks the current position in the file, is set to the beginning of the file.

On error, *_rtl_open* returns –1. The global variable *errno* is set to one of the following:

| | |
|---|---|
| EACCES | Permission denied |
| EINVACC | Invalid access code |
| EMFILE | Too many open files |
| ENOENT | Path or file not found |

**See also**

*open*, *_rtl_read*, *sopen*



## _rtl_read — io.h, dos.h

**Function**

Reads from file.

**Syntax**

```
int _rtl_read(int handle, void *buf, unsigned len);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks**

*_rtl_read* attempts to read *len* bytes from the file associated with *handle* into the buffer pointed to by *buf*.

When a file is opened in text mode, *_rtl_read* does not remove carriage returns.

The argument *handle* is a file handle obtained from a *creat*, *open*, *dup*, or *dup2* call.

On disk files *_rtl_read* begins reading at the current file pointer. When the reading is complete, the function increments the file pointer by the number of bytes read. On devices, the bytes are read directly from the device.

The maximum number of bytes that *_rtl_read* can read is UINT_MAX –1, because UINT_MAX is the same as –1, the error return indicator. UINT_MAX is defined in limits.h.

**Return value**

On successful completion, *_rtl_read* returns a positive integer indicating the number of bytes placed in the buffer. On end-of-file, *_rtl_read* returns zero. On error, it returns –1, and the global variable *errno* is set to one of the following values:

| | |
|---|---|
| EACCES | Permission denied |
| EBADF | Bad file number |

**See also**

*read*, *_rtl_open*, *_rtl_write*

## _rtl_write — io.h

**Function**

Writes to a file.

**Syntax**

```
int _rtl_write(int handle, void *buf, unsigned len);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | |

**Remarks**

*_rtl_write* attempts to write *len* bytes from the buffer pointed to by *buf* to the file associated with *handle*. The maximum number of bytes that *_rtl_write* can write is UINT_MAX –1, because UINT_MAX is the same as –1, which is the error return indicator for *_rtl_write*. UINT_MAX is defined in limits.h. *_rtl_write* does not translate a linefeed character (LF) to a CR/LF pair because all its files are binary files.

If the number of bytes actually written is less than that requested, the condition should be considered an error and probably indicates a full disk.

For disk files, writing always proceeds from the current file pointer. On devices, bytes are directly sent to the device.

For files opened with the O_APPEND option, the file pointer is not positioned to EOF by *_rtl_write* before writing the data.

**Return value**

*_rtl_write* returns the number of bytes written. In case of error, *_rtl_write* returns –1 and sets the global variable *errno* to one of the following values:

| | |
|---|---|
| EACCES | Permission denied |
| EBADF | Bad file number |

**See also**

*lseek, _rtl_read, write*

## scanf — stdio.h

**Function**

Scans and formats input from the stdin stream.

**Syntax**

```
int scanf(const char *format[, address, ...]);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | | ■ | ■ | ■ | ■ |

**Remarks**

*scanf* scans a series of input fields, one character at a time, reading from the stdin stream. Then each field is formatted according to a format specifier passed to *scanf* in the format string pointed to by *format*. Finally, *scanf* stores the formatted input at an address passed to it as an argument following *format*. There must be the same number of format specifiers and addresses as there are input fields.

➥ For Win32s or Win32 GUI applications, stdin must be redirected.

**The format string**

The format string present in *scanf* and the related functions *cscanf, fscanf, sscanf, vscanf, vfscanf,* and *vsscanf* controls how each function scans, converts, and stores its input fields. *There must be enough address arguments for the given format specifiers; if not, the results will be unpredictable and possibly disastrous.* Excess address arguments (more than required by the format) are ignored.



➥ *scanf* often leads to unexpected results if you diverge from an expected pattern. You need to remember to teach *scanf* how to synchronize at the end of a line. The combination of *gets* or *fgets* followed by *sscanf* is safe and easy, and therefore preferred.

The format string is a character string that contains three types of objects: *whitespace characters, non-whitespace characters,* and *format specifiers.*

- The whitespace characters are blank, tab (**\t**) or newline (**\n**). If a *...scanf* function encounters a whitespace character in the format string, it will read, but not store, all consecutive whitespace characters up to the next non-whitespace character in the input.

- The non-whitespace characters are all other ASCII characters except the percent sign (%). If a *...scanf* function encounters a non-whitespace character in the format string, it will read, but not store, a matching non-whitespace character.
- The format specifiers direct the *...scanf* functions to read and convert characters from the input field into specific types of values, then store them in the locations given by the address arguments.

Trailing whitespace is left unread (including a newline), unless explicitly matched in the format string.

### *Format specifiers*

*...scanf* format specifiers have the following form:

```
% [*] [width] [F|N] [h|l|L] type_character
```

Each format specifier begins with the percent character (%). After the % come the following, in this order:

- An optional assignment-suppression character, `[*]`
- An optional width specifier, `[width]`
- An optional pointer size modifier, `[F|N]`
- An optional argument-type modifier, `[h|l|L]`
- The type character

**Optional format string components**

These are the general aspects of input formatting controlled by the optional characters and specifiers in the *...scanf* format string:

| Character or specifier | What it controls or specifies |
|---|---|
| * | Suppresses assignment of the next input field. |
| width | Maximum number of characters to read; fewer characters might be read if the *...scanf* function encounters a whitespace or unconvertible character. |
| size | Overrides default size of address argument:<br>*N* = **near** pointer<br>*F* = **far** pointer |
| argument type | Overrides default type of address argument:<br>*h* = **short int**<br>*l* = **long int** (if the type character specifies an integer conversion)<br>*l* = **double** (if the type character specifies a floating-point conversion)<br>*L* = **long double** (valid only with floating-point conversions) |

**...scanf type characters**

The following table lists the *...scanf* type characters, the type of input expected by each, and in what format the input will be stored.

The information in this table is based on the assumption that no optional characters, specifiers, or modifiers (*, width, or size) were included in the format specifier.

To see how the addition of the optional elements affects the *...scanf* input, refer to the tables following this one.

| Type character | Expected input | Type of argument |
|---|---|---|
| ***Numerics*** | | |
| **d** | Decimal integer | Pointer to **int** (**int** **arg*). |
| **D** | Decimal integer | Pointer to **long** (**long** **arg*). |
| **o** | Octal integer | Pointer to **int** (**int** **arg*). |
| **O** | Octal integer | Pointer to **long** (**long** **arg*). |
| **i** | Decimal, octal, or hexadecimal integer | Pointer to **int** (**int** **arg*). |
| **I** | Decimal, octal, or hexadecimal integer | Pointer to **long** (**long** **arg*). |
| **u** | Unsigned decimal integer | Pointer to **unsigned int** (**unsigned int** **arg*). |
| **U** | Unsigned decimal integer | Pointer to **unsigned long** (**unsigned long** **arg*). |
| **x** | Hexadecimal integer | Pointer to **int** (**int** **arg*). |
| **X** | Hexadecimal integer | Pointer to **int** (**int** **arg*). |
| **e, E** | Floating point | Pointer to **float** (**float** **arg*). |
| **f** | Floating point | Pointer to **float** (**float** **arg*). |
| **g, G** | Floating point | Pointer to **float** (**float** **arg*). |
| ***Characters*** | | |
| **s** | Character string | Pointer to array of characters (**char** *arg[]*). |
| **c** | Character | Pointer to character (**char** **arg*) if a field width *W* is given along with the *c*-type character (such as %5*c*). |
| | | Pointer to array of *W* characters (**char** *arg*[*W*]). |
| **%** | % character | No conversion done; % is stored. |

| Type character | Expected input | Type of argument |
|---|---|---|
| *Pointers* | | |
| **n** | | Pointer to **int** (**int** **arg*). The number of characters read successfully up to %*n* is stored in this **int**. |
| **p** | Hexadecimal form YYYY:ZZZZ or ZZZZ | Pointer to an object. (**far*** or **near***) %*p* conversions default to the pointer size native to the memory model. |

**Input fields**

Any one of the following is an input field:

- All characters up to (but not including) the next whitespace character
- All characters up to the first one that cannot be converted under the current format specifier (such as an 8 or 9 under octal format)
- Up to *n* characters, where *n* is the specified field width

**Conventions**

Certain conventions accompany some of these format specifiers. The decimal-point character used in the output is determined by the current locale's LC_NUMERIC category. The conventions are summarized here.

***%c conversion***

This specification reads the next character, including a whitespace character. To skip one whitespace character and read the next non-whitespace character, use %1*s*.

***%Wc conversion (W = width specification)***

The address argument is a pointer to an array of characters; the array consists of *W* elements (**char** *arg*[*W*]).

***%s conversion***

The address argument is a pointer to an array of characters (**char** *arg*[]).

The array size must be *at least* (*n*+1) bytes, where *n* equals the length of string *s* (in characters). A space or newline terminates the input field; the terminator is not scanned or stored. A null-terminator is automatically appended to the string and stored as the last element in the array.

***%[search_set] conversion***

The set of characters surrounded by square brackets can be substituted for the *s*-type character. The address argument is a pointer to an array of characters (**char** *arg*[]).

These square brackets surround a set of characters that define a *search set* of possible characters making up the string (the input field).

If the first character in the brackets is a caret (^), the search set is inverted to include all ASCII characters except those between the square brackets. (Normally, a caret will be included in the inverted search set unless explicitly listed somewhere after the first caret.)

The input field is a string not delimited by whitespace. ...*scanf* reads the corresponding input field up to the first character it reaches that does not appear in the search set (or in the inverted search set). Two examples of this type of conversion are

| | |
|---|---|
| `%[abcd]` | Searches for any of the characters *a*, *b*, *c*, and *d* in the input field. |
| `%[^abcd]` | Searches for any characters *except a*, *b*, *c*, and *d* in the input field. |

You can also use a range facility shortcut to define a range of characters (numerals or letters) in the search set. For example, to catch all decimal digits, you could define the search set by using `%[0123456789]`, or you could use the shortcut to define the same search set by using `%[0-9]`.

To catch alphanumeric characters, use the following shortcuts:

| | |
|---|---|
| `%[A-Z]` | Catches all uppercase letters. |
| `%[0-9A-Za-z]` | Catches all decimal digits and all letters (uppercase and lowercase). |
| `%[A-FT-Z]` | Catches all uppercase letters from *A* through *F* and from *T* through Z. |

The rules covering these search set ranges are straightforward:

- The character prior to the hyphen (-) must be lexically less than the one after it.
- The hyphen must not be the first nor the last character in the set. (If it is first or last, it is considered to be the hyphen character, not a range definer.)
- The characters on either side of the hyphen must be the ends of the range and not part of some other range.

Here are some examples where the hyphen just means the hyphen character, not a range between two ends:

| | |
|---|---|
| `%[-+*/]` | The four arithmetic operations. |
| `%[z-a]` | The characters z, –, and *a*. |
| `%[+0-9-A-Z]` | The characters + and – and the ranges 0-9 and *A*-Z. |
| `%[+0-9A-Z-]` | Also the characters + and – and the ranges 0-9 and *A*-Z. |
| `%[^-0-9+A-Z]` | All characters except + and – and those in the ranges 0-9 and *A*-Z. |

***%e, %E. %f, %g, and %G (floating-point) conversions***

Floating-point numbers in the input field must conform to the following generic format:

```
[+/-] ddddddddd [.] dddd [E | e] [+/-] ddd
```

where [*item*] indicates that *item* is optional, and *ddd* represents decimal, octal, or hexadecimal digits.

INF = INFinity; NAN = Not-A-Number

In addition, +INF, –INF, +NAN, and –NAN are recognized as floating-point numbers. Note that the sign and capitalization are required.

***%d, %i, %o, %x, %D, %I, %O, %X, %c, %n conversions***

A pointer to **unsigned** character, **unsigned** integer, or **unsigned long** can be used in any conversion where a pointer to a character, integer, or **long** is allowed.

**Assignment-suppression character**

The assignment-suppression character is an asterisk (*); it is not to be confused with the C indirection (pointer) operator (also an asterisk).

If the asterisk follows the percent sign (%) in a format specifier, the next input field will be scanned but not assigned to the next address argument. The suppressed input data is assumed to be of the type specified by the type character that follows the asterisk character.

The success of literal matches and suppressed assignments is not directly determinable.

**Width specifiers**

The width specifier (*n*), a decimal integer, controls the maximum number of characters that will be read from the current input field.

If the input field contains fewer than *n* characters, *...scanf* reads all the characters in the field, then proceeds with the next field and format specifier.

If a whitespace or nonconvertible character occurs before width characters are read, the characters up to that character are read, converted, and stored, then the function attends to the next format specifier.

A nonconvertible character is one that cannot be converted according to the given format (such as an 8 or 9 when the format is octal, or a *J* or *K* when the format is hexadecimal or decimal).

| Width specifier | How width of stored input is affected |
|---|---|
| n | Up to *n* characters are read, converted, and stored in the current address argument. |

**Input-size and argument-type modifiers**

The input-size modifiers (*N* and *F*) and argument-type modifiers (*h*, *l*, and *L*) affect how the *...scanf* functions interpret the corresponding address argument *arg[f]*.

*F* and *N* override the default or declared size of *arg*.

*h*, *l*, and *L* indicate which type (version) of the following input data is to be used (*h* = **short**, *l* = **long**, *L* = **long double**). The input data will be converted to the specified version, and the *arg* for that input data should point to an object of the corresponding size (**short** object for **%h**, **long** or **double** object for **%l**, and **long double** object for **%L**).

| Modifier | How conversion is affected |
|---|---|
| **F** | Overrides default or declared size; *arg* interpreted as **far** pointer. |
| **N** | Overrides default or declared size; *arg* interpreted as **near** pointer. Cannot be used with any conversion in **huge** model. |
| **h** | For *d, i, o, u, x* types, convert input to **short int**, store in **short** object. |
| | For *D, I, O, U, X* types, no effect. |
| | For *e, f, c, s, n, p* types, no effect. |
| **l** | For *d, i, o, u, x* types, convert input to **long int**, store in **long** object. |
| | For *e, f, g* types, convert input to **double**, store in **double** object. |
| | For *D, I, O, U, X* types, no effect. |
| | For *c, s, n, p* types, no effect. |
| **L** | For *e, f, g* types, convert input to a **long double**, store in **long double** object. **L** has no effect on other formats. |

**When scanf stops scanning**

*scanf* might stop scanning a particular field before reaching the normal field-end character (whitespace), or might terminate entirely, for a variety of reasons.

*scanf* stops scanning and storing the current field and proceed to the next input field if any of the following occurs:

- An assignment-suppression character (*) appears after the percent character in the format specifier; the current input field is scanned but not stored.
- *width* characters have been read (*width* = width specification, a positive decimal integer in the format specifier).
- The next character read cannot be converted under the current format (for example, an *A* when the format is decimal).

- The next character in the input field does not appear in the search set (or does appear in an inverted search set).

When *scanf* stops scanning the current input field for one of these reasons, the next character is assumed to be unread and to be the first character of the following input field, or the first character in a subsequent read operation on the input.

*scanf* will terminate under the following circumstances:

- The next character in the input field conflicts with a corresponding non-whitespace character in the format string.
- The next character in the input field is EOF.
- The format string has been exhausted.

If a character sequence that is not part of a format specifier occurs in the format string, it must match the current sequence of characters in the input field; *scanf* will scan but not store the matched characters. When a conflicting character occurs, it remains in the input field as if it were never read.

**Return value**

*scanf* returns the number of input fields successfully scanned, converted, and stored; the return value does not include scanned fields that were not stored. If *scanf* attempts to read at end-of-file, the return value is EOF. If no fields were stored, the return value is 0.

**See also**

*atof, cscanf, fscanf, freopen, getc, printf, sscanf, vfscanf, vscanf, vsscanf*

## _searchenv — stdlib.h

**Function**

Searches an environment path for a file.

**Syntax**

```
void _searchenv(const char *file, const char *varname, char *buf);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks**

*_searchenv* attempts to locate *file*, searching along the path specified by the operating system environment variable *varname*. Typical environment variables that contain paths are PATH, LIB, and INCLUDE.

*_searchenv* searches for the file in the current directory of the current drive first. If the file is not found there, the environment variable *varname* is fetched, and each directory in the path it specifies is searched in turn until the file is found, or the path is exhausted.

When the file is located, the full path name is stored in the buffer pointed to by *buf*. This string can be used in a call to access the file (for example, with *fopen* or *exec...*). The buffer is assumed to be large enough to store any possible file name. If the file cannot be successfully located, an empty string (consisting of only a null character) will be stored at *buf*.

**Return value** None.

**See also** *_dos_findfirst, _dos_findnext, exec..., spawn..., system*

## searchpath — dir.h

**Function** Searches the operating system path for a file.

**Syntax**

```
char *searchpath(const char *file);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *searchpath* attempts to locate *file*, searching along the operating system path, which is the PATH=... string in the environment. A pointer to the complete path-name string is returned as the function value.

*searchpath* searches for the file in the current directory of the current drive first. If the file is not found there, the PATH environment variable is fetched, and each directory in the path is searched in turn until the file is found, or the path is exhausted.

When the file is located, a string is returned containing the full path name. This string can be used in a call to access the file (for example, with *fopen* or *exec...*).

The string returned is located in a static buffer and is overwritten on each subsequent call to *searchpath*.

**Return value** *searchpath* returns a pointer to a file name string if the file is successfully located; otherwise, *searchpath* returns null.

**See also** *exec..., findfirst, findnext, spawn..., system*

## _searchstr — stdlib.h

**Function** Searches a list of directories for a file.

**Syntax**

```
void _searchstr(const char *file, const char *ipath, char *buf);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks**

*_searchstr* attempt to locate *file*, searching along the path specified by the string *ipath*.

*_searchstr* searches for the file in the current directory of the current drive first. If the file is not found there, each directory in *ipath* is searched in turn until the file is found, or the path is exhausted. The directories in *ipath* must be separated by semicolons.

When the file is located, the full path name is stored in the buffer pointed by by *buf*. This string can be used in a call to access the file (for example, with *fopen* or *exec...*). The buffer is assumed to be large enough to store any possible file name. The constant _MAX_PATH, defined in stdlib.h, is the size of the largest file name. If the file cannot be successfully located, an empty string (consisting of only a null character) will be stored at *buf*.

**Return value**

None.

**See also**

*_searchenv*

## segread — dos.h

**Function**

Reads segment registers.

**Syntax**

```
void segread(struct SREGS *segp);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks**

*segread* places the current values of the segment registers into the structure pointed to by *segp*.

This call is intended for use with *intdosx* and *int86x*.

**Return value**

None.

**See also**

*FP_OFF, int86, int86x, intdos, intdosx, MK_FP, movedata*

## setbuf — stdio.h

**Function**

Assigns buffering to a stream.

**Syntax**

```
void setbuf(FILE *stream, char *buf);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks**

*setbuf* causes the buffer *buf* to be used for I/O buffering instead of an automatically allocated buffer. It is used after *stream* has been opened.

If *buf* is null, I/O will be unbuffered; otherwise, it will be fully buffered. The buffer must be BUFSIZ bytes long (specified in stdio.h).

*stdin* and *stdout* are unbuffered if they are not redirected; otherwise, they are fully buffered. *setbuf* can be used to change the buffering style used.

*Unbuffered* means that characters written to a stream are immediately output to the file or device, while *buffered* means that the characters are accumulated and written as a block.

*setbuf* produces unpredictable results unless it is called immediately after opening *stream* or after a call to *fseek*. Calling *setbuf* after *stream* has been unbuffered is legal and will not cause problems.

A common cause for error is to allocate the buffer as an automatic (local) variable and then fail to close the file before returning from the function where the buffer was declared.

**Return value**

None.

**See also**

*fflush, fopen, fseek, setvbuf*

## setcbrk — dos.h

**Function**

Sets control-break setting.

**Syntax**

```
int setcbrk(int cbrkvalue);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks**

*setcbrk* uses the DOS system call 0x33 to turn control-break checking on or off.

*cbrkvalue* = 0 Turns checking off (check only during I/O to console, printer, or communications devices).

*cbrkvalue* = 1 Turns checking on (check at every system call).

**Return value**

*setcbrk* returns *cbrkvalue*, the value passed.

**See also**

*getcbrk*

## _setcursortype — conio.h

**Function** Selects cursor appearance.

**Syntax**

```
void _setcursortype(int cur_t);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | | ■ | | | ■ |

**Remarks** Sets the cursor type to

| | |
|---|---|
| _NOCURSOR | Turns off the cursor |
| _NORMALCURSOR | Normal underscore cursor |
| _SOLIDCURSOR | Solid block cursor |

This function should not be used in Win32s or Win32 GUI applications.

**Return value** None.

## setdate

See *_dos_getdate*.

## setdisk

See *getdisk*.

## setdta — dos.h

**Function** Sets disk-transfer address.

**Syntax**

```
void setdta(char far *dta);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | |

**Remarks** *setdta* changes the current setting of the DOS disk-transfer address (DTA) to the value given by *dta*.

**Return value** None.

**See also** *getdta*

## setftime

See *getftime*.

## setjmp setjmp.h

**Function** Sets up for nonlocal goto.

**Syntax**

```
int setjmp(jmp_buf jmpb);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *setjmp* captures the complete *task state* in *jmpb* and returns 0.

A later call to *longjmp* with *jmpb* restores the captured task state and returns in such a way that *setjmp* appears to have returned with the value *val*.

A task state includes:

| Win 16 | Win 32 |
|---|---|
| All segment registers CS, DS, ES, SS | No segment registers are saved |
| Register variables DI and SI | Register variables EBX, EDI, ESI |
| Stack pointer SP | Stack pointer ESP |
| Frame pointer BP | Frame pointer EBP |
| Flags | Flags are not saved |

A task state is complete enough that *setjmp* can be used to implement coroutines.

*setjmp* must be called before *longjmp*. The routine that calls *setjmp* and sets up *jmpb* must still be active and cannot have returned before the *longjmp* is called. If it has returned, the results are unpredictable.

*setjmp* is useful for dealing with errors and exceptions encountered in a low-level subroutine of a program.

**Return value** *setjmp* returns 0 when it is initially called. If the return is from a call to *longjmp*, *setjmp* returns a nonzero value (as in the example).

**See also** *longjmp*, *signal*

# setlocale — locale.h

**Function**

Selects or queries a locale.

**Syntax**

```
char *setlocale(int category, const char *locale);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | ■ | ■ | ■ |

**Remarks**

Borland C++ supports the following locales at present:

| Module | Locale supported |
|---|---|
| de_DE | German |
| fr_FR | French |
| en_GB | English (Great Britain) |
| en_US | English (United States) |

Future releases of Borland C++ will increase the number of locales supported.

For each locale, the following character sets are supported:

| | |
|---|---|
| DOS437 | English |
| DOS850 | Multilingual (Latin I) |
| WIN1252 | Windows, Multilingual |

For a description of DOS character sets, see *MS-DOS User's Guide and Reference*. See also *MS Windows 3.1 Programmer's Reference, Volume 4* for a discussion of the WIN1252 character set.

The possible values for the *category* argument are as follows:

| Value | Description |
|---|---|
| LC_ALL | Affects all the following categories. |
| LC_COLLATE | Affects *strcoll* and *strxfrm*. |
| LC_CTYPE | Affects single-byte character handling functions. The *mbstowcs* and *mbtowc* functions are not affected. |
| LC_MONETARY | Affects monetary formatting by the *localeconv* function. |
| LC_NUMERIC | Affects the decimal point of non-monetary data formatting. This includes the *printf* family of functions, and the information returned by *localeconv*. |
| LC_TIME | Affects *strftime*. |

The *locale* argument is a pointer to the name of the locale or named locale category. Passing a NULL pointer returns the current locale in effect. Passing a pointer that points to a null string requests *setlocale* to look for

environment variables to determine which locale to set. The locale names are case sensitive.

The LOCALE.BLL file is installed in BC4\BIN directory.

If you specify a locale other than the default C locale, *setlocale* tries to access the locale library file named LOCALE.BLL to obtain the locale data. This file is located using the following strategies:

1. Searching the directory where the application's executable resides.
2. Searching in the current default directory.
3. Accessing the "PATH" environment variable and searching in each of the specified directories.

If the locale library is not found, *setlocale* terminates.

When *setlocale* is unable to honor a locale request, the preexisting locale in effect is unchanged and a null pointer is returned.

If the *locale* argument is a NULL pointer, the locale string for the category is returned. If *category* is LC_ALL, a complete locale string is returned. The structure of the complete locale string consists of the names of all the categories in the current locale concatenated and separated by semicolons. This string can be used as the locale parameter when calling *setlocale* with LC_ALL. This will reinstate all the locale categories that are named in the complete locale string, and allows saving and restoring of locale states. If the complete locale string is used with a single category, for example, LC_TIME, only that category will be restored from the locale string.

ANSI C states that if an empty string "" is used as the locale parameter an implementation defined locale is used. *setlocale* has been implemented to look for corresponding environment variables in this instance as POSIX suggests.

If the environment variable LC_ALL exists, the category will be set according to this variable. If the variable does not exist, the environment variable that has the same name as the requested category is looked for and the category is set accordingly.

If none of the above are satisfied, the environment variable named LANG is used. Otherwise, *setlocale* fails and returns a NULL pointer.

See the *Programmer's Guide*, Chapter 5, for information about defining options.

To take advantage of dynamically loadable locales in your application, define __USELOCALES__ for each module. If __USELOCALES__ is not defined, all locale-sensitive functions and macros will work only with the default C locale.

If a NULL pointer is used as the argument for the *locale* parameter, *setlocale* returns a string that specifies the current locale in effect. If the *category* parameter specifies a single category, such as LC_COLLATE, the string

pointed to will be the name of that category. If LC_ALL is used as the *category* parameter then the string pointed to will be a full locale string that will indicate the name of each category in effect.

```
⋮
localenameptr = setlocale( LC_COLLATE, NULL );

if (localenameptr)
   printf( "%s\n", localenameptr );
⋮
```

The output here will be one of the module names together with the specified code page. For example, the output could be `fr_FR.DOS850@dbase`.

```
⋮
localenameptr = setlocale( LC_ALL, NULL );

if (localenameptr)
   printf( "%s\n", localenameptr );
⋮
```

An example of the output here could be the following:

```
fr_FR.DOS850@dbase;fr_FR.DOS850;fr_FR.DOS850;fr_FR.DOS850;
fr_FR.DOS850;fr_FR.DOS850;;
```

Each category in this full string is delimited by a semicolon. This string can be copied and saved by an application and then used again to restore the same locale categories at another time. Each delimited name corresponds to the locale category constants defined in locale.h. Therefore, the first name is the name of the LC_COLLATE category, the second is the LC_CTYPE category, and so on. Any other categories named in the locale.h header file are reserved for future implementation.

Here are some examples of setting locales by using *setlocale*:

Set all default categories for the specified French locale:

```
setlocale( LC_ALL, "fr_FR.DOS850" );
```

Set French locale to named collation *dbase*:

```
setlocale( LC_COLLATE, "fr_FR.DOS850@dbase" )
```

The default collation is named *dbase*. Therefore, whether you specify *dbase* or nothing at all, you get the same collation. However, *dbase* might not be the default in future releases.

When a category is loaded from the locale library, the default category is the one that will be loaded unless a modifier name is used. For example:

```
setlocale( LC_COLLATE, "fr_FR.DOS850" )
```

causes the default LC_COLLATE category to be loaded. It might or might not have a specific name.

```
setlocale( LC_COLLATE, "fr_FR.DOS850@dbase" )
```

specifies that the LC_COLLATE category named `dbase` to be loaded. This might or might not be the default.

*setlocale* updates the *lconv* locale structure when a request has been fulfilled.

When an application exits, any allocated memory used for the locale object is deallocated.

**Return value** If selection is successful, *setlocale* returns a pointer to a string that is associated with the selected category (or possibly all categories) for the new locale.

On failure, a NULL pointer is returned and the locale is unchanged. All other possible returns are discussed in the Remarks section above.

**See also** *localeconv*

## setmem — mem.h

**Function** Assigns a value to a range of memory.

**Syntax**

```
void setmem(void *dest, unsigned length, char value);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks** *setmem* sets a block of *length* bytes, pointed to by *dest*, to the byte *value*.

**Return value** None.

**See also** *memset*, *strset*

## setmode — fcntl.h

**Function** Sets mode of an open file.

**Syntax**

```
int setmode(int handle, int amode);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *setmode* sets the mode of the open file associated with *handle* to either binary or text. The argument *amode* must have a value of either O_BINARY or O_TEXT, never both. (These symbolic constants are defined in fcntl.h.)

**Return value** *setmode* returns the previous translation mode if successful. On error it returns –1 and sets the global variable *errno* to

EINVAL Invalid argument

**See also** *_rtl_creat, creat, _rtl_open, open*

## settime

See *gettime* on page 133.

## setvbuf stdio.h

**Function** Assigns buffering to a stream.

**Syntax**

```
int setvbuf(FILE *stream, char *buf, int type, size_t size);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *setvbuf* causes the buffer *buf* to be used for I/O buffering instead of an automatically allocated buffer. It is used after the given stream is opened.

If *buf* is null, a buffer will be allocated using *malloc*; the buffer will use *size* as the amount allocated. The buffer will be automatically freed on close. The *size* parameter specifies the buffer size and must be greater than zero.

The parameter *size* is limited by the constant UINT_MAX as defined in limits.h.

*stdin* and *stdout* are unbuffered if they are not redirected; otherwise, they are fully buffered. *Unbuffered* means that characters written to a stream are immediately output to the file or device, while *buffered* means that the characters are accumulated and written as a block.

The *type* parameter is one of the following:

- _IOFBF The file is *fully buffered*. When a buffer is empty, the next input operation will attempt to fill the entire buffer. On output, the buffer will be completely filled before any data is written to the file.
- _IOLBF The file is *line buffered*. When a buffer is empty, the next input operation will still attempt to fill the entire buffer. On output,

however, the buffer will be flushed whenever a newline character is written to the file.

- _IONBF — The file is *unbuffered*. The *buf* and *size* parameters are ignored. Each input operation will read directly from the file, and each output operation will immediately write the data to the file.

A common cause for error is to allocate the buffer as an automatic (local) variable and then fail to close the file before returning from the function where the buffer was declared.

**Return value** *setvbuf* returns 0 on success. It returns nonzero if an invalid value is given for *type* or *size*, or if there is not enough space to allocate a buffer.

**See also** *fflush, fopen, setbuf*

## setvect

See *getvect*.

## setverify — dos.h

**Function** Sets the state of the verify flag in the operating system.

**Syntax**

```
void setverify(int value);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | | | | ■ |

**Remarks** *setverify* sets the current state of the verify flag to *value*, which can be either 0 (off) or 1 (on).

The verify flag controls output to the disk. When verify is off, writes are not verified; when verify is on, all disk writes are verified to ensure proper writing of the data.

**Return value** None.

**See also** *getverify*

## signal — signal.h

**Function** Specifies signal-handling actions.

**Syntax**

```
void (_USERENTRY *signal(int sig, void (_USERENTRY *func)
                                    (int sig[, int subcode])))(int);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks**

*signal* determines how receipt of signal number *sig* will subsequently be treated. You can install a user-specified handler routine (specified by the argument *func*) or use one of the two predefined handlers, SIG_DFL and SIG_IGN, in signal.h. The function *func* must be used with the _USERENTRY calling convention.

| Function pointer | Description |
|---|---|
| SIG_DFL | Terminates the program |
| SIG_ERR | Indicates an error return from *signal* |
| SIG_IGN | Ignore this type signal |

The signal types and their defaults are as follows:

| Signal type | Description |
|---|---|
| SIGBREAK | Keyboard must be in raw mode. |
| SIGABRT | Abnormal termination. Default action is equivalent to calling *_exit*(3). |
| SIGFPE | Arithmetic error caused by division by 0, invalid operation, and the like. Default action is equivalent to calling *_exit*(1). |
| SIGILL | Illegal operation. Default action is equivalent to calling *_exit*(1). |
| SIGINT | *Ctrl-C* interrupt. Default action is to do an INT 23h. |
| SIGSEGV | Illegal storage access. Default action is equivalent to calling *_exit*(1). |
| SIGTERM | Request for program termination. Default action is equivalent to calling *_exit*(1). |
| SIGUSR1, SIGUSR2, SIGUSR3 | User-defined signals that can be generated only by calling *raise*. Default action is to ignore the signal. |


signal.h defines a type called *sig_atomic_t*, the largest integer type the processor can load or store atomically in the presence of asynchronous interrupts (for the 8086 family, this is a 16-bit word; for 80386 and higher number processors, it is a 32-bit word—a Borland C++ integer).

When a signal is generated by the *raise* function or by an external event, the following two things happen:

- If a user-specified handler has been installed for the signal, the action for that signal type is set to SIG_DFL.
- The user-specified function is called with the signal type as the parameter.

User-specified handler functions can terminate by a return or by a call to *abort*, *_exit*, *exit*, or *longjmp*. If your handler function is expected to continue to receive and handle more signals, you must have the handler function call *signal* again.

Borland C++ implements an extension to ANSI C when the signal type is SIGFPE, SIGSEGV, or SIGILL. The user-specified handler function is called with one or two extra parameters. If SIGFPE, SIGSEGV, or SIGILL has been raised as the result of an explicit call to the *raise* function, the user-specified handler is called with one extra parameter, an integer specifying that the handler is being explicitly invoked. The explicit activation values for SIGFPE, SIGSEGV and SIGILL are as follows (see declarations in float.h):

| Signal | Meaning |
|---|---|
| SIGFPE | FPE_EXPLICITGEN |
| SIGSEGV | SEGV_EXPLICITGEN |
| SIGILL | ILL_EXPLICITGEN |

If SIGFPE is raised because of a floating-point exception, the user handler is called with one extra parameter that specifies the FPE_xxx type of the signal. If SIGSEGV, SIGILL, or the integer-related variants of SIGFPE signals (FPE_INTOVFLOW or FPE_INTDIV0) are raised as the result of a processor exception, the user handler is called with two extra parameters:

1. The SIGFPE, SIGSEGV, or SIGILL exception type (see float.h for all these types). This first parameter is the usual ANSI signal type.
2. An integer pointer into the stack of the interrupt handler that called the user-specified handler. This pointer points to a list of the processor registers saved when the exception occurred. The registers are in the same order as the parameters to an interrupt function; that is, BP, DI, SI, DS, ES, DX, CX, BX, AX, IP, CS, FLAGS. To have a register value changed when the handler returns, change one of the locations in this list. For example, to have a new SI value on return, do something like this:

   ```
   *((int*)list_pointer + 2) = new_SI_value;
   ```

   In this way, the handler can examine and make any adjustments to the registers that you want.

The following SIGFPE-type signals can occur (or be generated). They correspond to the exceptions that the 8087 family is capable of detecting, as well as the "INTEGER DIVIDE BY ZERO" and the "INTERRUPT ON OVERFLOW" on the main CPU. (The declarations for these are in float.h.)

| SIGFPE signal | Meaning |
|---|---|
| FPE_INTOVFLOW | INTO executed with OF flag set |
| FPE_INTDIV0 | Integer divide by zero |
| FPE_INVALID | Invalid operation |
| FPE_ZERODIVIDE | Division by zero |
| FPE_OVERFLOW | Numeric overflow |
| FPE_UNDERFLOW | Numeric underflow |
| FPE_INEXACT | Precision |
| FPE_EXPLICITGEN | User program executed *raise*(SIGFPE) |
| FPE_STACKFAULT | Floating-point stack overflow or underflow |

The FPE_INTOVFLOW and FPE_INTDIV0 signals are generated by integer operations, and the others are generated by floating-point operations. Whether the floating-point exceptions are generated depends on the coprocessor control word, which can be modified with *_control87*. Denormal exceptions are handled by Borland C++ and not passed to a signal handler.

The following SIGSEGV-type signals can occur:

| | |
|---|---|
| SEGV_BOUND | Bound constraint exception |
| SEGV_EXPLICITGEN | *raise*(SIGSEGV) was executed |

The 8088 and 8086 processors *don't* have a bound instruction. The 186, 286, 386, and NEC V series processors *do* have this instruction. So, on the 8088 and 8086 processors, the SEGV_BOUND type of SIGSEGV signal won't occur. Borland C++ doesn't generate bound instructions, but they can be used in inline code and separately compiled assembler routines that are linked in.

The following SIGILL-type signals can occur:

| | |
|---|---|
| ILL_EXECUTION | Illegal operation attempted |
| ILL_EXPLICITGEN | *raise*(SIGILL) was executed |

The 8088, 8086, NEC V20, and NEC V30 processors *don't* have an illegal operation exception. The 186, 286, 386, NEC V40, and NEC V50 processors *do* have this exception type. So, on 8088, 8086, NEC V20, and NEC V30 processors, the ILL_EXECUTION type of SIGILL won't occur.

When the signal type is SIGFPE, SIGSEGV, or SIGILL, a return from a signal handler is generally not advisable if the state of the 8087 is corrupt, the results of an integer division are wrong, an operation that shouldn't

have overflowed did, a bound instruction failed, or an illegal operation was attempted. The only time a return is reasonable is when the handler alters the registers so that a reasonable return context exists *or* the signal type indicates that the signal was generated explicitly (for example, FPE_EXPLICITGEN, SEGV_EXPLICITGEN, or ILL_EXPLICITGEN). Generally in this case you would print an error message and terminate the program using *_exit*, *exit*, or *abort*. If a return is executed under any other conditions, the program's action will probably be unpredictable upon resuming.

**Return value** If the call succeeds, *signal* returns a pointer to the previous handler routine for the specified signal type. If the call fails, *signal* returns SIG_ERR, and the external variable *errno* is set to EINVAL.

**See also** *abort*, *_control87*, *ctrlbrk*, *exit*, *longjmp*, *raise*, *setjmp*

## sin, sinl — math.h

**Function** Calculates sine.

**Syntax**

```
double sin(double x);
long double sinl(long double x);
```

| | DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|---|
| *sin* | ■ | ■ | ■ | ■ | ■ | | ■ |
| *sinl* | ■ | | ■ | ■ | | | ■ |

**Remarks** *sin* computes the sine of the input value. Angles are specified in radians.

*sinl* is the **long double** version; it takes a **long double** argument and returns a **long double** result. Error handling for these functions can be modified through the functions *_matherr* and *_matherrl*.

This function can be used with *bcd* and *complex* types.

**Return value** *sin* and *sinl* return the sine of the input value.

**See also** *acos*, *asin*, *atan*, *atan2*, *bcd*, *complex*, *cos*, *tan*

## sinh, sinhl — math.h

**Function** Calculates hyperbolic sine.

**Syntax**

```
double sinh(double x);
long double sinhl(long double x);
```

| | DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|---|
| *sinh* | ■ | ■ | ■ | ■ | ■ | | ■ |
| *sinhl* | ■ | | ■ | ■ | | | ■ |

**Remarks**

*sinh* computes the hyperbolic sine, $(e^x - e^{-x})/2$.

*sinl* is the **long double** version; it takes a **long double** argument and returns a **long double** result. Error handling for *sinh* and *sinhl* can be modified through the functions *_matherr* and *_matherrl*.

This function can be used with *bcd* and *complex* types.

**Return value**

*sinh* and *sinhl* return the hyperbolic sine of *x*.

When the correct value overflows, these functions return the value HUGE_VAL (*sinh*) or _LHUGE_VAL (*sinhl*) of appropriate sign. Also, the global variable *errno* is set to ERANGE.

**See also**

*acos, asin, atan, atan2, bcd, complex, cos, cosh, sin, tan, tanh*

## sleep — dos.h

**Function**

Suspends execution for an interval (seconds).

**Syntax**

```
void sleep(unsigned seconds);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | | ■ | | | ■ |

**Remarks**

With a call to *sleep*, the current program is suspended from execution for the number of seconds specified by the argument *seconds*. The interval is accurate only to the nearest hundredth of a second or to the accuracy of the operating system clock, whichever is less accurate.

**Return value**

None.

## sopen — fcntl.h, sys\stat.h, share.h, io.h

**Function**

Opens a shared file.

**Syntax**

```
int sopen(char *path, int access, int shflag[, int mode]);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks**

*sopen* opens the file given by *path* and prepares it for shared reading or writing, as determined by *access*, *shflag*, and *mode*.

For *sopen*, *access* is constructed by ORing flags bitwise from the following two lists. Only one flag from the first list can be used; the remaining flags can be used in any logical combination.

**List 1: Read/write flags**

| | |
|---|---|
| O_RDONLY | Open for reading only. |
| O_WRONLY | Open for writing only. |
| O_RDWR | Open for reading and writing. |

**List 2: Other access flags**

| | |
|---|---|
| O_NDELAY | Not used; for UNIX compatibility. |
| O_APPEND | If set, the file pointer is set to the end of the file prior to each write. |
| O_CREAT | If the file exists, this flag has no effect. If the file does not exist, the file is created, and the bits of *mode* are used to set the file attribute bits as in *chmod*. |
| O_TRUNC | If the file exists, its length is truncated to 0. The file attributes remain unchanged. |
| O_EXCL | Used only with O_CREAT. If the file already exists, an error is returned. |
| O_BINARY | This flag can be given to explicitly open the file in binary mode. |
| O_TEXT | This flag can be given to explicitly open the file in text mode. |
| O_NOINHERIT | The file is not passed to child programs. |

These O_... symbolic constants are defined in fcntl.h.

If neither O_BINARY nor O_TEXT is given, the file is opened in the translation mode set by the global variable *_fmode*.

If the O_CREAT flag is used in constructing *access*, you need to supply the *mode* argument to *sopen* from the following symbolic constants defined in sys\stat.h.

| Value of *mode* | Access permission |
|---|---|
| S_IWRITE | Permission to write |
| S_IREAD | Permission to read |
| S_IREADIS_IWRITE | Permission to read/write |

*shflag* specifies the type of file-sharing allowed on the file *path*. Symbolic constants for *shflag* are defined in share.h.

| Value of *shflag* | What it does |
|---|---|
| SH_COMPAT | Sets compatibility mode. |
| SH_DENYRW | Denies read/write access. |
| SH_DENYWR | Denies write access. |
| SH_DENYRD | Denies read access. |
| SH_DENYNONE | Permits read/write access. |
| SH_DENYNO | Permits read/write access. |

**Return value**

On successful completion, *sopen* returns a nonnegative integer (the file handle), and the file pointer (that marks the current position in the file) is set to the beginning of the file. On error, it returns –1, and the global variable *errno* is set to

| | |
|---|---|
| EACCES | Permission denied |
| EINVACC | Invalid access code |
| EMFILE | Too many open files |
| ENOENT | Path or file function not found |

**See also**

*chmod, close, creat, lock, lseek, _rtl_open, open, unlock, umask*

## spawnl, spawnle, spawnlp, spawnlpe, spawnv, spawnve, spawnvp, spawnvpe — process.h, stdio.h

**Function**

Creates and runs child processes.

**Syntax**

```
int spawnl(int mode, char *path, char *arg0, arg1, ..., argn, NULL);
int spawnle(int mode, char *path, char *arg0, arg1, ..., argn, NULL, char *envp[]);
int spawnlp(int mode, char *path, char *arg0, arg1, ..., argn, NULL);
int spawnlpe(int mode, char *path, char *arg0, arg1, ..., argn, NULL,
        char *envp[]);
```

The last string must be NULL in functions spawnle, spawnlpe, spawnv, spawnve, spawnvp, and spawnvpe.

```
int spawnv(int mode, char *path, char *argv[]);
int spawnve(int mode, char *path, char *argv[], char *envp[]);
int spawnvp(int mode, char *path, char *argv[]);
int spawnvpe(int mode, char *path, char *argv[], char *envp[]);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | | ■ | | | ■ |

**Remarks**

The functions in the *spawn...* family create child processes that run (execute) their own files. There must be sufficient memory available for loading and executing a child process.

The value of *mode* determines what action the calling function (the *parent process*) takes after the *spawn...* call. The possible values of *mode* are

| | |
|---|---|
| P_WAIT | Puts parent process "on hold" until child process completes execution. |
| P_NOWAIT | Continues to run parent process while child process runs. The child process ID is returned, so that the parent can wait for completion using *cwait* or *wait*. |
| P_NOWAITO | Identical to P_NOWAIT except that the child process ID isn't saved by the operating system, so the parent process can't wait for it using *cwait* or *wait*. |
| P_DETACH | Identical to P_NOWAITO, except that the child process is executed in the background with no access to the keyboard or the display. |
| P_OVERLAY | Overlays child process in memory location formerly occupied by parent. Same as an *exec...* call. |

*path* is the file name of the called child process. The *spawn...* function calls search for *path* using the standard operating system search algorithm:

- If no explicit extension is given, the functions search for the file as given. If the file is not found, they add .EXE and search again. If not found, they add .COM and search again. If still not found, they add .BAT and search once more. The command processor COMSPEC is used to run the executable file.
- If an extension is given, they search only for the exact file name.
- If only a period is given, they search only for the file name with no extension.
- If *path* does not contain an explicit directory, *spawn...* functions that have the ***p*** suffix search the current directory, then the directories set with the operating system PATH environment variable.

The suffixes *p*, *l*, and *v*, and *e* added to the *spawn...* "family name" specify that the named function operates with certain capabilities.

**p** The function searches for the file in those directories specified by the PATH environment variable. Without the *p* suffix, the function searches only the current working directory.

**l** The argument pointers *arg0*, *arg1*, ..., *argn* are passed as separate arguments. Typically, the *l* suffix is used when you know in advance the number of arguments to be passed.

**v** The argument pointers *argv[0], ..., arg[n]* are passed as an array of pointers. Typically, the *v* suffix is used when a variable number of arguments is to be passed.

**e** The argument *envp* can be passed to the child process, letting you alter the environment for the child process. Without the *e* suffix, child processes inherit the environment of the parent process.

Each function in the *spawn...* family *must* have one of the two argument-specifying suffixes (either *l* or *v*). The path search and environment inheritance suffixes (*p* and *e*) are optional.

For example,

- *spawnl* takes separate arguments, searches only the current directory for the child, and passes on the parent's environment to the child.
- *spawnvpe* takes an array of argument pointers, incorporates PATH in its search for the child process, and accepts the *envp* argument for altering the child's environment.

The *spawn...* functions must pass at least one argument to the child process (*arg0* or *argv[0]*). This argument is, by convention, a copy of *path*. (Using a different value for this $0^{th}$ argument won't produce an error.) If you want to pass an empty argument list to the child process, then *arg0* or *argv[0]* must be NULL.

Under DOS 3.x, *path* is available for the child process; under earlier versions, the child process cannot use the passed value of the $0^{th}$ argument (*arg0* or *argv[0]*).

When the *l* suffix is used, *arg0* usually points to *path*, and *arg1, ...., argn* point to character strings that form the new list of arguments. A mandatory null following *argn* marks the end of the list.

When the *e* suffix is used, you pass a list of new environment settings through the argument *envp*. This environment argument is an array of character pointers. Each element points to a null-terminated character string of the form

*envvar = value*

where *envvar* is the name of an environment variable, and *value* is the string value to which *envvar* is set. The last element in *envp[]* is null. When *envp* is null, the child inherits the parents' environment settings.

The combined length of *arg0 + arg1 + ... + argn* (or of *argv[0] + argv[1] + ... + argv[n]*), including space characters that separate the arguments, must be < 260 bytes. Null-terminators are not counted.

When a *spawn...* function call is made, any open files remain open in the child process.

**Return value**

On a successful execution, the *spawn...* functions where *mode* is P_WAIT return the child process' exit status (0 for a normal termination). If the child specifically calls *exit* with a nonzero argument, its exit status can be set to a nonzero value. If *mode* is P_NOWAIT or P_NOWAITO, the spawn functions return the process ID of the child process. This ID can be passed to *cwait*.

On error, the *spawn...* functions return –1, and the global variable *errno* is set to one of the following:

| | |
|---|---|
| E2BIG | Arg list too long |
| EINVAL | Invalid argument |
| ENOENT | Path or file name not found |
| ENOEXEC | Exec format error |
| ENOMEM | Not enough memory |

**See also**

*abort, atexit, cwait, _exit, exit, exec..., _fpreset, searchpath, system, wait*

## _splitpath — stdlib.h

**Function**

Splits a full path name into its components.

**Syntax**

```
void _splitpath(const char *path, char *drive, char *dir, char *name, char *ext);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks**

*_splitpath* takes a file's full path name (*path*) as a string in the form

```
X:\DIR\SUBDIR\NAME.EXT
```

and splits *path* into its four components. It then stores those components in the strings pointed to by *drive, dir, name,* and *ext*. (All five components must be passed, but any of them can be a null, which means the corresponding component will be parsed but not stored.) The maximum sizes for these strings are given by the constants _MAX_DRIVE _MAX_DIR _MAX_PATH _MAX_FNAME and _MAX_EXT) (defined in stdlib.h), and each size includes space for the null-terminator. These constants are defined in stdlib.h.

S

| Constant | String |
|---|---|
| _MAX_PATH | *path* |
| _MAX_DRIVE | *drive*; includes colon (:) |
| _MAX_DIR | *dir*; includes leading and trailing backslashes (\) |
| _MAX_FNAME | *name* |
| _MAX_EXT | *ext*; includes leading dot (.) |

*_splitpath* assumes that there is enough space to store each non-null component.

When *_splitpath* splits *path*, it treats the punctuation as follows:

- *drive* includes the colon (C:, A:, and so on).
- *dir* includes the leading and trailing backslashes (\BC\include\, \source\, and so on).
- *name* includes the file name.
- *ext* includes the dot preceding the extension (.C, .EXE, and so on).

*_makepath* and *_splitpath* are invertible; if you split a given *path* with *_splitpath*, then merge the resultant components with *_makepath*, you end up with *path*.

**Return value** None.

**See also** *_fullpath, _makepath*

## sprintf — stdio.h

**Function** Writes formatted output to a string.

**Syntax**

```
int sprintf(char *buffer, const char *format[, argument, ...]);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *sprintf* accepts a series of arguments, applies to each a format specifier contained in the format string pointed to by *format*, and outputs the formatted data to a string.

See *printf* for details on format specifiers.

*sprintf* applies the first format specifier to the first argument, the second to the second, and so on. There must be the same number of format specifiers as arguments.

**Return value** *sprintf* returns the number of bytes output. *sprintf* does not include the terminating null byte in the count. In the event of error, *sprintf* returns EOF.

**See also** *fprintf, printf*

## sqrt, sqrtl — math.h

**Function** Calculates the positive square root.

**Syntax**

```
double sqrt(double x);
long double sqrtl(long double x);
```

| | DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|---|
| *sqrt* | ■ | ■ | ■ | ■ | ■ | | ■ |
| *sqrtl* | ■ | | ■ | ■ | | | ■ |

**Remarks** *sqrt* calculates the positive square root of the argument *x*.

*sqrtl* is the **long double** version; it takes a **long double** argument and returns a **long double** result. Error handling for these functions can be modified through the functions *_matherr* and *_matherrl*.

This function can be used with *bcd* and *complex* types.

**Return value** On success, *sqrt* and *sqrtl* return the value calculated, the square root of *x*. If *x* is real and positive, the result is positive. If *x* is real and negative, the global variable *errno* is set to

EDOM Domain error

**See also** *bcd, complex, exp, log, pow*

## srand — stdlib.h

**Function** Initializes random number generator.

**Syntax**

```
void srand(unsigned seed);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** The random number generator is reinitialized by calling *srand* with an argument value of 1. It can be set to a new starting point by calling *srand* with a given *seed* number.

**Return value** None.

**See also** *rand, random, randomize*

## sscanf — stdio.h

**Function**

Scans and formats input from a string.

**Syntax**

```
int sscanf(const char *buffer, const char *format[, address, ...]);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks**

*See scanf for details on format specifiers.*

*sscanf* scans a series of input fields, one character at a time, reading from a string. Then each field is formatted according to a format specifier passed to *sscanf* in the format string pointed to by *format*. Finally, *sscanf* stores the formatted input at an address passed to it as an argument following *format*. There must be the same number of format specifiers and addresses as there are input fields.

*sscanf* might stop scanning a particular field before it reaches the normal end-of-field (whitespace) character, or it might terminate entirely, for a number of reasons. See *scanf* for a discussion of possible causes.

**Return value**

*sscanf* returns the number of input fields successfully scanned, converted, and stored; the return value does not include scanned fields that were not stored. If no fields were stored, the return value is 0.

If *sscanf* attempts to read at end-of-string, the return value is EOF.

**See also**

*fscanf, scanf*

## stackavail — malloc.h

**Function**

Gets the amount of available stack memory.

**Syntax**

```
size_t stackavail(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks**

*stackavail* returns the number of bytes available on the stack. This is the amount of dynamic memory that *alloca* can access.

**Return value**

*stackavail* returns a *size_t* value indicating the number of bytes available.

**See also**

*alloca*

## stat

See *fstat*.

## _status87 — float.h

**Function** Gets floating-point status.

**Syntax**

```
unsigned int _status87(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *_status87* gets the floating-point status word, which is a combination of the 80x87 status word and other conditions detected by the 80x87 exception handler.

**Return value** The bits in the return value give the floating-point status. See float.h for a complete definition of the bits returned by *_status87*.

## stime — time.h

**Function** Sets system date and time.

**Syntax**

```
int stime(time_t *tp);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | | | | ■ |

**Remarks** *stime* sets the system time and date. *tp* points to the value of the time as measured in seconds from 00:00:00 GMT, January 1, 1970.

**Return value** *stime* returns a value of 0.

**See also** *asctime, ftime, gettime, gmtime, localtime, time, tzset*

## stpcpy — string.h

**Function** Copies one string into another.

**Syntax**

```
char *stpcpy(char *dest, const char *src);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks** *stpcpy* copies the string *src* to *dest*, stopping after the terminating null character of *src* has been reached.

**Return value** *stpcpy* returns *dest* + *strlen*(*src*).

**See also** *strcpy*

## strcat, _fstrcat — string.h

**Function** Appends one string to another.

**Syntax**

```
char *strcat(char *dest, const char *src);
char far * far _fstrcat(char far *dest, const char far *src)
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *strcat* appends a copy of *src* to the end of *dest*. The length of the resulting string is *strlen*(*dest*) + *strlen*(*src*).

**Return value** *strcat* returns a pointer to the concatenated strings.

**See also** *_fstr**

## strchr, _fstrchr — string.h

**Function** Scans a string for the first occurrence of a given character.

**Syntax**

```
char *strchr(const char *s, int c);                          /* C only */
char far * far _fstrchr(const char far *s, int c)            /* C and C++ */
const char *strchr(const char *s, int c);                    // C++ only
char *strchr( char *s, int c);                               // C++ only
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *strchr* scans a string in the forward direction, looking for a specific character. *strchr* finds the *first* occurrence of the character *c* in the string *s*. The null-terminator is considered to be part of the string, so that, for example,

```
strchr(strs,0)
```

returns a pointer to the terminating null character of the string *strs*.

**Return value** *strchr* returns a pointer to the first occurrence of the character *c* in *s*; if *c* does not occur in *s*, *strchr* returns null.

**See also** *_fstr*, strcspn, strrchr*

## strcmp, _fstrcmp — string.h

**Function** Compares one string to another.

**Syntax**

```
int strcmp(const char *s1, const char *s2);
int far _fstrcmp(const char far *s1, const char far *s2);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *strcmp* performs an unsigned comparison of *s1* to *s2*, starting with the first character in each string and continuing with subsequent characters until the corresponding characters differ or until the end of the strings is reached.

**Return value** *strcmp* returns a value that is

- < 0 if *s1* is less than *s2*
- == 0 if *s1* is the same as *s2*
- > 0 if *s1* is greater than *s2*

**See also** _fstr*, *strcmpi, strcoll, stricmp, strncmp, strncmpi, strnicmp*

## strcmpi — string.h

**Function** Compares one string to another, without case sensitivity.

**Syntax**

```
int strcmpi(const char *s1, const char *s2);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | | ■ | | | ■ |

**Remarks**

*strcmpi* performs an unsigned comparison of *s1* to *s2*, without case sensitivity (same as *stricmp*—implemented as a macro).

It returns a value (< 0, 0, or > 0) based on the result of comparing *s1* (or part of it) to *s2* (or part of it).

The routine *strcmpi* is the same as *stricmp*. *strcmpi* is implemented through a macro in string.h and translates calls from *strcmpi* to *stricmp*. Therefore, to use *strcmpi*, you must include the header file string.h for the macro to be available. This macro is provided for compatibility with other C compilers.

**Return value**

*strcmpi* returns an **int** value that is

- < 0 if *s1* is less than *s2*
- == 0 if *s1* is the same as *s2*
- > 0 if *s1* is greater than *s2*

**See also**

*strcmp, strcoll, stricmp, strncmp, strncmpi, strnicmp*

## strcoll — string.h

**Function**

Compares two strings.

**Syntax**

```
int strcoll(char *s1, char *s2);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | ■ | ■ | ■ |

**Remarks**

*strcoll* compares the string pointed to by *s1* to the string pointed to by *s2*, according to the current locale's LC_COLLATE category.

**Return value**

*strcoll* returns a value that is

- < 0 if *s1* is less than *s2*
- == 0 if *s1* is the same as *s2*
- > 0 if *s1* is greater than *s2*

**See also**

*strcmp, strcmpi, stricmp, strncmp, strncmpi, strnicmp, strxfrm*

## strcpy, _fstrcpy — string.h

**Function** Copies one string into another.

**Syntax**

```
char *strcpy(char *dest, const char *src);
char far * far _fstrcpy(char far *dest, const char far *src);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** Copies string *src* to *dest*, stopping after the terminating null character has been moved.

**Return value** *strcpy* returns *dest*.

**See also** *_fstr*, stpcpy*

## strcspn, _fstrcspn — string.h

**Function** Scans a string for the initial segment not containing any subset of a given set of characters.

**Syntax**

```
size_t strcspn(const char *s1, const char *s2);
size_t far *far _fstrcspn(const char far *s1, const char far *s2)
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** The *strcspn* functions search *s2* until any one of the characters contained in *s1* is found. The number of characters which were read in *s2* is the return value. The string termination character is not counted. Neither string is altered during the search.

**Return value** *strcspn* returns the length of the initial segment of string *s1* that consists entirely of characters *not* from string *s2*.

**See also** _fstr*, *strchr, strrchr*

## _strdate — time.h

**Function** Converts current date to string.

**Syntax**

```
char *_strdate(char *buf);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *_strdate* converts the current date to a string, storing the string in the buffer *buf*. The buffer must be at least 9 characters long.

The string has the form `MM/DD/YY` where MM, DD, and YY are all two-digit numbers representing the month, day, and year. The string is terminated by a null character.

**Return value** *_strdate* returns *buf*, the address of the date string.

**See also** *asctime, ctime, localtime, strftime, _strtime, time*

## strdup, _fstrdup — string.h

**Function** Copies a string into a newly created location.

**Syntax**

```
char *strdup(const char *s);
char far * far _fstrdup(const char far *s)
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks** *strdup* makes a duplicate of string *s*, obtaining space with a call to *malloc*. The allocated space is (*strlen*(*s*) + 1) bytes long. The user is responsible for freeing the space allocated by *strdup* when it is no longer needed.

**Return value** *strdup* returns a pointer to the storage location containing the duplicated string, or returns null if space could not be allocated.

**See also** *free ,_fstr**

## _strerror — string.h

**Function** Builds a customized error message.

**Syntax**

```
char  *_strerror(const char *s);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *_strerror* lets you generate customized error messages; it returns a pointer to a null-terminated string containing an error message.

- If *s* is null, the return value points to the most recent error message.
- If *s* is not null, the return value contains *s* (your customized error message), a colon, a space, the most-recently generated system error message, and a new line. *s* should be 94 characters or less.

**Return value** *_strerror* returns a pointer to a constructed error string. The error message string is constructed in a static buffer that is overwritten with each call to *_strerror*.

**See also** *perror, strerror*

## strerror — string.h

**Function** Returns a pointer to an error message string.

**Syntax**

```
char *strerror(int errnum);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | ■ | ■ | ■ |

**Remarks** *strerror* takes an **int** parameter *errnum*, an error number, and returns a pointer to an error message string associated with *errnum*.

**Return value** *strerror* returns a pointer to a constructed error string. The error message string is constructed in a static buffer that is overwritten with each call to *strerror*.

**See also** *perror, _strerror*

## strftime — time.h

**Function** Formats time for output.

**Syntax**

```
size_t strftime(char *s, size_t maxsize, const char *fmt, const struct tm *t);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | ■ | ■ | ■ |

**Remarks** *strftime* formats the time in the argument *t* into the array pointed to by the argument *s* according to the *fmt* specifications. The format string consists of zero or more directives and ordinary characters. Like *printf*, a directive consists of the % character followed by a character that determines the substitution that is to take place. All ordinary characters are copied unchanged. No more than *maxsize* characters are placed in *s*.

The time is formatted according to the current locale's LC_TIME category.

The following table describes the ANSI-defined format specifiers.

| Format specifier | Substitutes |
|---|---|
| %% | Character % |
| %a | Abbreviated weekday name |
| %A | Full weekday name |
| %b | Abbreviated month name |
| %B | Full month name |
| %c | Date and time |
| %d | Two-digit day of the month (01 to 31) |
| %H | Two-digit hour (00 to 23) |
| %I | Two-digit hour (01 to 12) |
| %j | Three-digit day of the year (001 to 366) |
| %m | Two-digit month as a decimal number (1 – 12) |
| %M | Two-digit minute (00 to 59) |
| %p | AM or PM |
| %S | Two-digit second (00 to 59) |
| %U | Two-digit week number where Sunday is the first day of the week (00 to 53) |
| %w | Weekday where 0 is Sunday (0 to 6) |
| %W | Two-digit week number where Monday is the first day of the week (00 to 53) |
| %x | Date |
| %X | Time |
| %y | Two-digit year without century (00 to 99) |
| %Y | Year with century |
| %Z | Time zone name, or no characters if no time zone |

In addition to the ANSI C-defined format descriptors, the following POSIX-defined descriptors are also supported. Each format specifier begins with the percent character (%).

You must define __USELOCALES__ in order to use these descriptors.

| Format specifier | Substitutes |
|---|---|
| %C | Century as a decimal number (00-99). For example, 1992 => 19 |
| %D | Date in the format mm/dd/yy |
| %e | Day of the month as a decimal number in a two-digit field with leading space (1-31) |
| %h | A synonym for %b |
| %n | Newline character |
| %r | 12-hour time (01-12) format with am/pm string i.e. "%I:%M:%S %p" |
| %t | Tab character |
| %T | 24-hour time (00-23) in the format "HH:MM:SS" |
| %u | Weekday as a decimal number (1 Monday – 7 Sunday) |

In addition to these descriptors, *strftime* also supports the descriptor modifiers as defined by POSIX on the following descriptors:

You must define _ _USELOCALES_ _ in order to use these descriptors.

| Descriptor modifier | Substitutes |
|---|---|
| %Od | Day of the month using alternate numeric symbols |
| %Oe | Day of the month using alternate numeric symbols |
| %OH | Hour (24 hour) using alternate numeric symbols |
| %OI | Hour (12 hour) using alternate numeric symbols |
| %Om | Month using alternate numeric symbols |
| %OM | Minutes using alternate numeric symbols |
| %OS | Seconds using alternate numeric symbols |
| %Ou | Weekday as a number using alternate numeric symbols |
| %OU | Week number of the year using alternate numeric symbols |
| %Ow | Weekday as number using alternate numeric symbols |
| %OW | Week number of the year using alternate numeric symbols |
| %Oy | Year (offset from %C) using alternate numeric symbols |

%O modifier – when this modifier is used before any of the above supported numeric format descriptors, for example %Od, the numeric value is converted to the corresponding ordinal string, if it exists. If an ordinal string does not exist then the basic format descriptor is used unmodified.

For example, on 8/20/88 a %d format descriptor would produce 20 but %Od on the same day would produce 20th.

**Return value**

*strftime* returns the number of characters placed into *s*. If the number of characters required is greater than *maxsize*, *strftime* returns 0.

**See also**

*localtime, mktime, time*

## stricmp, _fstricmp — string.h



**Function**

Compares one string to another, without case sensitivity.

**Syntax**

```
int stricmp(const char *s1, const char *s2);
int far _fstricmp(const char far *s1, const char far *s2)
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks**

*stricmp* performs an unsigned comparison of *s1* to *s2*, starting with the first character in each string and continuing with subsequent characters until the corresponding characters differ or until the end of the strings is reached. The comparison is not case sensitive.

It returns a value (< 0, 0, or > 0) based on the result of comparing *s1* (or part of it) to *s2* (or part of it).

The routines *stricmp* and *strcmpi* are the same; *strcmpi* is implemented through a macro in string.h that translates calls from *strcmpi* to *stricmp*. Therefore, in order to use *strcmpi*, you must include the header file string.h for the macro to be available.

**Return value**

*stricmp* returns an **int** value that is

- < 0 if *s1* is less than *s2*
- == 0 if *s1* is the same as *s2*
- > 0 if *s1* is greater than *s2*

**See also**

*_fstr*, strcmp, strcmpi, strcoll, strncmp, strncmpi, strnicmp*

## strlen, _fstrlen — string.h

**Function**

Calculates the length of a string.

**Syntax**

```
size_t strlen(const char *s);
size_t far _fstrlen(const char far *s)
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks**

*strlen* calculates the length of *s*.

**Return value**

*strlen* returns the number of characters in *s*, not counting the null-terminating character.

**See also**

*_fstr**

## strlwr, _fstrlwr — string.h

**Function**

Converts uppercase letters in a string to lowercase.

**Syntax**

```
char *strlwr(char *s);
char far * far _fstrlwr(char char far *s)
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks**

*strlwr* converts uppercase letters in string *s* to lowercase according to the current locale's LC_CTYPE category. For the C locale, the conversion is

from uppercase letters (*A* to Z) to lowercase letters (*a* to z). No other characters are changed.

**Return value** *strlwr* returns a pointer to the string *s*.

**See also** *_fstr*, strupr*

## strncat, _fstrncat — string.h

**Function** Appends a portion of one string to another.

**Syntax**

```
char *strncat(char *dest, const char *src, size_t maxlen);
char far * far _fstrncat(char far *dest, const char far *src, size_t maxlen)
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *strncat* copies at most *maxlen* characters of *src* to the end of *dest* and then appends a null character. The maximum length of the resulting string is *strlen(dest)* + *maxlen*.

**Return value** *strncat* returns *dest*.

**See also** *_fstr**

## strncmp, _fstrncmp — string.h

**Function** Compares a portion of one string to a portion of another.

**Syntax**

```
int strncmp(const char *s1, const char *s2, size_t  maxlen);
int far _fstrncmp(const char far *s1, const char far *s2, size_t maxlen)
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *strncmp* makes the same unsigned comparison as *strcmp*, but looks at no more than *maxlen* characters. It starts with the first character in each string and continues with subsequent characters until the corresponding characters differ or until it has examined *maxlen* characters.

**Return value** *strncmp* returns an *int* value based on the result of comparing *s1* (or part of it) to *s2* (or part of it):

- < 0 if *s1* is less than *s2*
- == 0 if *s1* is the same as *s2*
- > 0 if *s1* is greater than *s2*

**See also** *_fstr*, strcmp, strcoll, stricmp, strncmpi, strnicmp*

## strncmpi — string.h

**Function**

Compares a portion of one string to a portion of another, without case sensitivity.

**Syntax**

```
int strncmpi(const char *s1, const char *s2, size_t n);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | | | | |

**Remarks**

*strncmpi* performs a signed comparison of *s1* to *s2*, for a maximum length of *n* bytes, starting with the first character in each string and continuing with subsequent characters until the corresponding characters differ or until *n* characters have been examined. The comparison is not case sensitive. (*strncmpi* is the same as *strnicmp*—implemented as a macro). It returns a value (< 0, 0, or > 0) based on the result of comparing *s1* (or part of it) to *s2* (or part of it).

The routines *strnicmp* and *strncmpi* are the same; *strncmpi* is implemented through a macro in string.h that translates calls from *strncmpi* to *strnicmp*. Therefore, in order to use *strncmpi*, you must include the header file string.h for the macro to be available. This macro is provided for compatibility with other C compilers.

**Return value**

*strncmpi* returns an **int** value that is

- < 0 if *s1* is less than *s2*
- == 0 if *s1* is the same as *s2*
- > 0 if *s1* is greater than *s2*

## strncpy, _fstrncpy — string.h

**Function**

Copies a given number of bytes from one string into another, truncating or padding as necessary.

**Syntax**

```
char *strncpy(char *dest, const char *src, size_t maxlen);
char far * far _fstrncpy(char far *dest, const char far *src, size_t maxlen)
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *strncpy* copies up to *maxlen* characters from *src* into *dest*, truncating or null-padding *dest*. The target string, *dest*, might not be null-terminated if the length of *src* is *maxlen* or more.

**Return value** *strncpy* returns *dest*.

**See also** *_fstr**

## strnicmp, _fstrnicmp — string.h

**Function** Compares a portion of one string to a portion of another, without case sensitivity.

**Syntax**

```
int strnicmp(const char *s1, const char *s2, size_t maxlen);
int far _fstrnicmp(const char far *s1, const char far *s2, size_t maxlen)
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *strnicmp* performs a signed comparison of *s1* to *s2*, for a maximum length of *maxlen* bytes, starting with the first character in each string and continuing with subsequent characters until the corresponding characters differ or until the end of the strings is reached. The comparison is not case sensitive.

It returns a value (< 0, 0, or > 0) based on the result of comparing *s1* (or part of it) to *s2* (or part of it).

**Return value** *strnicmp* returns an **int** value that is

- < 0 if *s1* is less than *s2*
- == 0 if *s1* is the same as *s2*
- > 0 if *s1* is greater than *s2*

**See also** *_fstr**

## strnset, _fstrnset — string.h

**Function** Sets a specified number of characters in a string to a given character.

**Syntax**

```
char *strnset(char *s, int ch, size_t n);
char far * far _fstrnset(char far *s, int ch, size_t n)
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *strnset* copies the character *ch* into the first *n* bytes of the string *s*. If *n* > *strlen(s)*, then *strlen(s)* replaces *n*. It stops when *n* characters have been set, or when a null character is found.

**Return value** *strnset* returns *s*.

**See also** *_fstr**

## strpbrk, _fstrpbrk — string.h

**Function** Scans a string for the first occurrence of any character from a given set.

**Syntax**

```
char *strpbrk(const char *s1, const char *s2);                    /* C only */
char far *far _fstrpbrk(const char far *s1, const char far *s2)   /* C and C++ */
const char *strpbrk(const char *s1, const char *s2);               // C++ only
char *strpbrk(char *s1, const char *s2);                           // C++ only
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *strpbrk* scans a string, *s1*, for the first occurrence of any character appearing in *s2*.

**Return value** *strpbrk* returns a pointer to the first occurrence of any of the characters in *s2*. If none of the *s2* characters occur in *s1*, *strpbrk* returns null.

**See also** *_fstr**

## strrchr, _fstrrchr — string.h

**Function** Scans a string for the last occurrence of a given character.

**Syntax**

```
char *strrchr(const char *s, int c);                    /* C only */
char far * far _fstrrchr(const char far *s, int c)      /* C and C++ */
const char *strrchr(const char *s, int c);               // C++ only
char *strrchr(char *s, int c);                           // C++ only
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *strrchr* scans a string in the reverse direction, looking for a specific character. *strrchr* finds the *last* occurrence of the character *c* in the string *s*. The null-terminator is considered to be part of the string.

**Return value** *strrchr* returns a pointer to the last occurrence of the character *c*. If *c* does not occur in *s*, *strrchr* returns null.

**See also** *_fstr*, strcspn, strchr*

## strrev, _fstrrev — string.h

**Function** Reverses a string.

**Syntax**

```
char *strrev(char *s);
char far * far _fstrrev(char far *s)
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *strrev* changes all characters in a string to reverse order, except the terminating null character. (For example, it would change *string\0* to *gnirts\0*.)

**Return value** *strrev* returns a pointer to the reversed string.

**See also** *_fstr**

## strset, _fstrset — string.h

**Function** Sets all characters in a string to a given character.

**Syntax**

```
char *strset(char *s, int ch);
char far * far _fstrset(char far *s, int ch)
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *strset* sets all characters in the string *s* to the character *ch*. It quits when the terminating null character is found.

**Return value** *strset* returns *s*.

**See also** *_fstr*, setmem*

## strspn, _fstrspn — string.h

**Function**

Scans a string for the first segment that is a subset of a given set of characters.

**Syntax**

```
size_t strspn(const char *s1, const char *s2);
size_t far _fstrspn(const char far *s1, const char far *s2);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks**

*strspn* finds the initial segment of string *s1* that consists entirely of characters from string *s2*.

**Return value**

*strspn* returns the length of the initial segment of *s1* that consists entirely of characters from *s2*.

**See also**

*_fstr**

## strstr, _fstrstr — string.h

**Function**

Scans a string for the occurrence of a given substring.

**Syntax**

```
char *strstr(const char *s1, const char *s2);                          /* C only */
char far * far _fstrstr(const char far *s1, const char far *s2);       /* C and C++ */
const char *strstr(const char *s1, const char *s2);                    // C++ only
char *strstr(char *s1, const char *s2);                                // C++ only
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks**

*strstr* scans *s1* for the first occurrence of the substring *s2*.

**Return value**

*strstr* returns a pointer to the element in *s1*, where *s2* begins (points to *s2* in *s1*). If *s2* does not occur in *s1*, *strstr* returns null.

**See also**

*_fstr**

## _strtime — time.h

**Function**

Converts current time to string.

**Syntax**

```
char *_strtime(char *buf);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *_strtime* converts the current time to a string, storing the string in the buffer *buf*. The buffer must be at least 9 characters long.

The string has the following form:

```
HH:MM:SS
```

where HH, MM, and SS are all two-digit numbers representing the hour, minute, and second, respectively. The string is terminated by a null character.

**Return value** *_strtime* returns *buf*, the address of the time string.

**See also** *asctime, ctime, localtime, strftime, _strdate, time*

## strtod, _strtold — stdlib.h

**Function** Convert a string to a **double** or **long double** value.

**Syntax**

```
double strtod(const char *s, char **endptr);
long double _strtold(const char *s, char **endptr);
```

| | DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|---|
| *strtod* | ■ | ■ | ■ | ■ | ■ | ■ | ■ |
| *_strtold* | ■ | | ■ | ■ | | | ■ |

**Remarks** *strtod* converts a character string, *s*, to a **double** value. *s* is a sequence of characters that can be interpreted as a **double** value; the characters must match this generic format:

```
[ws] [sn] [ddd] [.] [ddd] [fmt[sn]ddd]
```

where

*[ws]* = optional whitespace
*[sn]* = optional sign (+ or –)
*[ddd]* = optional digits
*[fmt]* = optional *e* or *E*
*[.]* = optional decimal point

*strtod* also recognizes +INF and –INF for plus and minus infinity, and +NAN and –NAN for Not-a-Number.

For example, here are some character strings that *strtod* can convert to **double**:

+ 1231.1981 *e*-1
502.85E2
+ 2010.952

*strtod* stops reading the string at the first character that cannot be interpreted as an appropriate part of a **double** value.

If *endptr* is not null, *strtod* sets **endptr* to point to the character that stopped the scan (**endptr* = &*stopper*). *endptr* is useful for error detection.

*_strtold* is the **long double** version; it converts a string to a **long double** value.

**Return value** These functions return the value of *s* as a **double** (*strtod*) or a **long double** (*_strtold*). In case of overflow, they return plus or minus HUGE_VAL (*strtod*) or _LHUGE_VAL (*_strtold*).

**See also** *atof*

## strtok, _fstrtok — string.h

**Function** Searches one string for tokens, which are separated by delimiters defined in a second string.

**Syntax**

```
char *strtok(char *s1, const char *s2);
char far * far _fstrtok(char far *s1, const char far *s2)
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *strtok* considers the string *s1* to consist of a sequence of zero or more text tokens, separated by spans of one or more characters from the separator string *s2*.

The first call to *strtok* returns a pointer to the first character of the first token in *s1* and writes a null character into *s1* immediately following the returned token. Subsequent calls with null for the first argument will work through the string *s1* in this way, until no tokens remain.

The separator string, *s2*, can be different from call to call.

**Return value** *strtok* returns a pointer to the token found in *s1*. A NULL pointer is returned when there are no more tokens.

**See also** *_fstr**

# strtol — stdlib.h

**Function**

Converts a string to a **long** value.

**Syntax**

```
long strtol(const char *s, char **endptr, int radix);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | ■ | ■ | ■ |

**Remarks**

*strtol* converts a character string, *s*, to a **long** integer value. *s* is a sequence of characters that can be interpreted as a **long** value; the characters must match this generic format:

```
[ws] [sn] [0] [x] [ddd]
```

where

*[ws]* = optional whitespace
*[sn]* = optional sign (+ or –)
*[0]* = optional zero (0)
*[x]* = optional x or X
*[ddd]* = optional digits

*strtol* stops reading the string at the first character it doesn't recognize.

If *radix* is between 2 and 36, the long integer is expressed in base *radix*. If *radix* is 0, the first few characters of *s* determine the base of the value being converted.

| First character | Second character | String interpreted as |
|---|---|---|
| 0 | 1 – 7 | Octal |
| 0 | *x* or *X* | Hexadecimal |
| 1 – 9 | | Decimal |

If *radix* is 1, it is considered to be an invalid value. If *radix* is less than 0 or greater than 36, it is considered to be an invalid value.

Any invalid value for *radix* causes the result to be 0 and sets the next character pointer **endptr* to the starting string pointer.

If the value in *s* is meant to be interpreted as octal, any character other than 0 to 7 will be unrecognized.

If the value in *s* is meant to be interpreted as decimal, any character other than 0 to 9 will be unrecognized.

If the value in *s* is meant to be interpreted as a number in any other base, then only the numerals and letters used to represent numbers in that base will be recognized. (For example, if *radix* equals 5, only 0 to 4 will be recognized; if *radix* equals 20, only 0 to 9 and *A* to *J* will be recognized.)

If *endptr* is not null, *strtol* sets **endptr* to point to the character that stopped the scan (**endptr* = *&stopper*).

**Return value** *strtol* returns the value of the converted string, or 0 on error.

**See also** *atoi, atol, strtoul*

## _strtold

See *strtod*.

## strtoul — stdlib.h

**Function** Converts a string to an **unsigned long** in the given radix.

**Syntax**

```
unsigned long strtoul(const char *s, char **endptr, int radix);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | ■ | ■ | ■ |

**Remarks** *strtoul* operates the same as *strtol*, except that it converts a string *str* to an **unsigned long** value (where *strtol* converts to a **long**). Refer to the entry for *strtol* for more information.

**Return value** *strtoul* returns the converted value, an **unsigned long**, or 0 on error.

**See also** *atol, strtol*

## strupr, _fstrupr — string.h

**Function** Converts lowercase letters in a string to uppercase.

**Syntax**

```
char *strupr(char *s);
char far * far _fstrupr(char far *s)
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *strupr* converts lowercase letters in string *s* to uppercase according to the current locale's LC_CTYPE category. For the default C locale, the conversion is from lowercase letters (*a* to *z*) to uppercase letters (*A* to Z). No other characters are changed.

**Return value** *strupr* returns *s*.

**See also** *_fstr*, strlwr*

## strxfrm — string.h

**Function** Transforms a portion of a string to a specified collation.

**Syntax**

```
size_t strxfrm(char *target, const char *source, size_t n);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | ■ | ■ | ■ |

**Remarks** *strxfrm* transforms the string pointed to by *source* into the string *target* for no more than *n* characters. The transformation is such that if the *strcmp* function is applied to the resulting strings, its return corresponds with the return values of the *strcoll* function.

No more than *n* characters, including the terminating null character, are copied to *target*.

*strxfrm* transforms a character string into a special string according to the current locale's LC_COLLATE category. The special string that is built can be compared with another of the same type, byte for byte, to achieve a locale-correct collation result. These special strings, which can be thought of as keys or tokenized strings, are not compatible across the different locales.

The tokens in the tokenized strings are built from the collation weights used by *strcoll* from the active locale's collation tables.

Processing stops only after all levels have been processed for the character string or the length of the tokenized string is equal to the maxlen parameter.

All redundant tokens are removed from each level's set of tokens.

The tokenized string buffer must be large enough to contain the resulting tokenized string. The length of this buffer depends on the size of the character string, the number of collation levels, the rules for each level and whether there are any special characters in the character string. Certain special characters can cause extra character processing of the string

resulting in more space requirements. For example, the French character "œ" will take double the space for itself because in some locales, it expands to two collation weights at each level. Substrings that have substitutions will also cause extra space requirements.

There is no safe formula to determine the required string buffer size, but at least (levels * string length) are required.

**Return value** Number of characters copied not including the terminating null character. If the value returned is greater than or equal to *n*, the content of *target* is indeterminate.

**See also** *strcmp, strcoll, strncpy*

## swab — stdlib.h

**Function** Swaps bytes.

**Syntax**

```
void swab(char *from, char *to, int nbytes);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks** *swab* copies *nbytes* bytes from the *from* string to the *to* string. Adjacent even- and odd-byte positions are swapped. This is useful for moving data from one machine to another machine with a different byte order. *nbytes* should be even.

**Return value** None.

## system — stdlib.h

**Function** Issue an operating system command.

**Syntax**

```
int system(const char *command);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | | ■ | | | ■ |

**Remarks** *system* invokes the operating system command processor to execute an operating system command, batch file, or other program named by the string *command*, from inside an executing C program.

To be located and executed, the program must be in the current directory or in one of the directories listed in the PATH string in the environment.

The COMSPEC environment variable is used to find the command processor program file, so that file need not be in the current directory.

**Return value**

If *command* is a NULL pointer, *system* returns nonzero if a command processor is available.

If *command* is not a NULL pointer, *system* returns 0 if the command processor was successfully started.

If an error occurred, a –1 is returned and *errno* is set to one of the following:

| | |
|---|---|
| ENOENT | Path or file function not found |
| ENOEXEC | Exec format error |
| ENOMEM | Not enough memory |

**See also**

*exec..., _fpreset, searchpath, spawn...*

## tan, tanl — math.h

**Function**

Calculates the tangent.

**Syntax**

```
double tan(double x);
long double tanl(long double x);
```

| | DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|---|
| *tan* | ■ | ■ | ■ | ■ | ■ | ■ | ■ |
| *tanl* | ■ | | ■ | ■ | | | ■ |

**Remarks**

*tan* calculates the tangent. Angles are specified in radians.

*tanl* is the **long double** version; it takes a **long double** argument and returns a **long double** result. Error handling for these routines can be modified through the functions *_matherr* and *_matherrl*.

This function can be used with *bcd* and *complex* types.

**Return value**

*tan* and *tanl* return the tangent of $x$, $sin(x)/cos(x)$.

**See also**

*acos, asin, atan, atan2, bcd, complex, cos, sin*

## tanh, tanhl — math.h

**Function**

Calculates the hyperbolic tangent.

**Syntax**

```
double tanh(double x);
long double tanhl(long double x);
```

| | DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|---|
| *tanh* | ■ | ■ | ■ | ■ | ■ | ■ | ■ |
| *tanhl* | ■ | | ■ | ■ | | | ■ |

**Remarks**

*tanh* computes the hyperbolic tangent, $sinh(x)/cosh(x)$.

*tanhl* is the **long double** version; it takes a **long double** argument and returns a **long double** result. Error handling for these functions can be modified through the functions *_matherr* and *_matherrl*.

This function can be used with *bcd* and *complex* types.

**Return value**

*tanh* and *tanhl* return the hyperbolic tangent of $x$.

**See also**

*bcd, complex, cos, cosh, sin, sinh, tan*

## tell — io.h

**Function**

Gets the current position of a file pointer.

**Syntax**

```
long tell(int handle);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks**

*tell* gets the current position of the file pointer associated with *handle* and expresses it as the number of bytes from the beginning of the file.

**Return value**

*tell* returns the current file pointer position. A return of –1 (**long**) indicates an error, and the global variable *errno* is set to

EBADF Bad file number

**See also**

*fgetpos, fseek, ftell, lseek*

## tempnam — stdio.h

**Function**

Creates a unique file name in specified directory.

**Syntax**

```
char *tempnam(char *dir, char *prefix)
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks**

The *tempnam* function creates a unique file name in arbitrary directories. The unique file is not actually created; *tempnam* only verifies that it does not currently exist. It attempts to use the following directories, in the order shown, when creating the file name:

- The directory specified by the TMP environment variable.
- The *dir* argument to *tempnam*.
- The *P_tmpdir* definition in stdio.h. If you edit stdio.h and change this definition, *tempnam* will *not* use the new definition.
- The current working directory.

If any of these directories is NULL, or undefined, or does not exist, it is skipped.

The *prefix* argument specifies the first part of the file name; it cannot be longer than 5 characters, and cannot contain a period (.). A unique file name is created by concatenating the directory name, the *prefix*, and 6 unique characters. Space for the resulting file name is allocated with *malloc*; when this file name is no longer needed, the caller should call *free* to free it.

If you do create a temporary file using the name constructed by *tempnam*, it is your responsibility to delete the file name (for example, with a call to *remove*). It is not deleted automatically. (*tmpfile does* delete the file name.)

**Return value**

If *tempnam* is successful, it returns a pointer to the unique temporary file name, which the caller can pass to *free* when it is no longer needed. Otherwise, if *tempnam* cannot create a unique file name, it returns NULL.

**See also**

*mktemp, tmpfile, tmpnam*

## textattr — conio.h



**Function**

Sets text attributes.

**Syntax**

```
void textattr(int newattr);
```

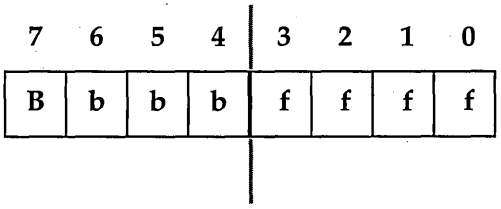
| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | | ■ | | | ■ |

**Remarks**

*textattr* lets you set both the foreground and background colors in a single call. (Normally, you set the attributes with *textcolor* and *textbackground*.)

This function does not affect any characters currently onscreen; it affects only those characters displayed by functions (such as *cprintf*) performing text mode, direct video output *after* this function is called.

The color information is encoded in the *newattr* parameter as follows:




In this 8-bit *newattr* parameter,

- *ffff* is the 4-bit foreground color (0 to 15).
- *bbb* is the 3-bit background color (0 to 7).
- *B* is the blink-enable bit.

If the blink-enable bit is on, the character blinks. This can be accomplished by adding the constant BLINK to the attribute.

If you use the symbolic color constants defined in conio.h for creating text attributes with *textattr*, note the following limitations on the color you select for the background:

- You can select only one of the first eight colors for the background.
- You must shift the selected background color left by 4 bits to move it into the correct bit positions.

These symbolic constants are listed in the following table:

| Symbolic constant | Numeric value | Foreground or background? |
|---|---|---|
| BLACK | 0 | Both |
| BLUE | 1 | Both |
| GREEN | 2 | Both |
| CYAN | 3 | Both |
| RED | 4 | Both |
| MAGENTA | 5 | Both |
| BROWN | 6 | Both |
| LIGHTGRAY | 7 | Both |
| DARKGRAY | 8 | Foreground only |
| LIGHTBLUE | 9 | Foreground only |
| LIGHTGREEN | 10 | Foreground only |
| LIGHTCYAN | 11 | Foreground only |

| Symbolic constant | Numeric value | Foreground or background? |
|---|---|---|
| LIGHTRED | 12 | Foreground only |
| LIGHTMAGENTA | 13 | Foreground only |
| YELLOW | 14 | Foreground only |
| WHITE | 15 | Foreground only |
| BLINK | 128 | Foreground only |

➥ This function should not be used in Win32s or Win32 GUI applications.

**Return value** None.

**See also** *gettextinfo, highvideo, lowvideo, normvideo, textbackground, textcolor*

## textbackground — conio.h

**Function** Selects new text background color.

**Syntax**

```
void textbackground(int newcolor);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | | ■ | | | ■ |

**Remarks** *textbackground* selects the background color. This function works for functions that produce output in text mode directly to the screen. *newcolor* selects the new background color. You can set *newcolor* to an integer from 0 to 7, or to one of the symbolic constants defined in conio.h. If you use symbolic constants, you must include conio.h.

Once you have called *textbackground*, all subsequent functions using direct video output (such as *cprintf*) will use *newcolor*. *textbackground* does not affect any characters currently onscreen.

The following table lists the symbolic constants and the numeric values of the allowable colors:

| Symbolic constant | Numeric value |
|---|---|
| BLACK | 0 |
| BLUE | 1 |
| GREEN | 2 |
| CYAN | 3 |
| RED | 4 |
| MAGENTA | 5 |
| BROWN | 6 |
| LIGHTGRAY | 7 |

This function should not be used in Win32s or Win32 GUI applications.

**Return value** None.

**See also** *gettextinfo, textattr, textcolor*

## textcolor — conio.h

**Function** Selects new character color in text mode.

**Syntax**

```
void textcolor(int newcolor);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | | ■ | | | ■ |

**Remarks** *textcolor* selects the foreground character color. This function works for the console output functions. *newcolor* selects the new foreground color. You can set *newcolor* to an integer as given in the table below, or to one of the symbolic constants defined in conio.h. If you use symbolic constants, you must include conio.h.

Once you have called *textcolor*, all subsequent functions using direct video output (such as *cprintf*) will use *newcolor*. *textcolor* does not affect any characters currently onscreen.

The following table lists the allowable colors (as symbolic constants) and their numeric values:

| Symbolic constant | Numeric value |
|---|---|
| BLACK | 0 |
| BLUE | 1 |
| GREEN | 2 |
| CYAN | 3 |
| RED | 4 |
| MAGENTA | 5 |
| BROWN | 6 |
| LIGHTGRAY | 7 |
| DARKGRAY | 8 |
| LIGHTBLUE | 9 |
| LIGHTGREEN | 10 |
| LIGHTCYAN | 11 |
| LIGHTRED | 12 |
| LIGHTMAGENTA | 13 |

| Symbolic constant | Numeric value |
|---|---|
| YELLOW | 14 |
| WHITE | 15 |
| BLINK | 128 |

You can make the characters blink by adding 128 to the foreground color. The predefined constant BLINK exists for this purpose; for example,

```
textcolor(CYAN + BLINK);
```

Some monitors do not recognize the intensity signal used to create the eight "light" colors (8-15). On such monitors, the light colors are displayed as their "dark" equivalents (0-7). Also, systems that do not display in color can treat these numbers as shades of one color, special patterns, or special attributes (such as underlined, bold, italics, and so on). Exactly what you'll see on such systems depends on your hardware.

This function should not be used in Win32s or Win32 GUI applications.

**Return value** None.

**See also** *gettextinfo, highvideo, lowvideo, normvideo, textattr, textbackground*

## textmode — conio.h

**Function** Puts screen in text mode.

**Syntax**

```
void textmode(int newmode);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | | ■ | | | ■ |

**Remarks** *textmode* selects a specific text mode.

You can give the text mode (the argument *newmode*) by using a symbolic constant from the enumeration type *text_modes* (defined in conio.h).

The most commonly used *text_modes* type constants and the modes they specify are given in the following table. Some additional values are defined in conio.h.

| Symbolic constant | Text mode |
|---|---|
| LASTMODE | Previous text mode |
| BW40 | Black and white, 40 columns |
| C40 | Color, 40 columns |

| Symbolic constant | Text mode |
|---|---|
| BW80 | Black and white, 80 columns |
| C80 | Color, 80 columns |
| MONO | Monochrome, 80 columns |
| C4350 | EGA 43-line and VGA 50-line modes |

When *textmode* is called, the current window is reset to the entire screen, and the current text attributes are reset to normal, corresponding to a call to *normvideo*.

Specifying LASTMODE to *textmode* causes the most recently selected text mode to be reselected.

*textmode* should be used only when the screen or window is in text mode (presumably to change to a different text mode). This is the only context in which *textmode* should be used. When the screen is in graphics mode, use *restorecrtmode* instead to escape temporarily to text mode.

This function should not be used in Win32s or Win32 GUI applications.

**Return value** None.

**See also** *gettextinfo, window*

## time — time.h

**Function** Gets time of day.

**Syntax**

```
time_t time(time_t *timer);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *time* gives the current time, in seconds, elapsed since 00:00:00 GMT, January 1, 1970, and stores that value in the location pointed to by *timer*, provided that *timer* is not a NULL pointer.

**Return value** *time* returns the elapsed time in seconds, as described.

**See also** *asctime, ctime, difftime, ftime, gettime, gmtime, localtime, settime, stime, tzset*

## tmpfile — stdio.h

**Function** Opens a "scratch" file in binary mode.

**Syntax**

```
FILE *tmpfile(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks**

*tmpfile* creates a temporary binary file and opens it for update (*w* + *b*). The file is automatically removed when it's closed or when your program terminates.

**Return value**

*tmpfile* returns a pointer to the stream of the temporary file created. If the file can't be created, *tmpfile* returns NULL.

**See also**

*fopen, tmpnam*

## tmpnam — stdio.h

**Function**

Creates a unique file name.

**Syntax**

```
char *tmpnam(char *s);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks**

*tmpnam* creates a unique file name, which can safely be used as the name of a temporary file. *tmpnam* generates a different string each time you call it, up to TMP_MAX times. TMP_MAX is defined in stdio.h as 65,535.

The parameter to *tmpnam*, *s*, is either null or a pointer to an array of at least *L_tmpnam* characters. *L_tmpnam* is defined in stdio.h. If *s* is NULL, *tmpnam* leaves the generated temporary file name in an internal static object and returns a pointer to that object. If *s* is not NULL, *tmpnam* places its result in the pointed-to array, which must be at least *L_tmpnam* characters long, and returns *s*.

If you do create such a temporary file with *tmpnam*, it is your responsibility to delete the file name (for example, with a call to *remove*). It is not deleted automatically. (*tmpfile does* delete the file name.)



**Return value**

If *s* is null, *tmpnam* returns a pointer to an internal static object. Otherwise, *tmpnam* returns *s*.

**See also**

*tmpfile*

## toascii — ctype.h

**Function**

Translates characters to ASCII format.

**Syntax**

```
int toascii(int c);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks**

*toascii* is a macro that converts the integer *c* to ASCII by clearing all but the lower 7 bits; this gives a value in the range 0 to 127.

**Return value**

*toascii* returns the converted value of *c*.

## _tolower — ctype.h

**Function**

Translates characters to lowercase.

**Syntax**

```
int _tolower(int ch);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks**

*_tolower* is a macro that does the same conversion as *tolower*, except that it should be used only when *ch* is known to be uppercase (*A*-*Z*).

To use *_tolower*, you must include ctype.h.

**Return value**

*_tolower* returns the converted value of *ch* if it is uppercase; otherwise, the result is undefined.

## tolower — ctype.h

**Function**

Translates characters to lowercase.

**Syntax**

```
int tolower(int ch);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks**

*tolower* is a function that converts an integer *ch* (in the range EOF to 255) to its lowercase value. The function is affected by the current locale's LC_CTYPE category. For the default C locale, *ch* is converted to a lowercase letter (*a* to *z*, if it was uppercase, *A* to Z). All others are left unchanged.

**Return value** *tolower* returns the converted value of *ch* if it is uppercase; it returns all others unchanged.

## _toupper ctype.h

**Function** Translates characters to uppercase.

**Syntax**

```
int _toupper(int ch);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks** *_toupper* is a macro that does the same conversion as *toupper*, except that it should be used only when *ch* is known to be lowercase letter (*a* to *z*).

To use *_toupper*, you must include ctype.h.

**Return value** *_toupper* returns the converted value of *ch* if it is lowercase; otherwise, the result is undefined.

## toupper ctype.h

**Function** Translates characters to uppercase.

**Syntax**

```
int toupper(int ch);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *toupper* is a function that converts an integer *ch* (in the range EOF to 255) to its uppercase value. The function is affected by the current locale's LC_CTYPE category. For the default C locale, *ch* is converted to an uppercase letter (*A* to *Z*; if it was lowercase, *a* to *z*). All others are left unchanged.

**Return value** *toupper* returns the converted value of *ch* if it is lowercase; it returns all others unchanged.

T-Z

## tzset time.h

**Function** Sets value of global variables *_daylight*, *_timezone*, and *_tzname*.

**Syntax**

```
void tzset(void)
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks**

*tzset* is available on XENIX systems.

*tzset* sets the *_daylight*, *_timezone*, and *_tzname* global variables based on the environment variable *TZ*. The library functions *ftime* and *localtime* use these global variables to adjust Greenwich Mean Time (GMT) to the local time zone. The format of the *TZ* environment string is:

```
TZ = zzz[+/-]d[d][lll]
```

where *zzz* is a three-character string representing the name of the current time zone. All three characters are required. For example, the string "PST" could be used to represent pacific standard time.

[+/-]*d*[*d*] is a required field containing an optionally signed number with 1 or more digits. This number is the local time zone's difference from GMT in hours. Positive numbers adjust westward from GMT. Negative numbers adjust eastward from GMT. For example, the number 5 = EST, +8 = PST, and –1 = continental Europe. This number is used in the calculation of the global variable *_timezone*. *_timezone* is the difference in seconds between GMT and the local time zone.

*lll* is an optional three-character field that represents the local time zone daylight saving time. For example, the string "PDT" could be used to represent pacific daylight saving time. If this field is present, it causes the global variable *_daylight* to be set nonzero. If this field is absent, *_daylight* is set to zero.

If the *TZ* environment string isn't present or isn't in the preceding form, a default *TZ* = "EST5EDT" is presumed for the purposes of assigning values to the global variables *_daylight*, *_timezone*, and *_tzname*.

The global variable *_tzname*[0] points to a three-character string with the value of the time-zone name from the *TZ* environment string. *_tzname*[1] points to a three-character string with the value of the daylight saving time-zone name from the *TZ* environment string. If no daylight saving name is present, *_tzname*[1] points to a null string.

**Return value**

None.

**See also**

*asctime*, *ctime*, *ftime*, *gmtime*, *localtime*, *stime*, *time*

## ultoa — stdlib.h

**Function** Converts an **unsigned long** to a string.

**Syntax**

```
char *ultoa(unsigned long value, char *string, int radix);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *ultoa* converts *value* to a null-terminated string and stores the result in *string*. *value* is an **unsigned long**.

*radix* specifies the base to be used in converting *value*; it must be between 2 and 36, inclusive. *ultoa* performs no overflow checking, and if *value* is negative and *radix* equals 10, it does not set the minus sign.

➥ The space allocated for *string* must be large enough to hold the returned string, including the terminating null character (\0). *ultoa* can return up to 33 bytes.

**Return value** *ultoa* returns *string*.

**See also** *itoa, ltoa*

## umask — io.h

**Function** Sets file read/write permission mask.

**Syntax**

```
unsigned umask(unsigned mode);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** The *umask* function sets the access permission mask used by *open* and *creat*. Bits that are set in *mode* will be cleared in the access permission of files subsequently created by *open* and *creat*.

The *mode* can have one of the following values, defined in sys\stat.h:

| Value of *mode* | Access permission |
|---|---|
| S_IWRITE | Permission to write |
| S_IREAD | Permission to read |
| S_IREADIS_IWRITE | Permission to read and write |

T-Z

**Return value** The previous value of the mask. There is no error return.

**See also** *creat, open*

## ungetc — stdio.h

**Function** Pushes a character back into input stream.

**Syntax**

```
int ungetc(int c, FILE *stream);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *ungetc* pushes the character *c* back onto the named input *stream*, which must be open for reading. This character will be returned on the next call to *getc* or *fread* for that *stream*. One character can be pushed back in all situations. A second call to *ungetc* without a call to *getc* will force the previous character to be forgotten. A call to *fflush, fseek, fsetpos,* or *rewind* erases all memory of any pushed-back characters.

**Return value** On success, *ungetc* returns the character pushed back; it returns EOF if the operation fails.

**See also** *fgetc, getc, getchar*

## ungetch — conio.h

**Function** Pushes a character back to the keyboard buffer.

**Syntax**

```
int ungetch(int ch);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | | ■ | | | ■ |

**Remarks** *ungetch* pushes the character *ch* back to the console, causing *ch* to be the next character read. The *ungetch* function fails if it is called more than once before the next read.

**Return value** *ungetch* returns the character *ch* if it is successful. A return value of EOF indicates an error.

➥ This function should not be used in Win32s or Win32 GUI applications.

**See also** *getch, getche*

## unixtodos — dos.h

**Function** Converts date and time from UNIX to DOS format.

**Syntax**

```
void unixtodos(long time, struct date *d, struct time *t);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *unixtodos* converts the UNIX-format time given in *time* to DOS format and fills in the *date* and *time* structures pointed to by *d* and *t*.

*time* must not represent a calendar time earlier than Jan. 1, 1980 00:00:00.

**Return value** None.

**See also** *dostounix*

## unlink — io.h

**Function** Deletes a file.

**Syntax**

```
int unlink(const char *filename);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks** *unlink* deletes a file specified by *filename*. Any drive, path, and file name can be used as a *filename*. Wildcards are not allowed.

Read-only files cannot be deleted by this call. To remove read-only files, first use *chmod* or *_rtl_chmod* to change the read-only attribute.

➥ If your file is open, be sure to close it before unlinking it.

**Return value** On successful completion, *unlink* returns 0. On error, it returns –1 and the global variable *errno* is set to one of the following values:

| | |
|---|---|
| EACCES | Permission denied |
| ENOENT | Path or file name not found |

**See also** *chmod, remove*

## unlock — io.h

**Function**

Releases file-sharing locks.

**Syntax**

```
int unlock(int handle, long offset, long length);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks**

*unlock* provides an interface to the operating system file-sharing mechanism. *unlock* removes a lock previously placed with a call to *lock*. To avoid error, all locks must be removed before a file is closed. A program must release all locks before completing.

**Return value**

*unlock* returns 0 on success, –1 on error.

**See also**

*lock, locking, sopen*

## utime — utime.h

**Function**

Sets file time and date.

**Syntax**

```
int utime(char *path, struct utimbuf *times);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks**

*utime* sets the modification time for the file *path*. The modification time is contained in the *utimbuf* structure pointed to by *times*. This structure is defined in utime.h, and has the following format:

```
struct utimbuf {
   time_t   actime;    /* access time */
   time_t   modtime;   /* modification time */
   };
```

The FAT file system supports only a modification time; therefore, on FAT file systems *utime* ignores *actime* and uses only *modtime* to set the file's modification time.

If *times* is NULL, the file's modification time is set to the current time.

**Return value**

*utime* returns 0 if it is successful. Otherwise, it returns –1, and the global variable *errno* is set to one of the following:

| | |
|---|---|
| EACCES | Permission denied |
| EMFILE | Too many open files |
| ENOENT | Path or file name not found |

**See also** *setftime, stat, time*

## va_arg, va_end, va_start — stdarg.h

**Function** Implement a variable argument list.

**Syntax**

```
void va_start(va_list ap, lastfix);
type va_arg(va_list ap, type);
void va_end(va_list ap);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** Some C functions, such as *vfprintf* and *vprintf*, take variable argument lists in addition to taking a number of fixed (known) parameters. The *va_arg*, *va_end*, and *va_start* macros provide a portable way to access these argument lists. They are used for stepping through a list of arguments when the called function does not know the number and types of the arguments being passed.

The header file stdarg.h declares one type (***va_list***) and three macros (*va_start*, *va_arg*, and *va_end*).

- ***va_list***: This array holds information needed by *va_arg* and *va_end*. When a called function takes a variable argument list, it declares a variable *ap* of type ***va_list***.
- *va_start*: This routine (implemented as a macro) sets *ap* to point to the first of the variable arguments being passed to the function. *va_start* must be used before the first call to *va_arg* or *va_end*.
- *va_start* takes two parameters: *ap* and *lastfix*. (*ap* is explained under *va_list* in the preceding paragraph; *lastfix* is the name of the last fixed parameter being passed to the called function.)
- *va_arg*: This routine (also implemented as a macro) expands to an expression that has the same type and value as the next argument being passed (one of the variable arguments). The variable *ap* to *va_arg* should be the same *ap* that *va_start* initialized.

➥ Because of default promotions, you can't use **char**, **unsigned char**, or **float** types with *va_arg*.

The first time *va_arg* is used, it returns the first argument in the list. Each successive time *va_arg* is used, it returns the next argument in the list. It does this by first dereferencing *ap*, and then incrementing *ap* to point to the following item. *va_arg* uses the *type* to both perform the dereference and to locate the following item. Each successive time *va_arg* is invoked, it modifies *ap* to point to the next argument in the list.

- *va_end*: This macro helps the called function perform a normal return. *va_end* might modify *ap* in such a way that it cannot be used unless *va_start* is recalled. *va_end* should be called after *va_arg* has read all the arguments; failure to do so might cause strange, undefined behavior in your program.

**Return value**

*va_start* and *va_end* return no values; *va_arg* returns the current argument in the list (the one that *ap* is pointing to).

**See also**

*v...printf, v...scanf*

## vfprintf — stdio.h

**Function**

Writes formatted output to a stream.

**Syntax**

```
int vfprintf(FILE *stream, const char *format, va_list arglist);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks**

The *v...printf* functions are known as *alternate entry points* for the *...printf* functions. They behave exactly like their *...printf* counterparts, but they accept a pointer to a list of arguments instead of an argument list.

See *printf* for details on format specifiers.

*vfprintf* accepts a pointer to a series of arguments, applies to each argument a format specifier contained in the format string pointed to by *format*, and outputs the formatted data to a stream. There must be the same number of format specifiers as arguments.

**Return value**

*vfprintf* returns the number of bytes output. In the event of error, *vfprintf* returns EOF.

**See also**

*printf, va_arg, va_end, va_start*

## vfscanf — stdio.h

**Function**

Scans and formats input from a stream.

**Syntax**

```
int vfscanf(FILE *stream, const char *format, va_list arglist);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks**

The *v...scanf* functions are known as *alternate entry points* for the *...scanf* functions. They behave exactly like their *...scanf* counterparts, but they accept a pointer to a list of arguments instead of an argument list.

See *scanf* for details on format specifiers.

*vfscanf* scans a series of input fields, one character at a time, reading from a stream. Then each field is formatted according to a format specifier passed to *vfscanf* in the format string pointed to by *format*. Finally, *vfscanf* stores the formatted input at an address passed to it as an argument following *format*. There must be the same number of format specifiers and addresses as there are input fields.

*vfscanf* might stop scanning a particular field before it reaches the normal end-of-field (whitespace) character, or it might terminate entirely, for a number of reasons. See *scanf* for a discussion of possible causes.

**Return value**

*vfscanf* returns the number of input fields successfully scanned, converted, and stored; the return value does not include scanned fields that were not stored. If no fields were stored, the return value is 0.

If *vfscanf* attempts to read at end-of-file, the return value is EOF.

**See also**

*fscanf, scanf, va_arg, va_end, va_start*

## vprintf — stdarg.h

**Function**

Writes formatted output to stdout.

**Syntax**

```
int vprintf(const char *format, va_list arglist);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | | ■ | ■ | | ■ |

**Remarks**

The *v...printf* functions are known as *alternate entry points* for the *...printf* functions. They behave exactly like their *...printf* counterparts, but they accept a pointer to a list of arguments instead of an argument list.

See *printf* for details on format specifiers.

*vprintf* accepts a pointer to a series of arguments, applies to each a format specifier contained in the format string pointed to by *format*, and outputs the formatted data to stdout. There must be the same number of format specifiers as arguments.

T-Z

When you use the SS!=DS flag in 16-bit applications, *vprintf* assumes that the address being passed is in the SS segment.

For Win32s or Win32 GUI applications, stdout must be redirected.

**Return value**

*vprint* returns the number of bytes output. In the event of error, *vprint* returns EOF.

**See also**

*freopen, printf, va_arg, va_end, va_start*

## vscanf — stdarg.h

**Function**

Scans and formats input from stdin.

**Syntax**

```
int vscanf(const char *format, va_list arglist);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks**

The *v...scanf* functions are known as *alternate entry points* for the *...scanf* functions. They behave exactly like their *...scanf* counterparts, but they accept a pointer to a list of arguments instead of an argument list.

See *scanf* for details on format specifiers.

*vscanf* scans a series of input fields, one character at a time, reading from stdin. Then each field is formatted according to a format specifier passed to *vscanf* in the format string pointed to by *format*. Finally, *vscanf* stores the formatted input at an address passed to it as an argument following *format*. There must be the same number of format specifiers and addresses as there are input fields.

*vscanf* might stop scanning a particular field before it reaches the normal end-of-field (whitespace) character, or it might terminate entirely, for a number of reasons. See *scanf* for a discussion of possible causes.

For Win32s or Win32 GUI applications, stdin must be redirected.

**Return value**

*vscanf* returns the number of input fields successfully scanned, converted, and stored; the return value does not include scanned fields that were not stored. If no fields were stored, the return value is 0.

If *vscanf* attempts to read at end-of-file, the return value is EOF.

**See also**

*freopen, fscanf, scanf, va_arg, va_end, va_start*

## vsprintf — stdarg.h

**Function**

Writes formatted output to a string.

**Syntax**

```
int vsprintf(char *buffer, const char *format, va_list arglist);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks**

The *v...printf* functions are known as *alternate entry points* for the *...printf* functions. They behave exactly like their *...printf* counterparts, but they accept a pointer to a list of arguments instead of an argument list.

See *printf* for details on format specifiers.

*vsprintf* accepts a pointer to a series of arguments, applies to each a format specifier contained in the format string pointed to by *format*, and outputs the formatted data to a string. There must be the same number of format specifiers as arguments.

**Return value**

*vsprintf* returns the number of bytes output. In the event of error, *vsprintf* returns EOF.

**See also**

*printf, va_arg, va_end, va_start*

## vsscanf — stdarg.h

**Function**

Scans and formats input from a stream.

**Syntax**

```
int vsscanf(const char *buffer, const char *format, va_list arglist);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks**

The *v...scanf* functions are known as *alternate entry points* for the *...scanf* functions. They behave exactly like their *...scanf* counterparts, but they accept a pointer to a list of arguments instead of an argument list.

See *scanf* for details on format specifiers.

*vsscanf* scans a series of input fields, one character at a time, reading from a stream. Then each field is formatted according to a format specifier passed to *vsscanf* in the format string pointed to by *format*. Finally, *vsscanf* stores the formatted input at an address passed to it as an argument following *format*. There must be the same number of format specifiers and addresses as there are input fields.

*vsscanf* might stop scanning a particular field before it reaches the normal end-of-field (whitespace) character, or it might terminate entirely, for a number of reasons. See *scanf* for a discussion of possible causes.

**Return value**

*vsscanf* returns the number of input fields successfully scanned, converted, and stored; the return value does not include scanned fields that were not stored. If no fields were stored, the return value is 0.

If *vsscanf* attempts to read at end-of-string, the return value is EOF.

**See also**

*fscanf, scanf, sscanf, va_arg, va_end, va_start, vfscanf*

## wait — process.h

**Function**

Waits for one or more child processes to terminate.

**Syntax**

```
int wait(int *statloc);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| | | | ■ | | | ■ |

**Remarks**

The *wait* function waits for one or more child processes to terminate. The child processes must be those created by the calling program; *wait* cannot wait for grandchildren (processes spawned by child processes). If *statloc* is not NULL, it points to location where *wait* will store the termination status. If the child process terminated normally (by calling *exit*, or returning from *main*), the termination status word is defined as follows:

**Bits 0-7** Zero.

**Bits 8-15** The least significant byte of the return code from the child process. This is the value that is passed to *exit*, or is returned from *main*. If the child process simply exited from *main* without returning a value, this value will be unpredictable.

If the child process terminated abnormally, the termination status word is defined as follows:

**Bits 0-7** Termination information about the child:

1 Critical error abort.
2 Execution fault, protection exception.
3 External termination signal.

**Bits 8-15** Zero.

**Return value**

When *wait* returns after a normal child process termination it returns the process ID of the child.

When *wait* returns after an abnormal child termination it returns –1 to the parent and sets *errno* to EINTR.

If *wait* returns without a child process completion it returns a –1 value and sets *errno* to

ECHILD No child process exists

**See also** *cwait, spawn*

## wcstombs — stdlib.h

**Function** Converts a wchar_t array into a multibyte string.

**Syntax**

```
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** *wcstombs* converts the type wchar_t elements contained in *pwcs* into a multibyte character string *s*. The process terminates if either a null character or an invalid multibyte character is encountered.

No more than *n* bytes are modified. If *n* number of bytes are processed before a null character is reached, the array *s* is not null terminated.

The behavior of *wcstombs* is affected by the setting of LC_CTYPE category of the current locale.

**Return value** If an invalid multibyte character is encountered, *wcstombs* returns (size_t) –1. Otherwise, the function returns the number of bytes modified, not including the terminating code, if any.

## wctomb — stdlib.h

T-Z

**Function** Converts wchar_t code to a multibyte character.

**Syntax**

```
int wctomb(char *s, wchar_t wc);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ |

**Remarks** If *s* is not null, *wctomb* determines the number of bytes needed to represent the multibyte character corresponding to *wc* (including any change in shift state). The multibyte character is stored in *s*. At most *MB_CUR_MAX*

characters are stored. If the value of *wc* is zero, *wctomb* is left in the initial state.

The behavior of *wctomb* is affected by the setting of LC_CTYPE category of the current locale.

**Return value**

If *s* is a NULL pointer, *wctomb* returns a nonzero value if multibyte character encodings do have state-dependent encodings, and a zero value if they do not.

If *s* is not a NULL pointer, *wctomb* returns –1 if the *wc* value does not represent a valid multibyte character. Otherwise, *wctomb* returns the number of bytes that are contained in the multibyte character corresponding to *wc*. In no case will the return value be greater than the value of *MB_CUR_MAX* macro.

## wherex — conio.h

**Function**

Gives horizontal cursor position within window.

**Syntax**

```
int wherex(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks**

*wherex* returns the x-coordinate of the current cursor position (within the current text window).

➥ This function should not be used in Win32s or Win32 GUI applications.

**Return value**

*wherex* returns an integer in the range 1 to the number of columns in the current video mode.

**See also**

*gettextinfo, gotoxy, wherey*

## wherey — conio.h

**Function**

Gives vertical cursor position within window.

**Syntax**

```
int wherey(void);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *wherey* returns the y-coordinate of the current cursor position (within the current text window).

➥ This function should not be used in Win32s or Win32 GUI applications.

**Return value** *wherey* returns an integer in the range 1 to the number of rows in the current video mode.

**See also** *gettextinfo, gotoxy, wherex*

## window — conio.h

**Function** Defines active text mode window.

**Syntax**

```
void window(int left, int top, int right, int bottom);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | | ■ | | | ■ |

**Remarks** *window* defines a text window onscreen. If the coordinates are in any way invalid, the call to *window* is ignored.

*left* and *top* are the screen coordinates of the upper left corner of the window. *right* and *bottom* are the screen coordinates of the lower right corner.

The minimum size of the text window is one column by one line. The default window is full screen, with the coordinates:

1,1,C,R

where C is the number of columns in the current video mode, and R is the number of rows.

➥ This function should not be used in Win32s or Win32 GUI applications.

**Return value** None.

**See also** *clreol, clrscr, delline, gettextinfo, gotoxy, insline, puttext, textmode*

T-Z

## _write — io.h

**Remarks** Obsolete function. See *_rtl_write.*

# write — io.h

**Function**

Writes to a file.

**Syntax**

```
int write(int handle, void *buf, unsigned len);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | | | ■ |

**Remarks**

*write* writes a buffer of data to the file or device named by the given *handle*. *handle* is a file handle obtained from a *creat*, *open*, *dup*, or *dup2* call.

This function attempts to write *len* bytes from the buffer pointed to by *buf* to the file associated with *handle*. Except when *write* is used to write to a text file, the number of bytes written to the file will be no more than the number requested. The maximum number of bytes that *write* can write is UINT_MAX –1, because UINT_MAX is the same as –1, which is the error return indicator for *write*. On text files, when *write* sees a linefeed (LF) character, it outputs a CR/LF pair. UINT_MAX is defined in limits.h.

If the number of bytes actually written is less than that requested, the condition should be considered an error and probably indicates a full disk. For disks or disk files, writing always proceeds from the current file pointer. For devices, bytes are sent directly to the device. For files opened with the O_APPEND option, the file pointer is positioned to EOF by *write* before writing the data.

**Return value**

*write* returns the number of bytes written. A *write* to a text file does not count generated carriage returns. In case of error, *write* returns –1 and sets the global variable *errno* to one of the following values:

| | |
|---|---|
| EACCES | Permission denied |
| EBADF | Bad file number |

**See also**

*creat*, *lseek*, *open*, *read*, *_rtl_write*

CHAPTER 4

# Global variables

Borland C++ provides you with predefined global variables for many common needs, such as dates, times, command-line arguments, and so on. This chapter defines and describes them.

## _8087 — dos.h

**Function**

Coprocessor chip flag.

**Syntax**

```
extern int _8087;
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | |

**Remarks**

In a 16-bit Windows program, the value is 1 if any coprocessor is detected.

The *_8087* variable is set to a nonzero value (1, 2, or 3) if the startup code autodetection logic detects a floating-point coprocessor (an 8087, 80287, or 80387, respectively). The *_8087* variable is set to 0 otherwise.

The autodetection logic can be overridden by setting the 87 environment variable to `YES` or `NO`. (The commands are `SET 87=YES` and `SET 87=NO`; it is essential that there be no spaces before or after the equal sign.) If you use the *87* environment variable, the *_8087* variable will reflect the override.

Refer to Chapter 8 in the *Programmer's Guide* for more information about the 87 environment variable.

## _argc — dos.h

**Function**

Keeps a count of command-line arguments.

**Syntax**

```
extern int _argc;
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *_argc* has the value of *argc* passed to *main* when the program starts.

## _argv — dos.h

**Function** An array of pointers to command-line arguments.

**Syntax**

```
extern char **_argv;
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *_argv* points to an array containing the original command-line arguments (the elements of *argv[]*) passed to *main* when the program starts.

## _ctype — ctype.h

**Function** An array of character attribute information.

**Syntax**

```
extern char _ctype[];
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *_ctype* is an array of character attribute information indexed by ASCII value + 1. Each entry is a set of bits describing the character.

This array is used only by routines affected by the C locale, such as *isdigit*, *isprint*, and so on.

## _daylight — time.h

**Function** Indicates whether daylight saving time adjustments will be made.

**Syntax**

```
extern int _daylight;
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *_daylight* is used by the time and date functions. It is set by the *tzset*, *ftime*, and *localtime* functions to 1 for daylight saving time, 0 for standard time.

**See also** *_timezone*

## _directvideo — conio.h

**Function**

Flag that controls video output.

**Syntax**

```
extern int _directvideo;
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | |

**Remarks**

*_directvideo* controls whether your program's console output (from *cputs*, for example) goes directly to the video RAM (*_directvideo* = 1) or goes via ROM BIOS calls (*_directvideo* = 0).

The default value is *_directvideo* = 1 (console output goes directly to video RAM). To use *_directvideo* = 1, your system's video hardware must be identical to IBM display adapters. Setting *_directvideo* = 0 allows your console output to work on any system that is IBM BIOS-compatible.

*_directvideo* should be used only in character-based applications. It should not be used in 16-bit Windows, Win32s, or Win32 GUI applications.

## _environ — dos.h

**Function**

Accesses the operating system environment variables.

**Syntax**

```
extern char ** _environ;
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks**

*_environ* is an array of pointers to strings; it is used to access and alter the operating system environment variables. Each string is of the form

*envvar = varvalue*

where *envvar* is the name of an environment variable (such as `PATH`), and *varvalue* is the string value to which *envvar* is set (such as `C:\BIN;C:\DOS`). The string *varvalue* can be empty.

When a program begins execution, the operating system environment settings are passed directly to the program. Note that *env*, the third argument to *main*, is equal to the initial setting of *_environ*.

The *_environ* array can be accessed by *getenv*; however, the *putenv* function is the only routine that should be used to add, change or delete the *_environ* array entries. This is because modification can resize and relocate the

process environment array, but *_environ* is automatically adjusted so that it always points to the array.

**See also** *getenv, putenv*

## errno, _doserrno, _sys_errlist, _sys_nerr — dos.h, errno.h

**Function** Enable *perror* to print error messages.

**Syntax**

```
extern int _doserrno;
extern int errno;
extern char **_sys_errlist;
extern int _sys_nerr;
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *errno*, *_sys_errlist*, and *_sys_nerr* are used by *perror* to print error messages when certain library routines fail to accomplish their appointed tasks. *_doserrno* is a variable that maps many operating-system error codes to *errno*; however, *perror* does not use *_doserrno* directly. See the header files winbase.h and winerror.h for the list of operating-system errors.

- *errno*: When an error in a math or system call occurs, *errno* is set to indicate the type of error. Sometimes *errno* and *_doserrno* are equivalent. At other times, *errno* does not contain the actual operating system error code, which is contained in *_doserrno* instead. Still other errors might occur that set only *errno*, not *_doserrno*.
- *_doserrno*: When an operating-system call results in an error, *_doserrno* is set to the actual operating-system error code. *errno* is a parallel error variable inherited from UNIX.
- *_sys_errlist*: To provide more control over message formatting, the array of message strings is provided in *_sys_errlist*. You can use *errno* as an index into the array to find the string corresponding to the error number. The string does not include any newline character.
- *_sys_nerr*: This variable is defined as the number of error message strings in *_sys_errlist*.

The following table gives mnemonics and their meanings for the values stored in *_sys_errlist*. The list is alphabetically ordered for easier reading. For the numerical ordering, see the header file errno.h.

| Mnemonic | 16-bit description | 32-bit description |
|---|---|---|
| E2BIG | Arg list too long | Arg list too long |
| EACCES | Permission denied | Permission denied |
| EBADF | Bad file number | Bad file number |
| ECHILD | | No child process |
| ECONTR | Memory blocks destroyed | Memory blocks destroyed |
| ECURDIR | Attempt to remove CurDir | Attempt to remove CurDir |
| EDEADLOCK | | Locking violation |
| EDOM | Domain error | Math argument |
| EEXIST | File already exists | File already exists |
| EFAULT | Unknown error | Unknown error |
| EINTR | | Interrupted function call |
| EINVACC | Invalid access code | Invalid access code |
| EINVAL | Invalid argument | Invalid argument |
| EINVDAT | Invalid data | Invalid data |
| EINVDRV | Invalid drive specified | Invalid drive specified |
| EINVENV | Invalid environment | Invalid environment |
| EINVFMT | Invalid format | Invalid format |
| EINVFNC | Invalid function number | Invalid function number |
| EINVMEM | Invalid memory block address | Invalid memory block address |
| EIO | | Input/Output error |
| EMFILE | Too many open files | Too many open files |
| ENAMETOOLONG | | File name too long |
| ENFILE | | Too many open files |
| ENMFILE | No more files | No more files |
| ENODEV | No such device | No such device |
| ENOENT | No such file or directory | No such file or directory |
| ENOEXEC | Exec format error | Exec format error |
| ENOFILE | No such file or directory | File not found |
| ENOMEM | Not enough memory | Not enough core |
| ENOPATH | Path not found | Path not found |
| ENOSPC | | No space left on device |
| ENOTSAM | Not same device | Not same device |
| ENXIO | | No such device or address |
| EPERM | | Operation not permitted |
| EPIPE | | Broken pipe |
| ERANGE | Result out of range | Result too large |
| EROFS | | Read-only filesystem |
| ESPIPE | | Illegal seek |
| EXDEV | Cross-device link | Cross-device link |
| EZERO | Error 0 | Error 0 |

The following list gives mnemonics for the actual DOS error codes to which *_doserrno* can be set. (This value of *_doserrno* may or may not be mapped (through *errno*) to an equivalent error message string in *_sys_errlist*.

| Mnemonic | DOS error code |
|---|---|
| E2BIG | Bad environ |
| EACCES | Access denied |
| EACCES | Bad access |
| EACCES | Is current dir |
| EBADF | Bad handle |
| EFAULT | Reserved |
| EINVAL | Bad data |
| EINVAL | Bad function |
| EMFILE | Too many open |
| ENOENT | No such file or directory |
| ENOEXEC | Bad format |
| ENOMEM | Out of memory |
| ENOMEM | Bad block |
| EXDEV | Bad drive |
| EXDEV | Not same device |

Refer to your DOS reference manual for more information about DOS error return codes.

## _floatconvert — stdio.h

**Function**

Links the floating-point formats.

**Syntax**

```
extern int _floatconvert;
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks**

Floating-point output requires linking of conversion routines used by *printf*, *scanf*, and any variants of these functions. To reduce executable size, the floating-point formats are not automatically linked. However, this linkage is done automatically whenever your program uses a mathematical routine or the address is taken of some floating-point number. If neither of these actions occur the missing floating-point formats can result in a run-time error.

The following program illustrates how to set up your program to properly execute.

```
/* PREPARE TO OUTPUT FLOATING-POINT NUMBERS. */
#include <stdio.h>

#pragma extref _floatconvert

void main() {
    printf("d = %lf\n", 1);
}
```

## _fmode

**fcntl.h**

**Function**

Determines default file-translation mode.

**Syntax**

```
extern int _fmode;
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks**

*_fmode* determines in which mode (text or binary) files will be opened and translated. The value of *_fmode* is O_TEXT by default, which specifies that files will be read in text mode. If *_fmode* is set to O_BINARY, the files are opened and read in binary mode. (O_TEXT and O_BINARY are defined in fcntl.h.)

In text mode, carriage-return/linefeed (CR/LF) combinations are translated to a single linefeed character (LF) on input. On output, the reverse is true: LF characters are translated to CR/LF combinations.

In binary mode, no such translation occurs.

You can override the default mode as set by *_fmode* by specifying a *t* (for text mode) or *b* (for binary mode) in the argument *type* in the library functions *fopen*, *fdopen*, and *freopen*. Also, in the function *open*, the argument *access* can include either O_BINARY or O_TEXT, which will explicitly define the file being opened (given by the *open pathname* argument) to be in either binary or text mode.

## _new_handler

**Function**

Traps new allocation miscues.

**Syntax**

```
typedef void (*pvf)();
pvf _new_handler;
```

As an alternative, you can set using the function *set_new_handler*, like this:

```
pvf set_new_handler(pvf p);
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks**

*_new_handler* contains a pointer to a function that takes no arguments and returns **void**. If **operator new()** is unable to allocate the space required, it will call the function pointed to by *_new_handler*; if that function returns it will try the allocation again. By default, the function pointed to by *_new_handler* terminates the application. The application can replace this handler, however, with a function that can try to free up some space. This is done by assigning directly to *_new_handler* or by calling the function *set_new_handler*, which returns a pointer to the former handler.

*_new_handler* is provided primarily for compatibility with C++ version 1.2. In most cases this functionality can be better provided by overloading **operator new()**.

## _osmajor, _osminor, _osversion — dos.h

**Function**

Contain the major and minor operating-system version numbers.

**Syntax**

```
extern unsigned char _osmajor;
extern unsigned char _osminor;
extern unsigned _osversion;
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks**

The major and minor version numbers are available individually through *_osmajor* and *_osminor*. *_osmajor* is the major version number, and *_osminor* is the minor version number. For example, if you are running DOS version 3.2, *_osmajor* will be 3 and *_osminor* will be 20.

*_osversion* is functionally identical to *_version*. See the discussion of *_version*.

These variables can be useful when you want to write modules that will run on DOS versions 2.x and 3.x. Some library routines behave differently depending on the DOS version number; other routines work under DOS 3.x only. (For example, refer to *_rtl_open*, *creatnew*, and *ioctl* in this book.)

## _psp — dos.h

**Function**

Contains the segment address of the program segment prefix (PSP) for the current program.

**Syntax**

```
extern unsigned int _psp;
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks**

The PSP is a DOS process descriptor; it contains initial DOS information about the program.

Refer to the *DOS Programmer's Reference Manual* for more information on the PSP.

## _threadid — stddef.h

**Function**

Pointer to thread ID.

**Syntax**

```
extern long _threadid;
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | | ■ | | | ■ |

**Remarks**

*_threadid* is a long integer that contains the ID of the currently executing thread. It is implemented as a macro, and should be declared only by including stddef.h.

## _ _throwExceptionName, _ _throwFileName, _ _throwLineNumber — except.h

**Function**

Generates information about a thrown exception.

**Syntax**

```
extern char * _ _throwExceptionName;
extern char * _ _throwFileName;
extern char * _ _throwLineNumber;
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks**

Use these global variables to get the name and location of a thrown exception. The output for each of the variables is a printable character string.

To get the file name and line number for a thrown exception with *__throwFileName* and *__throwLineNumber*, you must compile the module with the **–xp** compiler option.

## _timezone — time.h

**Function** Contains difference in seconds between local time and GMT.

**Syntax**

```
extern long _timezone;
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** *_timezone* is used by the time-and-date functions.

This variable is calculated by then *tzset* function; it is assigned a long value that is the difference, in seconds, between the current local time and Greenwich mean time.

**See also** *_daylight*

## _tzname — time.h

**Function** Array of pointers to time-zone names.

**Syntax**

```
extern char * _tzname[2]
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks** The global variable *_tzname* is an array of pointers to strings containing abbreviations for time-zone names. *_tzname*[0] points to a three-character string with the value of the time-zone name from the *TZ* environment string. The global variable *_tzname*[1] points to a three-character string with the value of the daylight-saving time-zone name from the *TZ* environment string. If no daylight saving name is present, *_tzname*[1] points to a null string.

## _version — dos.h

**Function** Contains the operating-system version number.

**Syntax**

```
extern unsigned _version;
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks**

*_version* contains the operating-system version number, with the major version number in the high byte and the minor version number in the low byte. For a 32-bit application, this layout of the version number is in the low word. (For DOS version *x.y*, the *x* is the major version number, and *y* is the minor.)

## _wscroll — conio.h

**Function**

Enables or disables scrolling in console I/O functions.

**Syntax**

```
extern int _wscroll
```

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

**Remarks**

*_wscroll* is a console I/O flag. Its default value is 1. If you set *_wscroll* to 0, scrolling is disabled. This can be useful for drawing along the edges of a window without having your screen scroll.

*_wscroll* should be used only in character-based applications. It is available for EasyWin but it should not be used in any GUI application.

C H A P T E R 5

# The C++ iostream classes

Online help provides sample programs for many iostream classes.

The stream class library in C++ consists of several classes distributed in two separate hierarchical trees. See the *Programmer's Guide*, Chapter 6, for an illustration of the class hierarchies. This reference presents some of the most useful details of these classes, in alphabetical order. The following cross-reference table tells which classes belong to which header files.

| Header file | Classes |
|---|---|
| constrea.h | *conbuf*, *constream* (These classes are available only for console-mode applications.) |
| iostream.h | *ios*, *iostream*, *iostream_withassign*, *istream*, *istream_withassign*, *ostream*, *ostream_withassign*, *streambuf* |
| fstream.h | *filebuf*, *fstream*, *fstreambase*, *ifstream*, *ofstream* |
| strstrea.h | *istrstream*, *ostrstream*, *strstream*, *strstreambase*, *strstreambuf* |

## conbuf class — constrea.h

conbuf is available only for console-mode applications.

Specializes *streambuf* to handle console output.

### Public constructor

**Constructor**

```
conbuf()
```

Makes an unattached *conbuf*.

### Public member functions

**clreol**

```
void clreol()
```

Clears to end of line in text window.

**clrscr**

```
void clrscr()
```

Clears the defined screen.

**delline**

```
void delline()
```

Deletes a line in the window.

**gotoxy**

```
void gotoxy(int x, int y)
```

Positions the cursor in the window at the specified location.

**highvideo**

```
void highvideo()
```

Selects high-intensity characters.

**insline**

```
void insline()
```

Inserts a blank line.

**lowvideo**

```
void lowvideo()
```

Selects low-intensity characters.

**normvideo**

```
void normvideo()
```

Selects normal-intensity characters.

**overflow**

```
virtual int overflow( int = EOF )
```

Flushes the conbuf to its destination.

**setcursortype**

```
void setcursortype(int cur_type)
```

Selects the cursor appearance.

**textattr**

```
void textattr(int newattribute)
```

Selects cursor appearance.

**textbackground**

```
void textbackground(int newcolor)
```

Selects the text background color.

**textcolor**

```
void textcolor( int newcolor)
```

Selects character color in text mode.

**textmode**

```
static void textmode(int newmode)
```

Puts the screen in text mode.

**wherex**

```
int wherex()
```

Gets the horizontal cursor position.

**wherey**

```
int wherey()
```

Gets the vertical cursor position.

**window**

```
void window(int left, int top, int right, int bottom)
```

Defines the active window.

## constream class — constrea.h

constream is available only for console-mode applications.

Provides console output streams. This class is derived from *ostream*.

### Public constructor

**Constructor**

```
constream()
```

Provides an unattached output stream to the console.

### Public member functions

**clrscr**

```
void clrscr()
```

Clears the screen.

**rdbuf**

```
conbuf *rdbuf()
```

Returns a pointer to this constream's assigned conbuf.

**textmode**

```
void textmode(int newmode)
```

Puts the screen in text mode.

**window**

```
void window(int left, int top, int right, int bottom)
```

Defines the active window.

## filebuf class — fstream.h

Specializes *streambuf* to use files for input and output of characters. The *filebuf* class manages buffer allocation and deletion, and seeking within a file. This class also permits unbuffered file I/O by using the appropriate constructor or the member function *filebuf*::*setbuf*. By default, files are opened in *openprot* mode to allow reading and writing. See page 319 for a list of file-opening modes.

The *filebuf* class only provides basic services for file I/O. Input and output to a filebuf can only be done with the low-level functions provided by *streambuf*. Higher level classes provide formatting services.

## Public constructors

**Constructor**

```
filebuf();
```

Makes a *filebuf* that isn't attached to a file.

```
filebuf(int fd);
```

Makes a *filebuf* attached to a file as specified by file descriptor *fd*.

**Constructor**

```
filebuf(int fd, char *buf, int n);
```

Makes a *filebuf* attached to a file specified by the file descriptor *fd*, and uses *buf* as the storage area. The size of *buf* is sufficient to store *n* bytes. If *buf* is NULL or *n* is non-positive, the *filebuf* is unbuffered.

## Public data members

**openprot**

```
static const int openprot
```

The default file protection. The exact value of *openprot* should not be of interest to the user. Its purpose is to set the file permissions to read and write.

## Public member functions

**attach**

```
filebuf* attach(int fd)
```

Connects this closed *filebuf* to a file specified by the file descriptor *fd*. If the file buffer is already open, *attach* fails and returns NULL. Otherwise, the file buffer is connected to *fd*.

**close**

```
filebuf* close()
```

Flushes and closes the file. Generally, it is not necessary to make an explicit call to *close* at your program's end because proper file closing is ensured by the *filebuf* destructor. An explicit call to *close* is useful when you want to disconnect the *filebuf* from your program.

Returns 0 on error, for example, if the file was already closed. Otherwise, the function returns a reference to the *filebuf* (the **this** pointer).

**fd**

```
int fd()
```

Returns the file descriptor or EOF.

**is_open**

```
int is_open();
```

Returns nonzero if the file is open.

**open**

```
filebuf* open(const char *filename, int mode,
              int prot = filebuf::openprot);
```

Opens the file specified by *filename* and connects to it. The file-opening mode is specified by *mode*.

**overflow**

```
virtual int overflow(int c = EOF);
```

Flushes a buffer to its destination. Every derived class should define the actions to be taken.

**seekoff**

```
virtual streampos seekoff(streamoff offset, dir ios::seek_dir, int mode);
```

Moves the file get/put pointer an *offset* number of bytes. The pointer is moved in the direction specified by *dir* relative to the current position. *mode* can specify read (*ios::in*), write (*ios::out*), or both. If *mode* is *ios::in*, the get pointer is adjusted. If *mode* is *ios::out*, the put pointer is adjusted.

If successful, the *seekoff* function returns a *streampos*-type value that indicates the new file pointer position.

The function can fail if the file does not support repositioning or you request an illegal pointer repositioning, for example, beyond the end of the file. On failure, *seekoff* returns EOF. The file pointer position is undefined.

**setbuf**

```
virtual streambuf* setbuf(char *buf, int len);
```

Allocates *buf* of size *len* for use by the *filebuf*. If *buf* is NULL or *len* is a non-positive value, the *filebuf* is unbuffered.

On success, *setbuf* returns a pointer to the *filebuf*. A failure occurs if the file is open and a buffer has been allocated. On failure, *setbuf* returns NULL and no changes are made to the buffering status.

**sync**

```
virtual int sync();
```

Establishes consistency between internal data structures and the external stream representation.

**underflow**

```
virtual int underflow();
```

Makes input available. This is called when no more data exists in the input buffer. Every derived class should define the actions to be taken.

## fstream class — fstream.h

This stream class, derived from *fstreambase* and *iostream*, provides for simultaneous input and output on a *filebuf*.

### Public constructors

**Constructor**

```
fstream();
```

Makes an *fstream* that isn't attached to a file.

**Constructor**

```
fstream(const char *name, int mode, int prot = filebuf::openprot);
```

Makes an *fstream*, opens a file with access as specified by *mode*, and connects to it. See page 319 for access options provided by *ios::open_mode*.

**Constructor**

```
fstream(int fd);
```

Makes an *fstream* and connects to an open-file descriptor specified by *fd*.

**Constructor**

```
fstream(int fd, char *buf, int n);
```

Makes a *fstream* attached to a file specified by the file descriptor *fd*, and uses *buf* as the storage area. The size of *buf* is sufficient to store *n* bytes. If *buf* is NULL or *n* is non-positive, the *fstream* is unbuffered.

### Public member functions

**open**

```
void open(const char *name, int mode, int prot = filebuf::openprot);
```

Opens a file specified by *name* for an *fstream*. The file-opening mode is specified by the variable *mode*.

**rdbuf**

```
filebuf* rdbuf();
```

Returns the *filebuf* used.

## fstreambase class — fstream.h

This stream class, derived from *ios*, provides operations common to file streams. It serves as a base for *fstream*, *ifstream*, and *ofstream*.

### Public constructors

**Constructor**

```
fstreambase();
```

Makes an *fstreambase* that isn't attached to a file.

**Constructor**

```
fstreambase(const char *name, int mode, int = filebuf::openprot);
```

Makes an *fstreambase*, opens a file specified by *name* in mode specified by *mode*, and connects to it.

**Constructor**

```
fstreambase(int fd);
```

Makes an *fstreambase* and connects to an open-file descriptor specified by *fd*.

**Constructor**

```
fstreambase(int fd, char *buf, int len);
```

Makes an *fstreambase* connected to an open-file descriptor specified by *fd*. The buffer is specified by *buf* and the buffer size is *len*.

### Public member functions

**attach**

```
void attach(int fd);
```

Connects to an open-file descriptor.

**close**

```
void close();
```

Closes the associated *filebuf* and file.

**open**

```
void open(const char *name, int mode, int prot = filebuf::openprot);
```

Opens a file for an *fstreambase*. The file-opening mode is specified by *mode*.

**rdbuf**

```
filebuf* rdbuf();
```

Returns the filebuf used.

**setbuf**

```
void setbuf(char *buf, int len);
```

Reserves an area of memory pointed to by *buf*. The area is sufficiently large to store *len* number of bytes.

## ifstream class — fstream.h

This stream class, derived from *fstreambase* and *istream*, provides input operations on a *filebuf*.

### Public constructors

**Constructor**

```
ifstream();
```

Makes an *ifstream* that isn't attached to a file.

**Constructor**

```
ifstream(const char *name, int mode = ios::in,
        int prot = filebuf::openprot);
```

Makes an *ifstream*, opens a file for input in protected mode, and connects to it. By default, the file is not created if it does not already exist.

**Constructor**

```
ifstream(int fd);
```

Makes an *ifstream* and connects to an open-file descriptor *fd*.

**Constructor**

```
ifstream(int fd, char *buf, int buf_len);
```

Makes an *ifstream* connected to an open file. The file is specified by its descriptor, *fd*. The *ifstream* uses the buffer specified by *buf* of length *buf_len*.

### Public member functions

**open**

```
void open(const char *name, int mode, int prot = filebuf::openprot);
```

Opens a file for an *ifstream*.

**rdbuf**

```
filebuf* rdbuf();
```

Returns the filebuf used.

## ios class — iostream.h

Provides operations common to both input and output. Its derived classes (*istream, ostream, iostream*) specialize I/O with high-level formatting operations. The *ios* class is a base for *istream, ostream, fstreambase,* and *strstreambase.*

### Public data members

The following three constants are used as the second parameter of the *setf* function:

```
static const long  adjustfield;  // left | right | internal
static const long  basefield;    // dec | oct | hex
static const long  floatfield;   // scientific | fixed
```

Stream seek direction:

```
enum seek_dir { beg=0, cur=1, end=2 };
```

Stream operation mode. These can be logically ORed:

```
enum open_mode  {
   app,              Append data—always write at end of file.
   ate,              Seek to end of file upon original open.
   in,               Open for input (default for ifstreams).
   out,              Open for output (default for ofstreams).
   binary,           Open file in binary mode.
   trunc,            Discard contents if file exists (default if out is specified
                     and neither ate nor app is specified).
   nocreate,         If file does not exist, open fails.
   noreplace,        If file exists, open for output fails unless ate or app is
                     set.
};
```

Format flags used with *flags*, *setf*, and *unsetf* member functions:

```
enum {
   skipws,           Skip whitespace on input.
   left,             Left-adjust output.
   right,            Right-adjust output.
   internal,         Pad after sign or base indicator.
   dec,              Decimal conversion.
   oct,              Octal conversion.
   hex,              Hexadecimal conversion.
   showbase,         Show base indicator on output.
   showpoint,        Show decimal point for floating-point output.
   uppercase,        Uppercase hex output.
   showpos,          Show '+' with positive integers.
   scientific,       Suffix floating-point numbers with exponential (E)
                     notation on output.
   fixed,            Use fixed decimal point for floating-point numbers.
   unitbuf,          Flush all streams after insertion.
   stdio,            Flush stdout, stderr after insertion.
};
```

## Protected data members

```
streambuf      *bp;          // The associated streambuf
int             x_fill;      // Padding character of output
long            x_flags;     // Formatting flag bits
int             x_precision; // Floating-point precision on output
```

```
int             state;      // Current state of the streambuf
ostream        *x_tie;      // The tied ostream, if any
int             x_width;    // Field width on output
```

## Public constructor

**Constructor**

```
ios(streambuf *);
```

Associates a given *streambuf* with the stream.

## Protected constructor

**Constructor**

```
ios();
```

Constructs an *ios* object that has no corresponding *streambuf*.

## Public member functions

**bad**

```
int bad();
```

Nonzero if error occurred.

**bitalloc**

```
static long bitalloc();
```

Acquires a new flag bit set. The return value can be used to set, clear, and test the flag. This is for user-defined formatting flags.

**clear**

```
void clear(int = 0);
```

Sets the stream state to the given value.

**eof**

```
int eof();
```

Nonzero on end of file.

**fail**

```
int fail();
```

Nonzero if an operation failed.

**fill**

```
char fill()
```

Returns the current fill character.

**fill**

```
char fill(char);
```

Resets the fill character; returns the previous character.

**flags**

```
long flags();
```

Returns the current format flags.

**flags**

```
long flags(long);
```

Sets the format flags to be identical to the given **long**; returns previous flags. Use *flags(0)* to set the default format.

**good**

```
int good();
```

Nonzero if no state bits were set (that is, no errors appeared).

**precision**

```
int precision();
```

Returns the current floating-point precision.

**precision**

```
int precision(int);
```

Sets the floating-point precision; returns previous setting.

**rdbuf**

```
streambuf* rdbuf();
```

Returns a pointer to this stream's assigned streambuf.

**rdstate**

```
int rdstate();
```

**setf**

```
long setf(long);
```

Sets the flags corresponding to those marked in the given **long**; returns previous settings.

**setf**

```
long setf(long _setbits, long _field);
```

The bits corresponding to those marked in *_field* are cleared, and then reset to be those marked in *_setbits*.

**sync_with_stdio**

```
static void sync_with_stdio();
```

Mixes stdio files and iostreams. This should not be used for new code.

**tie**

```
ostream* tie();
```

Returns the *tied stream*, or NULL if there is none. Tied streams are those that are connected such that when one is used, the other is affected. For example, *cin* and *cout* are tied; when *cin* is used, it flushes *cout* first.

**tie**

```
ostream* tie(ostream *out);
```

Ties another stream to the output stream *out* and returns the previously tied stream. If the stream was not previously tied, *tie* returns NULL.

When an input stream has characters to be consumed, or if an output stream needs more characters, the tied stream is first flushed automatically. By default, *cin*, *cerr* and *clog* are tied to *cout*.

**unsetf**

```
long unsetf(long f);
```

Clears the bits corresponding to *f* and returns a **long** that represents the previous settings.

**width**

```
int width();
```

Returns the current width setting.

**width**

```
int width(int);
```

Sets the width as given; returns the previous width.

**xalloc**

```
static int xalloc();
```

Returns an array index of previously unused words that can be used as user-defined formatting flags.

### Protected member functions

**init**

```
void init(streambuf *);
```

Provides the actual initialization.

**setstate**

```
void setstate(int);
```

Sets all status bits.

## iostream class — iostream.h

This class, derived from *istream* and *ostream*, is a mixture of its base classes, allowing both input and output on a stream. It is a base for *fstream* and *strstream*.

### Public constructor

**Constructor**

```
iostream(streambuf *);
```

Associates a given *streambuf* with the stream.

## iostream_withassign class — iostream.h

This class is an *iostream* with an added assignment operator.

### Public constructor

**Constructor**

```
iostream_withassign();
```

Default constructor (calls *iostream's* constructor).

### Public member functions

None (although the = operator is overloaded).

## istream class — iostream.h

Provides formatted and unformatted input from a *streambuf*. The >> operator is overloaded for all fundamental types, as explained in the narrative at the beginning of the chapter. This *ios* class is a base for *ifstream*, *iostream*, *istrstream*, and *istream_withassign*.

### Public constructor

**Constructor**

```
istream(streambuf *);
```

Associates a given *streambuf* with the stream.

### Public member functions

**gcount**

```
int gcount();
```

Returns the number of characters last extracted.

**get**

```
int get();
```

Extracts the next character or EOF.

**get**

```
istream& get(char *buf, int len, char delim = '\n');
istream& get(signed char *buf, int len, char delim = '\n');
istream& get(unsigned char *buf, int len, char delim = '\n');
```

Extracts characters and stores them in *buf* until the delimiter, specified by *delim*, or end-of-file is encountered, or until (*len* – 1) bytes have been read. A terminating null is always placed in the output string; the delimiter never is. The delimiter remains in the stream. Fails only if no characters were extracted.

The *get* function fails if it encounters the end of file before any characters are stored. On failure, *get* sets *ios*::*failbit*.

**get**

```
istream& get(char &ch);
istream& get(signed char &ch);
istream& get(unsigned char &ch);
```

Extracts a single character into the *ch* reference.

**get**

```
istream& get(streambuf &sbuf, char delim = '\n');
```

Extracts characters into the given *sbuf* reference until *delim* is encountered.

**getline**

```
istream& getline(char *buf, int len, char);
istream& getline(signed char *buf, int len, char delim = '\n');
istream& getline(unsigned char *buf, int len, char delim = '\n');
```

Same as *get*, except the delimiter is also extracted. Generally, the specified *delim* is not copied to *buf*. However, if the delimiter is encountered exactly when *len* characters have been extracted, *delim* is not extracted.

**ignore**

```
istream& ignore(int n = 1, int delim = EOF);
```

Causes up to *n* characters in the input stream to be skipped; stops if *delim* is encountered.

**ipfx**

```
istream& ipfx(int n = 0);
```

The *ipfx* function is called by input functions prior to fetching from an input stream. Functions that perform formatted input call *ipfx(0)*; unformatted input functions call *ipfx(1)*.

**peek**

```
int peek();
```

Returns next char without extraction.

**putback**

```
istream& putback(char);
```

Pushes back a character into the stream.

**read**

```
istream& read(char*, int);
istream& read(signed char*, int);
istream& read(unsigned char*, int);
```

Extracts a given number of characters into an array. Use *gcount* for the number of characters actually extracted if an error occurred.

**seekg**

```
istream& seekg(streampos pos);
```

Moves to an absolute position in the input stream.

**seekg**

```
istream& seekg(streamoff offset, seek_dir dir);
```

Moves *offset* number of bytes relative to the current position for the input stream. The offset is in the direction specified by *dir* following the definition: **enum** *seek_dir {beg, cur, end};*

Use *ostream*::*seekp* for positioning in an output stream.

Use *seekpos* or *seekoff* for positioning in a stream buffer.

**tellg**

```
streampos tellg();
```

Returns the current stream position. On failure, *tellg* returns a negative number.

Use *ostream*::*tellp* to find the position in an output stream.

### Protected member functions

**eatwhite**

```
void eatwhite();
```

Extract consecutive whitespace.

## istream_withassign class — iostream.h

This class is an *istream* with an added assignment operator.

### Public constructor

**Constructor**

```
istream_withassign();
```

Default constructor (calls *istream*'s constructor).

### Public member functions

None (although the = operator is overloaded).

## istrstream class — strstrea.h

Provides input operations on a *strstreambuf*. This class is derived from *strstreambase* and *istream*.

### Public constructors

**Constructor**

```
istrstream(char *);
istrstream(signed char *);
istrstream(unsigned char *);
```

Each of the constructors above makes an *istrstream* with a specified string (a null character is never extracted). See "The three char types" in Chapter 1 of the *Programmer's Guide* for a discussion of character types.

**Constructor**

```
istrstream(char *str, int n);
istrsteam(signed char *str, int);
istrstream(unsigned char *str, int);
```

Each of the three constructors above makes an *istrstream* using up to *n* bytes of *str*. See "The three char types" in Chapter 1 of the *Programmer's Guide* for a discussion of character types.

## ofstream class — fstream.h

Provides input operations on a *filebuf*. This class is derived from *fstreambase* and *ostream*.

### Public constructors

**Constructor**

```
ofstream();
```

Makes an *ofstream* that isn't attached to a file.

**Constructor**

```
ofstream(const char *name, int mode = ios::out,
        int prot = filebuf::openprot);
```

Makes an *ofstream*, opens a file, and connects to it.

**Constructor**

```
ofstream(int fd);
```

Makes an *ofstream* and connects to an open-file descriptor specified by *fd*.

**Constructor**

```
ofstream(int fd, char *buf, int len);
```

Makes an *ofstream* connected to an open-file descriptor specified by *fd*. The buffer specified by *buf* of *len* is used by the *ofstream*.

### Public member functions

**open**

```
void open(const char *name, int mode = ios::out,
        int prot = filebuf::openprot);
```

Opens a file for an *ofstream*.

**rdbuf**

```
filebuf* rdbuf();
```

Returns the *filebuf* used.

## ostream class — iostream.h

Provides formatted and unformatted output to a *streambuf*. The << operator is overloaded for all fundamental types. This *ios*-based class is a base for *constream*, *iostream*, *ofstream*, *ostrstream*, and *ostream_withassign*.

### Public constructor

**Constructor**

```
ostream(streambuf *);
```

Associates a given *streambuf* with the stream.

### Public member functions

**flush**

```
ostream& flush();
```

Flushes the stream.

**opfx**

```
int opfx();
```

The *opfx* function is called by output functions prior to inserting to an output stream. *opfx* returns 0 if the *ostream* has a nonzero error state. Otherwise, *opfx* returns a nonzero value.

**osfx**

```
void osfx();
```

The *osfx* function performs post output operations. If *ios*::*unitbuf* is on, *opfx* flushes the *ostream*. On failure, *opfx* sets *ios*:: *failbit*.

**put**

```
ostream& put(unsigned char ch);
ostream& put(char ch);
ostream& put(signed char ch);
```

Inserts the character.

**seekp**

```
ostream& seekp(streampos);
```

Moves to an absolute position (as returned from *tellp*).

**seekp**

```
ostream& seekp(streamoff, seek_dir);
```

Moves to a position relative to the current position, following the definition: **enum** *seek_dir {beg, cur, end};*

**tellp**

```
streampos tellp();
```

Returns the current stream position.

**write**

```
ostream& write(const signed char*, int n);
ostream& write(const unsigned char*, int n);
ostream& write(const char*, int n);
```

Inserts *n* characters (nulls included).

## ostream_withassign class — iostream.h

This class is an *ostream* with an added assignment operator.

### Public constructor

**Constructor**

```
ostream_withassign();
```

Default constructor (calls *ostream*'s constructor).

### Public member functions

None (although the = operator is overloaded).

## ostrstream class — strstrea.h

Provides output operations on a *strstreambuf*. This class is derived from *strstreambase* and *ostream*.

### Public constructors

**Constructor**

```
ostrstream();
```

Makes a dynamic *ostrstream*.

**Constructor**

```
ostrstream(char *buf, int len, int mode = ios::out);
ostrstream(signed char *buf, int len, int mode = ios::out);
ostrstream(unsigned char *buf, int len, int mode = ios::out);
```

Each of the three constructors above makes a *ostrstream* with a specified *len*-byte buffer. If the file-opening mode is *ios::app* or *ios::ate*, the get/put pointer is positioned at the null character of the string. See "The three char types" in Chapter 1 of the *Programmer's Guide* for a discussion of character types.

### Public member functions

**pcount**

```
int pcount();
```

Returns the number of bytes currently stored in the buffer.

**str**

```
char *str();
```

Returns and freezes the buffer. You must deallocate it if it was dynamic.

## streambuf class — iostream.h

This is a base class for all other buffering classes. It provides a buffer interface between your data and storage areas such as memory or physical devices. The buffers created by *streambuf* are referred to as get, put, and reserve areas. The contents are accessed and manipulated by pointers that point between characters.

Buffering actions performed by *streambuf* are rather primitive. Normally, applications gain access to buffers and buffering functions through a pointer to *streambuf* that is set by *ios*. Class *ios* provides a pointer to *streambuf* that provides a transparent access to buffer services for high-level classes. The high-level classes provide I/O formatting.

### Public constructors

**Constructor**

```
streambuf();
```

Creates an empty buffer object.

**Constructor**

```
streambuf(char *buf, int size);
```

Constructs an empty buffer *buf* and sets up a reserve area for *size* number of bytes.

## Public member functions

**in_avail**

```
int in_avail();
```

Returns the number of characters remaining in the input buffer.

**out_waiting**

```
int out_waiting();
```

Returns the number of characters remaining in the output buffer.

**sbumpc**

```
int sbumpc();
```

Returns the current character from the input buffer, then advances.

**seekoff**

```
virtual streampos seekoff(streamoff, ios::seek_dir,
                          int = (ios::in | ios::out);
```

Moves the get and/or put pointer (the third argument determines which one or both) relative to the current position.

**seekpos**

```
virtual streampos seekpos(streampos, int = (ios::in | ios::out));
```

Moves the get or put pointer to an absolute position.

**setbuf**

```
virtual streambuf* setbuf(char *, int);
```

Connects to a given buffer.

**sgetc**

```
int sgetc();
```

Peeks at the next character in the input buffer.

**sgetn**

```
int sgetn(char*, int n);
```

Gets the next *n* characters from the input buffer.

**snextc**

```
int snextc();
```

Advances to and returns the next character from the input buffer.

**sputbackc**

```
int sputbackc(char);
```

Returns a character to input.

**sputc**

```
int sputc(int);
```

Puts one character into the output buffer.

**sputn**

```
int sputn(const char*, int n);
```

Puts *n* characters into the output buffer.

**stossc**

```
void stossc();
```

Advances to the next character in the input buffer.

## Protected member functions

**allocate**

```
int allocate();
```

Sets up a buffer area.

**base**

```
char *base();
```

Returns the start of the buffer area.

**blen**

```
int blen();
```

Returns the length of the buffer area.

**eback**

```
char *eback();
```

Returns the base of the putback section of the get area.

**ebuf**

```
char *ebuf();
```

Returns the end+1 of the buffer area.

**egptr**

```
char *egptr();
```

Returns the end+1 of the get area.

**epptr**

```
char *epptr();
```

Returns the end+1 of the put area.

**gbump**

```
void gbump(int);
```

Advances the get pointer.

**gptr**

```
char *gptr();
```

Returns the next location in the get area.

**pbase**

```
char *pbase();
```

Returns the start of the put area.

**pbump**

```
void pbump(int);
```

Advances the put pointer.

**pptr**

```
char *pptr();
```

Returns the next location in the put area.

**setb**

```
void setb(char *, char *, int = 0 );
```

Sets the buffer area.

**setg**

```
void setg(char *, char *, char *);
```

Initializes the get pointers.

**setp**

```
void setp(char *, char *);
```

Initializes the put pointers.

**unbuffered**

```
void unbuffered(int);
```

Sets the buffering state.

**unbuffered**

```
int unbuffered();
```

Returns nonzero if not buffered.

## strstreambase class — strstrea.h

Specializes *ios* to string streams. This class is entirely protected except for the member function *strstreambase::rdbuf*. This class is a base for *strstream*, *istrstream*, and *ostrstream*.

### Public constructors

**Constructor**

```
strstreambase();
```

Makes an empty *strstreambase*.

**Constructor**

```
strstreambase(char *, int, char *start);
```

Makes an *strstreambase* with a specified buffer and starting position.

### Public member functions

**rdbuf**

```
strstreambuf * rdbuf();
```

Returns a pointer to the *strstreambuf* associated with this object.

## strstreambuf class — strstrea.h

Specializes *streambuf* for in-memory formatting.

## Public constructors

**Constructor**

```
strstreambuf();
```

Makes a dynamic *strstreambuf*. Memory will be dynamically allocated as needed.

**Constructor**

```
strstreambuf(void * (*)(long), void (*)(void *));
```

Makes a dynamic buffer with specified allocation and free functions.

**Constructor**

```
strstreambuf(int n);
```

Makes a dynamic *strstreambuf*, initially allocating a buffer of at least *n* bytes.

**Constructor**

```
strstreambuf(char*, int, char *strt = 0);
strstreambuf(signed char *, int, signed char *strt = 0);
strstreambuf(unsigned char *, int, unsigned char *strt = 0);
```

Each of the three constructors above makes a static *strstreambuf* with a specified buffer. If *strt* is not null, it delimits the buffer. See "The three char types" in Chapter 1 of the *Programmer's Guide* for a discussion of character types.

## Public member functions

**doallocate**

```
virtual int doallocate();
```

Performs low-level buffer allocation.

**freeze**

```
void freeze(int = 1);
```

If the input parameter is nonzero, disallows storing any characters in the buffer. Unfreeze by passing a zero.

**overflow**

```
virtual int overflow(int);
```

Flushes a buffer to its destination. Every derived class should define the actions to be taken.

**seekoff**

```
virtual streampos seekoff(streamoff, ios::seek_dir, int);
```

Moves the pointer relative to the current position.

**setbuf**

```
virtual streambuf* setbuf(char*, int);
```

Specifies the buffer to use.

**str**

```
char *str();
```

Returns a pointer to the buffer and freezes it.

**sync**

```
virtual int sync();
```

Establishes consistency between internal data structures and the external stream representation.

**underflow**

```
virtual int underflow();
```

Makes input available. This is called when a character is requested and the strstreambuf is empty. Every derived class should define the actions to be taken.

## strstream class — strstrea.h

Provides for simultaneous input and output on a *strstreambuf*. This class is derived from *strstreambase* and *iostream*.

### Public constructors

**Constructor**

```
strstream();
```

Makes a dynamic *strstream*.

**Constructor**

```
strstream(char *buf, int sz, int mode);
strstream(signed char *buf, int sz, int mode);
strstream(unsigned char *buf, int sz, int mode);
```

Each of the three constructors above makes a *strstream* with a specified *sz*-byte buffer. If *mode* is *ios::app* or *ios::ate*, the get/put pointer is positioned at the null character of the string. See "The three char types" in Chapter 1 of the *Programmer's Guide* for a discussion of character types.

### Public member function

**str**

```
char *str();
```

Returns and freezes the buffer. The user must deallocate it if it was dynamic.

C H A P T E R 6

# Persistent stream classes and macros

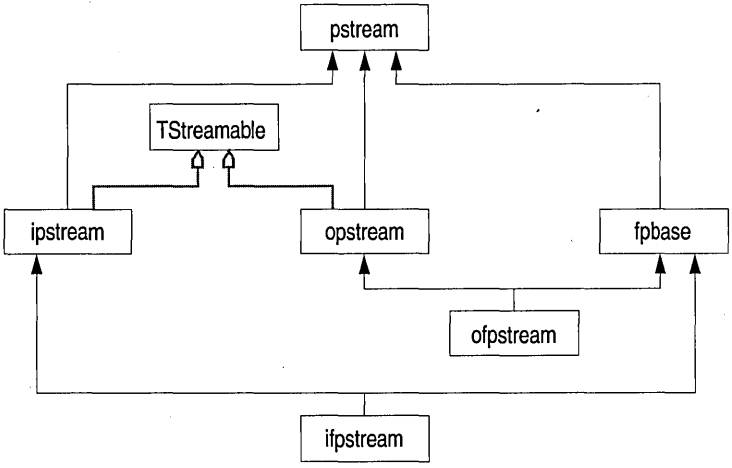
To learn how to use the persistent streams library, see Chapter 7 in the *Programmer's Guide*.

Borland support for persistent streams consists of a class hierarchy and macros to help you develop streamable objects. This chapter is a reference for these classes and macros. It alphabetically lists and describes all the public classes that support persistent objects. The class descriptions are followed by descriptions of the _ _DELTA macro and the streaming macros. The streaming macros are provided to simplify the declaration and definition of streamable classes.

## The persistent streams class hierarchy

The persistent streams class hierarchy is shown in the following figure:

Figure 6.1 Streamable class hierarchy




The gray arrows connecting *TStreamableBase* indicate that it is a friend class.

# fpbase class — objstm.h

Provides the basic operations common to all object file stream I/O.

## Constructors

**Constructor**

```
fpbase();
fpbase(const char _FAR *name, int omode, int prot = filebuf::openprot);
fpbase(int f);
fpbase(int f, char _FAR *b, int len);
```

Creates a buffered *fpbase* object. You can set the size and initial contents of the buffer with the *len* and *b* arguments. You can open a file and attach it to the stream by specifying the name, mode, and protection (*prot*) arguments, or by using the file descriptor, *f*.

## Public member functions

**attach**

```
void attach(int f);
```

Attaches the file with descriptor *f* to this stream if possible. Sets *ios::state* accordingly.

**close**

```
void close();
```

Closes the stream and associated file.

**open**

```
void open(const char _FAR *name, int mode, int prot = filebuf::openprot);
```

Opens the named file in the given *mode* (*app, ate, in, out, binary, trunc, nocreate, noreplace*) and protection. The opened file is attached to this stream.

**rdbuf**

```
filebuf _FAR * rdbuf();
```

Returns a pointer to the current file buffer.

**setbuf**

```
void setbuf(char _FAR *buf, int len);
```

Allocates a buffer of size *len*.

# ifpstream class — objstrm.h

Provides the base class for reading (extracting) streamable objects from file streams.

### Public constructors

**Constructor**

```
ifpstream();
ifpstream(const char _FAR *name, int mode = ios::in, int prot =
          filebuf::openprot);
ifpstream(int f);
ifpstream(int f, char _FAR *b, int len);
```

Creates a buffered *ifpstream* object. You can set the size and initial contents of the buffer with the *len* and *b* arguments. You can open a file and attach it to the stream by specifying the name, mode, and protection arguments, or via the file descriptor, *f*.

### Public member functions

**open**

```
void open(const char _FAR *name, int mode = ios::in, int prot =
          filebuf::openprot);
```

Opens the named file in the given *mode* (*app, ate, in, out, binary, trunc, nocreate,* or *noreplace*) and protection. The default mode is *in* (input) with *openprot* protection. The opened file is attached to this stream.

**rdbuf**

```
filebuf _FAR * rdbuf();
```

Returns a pointer to the current file buffer.

## ipstream class — objstrm.h

Provides the base class for reading (extracting) streamable objects.

### Public constructors

**Constructor**

```
ipstream(streambuf *buf);
```

Creates a buffered *ipstream* with the given buffer and sets the *bp* data member to *buf*. The state is set to 0.

### Public member functions

**find**

```
TStreamableBase _FAR * find(P_id_type Id);
```

Returns a pointer to the object corresponding to *Id*.

**freadBytes**

```
void freadBytes( void far *data, size_t sz );
```

Reads into the supplied far buffer (*data*) the number of bytes specified by *sz*.

**freadString**

```
char *freadString();
```

Reads a string from the stream. Determines the length of the string and allocates a far character array of the appropriate length. Reads the string into this array and returns a pointer to the string. The caller is expected to free the allocated memory block.

```
char *freadString( char far *buf, unsigned maxLen );
```

Reads a string from the stream into the supplied far buffer (*buf*). If the length of the string is greater than *maxLen*-1, reads nothing. Otherwise reads the string into the buffer and appends a null terminating byte.

**getVersion**

```
uint32 getVersion() const;
```

Returns the object version number.

**readByte**

```
uint8 readByte();
```

Returns the character at the current stream position.

**readBytes**

```
void readBytes(void _FAR *data, size_t sz);
```

Reads *sz* bytes from current stream position, and writes them to *data*.

**readString**

```
char _FAR * readString();
char _FAR * readString(char _FAR *buf, unsigned maxLen);
```

*readString()* allocates a buffer large enough to contain the string at the current stream position. Reads the string from the stream into the buffer. The caller must free the buffer.

*readString(Pchar buf, unsigned maxLen)* reads the string at the current stream position into the buffer specified by *buf*. Does not read more than *maxLen* bytes.

**readWord**

```
uint32 readWord();
```

Returns the 32-bit word at the current stream position.

**readWord16**

```
uint16 readWord16();
```

Returns the 16-bit word at the current stream position.

**readWord**

```
uint32 readWord32();
```

Returns the 32-bit word at the current stream position.

**registerObject**

```
void registerObject(TStreamableBase * adr);
```

Registers the object pointed to by *adr*.

**seekg**

```
ipstream& seekg(streampos pos);
ipstream& seekg(streamoff off, ios::seek_dir);
```

The first form moves the stream position to the absolute position given by *pos*. The second form moves to a position relative to the current position by an offset *off* (+ or –) starting at *ios::seek_dir*. *ios::seek_dir* can be set to *beg* (start of stream), *cur* (current stream position), or *end* (end of stream).

**tellg**

```
streampos tellg();
```

Returns the (absolute) current stream position.

## Protected constructors

**Constructor**

```
ipstream();
```

The protected form of the constructor does not initialize the buffer pointer *bp*. Use *init* to set the buffer and state.

## Protected member functions

**readData**

```
void _FAR * readData(const ObjectBuilder _FAR* ,TStreamableBase _FAR *&
                     mem);
```

Invokes the appropriate *read* function to read from the stream to the object pointed to by *mem*. If *mem* is 0, the appropriate *build* function is called first.

See also: *TStreamableClass*, and the *read* and *build* member functions of each streamable class

**readPrefix**

```
const ObjectBuilder _FAR * readPrefix();
```

Returns the *TStreamableClass* object corresponding to the class *name* stored at the current position.

**readSuffix**

```
void readSuffix();
```

Reads and checks the final byte of an object's name field.

See also: *ipstream::readPrefix*

**readVersion**

```
void readVersion();
```

Sets the version number of the input stream.

### Friends

**Operator >>**

```
friend ipstream& operator >> (ipstream& ps, signed char _FAR & ch);
friend ipstream& operator >> (ipstream& ps, unsigned char _FAR & ch);
friend ipstream& operator >> (ipstream& ps, signed short _FAR & sh);
friend ipstream& operator >> (ipstream& ps, unsigned short _FAR & sh);
friend ipstream& operator >> (ipstream& ps, signed int _FAR & i);
friend ipstream& operator >> (ipstream& ps, unsigned int _FAR & i);
friend ipstream& operator >> (ipstream& ps, signed long _FAR & l);
friend ipstream& operator >> (ipstream& ps, unsigned long _FAR & l);
friend ipstream& operator >> (ipstream& ps, float _FAR & f);
friend ipstream& operator >> (ipstream& ps, double _FAR & d);
friend ipstream& operator >> (ipstream& ps, long double _FAR & d);
friend ipstream& operator >> (ipstream& ps, TStreamableBase t);
friend ipstream& operator >> (ipstream& ps, void *t);
```

Extracts (reads) from the *ipstream ps*, to the given argument. A reference to the stream is returned, letting you chain **>>** operations in the usual way. The data type of the argument determines how the read is performed. For example, reading a signed *char* is implemented using *readByte*.

## ofpstream class — objstrm.h

Provides the base class for writing (inserting) streamable objects to file streams.

### Public constructors

**Constructor**

```
ofpstream();
ofpstream(const char _FAR *name, int mode = ios::out,
          int prot = filebuf::openprot);
ofpstream(int f);
ofpstream(int f, char _FAR *b, int len);
```

Creates a buffered *ofpstream* object. You can set the size and initial contents of the buffer with the *len* and *b* arguments. A file can be opened and attached to the stream by specifying the name, mode, and protection arguments, or by using the file descriptor, *f*.

### Public member functions

**open**

```
void open(char _FAR *name, int mode = ios::out,
          int prot = filebuf::openprot);
```

Opens the named file in the given *mode* (*app, ate, in, out, binary, trunc, nocreate,* or *noreplace*) and protection. The default mode is *out* (output) with *openprot* protection. The opened file is attached to this stream.

**rdbuf**

```
filebuf _FAR * rdbuf();
```

Returns the current file buffer.

## opstream class — objstrm.h

*opstream,* a specialized derivative of *pstream,* is the base class for writing (inserting) streamable objects.

### Public constructors and destructor

**Constructor**

```
opstream(streambuf _FAR *buf);
```

This constructor creates a buffered *opstream* with the given buffer and sets the *bp* data member to *buf.* The state is set to 0.

**Destructor**

```
~opstream();
```

Destroys the *opstream* object.

See also: *pstream::init*

### Public member functions

**findObject**

```
P_id_type findObject(TStreamableBase _FAR *adr);
```

Returns the type ID for the object pointed to by *adr.*

**findVB**

```
P_id_type findVB(TStreamableBase _FAR *adr);
```

Returns a pointer to the virtual base.

**flush**

```
opstream& flush();
```

Flushes the stream.

**fwriteBytes**

```
void fwriteBytes( const void *data, size_t sz );
```

Writes the specified number of bytes (*sz*) from the supplied far buffer (*data*) to the stream.

**fwriteString**

```
void fwriteString( const char *str );
```

Writes the specified far character string (*str*) to the stream.

**registerObject**

```
void registerObject(TStreamableBase _FAR *adr);
```

Registers the class of the object pointed to by *adr*.

**registerVB**

```
void registerVB(TStreamableBase _FAR *adr);
```

Registers a virtual base class.

**seekp**

```
opstream& seekp(streampos pos);
opstream& seekp(streamoff off,ios::seek_dir);
```

The first form moves the stream's current position to the absolute position given by *pos*. The second form moves to a position relative to the current position by an offset *off* (+ or –) starting at *ios::seek_dir*. *ios::seek_dir* can be set to *beg* (start of stream), *cur* (current stream position), or *end* (end of stream).

**tellp**

```
streampos tellp();
```

Returns the (absolute) current stream position.

**writeByte**

```
void writeByte(uint8 ch);
```

Writes the byte *ch* to the stream.

**writeBytes**

```
void writeBytes(const void *data, size_t sz);
void writeBytes(const void far *data, size_t sz);
```

Writes *sz* bytes from *data* buffer to the stream.

**writeObject**

```
void writeObject( const TStreamableBase _BIDSFAR *t );
```

Writes the object that is pointed to by *t* to the output stream.

**writeObjectPtr**

```
void writeObjectPtr(const TStreamableBase *t);
```

Writes the object pointer *t* to the output stream.

**writeString**

```
void writeString(const char _FAR *str);
```

Writes *str* to the stream (together with a leading length byte).

**writeWord**

```
void writeWord(uint32 us);
```

Writes the 32-bit word *us* to the stream.

**writeWord16**

```
void writeWord16(uint16 us);
```

Writes the 16-bit word *us* to the stream.

**writeWord32**

```
void writeWord32(uint32 us);
```

Writes the 32-bit word *us* to the stream.

## Protected constructors

**Constructor**

```
opstream();
```

This protected form of the constructor does not initialize the buffer pointer *bp*. Use *init* to set the buffer and state.

## Protected member functions

**writeData**

```
void writeData(TStreamableBase *t);
```

Writes data to the stream by calling the appropriate class's *write* member function for the object being written.

See also: *TStreamableBase* and the *write* functions in the streamable classes

**writePrefix**

```
void writePrefix(const TStreamableBase *t);
```

Writes the class name prefix to the stream. The << operator uses this function to write a prefix and suffix around the data written with *writeData*. The prefix/suffix is used to ensure type-safe stream I/O.

See also: *ipstream:readPrefix*

**writeSuffix**

```
void writeSuffix(const TStreamableBase *t);
```

Writes the class name suffix to the stream. The << operator uses this function to write a prefix and suffix around the data written with *writeData*. The prefix/suffix is used to ensure type-safe stream I/O.

See also: *ipstream:readPrefix*

## Friends

**Operator <<**

```
friend opstream& operator << (opstream&  ps, signed char ch);
friend opstream& operator << (opstream&  ps, unsigned char ch);
friend opstream& operator << (opstream&  ps, signed short sh);
friend opstream& operator << (opstream&  ps, unsigned short sh);
friend opstream& operator << (opstream&  ps, signed int i);
friend opstream& operator << (opstream&  ps, unsigned int i);
friend opstream& operator << (opstream&  ps, signed long l);
```

```
friend opstream& operator << (opstream&  ps, unsigned long l);
friend opstream& operator << (opstream&  ps, float f);
friend opstream& operator << (opstream&  ps, double d);
friend opstream& operator << (opstream&  ps, long double d);
friend opstream& operator << (opstream&  ps, TStreamableBase& t);
```

Inserts (writes) the given argument to the given *ipstream* object. The data type of the argument determines the form of write operation employed.

# pstream class — objstrm.h

*pstream* is the base class for handling streamable objects.

## Type definitions

**PointerTypes**

```
enum PointerTypes{ptNull, ptIndexed, ptObject};
```

Enumerates object pointer types.

## Public constructors and destructor

**Constructor**

```
pstream(streambuf _FAR *buf);
```

This constructor creates a buffered *pstream* with the given buffer and sets the *bp* data member to *buf*. The state is set to 0.

**Destructor**

```
virtual ~pstream();
```

Destroys the *pstream* object.

## Public member functions

**bad**

```
int bad() const;
```

Returns nonzero if an error occurs.

**clear**

```
void clear(int aState = 0);
```

Set the stream *state* to the given value (defaults to 0).

**eof**

```
int eof() const;
```

Returns nonzero on end of stream.

**fail**

```
int fail() const;
```

Returns nonzero if a stream operation fails.

**good**

```
int good() const;
```

Returns nonzero if no state bits are set (that is, if no errors occurred).

**rdbuf**

```
streambuf _FAR * rdbuf() const;
```

Returns the *pb* pointer to this stream's assigned buffer.

See also: *pstream::pb*

**rdstate**

```
int rdstate() const;
```

Returns the current *state* value.

## Operators

**Operator void *()**

```
operator void *() const;
```

Overloads the pointer-to-*void* cast operator. Returns 0 if the operation has failed (that is, if *pstream::fail* returned nonzero); otherwise, returns nonzero.

See also: *pstream::fail*

**Operator ! ()**

```
int operator ! () const;
```

Overloads the NOT operator. Returns the value returned by *pstream::fail*.

See also: *pstream::fail*

## Protected data members

**bp**

```
streambuf _FAR *bp;
```

Pointer to the stream buffer.

**state**

```
int state;
```

Format state flags. Use *rdstate* to access the current state.

See also: *pstream::rdstate*

## Protected constructors

**Constructor**

```
pstream();
```

This form of the constructor does not initialize the buffer pointer *bp*. Use *init* and *setstate* to set the buffer and state.

### Protected member functions

**init**

```
void init(streambuf _FAR *sbp);
```

Initializes the stream: sets *state* to 0 and *bp* to *sbp*.

**setstate**

```
void setstate(int b);
```

Updates the *state* data member with `state |= (b & 0xFF)`.

## TStreamableBase class — objstrm.h

```
class _EXPCLASS TStreamableBase : public TCastable
```

Classes that inherit from *TStreamableBase* are known as streamable classes, meaning their objects can be written to and read from streams. If you want to develop your own streamable classes, you should make sure that *TStreamableBase* is somewhere in their ancestry. Using an existing streamable class as a base, of course, is an obvious way of achieving this. Don't be afraid to use multiple inheritance to derive a class from *TStreamableBase* if your class must also fit into an existing class hierarchy.

### Type definitions

**Type_id**

```
typedef const char *Type_id;
```

Describes type identifiers.

### Public destructor

**Destructor**

```
virtual ~TStreamableBase() {};
```

Destroys the *TStreamableBase* object.

### Public member functions

**CastableID**

```
virtual Type_id CastableID() const = 0;
```

Provides support for typesafe downcasting. Returns a string containing the type name.

**FindBase**

```
virtual void *FindBase( Type_id id ) const;
```

Returns a pointer to the base class.

**MostDerived**

```
virtual void *MostDerived() const = 0;
```

Returns a **void** pointer to the actual streamed object.

## TStreamableClass class — streambl.h

Used by the private database class and *pstream* in streamable class registration.

### Public constructor

**Constructor**

```
TStreamableClass(const char *n, BUILDER b, int d=NoDelta, ModuleId
                 mid=GetModuleId());
```

Creates a *TStreamableClass* object with the given name (*n*) and the given builder function (*b*), then registers the type. Each streamable class, for example *TClassname*, has a *build* member function of type BUILDER. For type-safe object-stream I/O, the stream manager needs to access the names and the type information for each class. To ensure that the appropriate functions are linked into any application using the stream manager, you must provide a reference such as:

```
TStreamableClass RegClassName;
```

where *TClassName* is the name of the class for which objects need to be streamed. (Note that *RegClassName* is a single identifier.) This not only registers *TClassName* (telling the stream manager which *build* function to use), it also automatically registers any dependent classes. You can register a class more than once without any harm or overhead.

Invoke this function to provide raw memory of the correct size into which an object of the specified class can be read. Because the build procedure invokes a special constructor for the class, all virtual table pointers are initialized correctly.

The distance, in bytes, between the base of the streamable object and the beginning of the *TStreamableBase* component of the object is *d*. Calculate *d* by using the _ _DELTA macro. For example,

```
TStreamableClass RegTClassName = TStreamableClass("TClassName",
TClassName::build, _ _DELTA(TClassName));
```

See also: *TStreamableBase, ipstream, opstream*

### Friends

The classes *opstream* and *ipstream* are friends of *TStreamableClass*.

## TStreamer class — objstrm.h

```
class _BIDSCLASS _RTTI TStreamer
```

Base class for all streamable objects.

### Public member functions

**GetObject**

```
TStreamableBase *GetObject() const
```

Returns the address of the *TStreamableBase* component of the streamable object.

### Protected constructors

**Constructor**

```
TStreamer( TStreamableBase *obj )
```

Constructs the *TStreamer* object, and initializes the streamable object pointer.

### Protected member functions

**Read**

```
virtual void *Read( ipstream&, uint32 ) const = 0;
```

This pure virtual member function must be redefined for every streamable class. It must read the necessary data members for the streamable class from the supplied *ipstream*.

**StreamableName**

```
virtual const char *StreamableName() const = 0;
```

This pure virtual member function must be redefined for every streamable class. *StreamableName* returns the name of the streamable class, which is used by the stream manager to register the streamable class. The name returned must be a 0-terminated string.

**Write**

```
virtual void Write( opstream& ) const = 0;
```

This pure virtual function must be redefined for every streamable class. It must write the necessary streamable class data members to the supplied *opstream* object. *Write* is usually implemented by calling the base class's *Write* (if any), and then inserting any additional data members for the derived class.

## __DELTA macro — streambl.h

```
#define __DELTA( d ) (FP_OFF((TStreamable *)(d *)1)-1)
```

Calculates the distance, in bytes, between the base of the streamable object and the beginning of the *TStreamableBase* component of the object.

## DECLARE_STREAMABLE macro — objstrm.h

```
DECLARE_STREAMABLE(exp, cls, ver)
```

The DECLARE_STREAMABLE macro is used within a class definition to add the members that are needed for streaming. Because it contains access specifiers, it should be followed by an access specifier or be used at the end of the class definition. The first parameter should be a macro, which in turn should conditionally expand to either **__import** or **__export**, depending on whether or not the class is to be imported or exported from a DLL. The second parameter is the streamable class name. The third parameter is the object version number. DECLARE_STREAMABLE is defined as follows:

```
#define DECLARE_STREAMABLE(exp, cls, ver )                    \
    DECLARE_CASTABLE                                          \
    DECLARE_STREAMER(exp, cls, ver );                         \
    DECLARE_STREAMABLE_OPS( cls );                            \
    DECLARE_STREAMABLE_CTOR( cls )
```

See also: Chapter 9 in the *Programmer's Guide*

## DECLARE_STREAMABLE_FROM_BASE macro — objstrm.h

```
DECLARE_STREAMABLE_FROM_BASE(exp, cls, ver)
```

DECLARE_STREAMABLE_FROM_BASE is used in the same way as DECLARE_STREAMABLE; it should be used when the class being defined can be written and read using *Read* and *Write* functions defined in its base class without change. This usually occurs when a derived class overrides virtual functions in its base or provides different constructors, but does not add any data members. (If you used DECLARE_STREAMABLE in this situation, you would have to write *Read* and *Write* functions that merely called the base's *Read* and *Write* functions. Using DECLARE_STREAMABLE_FROM_BASE prevents this.)

DECLARE_STREAMABLE_FROM_BASE is defined as follows:

```
#define DECLARE_STREAMABLE_FROM_BASE( cls, base, ver )          \
    DECLARE_CASTABLE                                            \
    DECLARE_STREAMER_FROM_BASE(exp, cls, base, ver );           \
    DECLARE_STREAMABLE_OPS( cls );                              \
    DECLARE_STREAMABLE_CTOR( cls )
```

## DECLARE_ABSTRACT_STREAMABLE macro — objstrm.h

```
DECLARE_ABSTRACT_STREAMABLE(exp, cls, ver)
```

This macro is used in an abstract class. DECLARE_STREAMABLE doesn't work with an abstract class because an abstract class can never be instantiated, and the code that attempts to instantiate the object (*Build*) causes compiler errors. This macro expands to DECLARE_CASTABLE, DECLARE_ABSTRACT_STREAMER, DECLARE_STREAMABLE_OPS, and DECLARE_STREAMABLE_CTOR.

## DECLARE_STREAMER macro — objstrm.h

```
DECLARE_STREAMER(exp, cls, ver )
```

This macro defines a nested class within your streamable class; it contains the core of the streaming code. DECLARE_STREAMER declares the *Read* and *Write* function declarations, whose definitions you must provide, and the *Build* function that calls the *TStreamableClass* constructor. See DECLARE_STREAMABLE for an explanation of the parameters.

## DECLARE_STREAMER_FROM_BASE macro — objstrm.h

```
DECLARE_STREAMER_FROM_BASE( exp, cls, base )
```

This macro is used by DECLARE_STREAMABLE_FROM_BASE. It declares a nested *Streamer* class without the *Read* and *Write* functions. See DECLARE_STREAMABLE for a description of the parameters.

## DECLARE_ABSTRACT_STREAMER macro — objstrm.h

```
define DECLARE_ABSTRACT_STREAMER( exp, cls, ver )
```

This macro is used by DECLARE_ABSTRACT_STREAMABLE. It declares a nested *Streamer* class without the *Build* function. See DECLARE_STREAMABLE for an explanation of the parameters.

## DECLARE_CASTABLE macro — objstrm.h

```
DECLARE_CASTABLE
```

This macro provides declarations that provide a rudimentary typesafe downcast mechanism. This is useful for compilers that don't support run-time type information.

## DECLARE_STREAMABLE_OPS macro — objstrm.h

```
DECLARE_STREAMABLE_OPS(cls)
```

Declares the inserters and extractors. For template classes, DECLARE_STREAMABLE_OPS must use `class<...>` as the macro argument; other DECLAREs take only the class name.

## DECLARE_STREAMABLE_CTOR macro — objstrm.h

```
DECLARE_STREAMABLE_CTOR(cls)
```

Declares the constructor called by the *Streamer::Build* function.

## IMPLEMENT_STREAMABLE macros — objstrm.h

```
IMPLEMENT_STREAMABLE(cls)
IMPLEMENT_STREAMABLE1(cls, base1)
IMPLEMENT_STREAMABLE2(cls, base1, base2)
IMPLEMENT_STREAMABLE3(cls, base1, base2, base3)
IMPLEMENT_STREAMABLE4(cls, base1, base2, base3, base4)
IMPLEMENT_STREAMABLE5(cls, base1, base2, base3, base4, base5)
```

The IMPLEMENT_STREAMABLE macros generate the registration object for the class via IMPLEMENT_STREAMABLE_CLASS, and generate the various member functions that are needed for a streamable class via IMPLEMENT_ABSTRACT_STREAMABLE.

IMPLEMENT_STREAMABLE is used when the class has no base classes other than TStreamableBase. Its only parameter is the name of the class. The numbered versions (IMPLEMENT_STREAMABLE1, IMPLEMENT_STREAMABLE2, and so on) are for classes that have bases. Each base class, including all virtual bases, must be listed in the IMPLEMENT_STREAMABLE macro invocation.

The individual components comprising these macros can be used separately for special situations, such for as custom constructors.

## IMPLEMENT_STREAMABLE_CLASS macro — objstrm.h

```
IMPLEMENT_STREAMABLE_CLASS(cls)
```

Constructs a *TStreamableClass* class instance.

## IMPLEMENT_STREAMABLE_CTOR macros — objstrm.h

```
IMPLEMENT_STREAMABLE_CTOR(cls)
IMPLEMENT_STREAMABLE_CTOR1(cls, base1)
IMPLEMENT_STREAMABLE_CTOR2(cls, base1, base2)
IMPLEMENT_STREAMABLE_CTOR3(cls, base1, base2, base3)
IMPLEMENT_STREAMABLE_CTOR4(cls, base1, base2, base3, base4)
IMPLEMENT_STREAMABLE_CTOR5(cls, base1, base2, base3, base4, base5)
```

Defines the constructor called by the *Build* function. All base classes must be listed in the appropriate macro.

## IMPLEMENT_STREAMABLE_POINTER macro — objstrm.h

```
IMPLEMENT_STREAMABLE_POINTER(cls)
```

Creates the instance pointer extraction operator (>>).

## IMPLEMENT_CASTABLE_ID macro — objstrm.h

```
IMPLEMENT_CASTABLE_ID( cls )
```

Sets the typesafe downcast identifier.

## IMPLEMENT_CASTABLE macros — objstrm.h

```
IMPLEMENT_CASTABLE( cls )
IMPLEMENT_CASTABLE1( cls )
IMPLEMENT_CASTABLE2( cls )
IMPLEMENT_CASTABLE3( cls )
IMPLEMENT_CASTABLE4( cls )
IMPLEMENT_CASTABLE5( cls )
```

These macros implement code that supports the typesafe downcast mechanism.

## IMPLEMENT_STREAMER macro — objstrm.h

```
IMPLEMENT_STREAMER( cls )
```

Defines the *Streamer* constructor.

## IMPLEMENT_ABSTRACT_STREAMABLE macros — objstrm.h

```
IMPLEMENT_ABSTRACT_STREAMABLE( cls )
IMPLEMENT_ABSTRACT_STREAMABLE1( cls )
IMPLEMENT_ABSTRACT_STREAMABLE2( cls )
IMPLEMENT_ABSTRACT_STREAMABLE3( cls )
IMPLEMENT_ABSTRACT_STREAMABLE4( cls )
IMPLEMENT_ABSTRACT_STREAMABLE5( cls )
```

This macro expands to IMPLEMENT_STREAMER (which defines the *Streamer* constructor), IMPLEMENT_STREAMABLE_CTOR (which defines the *TStreamableClass* constructor), and IMPLEMENT_STREAMABLE_POINTER (which defines the instance pointer extraction operator).

## IMPLEMENT_STREAMABLE_FROM_BASE macro objstrm.h

```
IMPLEMENT_STREAMABLE_FROM_BASE( cls, base1 )
```

This macro expands to IMPLEMENT_STREAMABLE_CLASS (which constructs a *TStreamableClass* instance), IMPLEMENT_STREAMABLE_CTOR1 (which defines a one base class constructor that is called by *Build*), and IMPLEMENT_STREAMABLE_POINTER (which defines the instance pointer extraction operator).

C H A P T E R 7

# The C++ container classes

See Chapter 7 in the *Programmer's Guide* for information on using containers.

This chapter is a reference guide to the Borland C++ container classes. Each container class belongs to one of the following groups, which are listed here with their associated header-file names.

- Array (arrays.h)
- Association (assoc.h)
- Bag (bags.h)
- Binary tree (binimp.h)
- Dequeue (deques.h)
- Dictionary (dict.h)
- Double-linked list (dlistimp.h)
- Hash table (hashimp.h)
- List (listimp.h)
- Queue (queues.h)
- Set (sets.h)
- Stack (stacks.h)
- Vector (vectimp.h)

## TMArrayAsVector template — arrays.h

*TMArrayAsVector* implements a managed array of objects of type *T*, using a vector as the underlying implementation. It requires an == operator for type *T*.

### Type definitions

**CondFunc**

```
typedef int ( *CondFunc)(const T &, void *);
```

Function type used as a parameter to *FirstThat* and *LastThat* member functions.

**IterFunc**

```
typedef void ( *IterFunc)(T &, void *);
```

Function type used as a parameter to the *ForEach* member function.

### Public constructors

**Constructor**

```
TMArrayAsVector( int upper, int lower = 0, int delta = 0 )
```

Creates an array with an upper bound of *upper*, a lower bound of *lower*, and a growth delta of *delta*.

## Public member functions

**Add**

```
int Add( const T& t )
```

Adds a *T* object at the next available index at the end of an array. Adding an element beyond the upper bound leads to an overflow condition. If overflow occurs and *delta* is nonzero, the array is expanded (by sufficient multiples of *delta* bytes) to accommodate the addition. If *delta* is zero, *Add* fails. *Add* returns 0 if it couldn't add the object.

**AddAt**

```
int AddAt( const T& t, int loc )
```

Adds a *T* object at the specified index. If that index is occupied, it moves the object up to make room for the added object. If *loc* is beyond the upper bound, the array is expanded if *delta* (see the constructor) is nonzero. If *delta* is zero, attempting to *AddAt* beyond the upper bound gives an error.

**ArraySize**

```
unsigned ArraySize() const
```

Returns the current number of cells allocated.

**Destroy**

```
int Destroy( int i )
```

Removes the object at the given index. The object will be destroyed.

```
int Destroy( const T& t )
```

Removes the given object and destroys it.

**Detach**

```
int Detach( int loc, TShouldDelete::DeleteType dt =TShouldDelete::NoDelete )

int Detach( const T& t, TShouldDelete::DeleteType dt =
TShouldDelete::NoDelete )
```

The first version removes the object at *loc*; the second version removes the first object that compares equal to the specified object. The value of *dt* and the current ownership setting determine whether the object itself will be deleted. *DeleteType* is defined in the base class *TShouldDelete* as `enum { NoDelete, DefDelete, Delete }`. The default value of *dt*, *NoDelete*, means that the object will not be deleted regardless of ownership. With *dt* set to *Delete*, the object will be deleted regardless of ownership. If *dt* is set to *DefDelete*, the object will be deleted only if the array owns its elements.

See also: *TShouldDelete::ownsElements*

**FirstThat**

```
T *FirstThat(CondFunc, void *args ) const
```

Returns a pointer to the first object in the array that satisfies a given condition. You supply a test-function pointer *f* that returns true for a certain condition. You can pass arbitrary arguments via *args*. Returns 0 if no object in the array meets the condition.

See also: *LastThat*

**Flush**

```
void Flush( TShouldDelete::DeleteType dt = TShouldDelete::DefDelete )
```

Removes all elements from the array without destroying the array. The value of *dt* determines whether the elements themselves are destroyed. By default, the ownership status of the array determines their fate, as explained in the *Detach* member function. You can also set *dt* to *Delete* and *NoDelete*.

See also: *Detach*

**ForEach**

```
void ForEach(IterFunc, void *args )
```

*ForEach* creates an internal iterator to execute the given function for each element in the array. The *args* argument lets you pass arbitrary data to this function.

**GetItemsInContainer**

```
unsigned GetItemsInContainer() const
```

Returns the number of items in the array, as distinguished from *ArraySize*, which returns the size of the array.

**HasMember**

```
int HasMember( const T& t ) const
```

Returns 1 if the given object is found in the array; otherwise returns 0.

**IsEmpty**

```
int IsEmpty() const
```

Returns 1 if the array contains no elements; otherwise returns 0.

**IsFull**

```
int IsFull() const
```

Returns 1 if the array is full; otherwise returns 0. The array is full if *delta* is not equal to 0 and if the number of items in the container equals the value returned by *ArraySize*.

**LastThat**

```
T *LastThat( int ( * f)(const T &, void *), void *args ) const
```

Returns a pointer to the last object in the array that satisfies a given condition. You supply a test function pointer, *f*, that returns true for a certain condition. You can pass arbitrary arguments via *args*. Returns 0 if no object in the array meets the condition. Note that *LastThat* creates its own internal iterator, so you can treat it as a "search" function.

See also: *FirstThat*, *ForEach*

**LowerBound**

```
int LowerBound() const
```

Returns the array's *lowerbound*.

**UpperBound**

```
int UpperBound() const
```

Returns the array's current *upperbound*.

## Protected member functions

**BoundBase**

```
int BoundBase( unsigned loc ) const
```

*Boundbase* adjust vectors, which are zero-based, to arrays, which aren't zero-based. See *ZeroBase*.

**Find**

```
int Find( const T& t ) const
```

Finds the specified object and returns the object's index; otherwise returns INT_MAX.

**Grow**

```
void Grow( int loc )
```

Increases the size of the array, in either direction, so that *loc* is a valid index.

**InsertEntry**

```
void InsertEntry( int loc )
```

Creates an object and inserts it at *loc*, moving entries above *loc* up by one.

**ItemAt**

```
T ItemAt( int i ) const
```

Returns a copy of the object stored at location *i*.

**Reallocate**

```
int Reallocate( unsigned sz, unsigned offset = 0 )
```

If *delta* (see the constructor) is zero, *reallocate* returns 0. Otherwise, *reallocate* tries to create a new array of size *sz* (adjusted upwards to the nearest multiple of *delta*). The existing array is copied to the expanded array and then deleted. In an array of pointers, the entries are zeroed for each unused element. In an array of objects, the default constructor is invoked for each unused element. *offset* is the location in the new vector where the first element of the old vector should be copied. This is needed when the array has to be extended downward.

**RemoveEntry**

```
void RemoveEntry( int loc )
```

Removes element at the *loc* index into the array, and reduces the array by one element. Elements from index (*loc* + 1) upward are copied to positions *loc*, (*loc* + 1), and so on. The original element at *loc* is lost.

**SetData**

```
void SetData( int loc, const T& t )
```

The given *t* replaces the existing element at the index *loc*.

**ZeroBase**

```
unsigned ZeroBase( int loc ) const
```

Returns the location relative to *lowerbound* (*loc* – *lowerbound*).

### Operators

**operator [ ]**

```
T& operator [](  int loc )
T& operator [](  int loc ) const
```

Returns a reference to the element at the location specified by *loc*. the non-**const** version resizes the array if it's necessary to make *loc* a valid index. The **const** throws an exception in the debugging version on an attempt to index out of bounds.

## TMArrayAsVectorIterator template — arrays.h

Implements an iterator object to traverse *TMArrayAsVector* objects.

### Public constructors

**Constructor**

```
TMArrayAsVectorIterator( const TMArrayAsVector<T,Alloc> & a ) :
```

Creates an iterator object to traverse *TMArrayAsVector* objects.

### Public member functions

**Current**

```
Const T& Current();
```

Returns the current object.

**Restart**

```
void Restart();
void Restart( unsigned start, unsigned stop );
```

Restarts iteration from the beginning, or over the specified range.

### Operators

**operator ++**

```
Const T& operator ++(int);
```

Moves to the next object, and returns the object that was current before the move (post-increment).

```
Const T& operator ++();
```

Moves to the next object, and returns the object that was current after the move (pre-increment).

**operator int**

```
operator int() const
```

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

## TArrayAsVector template — arrays.h

*TArrayAsVector* implements an array of objects of type *T*, using a vector as the underlying implementation. *TStandardAllocator* is used to manage memory. See *TMArrayAsVector* on page 355 for members.

### Public constructors

**Constructor**

```
TArrayAsVector( int upper, int lower = 0, int delta = 0 ) :
```

Creates an array with an upper bound of *upper*, a lower bound of *lower*, and a growth delta of *delta*.

## TArrayAsVectorIterator template — arrays.h

Implements an iterator object to traverse *TArrayAsVector* objects. See *TMArrayAsVectorIterator* on page 359 for members.

### Public constructors

**Constructor**

```
TArrayAsVectorIterator( const TArrayAsVector<T> & a )
```

Creates an iterator object to traverse *TArrayAsVector* objects.

## TMIArrayAsVector template — arrays.h

Implements a managed, indirect array of objects of type *T*, using a vector as the underlying implementation.

### Type definitions

**CondFunc**

```
typedef int ( *CondFunc)(const T &, void *);
```

Function type used as a parameter to *FirstThat* and *LastThat* member functions.

**IterFunc**

```
typedef void ( *IterFunc)(T &, void *);
```

Function type used as a parameter to *ForEach* member function.

## Public constructors

**Constructor**

```
TMIArrayAsVector( int upper, int lower = 0, int delta = 0 )
```

Creates an indirect array with an upper bound of *upper*, a lower bound of *lower*, and a growth delta of *delta*.

## Public member functions

**Add**

```
int Add( T *t )
```

Adds a pointer to a *T* object at the next available index at the end of an array. Adding an element beyond the upper bound leads to an overflow condition. If overflow occurs and *delta* is nonzero, the array is expanded (by sufficient multiples of *delta* bytes) to accommodate the addition. If *delta* is zero, *Add* fails. *Add* returns 0 if the object couldn't be added.

**AddAt**

```
int AddAt( T *t, int loc )
```

Adds a pointer to a *T* object at the specified index. If that index is occupied, it moves the object up to make room for the added object. If *loc* is beyond the upper bound, the array is expanded if *delta* (see the constructor) is nonzero. If *delta* is zero, attempting to *AddAt* beyond the upper bound gives an error.

**ArraySize**

```
unsigned ArraySize() const
```

Returns the current number of cells allocated.

**Destroy**

```
int Destroy( int i )
```

Removes the object at the given index. The object will be deleted.

```
int Destroy( T *t )
```

Removes the object pointed to by *t* and deletes it.

**Detach**

```
int Detach( T *t, DeleteType dt = NoDelete )
int Detach( int loc, DeleteType dt = NoDelete )
```

The first version removes the object pointer at *loc*; the second version removes the specified pointer. The value of *dt* and the current ownership

setting determine whether the object itself will be deleted. *DeleteType* is defined in the base class *TShouldDelete* as `enum { NoDelete, DefDelete, Delete }`. The default value of *dt*, *NoDelete*, means that the object will not be deleted regardless of ownership. With *dt* set to *Delete*, the object will be deleted regardless of ownership. If *dt* is set to *DefDelete*, the object will be deleted only if the array owns its elements.

See also: *TShouldDelete::ownsElements*

**FirstThat**

```
T *FirstThat(CondFunc, void *args ) const
```

Returns a pointer to the first element in the array that satisfies a given condition. You supply a test-function pointer *f* that returns true for a certain condition. You can pass arbitrary arguments via *args*. Returns 0 if no object in the container meets the condition. Note that *FirstThat* creates its own internal iterator, so you can treat it as a "search" function.

See also: *LastThat*

**Find**

```
int Find( const T *t ) const
```

Finds the first specified object pointer and returns the index. Returns INT_MAX not found.

**Flush**

```
void Flush( DeleteType dt = DefDelete )
```

Removes all elements from the array without destroying the array. The value of *dt* determines whether the elements themselves are destroyed. By default, the ownership status of the array determines their fate, as explained in the *Detach* member function. You can also set *dt* to *Delete* and *NoDelete*.

See also: *Detach*

**ForEach**

```
void ForEach(IterFunc, void *args )
```

*ForEach* creates an internal iterator to execute the given function for each element in the container. The *args* argument lets you pass arbitrary data to this function.

**GetItemsInContainer**

```
unsigned GetItemsInContainer() const
```

Returns the number of items in the array.

**HasMember**

```
int HasMember( const T& t ) const
```

Returns 1 if the given object is found in the array; otherwise returns 0.

**IsEmpty**

```
int IsEmpty() const
```

Returns 1 if the array contains no elements; otherwise returns 0.

**IsFull**

```
int IsFull() const
```

Returns 1 if the array is full; otherwise returns 0.

**LastThat**

```
T *LastThat( int ( * f)(const T &, void *), void *args ) const
```

Returns a pointer to the last element in the array that satisfies a given condition. You supply a test function pointer, *f*, that returns true for a certain condition. You can pass arbitrary arguments via *args*. Returns 0 if no object in the container meets the condition. Note that *LastThat* creates its own internal iterator, so you can treat it as a "search" function.

See also: *FirstThat, ForEach*

**LowerBound**

```
int LowerBound() const
```

Returns the array's *lowerbound*.

**UpperBound**

```
int UpperBound() const
```

Returns the array's current *upperbound*.

## Protected member functions

**BoundBase**

```
int BoundBase( unsigned loc ) const
```

*Boundbase* adjust vectors, which are zero-based, to arrays, which aren't zero-based. See *ZeroBase*.

**Grow**

```
void Grow( int loc )
```

Increases the size of the array, in either direction, so that *loc* is a valid index.

**InsertEntry**

```
void InsertEntry( int loc )
```

Creates an object and inserts it at *loc*.

**ItemAt**

```
T ItemAt( int i ) const
```

Returns a copy of the object stored at location *i*.

**Reallocate**

```
int Reallocate( unsigned sz, unsigned offset = 0 )
```

If *delta* (see the constructor) is zero, *reallocate* returns 0. Otherwise, *reallocate* tries to create a new array of size *sz* (adjusted upward to the nearest multiple of *delta*). The existing array is copied to the expanded array and then deleted. In an array of pointers the entries are zeroed. In an array of objects the default constructor is invoked for each unused element. *offset* is the location in the new vector where the first element of the old vector should be copied. This is needed when the array has to be extended downward.

**RemoveEntry**

```
void RemoveEntry( int loc )
```

Removes element at *loc*, and reduces the array by one element. Elements from index (*loc* + 1) upward are copied to positions *loc*, (*loc* + 1), and so on. The original element at *loc* is lost.

**SetData**

```
void SetData( int loc, const T& t )
```

The given *t* replaces the existing element at the index *loc*.

**SqueezeEntry**

```
void SqueezeEntry( unsigned loc )
```

Removes element at *loc*, and reduces the array by one element. Elements from index (*loc* + 1) upward are copied to positions *loc*, (*loc* + 1), and so on. The original element at *loc* is lost.

**ZeroBase**

```
unsigned ZeroBase( int loc ) const
```

Returns the location relative to *lowerbound* (*loc* – *lowerbound*).

### Operators

**operator [ ]**

```
T * & operator [](  int loc )
T * & operator [](  int loc ) const
```

Returns a reference to the element at the location specified by *loc*. the non-**const** version resizes the array if it's necessary to make *loc* a valid index. The **const** throws an exception in the debugging version on an attempt to index out of bounds.

## TMIArrayAsVectorIterator template — arrays.h

Implements an iterator object to traverse *TMIArrayAsVector* objects. Based on *TMVectorIteratorImp*.

### Public constructors

**Constructor**

```
TMIArrayAsVectorIterator( const TMIArrayAsVector<T,Alloc> &a )
```

Creates an iterator object to traverse *TMArrayAsVector* objects.

### Public member functions

**Current**

```
T *Current();
```

Returns a pointer to the current object.

**Restart**

```
void Restart();
void Restart( unsigned start, unsigned sTop );
```

Restarts iteration from the beginning, or over the specified range.

### Operators

**operator ++**

```
Const T& operator ++(int);
```

Moves to the next object, and returns the object that was current before the move (post-increment).

```
Const T& operator ++();
```

Moves to the next object, and returns the object that was current after the move (pre-increment).

## TIArrayAsVector template — arrays.h

Implements an indirect array of objects of type *T*, using a vector as the underlying implementation. *TStandardAllocator* is used to manage memory. See *TMIArrayAsVector* on page 360 for members.

### Public constructors

**Constructor**

```
TIArrayAsVector( int upper, int lower = 0, int delta = 0 )
```

Creates an array with an upper bound of *upper*, a lower bound of *lower*, and a growth delta of *delta*.

## TIArrayAsVectorIterator template — arrays.h

Implements an iterator object to traverse *TIArrayAsVector* objects. Uses *TStandardAllocator* for memory management. See *TMIArrayAsVectorIterator* on page 364 for member functions and operators.

### Public constructors

**Constructor**

```
TIArrayAsVectorIterator( const TIArrayAsVector<T> &a ) :
TMIArrayAsVectorIterator<T,TStandardAllocator>(a)
```

Creates an iterator object to traverse *TIArrayAsVector* objects.

## TMSArrayAsVector template — arrays.h

Implements a sorted array of objects of type *T*, using a vector as the underlying implementation. With the exception of the *AddAt* member function, *TMSArrayAsVector* inherits its member functions and operators from *TMArrayAsVector*. See *TMArrayAsVector* on page 355 for members.

### Public constructors

**Constructor**

```
TMSArrayAsVector( int upper, int lower = 0, int delta = 0 )
```

Creates an array with an upper bound of *upper*, a lower bound of *lower*, and a growth delta of *delta*. It requires a < operator for type *T*.

## TMSArrayAsVectorIterator template — arrays.h

Implements an iterator object to traverse *TMSArrayAsVector* objects. See *TMArrayAsVectorIterator* on page 359 for members.

### Public constructors

**Constructor**

```
TMSArrayAsVectorIterator( const TMSArrayAsVector<T> & a ) :
```

Creates an iterator object to traverse *TSArrayAsVector* objects.

## TSArrayAsVector template — arrays.h

Implements a sorted array of objects of type *T*, using a vector as the underlying implementation. With the exception of the *AddAt* member function, *TSArrayAsVector* inherits its member functions and operators from *TMArrayAsVector*. See *TMArrayAsVector* 355 for members.

### Public constructors

**Constructor**

```
TSArrayAsVector( int upper, int lower = 0, int delta = 0 )
```

Creates an array with an upper bound of *upper*, a lower bound of *lower*, and a growth delta of *delta*. It requires a < operator for type *T*.

## TSArrayAsVectorIterator template — arrays.h

Implements an iterator object to traverse *TSArrayAsVector* objects. See *TMArrayAsVectorIterator* on page 359 for members.

### Public constructors

**Constructor**

```
TSArrayAsVectorIterator( const TSArrayAsVector<T> & a ) :
```

Creates an iterator object to traverse *TSArrayAsVector* objects.

## TISArrayAsVector template — arrays.h

Implements an indirect sorted array of objects of type *T*, using a vector as the underlying implementation. See *TMIArrayAsVector* on page 360 for members.

### Public constructors

**Constructor**

```
TISArrayAsVector( int upper, int lower = 0, int delta = 0 )
```

Creates an indirect array with an upper bound of *upper*, a lower bound of *lower*, and a growth delta of *delta*.

## TISArrayAsVectorIterator template — arrays.h

Implements an iterator object to traverse *TISArrayAsVector* objects. See *TMArrayAsVectorIterator* on page 359 for members.

### Public constructors

**Constructor**

```
TISArrayAsVectorIterator( const TISArrayAsVector<T> &a )
```

Creates an iterator object to traverse *TISArrayAsVector* objects.

## TMISArrayAsVector template — arrays.h

Implements a managed, indirect sorted array of objects of type *T*, using a vector as the underlying implementation. See *TMIArrayAsVector* on page 360 for members.

### Public constructors

**Constructor**

```
TMISArrayAsVector( int upper, int lower = 0, int delta = 0 )
```

Creates an indirect array with an upper bound of *upper*, a lower bound of *lower*, and a growth delta of *delta*.

## TMDDAssociation template — assoc.h

Implements a managed association, binding a direct key (*K*) with a direct value (*V*) . Assumes that *K* has a *HashValue* member function, or that a global function with the following prototype exists:

```
unsigned HashValue( K & );
```

*K* also must have a valid == operator. Class *A* represents the user-supplied storage manager.

### Public constructors

**Constructor**

```
TMDDAssociation()
```

The default constructor.

**Constructor**

```
TMDDAssociation( const K &k, const V &v )
```

Constructs an object that associates a copy of key object *k* with a copy of value object *v*.

### Public member functions

**HashValue**

```
unsigned HashValue()
```

Returns the hash value for the key.

**Key**

```
K Key()
```

Returns *KeyData*.

**Value**

```
V Value()
```

Returns *ValueData*.

### Operators

**operator ==**

Tests equality between keys.

## TDDAssociation template — assoc.h

Standard association (direct key, direct value). Implements an association, binding a direct key (*K*) with a direct value (*V*). Assumes that *K* has a *HashValue* member function, or that a global function with the following prototype exists:

```
unsigned HashValue( K & );
```

*K* also must have a valid == operator. See *TMDDAssociation* on page 368 for members.

### Public constructors

**Constructor**

```
TDDAssociation()
```

The default constructor.

**Constructor**

```
TDDAssociation( const K &k, const V &v )
```

Constructs an object that associates key object *k* with value object *v*.

## TMDIAssociation template — assoc.h

Implements a managed association, binding a direct key (*K*) with a indirect value (*V*) . Assumes that *K* has a *HashValue* member function, or that a global function with the following prototype exists:

```
unsigned HashValue( K & );
```

*K* also must have a valid == operator. Class *A* represents the user-supplied storage manager.

### Public constructors

**Constructor**

```
TMDIAssociation()
```

The default constructor.

**Constructor**

```
TMDIAssociation( K k, V * v )
```

Constructs an object that associates key object *k* with value object *v*.

### Public member functions

**HashValue**

```
unsigned HashValue()
```

Returns the hash value for the key.

**Key**

```
K Key()
```

Returns the key.

**Value**

```
const V * Value()
```

Returns a pointer to the data.

### Operators

**operator ==**

```
int operator == (const TMDDAssociation<K,V,A> & a)
```

Tests the equality between keys.

## TDIAssociation template — assoc.h

Implements an association, binding a direct key (*K*) with a indirect value (*V*). Assumes that *K* has a *HashValue* member function, or that a global function with the following prototype exists:

```
unsigned HashValue( K & );
```

*K* also must have a valid == operator. See *TMDIAssociation* on page 369 for members.

### Public constructors

**Constructor**

```
TDIAssociation()
```

The default constructor.

**Constructor** `TDIAssociation( K k, V * v )`

Constructs an object that associates key object *k* with value object *v*.

## TMIDAssociation template — assoc.h

Implements a managed association, binding an indirect key (*K*) with a direct value (*V*). Assumes that *K* has a *HashValue* member function, or that a global function with the following prototype exists:

```
unsigned HashValue( K & );
```

*K* also must have a valid == operator. Class *A* represents the user-supplied storage manager.

### Protected data members

**KeyData** `K KeyData;`

The key class passed into the template by the user.

**ValueData** `V ValueData;`

The value class passed into the template by the user.

### Public constructors

**Constructor** `TMIDAssociation()`

The default constructor.

**Constructor** `TMIDAssociation( K *k, V v )`

Constructs an object that associates key object *k* with value object *v*.

### Public member functions

**HashValue** `unsigned HashValue()`

Returns the hash value for the key.

**Key** `const K * Key()`

Returns a pointer to the key.

**Value** `V Value()`

Returns a copy of the data.

### Operators

**operator ==**

```
int operator == (const TMIDAssociation<K,V,A> & a)
```

Tests the equality between keys.

## TIDAssociation template — assoc.h

Implements an association, binding an indirect key (*K*) with a direct value (*V*) . Assumes that *K* has a *HashValue* member function, or that a global function with the following prototype exists:

```
unsigned HashValue( K & );
```

*K* also must have a valid == operator. See *TMIDAssociation* on page 371 for members.

### Public constructors

**Constructor**

```
TIDAssociation()
```

The default constructor.

**Constructor**

```
TIDAssociation( K * k, V v )
```

Constructs an object that associates key object **k* with value object *v*.

## TMIIAssociation template — assoc.h

Implements a managed association, binding an indirect key (*K*) with an indirect value (*V*) . Assumes that *K* has a *HashValue* member function, or that a global function with the following prototype exists:

```
unsigned HashValue( K & );
```

*K* also must have a valid == operator. Class *A* represents the user-supplied storage manager.

### Public constructors

**Constructor**

```
TMIIAssociation()
```

The default constructor.

**Constructor**

```
TMIIAssociation( K * k, V * v )
```

Constructs an object that associates key object **k* with value object **v*.

### Public member functions

**HashValue**

```
unsigned HashValue()
```

Returns the hash value for the key.

**Key**

```
const K * Key()
```

Returns a pointer to the key.

**Value**

```
V * Value()
```

Returns a pointer to the data.

### Operators

**operator ==**

```
int operator == (const TMIIAssociation<K,V,A> & a)
```

Tests equality between keys.

## TIIAssociation template — assoc.h

Standard association (indirect key, indirect value). Implements an association, binding an indirect key (*K*) with an indirect value (*V*) . Assumes that *K* has a *HashValue* member function, or that a global function with the following prototype exists:

```
unsigned HashValue( K & );
```

*K* also must have a valid == operator. See *TMIIAssociation* on page 372 for members.

### Public constructors

**Constructor**

```
TIIAssociation()
```

The default constructor.

**Constructor**

```
TIIAssociation( K *k, V * v )
```

Constructs an object that associates key object **k* with value object **v*.

# TMBagAsVector template — bags.h

Implements a managed bag of objects of type *T*, using a vector as the underlying implementation. Bags, unlike sets, can contain duplicate objects.

## Type definitions

**CondFunc**

```
typedef int ( *CondFunc)(const T &, void *);
```

Function type used as a parameter to *FirstThat* and *LastThat* member functions.

**IterFunc**

```
typedef void ( *IterFunc)(T &, void *);
```

Function type used as a parameter to *ForEach* member function.

## Public constructors

**Constructor**

```
TMBagAsVector( unsigned sz = DEFAULT_BAG_SIZE )
```

Constructs a managed, empty bag. *sz* represents the number of items the bag can hold.

## Public member functions

**Add**

```
int Add( const T& t )
```

Adds the given object to the bag.

**Detach**

```
int Detach( const T& t, TShouldDelete::DeleteType =
TShouldDelete::NoDelete )
```

Removes the specified object. The value of *dt* and the current ownership setting determine whether the object itself will be deleted. *DeleteType* is defined in the base class *TShouldDelete* as `enum { NoDelete, DefDelete, Delete }`. The default value of *dt*, *NoDelete*, means that the object will not be deleted regardless of ownership. With *dt* set to *Delete*, the object will be deleted regardless of ownership. If *dt* is set to *DefDelete*, the object will be deleted only if the bag owns its elements.

See also: *TShouldDelete::ownsElements*

**FindMember**

```
T* FindMember( const T& t ) const
```

Returns a pointer to the given object if found; otherwise returns 0.

**Flush**

```
void Flush( TShouldDelete::DeleteType = TShouldDelete::DefDelete )
```

Removes all the elements from the bag without destroying the bag. The value of *dt* determines whether the elements themselves are destroyed. By default, the ownership status of the bag determines their fate, as explained in the *Detach* member function. You can also set *dt* to *Delete* and *NoDelete*.

See also: *Detach*

**ForEach**

```
void ForEach(IterFunc, void *args )
```

*ForEach* creates an internal iterator to execute the given function for each element in the bag. The *args* argument lets you pass arbitrary data to this function.

**GetItemsInContainer**

```
int GetItemsInContainer() const
```

Returns the number of objects in the bag.

**HasMember**

```
int HasMember( const T& t ) const
```

Returns 1 if the given object is found; otherwise returns 0.

**IsEmpty**

```
int isEmpty() const
```

Returns 1 if the bag is empty; otherwise returns 0.

**IsFull**

```
int isFull() const
```

Returns 0.

### Protected member functions

**Find**

```
virtual T *Find( const T& ) const;
```

Returns a pointer to the given object if found; otherwise returns 0.

## TMBagAsVectorIterator template — bags.h

Implements an iterator object to traverse *TMBagAsVector* objects. See *TMArrayAsVectorIterator* on page 359 members.

### Public constructors

**Constructor**

```
TMBagAsVectorIterator( const TMBagAsVector<T,Alloc> & b )
```

Constructs an object that iterates on *TMBagAsVector* objects.

## TBagAsVector template — bags.h

Implements a bag of objects of type *T*, using a vector as the underlying implementation. *TStandardAllocator* is used to manage memory. See *TMBagAsVector* on page 374 for members.

### Public constructors

**Constructor**

```
TBagAsVector( unsigned sz = DEFAULT_BAG_SIZE )
```

Constructs an empty bag. *sz* represents the number of items the bag can hold.

## TBagAsVectorIterator template — bags.h

Implements an iterator object to traverse *TBagAsVector* objects. *TStandardAllocator* is used to manage memory. See *TMArrayAsVectorIterator* on page 359 for members.

### Public constructors

**Constructor**

```
TBagAsVectorIterator( const TBagAsVector<T> & b )
```

Constructs an object that iterates on *TBagAsVector* objects.

## TMIBagAsVector template — bags.h

Implements a managed bag of pointers to objects of type *T*, using a vector as the underlying implementation.

### Type definitions

**CondFunc**

```
typedef int ( *CondFunc)(const T &, void *);
```

Function type used as a parameter to *FirstThat* and *LastThat* member functions.

**IterFunc**

```
typedef void ( *IterFunc)(T &, void *);
```

Function type used as a parameter to *ForEach* member function.

## Public constructors

**Constructor**

```
TMIBagAsVector( unsigned sz = DEFAULT_BAG_SIZE )
```

Constructs an empty, managed, indirect bag. *sz* represents the initial number of slots allocated.

## Public member functions

**Add**

```
int Add( T *t )
```

Adds the given object pointer to the bag.

**Detach**

```
int Detach( T *t, DeleteType dt = NoDelete )
```

Removes the specified object pointer. The value of *dt* and the current ownership setting determine whether the object itself will be deleted. *DeleteType* is defined in the base class *TShouldDelete* as `enum { NoDelete, DefDelete, Delete }`. The default value of *dt*, *NoDelete*, means that the object will not be deleted regardless of ownership. With *dt* set to *Delete*, the object will be deleted regardless of ownership. If *dt* is set to *DefDelete*, the object will only be deleted if the bag owns its elements.

See also: *TShouldDelete::ownsElements*

**FindMember**

```
T *FindMember( T *t ) const
```

Returns a pointer to the object if found; otherwise returns 0.

**FirstThat**

```
T *FirstThat(CondFunc, void *args ) const
```

See: *TMBagAsVector::FirstThat*

**Flush**

```
void Flush( TShouldDelete::DeleteType dt = TShouldDelete::DefDelete )
```

Removes all the elements from the bag without destroying the bag. The value of *dt* determines whether the elements themselves are destroyed. By default, the ownership status of the bag determines their fate, as explained in the *Detach* member function. You can also set *dt* to *Delete* and *NoDelete*.

See also: *Detach*

**ForEach**

```
void ForEach(IterFunc, void *args )
```

*ForEach* creates an internal iterator to execute the given function for each element in the bag. The *args* argument lets you pass arbitrary data to this function.

**GetItemsInContainer**

```
int GetItemsInContainer() const
```

Returns the number of objects in the bag.

**HasMember**

```
int HasMember( const T& t ) const
```

Returns 1 if the given object is found; otherwise returns 0.

**IsEmpty**

```
int isEmpty() const
```

Returns 1 if the bag is empty; otherwise returns 0.

**IsFull**

```
int isFull() const
```

Returns 0.

**LastThat**

```
T *LastThat(CondFunc, void *args ) const
```

Returns a pointer to the last object in the array that satisfies a given condition. You supply a test function pointer, *f*, that returns true for a certain condition. You can pass arbitrary arguments via *args*. Returns 0 if no object in the array meets the condition. Note that *LastThat* creates its own internal iterator, so you can treat it as a "search" function.

## TMIBagAsVectorIterator template — bags.h

Implements an iterator object to traverse *TMIBagAsVector* objects. See *TMArrayAsVectorIterator* on page 359 for members.

### Public constructors

**Constructor**

```
TMIBagAsVectorIterator( const TMIBagAsVector<T,Alloc> & s )
```

Constructs an object that iterates on *TMIBagAsVector* objects.

## TIBagAsVector template — bags.h

Implements a bag of pointers to objects of type *T*, using a vector as the underlying implementation. *TStandardAllocator* is used to manage memory. See *TMIBagAsVector* on page 376 for members.

### Public constructors

**Constructor**

```
TIBagAsVector( unsigned sz = DEFAULT_BAG_SIZE )
```

Constructs an empty, managed, indirect bag. *sz* represents the initial number of slots allocated.

## TIBagAsVectorIterator template — bags.h

Implements an iterator object to traverse *TIBagAsVector* objects. *TStandardAllocator* is used to manage memory. See *TMArrayAsVectorIterator* on page 359 for members.

### Public constructors

**Constructor**

```
TIBagAsVectorIterator( const TIBagAsVector<T> & s )
```

Constructs an object that iterates on *TMIBagAsVector* objects.

## TBinarySearchTreeImp template — binimp.h

Implements an unbalanced binary tree. Class *T* must have < and == operators, and must have a default constructor.

### Public member functions

**Add**

```
int Add( const T& t )
```

Creates a new binary-tree node and inserts a copy of object *t* into it.

**Detach**

```
int Detach( const T& t, int del = 0 )
```

Removes the node containing item *t* from the tree.

**Find**

```
T * Find( const T& t ) const
```

Returns a pointer to the node containing item *t*.

**Flush**

```
void Flush;(int del=0);
```

Removes all items from the tree.

**ForEach**

```
void ForEach( IterFunc iter, void * args, IteratorOrder order = InOrder )
```

Creates an internal iterator that executes the given function *iter* for each item in the container. The *args* argument lets you pass arbitrary data to this function.

**GetItemsInContainer**

```
unsigned GetItemsInContainer();
```

Returns the number of items in the tree.

**Parent::IsEmpty**

```
int IsEmpty();
```

Returns 1 if the tree is empty; otherwise returns 0.

### Protected member functions

**EqualTo**

```
virtual int EqualTo( BinNode *n1, BinNode *n2 )
```

Tests the equality between two nodes.

**LessThan**

```
virtual int LessThan( BinNode *n1, BinNode *n2 )
```

Tests if node *n1* is less than node *n2*.

**DeleteNode**

```
virtual void DeleteNode( BinNode *node, int del)
```

Deletes *node*. The second parameter is ignored.

## TBinarySearchTreeIteratorImp template — binimp.h

Implements an iterator that traverses *TBinarySearchTreeImp* objects.

### Public constructors

**Constructor**

```
TBinarySearchTreeIteratorImp( TBinarySearchTreeImp<T>& tree,
TBinarySearchTreeBase::IteratorOrder order =
TBinarySearchTreeBase::InOrder ) : TBinaryTreeExternalIteratorBase( tree,
order ), CurNode(static_cast<TBinaryNodeImp<T>*>(Next()))
```

Constructs an iterator object that traverses a *TBinarySearchTreeImp* container.

### Public member functions

**Current**

```
const T& Current() const
```

Returns the current object.

**Restart**

```
void Restart()
```

Restarts iteration from the beginning of the tree.

### Operators

**operator int**

```
operator int() const
```

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

**operator ++**

```
const T& operator ++ ( int )
```

Moves to the next object in the tree, and returns the object that was current before the move (post-increment).

```
const T& operator ++ ()
```

Moves to the next object, and returns the object that was current after the move (pre-increment).

## TIBinarySearchTreeImp template — binimp.h

Implements an indirect unbalanced binary tree. Class *T* must have < and == operators, and must have a default constructor.

### Public member functions

**Add**

```
int Add( T * t )
```

Creates a new binary-tree node and inserts a pointer to object *t* into the tree.

**Detach**

```
int Detach( T * t, int del = 0 )
```

Removes the node containing item *t* from the tree. The item is deleted if *del* is 1.

**Find**

```
T * Find( T * t ) const
```

Returns a pointer to the node containing **t*.

**Flush**

```
void Flush;(int del=0);
```

Removes all items from the tree. The are deleted if *del* is 1. If *del* is 0 the items are not deleted.

**ForEach**

```
void ForEach( void ( *func)(T &, void *), void * args, IteratorOrder order
= InOrder )
```

Creates an internal iterator that executes the given function *f* for each item in the container. The *args* argument lets you pass arbitrary data to this function.

**GetItemsInContainer**

```
unsigned GetItemsInContainer();
```

Returns the number of items in the tree.

**Parent::IsEmpty**

```
int IsEmpty();
```

Returns 1 if the tree is empty; otherwise returns 0.

### Protected member functions

**EqualTo**

```
virtual int EqualTo( BinNode *n1, BinNode *n2 )
```

Tests the equality between two nodes.

**LessThan**

```
virtual int LessThan( BinNode *n1, BinNode *n2 )
```

Tests if node *n1* is less than node *n2*.

**DeleteNode**

```
virtual void DeleteNode( BinNode *node, int del)
```

Deletes *node*. The second parameter is ignored.

## TIBinarySearchTreeIteratorImp template — binimp.h

Implements an iterator that traverses *TIBinarySearchTreeImp* objects.

### Public constructors

**Constructor**

```
TIBinarySearchTreeIteratorImp( TIBinarySearchTreeImp<T>& tree,
TBinarySearchTreeBase::IteratorOrder order =
TBinarySearchTreeBase::InOrder ) :
TBinarySearchTreeIteratorImp<TVoidPointer>(tree,order)
```

Constructs an iterator object that traverses a *TIBinarySearchTreeImp* container.

### Public member functions

**Current**

```
T *Current() const
```

Returns a pointer to the current object.

**Restart**

```
void Restart()
```

Restarts iteration from the beginning of the tree.

### Operators

**operator int**

```
operator int() const
```

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

**operator ++**

```
T *operator ++ ( int i )
```

Moves to the next object in the tree, and returns a pointer to the object that was current before the move (post-increment).

```
T *operator ++ ()
```

Moves to the next object, and returns a pointer to the object that was current after the move (pre-increment).

## TMDequeAsVector template — deques.h

Implements a managed dequeue of *T* objects, using a vector as the underlying implementation.

### Type definitions

**CondFunc**

```
typedef int ( *CondFunc)(const T &, void *);
```

Function type used as a parameter to *FirstThat* and *LastThat* member functions.

**IterFunc**

```
typedef void ( *IterFunc)(T &, void *);
```

Function type used as a parameter to *ForEach* member function.

### Public constructors

**Constructor**

```
TMDequeAsVector( unsigned max = DEFAULT_DEQUE_SIZE )
```

Constructs a dequeue of *max* size.

### Public member functions

**FirstThat**

```
T *FirstThat(CondFunc, void *args ) const;
```

Returns a pointer to the first object in the dequeue that satisfies a given condition. You supply a test-function pointer *f* that returns true for a certain condition. You can pass arbitrary arguments via *args*. Returns 0 if no object in the array meets the condition.

See also: *LastThat*

**Flush**

```
void Flush( TShouldDelete::DeleteType = TShouldDelete::DefDelete )
```

Flushes the dequeue without destroying it. The fate of any objects removed depends on the current ownership status and the value of the *dt* argument.

See also: *TShouldDelete::ownsElements*

**ForEach**

```
void ForEach(IterFunc, void *args );
```

Executes function *f* for each dequeue element. *ForEach* creates an internal iterator to execute the given function for each element in the array. The *args* argument lets you pass arbitrary data to this function.

**GetItemsInContainer**

```
int GetItemsInContainer() const
```

Returns the number of items in the dequeue.

**GetLeft**

```
T GetLeft();
```

Returns the object at the left end and removes it from the dequeue. The debuggable version throws an exception when the dequeue is empty.

See also: *PeekLeft*

**GetRight**

```
T GetRight();
```

Same as *GetLeft*, except that the right end of the dequeue is returned.

See also: *PeekRight*

**IsEmpty**

```
int IsEmpty() const
```

Returns 1 if the dequeue has no elements; otherwise returns 0.

**IsFull**

```
int IsFull() const
```

Returns 1 if the dequeue is full; otherwise returns 0.

**LastThat**

```
T *LastThat(CondFunc, void *args ) const;
```

Returns a pointer to the last object in the dequeue that satisfies a given condition. You supply a test function pointer, *f*, that returns true for a certain condition. You can pass arbitrary arguments via *args*. Returns 0 if no object in the array meets the condition. Note that *LastThat* creates its own internal iterator, so you can treat it as a "search" function.

See also: *FirstThat*, *ForEach*

**PeekLeft**

```
Const T& PeekLeft() const
```

Returns the object at the left end (head) of the dequeue. The object stays in the dequeue.

See also: *GetLeft*

**PeekRight**

```
Const T& PeekRight() const
```

Returns the object at the right end (tail) of the dequeue. The object stays in the dequeue.

See also: *GetRight*

**PutLeft**

```
void PutLeft( const T& );
```

Adds (pushes) the given object at the left end (head) of the dequeue.

**PutRight**

```
void PutRight( const T& );
```

Adds (pushes) the given object at the right end (tail) of the dequeue.

### Protected data members

**Data**

```
Vect Data;
```

The vector containing the dequeue's data.

**Left**

```
unsigned Left;
```

Index to the leftmost element of the dequeue.

**Right**

```
unsigned Right;
```

Index to the rightmost element of the dequeue.

### Protected member functions

**Next**

```
unsigned Next( unsigned index ) const
```

Returns *index* + 1. Wraps around to the head of the dequeue.

See also: *Prev*

**Prev**

```
unsigned Prev( unsigned index ) const
```

Returns *index* – 1. Wraps around to the tail of the dequeue.

## TMDequeAsVectorIterator template — deques.h

Implements an iterator object for a managed, vector-based dequeue.

### Public constructors

**Constructor**

```
TMDequeAsVectorIterator( const TMDequeAsVector<T,Alloc> &d )
```

Constructs an object that iterates on *TMDequeAsVector* objects.

### Public member functions

**Current**

```
Const T& Current();
```

Returns the current object.

**Restart**

```
void Restart();
```

Restarts iteration.

### Operators

**operator ++**

```
Const T& operator ++ ( int );
```

Moves to the next object, and returns the object that was current before the move (post-increment).

```
Const T& operator ++ ();
```

Moves to the next object, and returns the object that was current after the move (pre-increment).

**operator int**

```
operator int();
```

Converts the iterator to an integer value for testing if objects remain in the iterator. Iterator converts to 0 if nothing remains in the iterator.

## TDequeAsVector template — deques.h

Implements a dequeue of *T* objects, using a vector as the underlying implementation. *TStandardAllocator* is used to manage memory. See *TMDequeAsVector* on page 383 for members.

### Public constructors

**Constructor**

```
TDequeAsVector( unsigned max = DEFAULT_DEQUE_SIZE )
```

Constructs a dequeue of *max* size.

## TDequeAsVectorIterator template — deques.h

Implements an iterator object for a vector-based dequeue. See *TMDequeAsVectorIterator* on page 385 for members.

### Public constructors

**Constructor**

```
TDequeAsVectorIterator( const TDequeAsVector<T> &d )
```

Constructs an object that iterates on *TMDequeAsVector* objects.

## TMIDequeAsVector template — deques.h

Implements a managed, indirect dequeue of pointers to objects of type *T*, using a vector as the underlying implementation.

### Type definitions

**CondFunc**

```
typedef int ( *CondFunc)(const T &, void *);
```

Function type used as a parameter to *FirstThat* and *LastThat* member functions.

**IterFunc**

```
typedef void ( *IterFunc)(T &, void *);
```

Function type used as a parameter to *ForEach* member function.

### Public constructors

**Constructor**

```
TMIDequeAsVector( unsigned sz = DEFAULT_DEQUE_SIZE )
```

Constructs an indirect dequeue of *max* size.

### Public member functions

**FirstThat**

```
T *FirstThat(CondFunc, void *args ) const;
```

Returns a pointer to the first object in the dequeue that satisfies a given condition. You supply a test-function pointer *f* that returns true for a certain condition. You can pass arbitrary arguments via *args*. Returns 0 if no object in the array meets the condition.

See also: *LastThat*

**Flush**

```
void Flush( TShouldDelete::DeleteType = TShouldDelete::DefDelete );
```

Flushes the dequeue without destroying it. The fate of any objects removed depends on the current ownership status and the value of the *dt* argument.

**ForEach**

```
void ForEach(IterFunc, void *args );
```

Executes function *f* for each dequeue element. *ForEach* creates an internal iterator to execute the given function for each element in the array. The *args* argument lets you pass arbitrary data to this function.

**GetItemsInContainer**

```
int GetItemsInContainer() const
```

Returns the number of items in the dequeue.

**GetLeft**

```
T *GetLeft()
```

Returns a pointer to the object at the left end and removes it from the dequeue. Returns 0 if the dequeue is empty.

See also: *PeekLeft*

**GetRight**

```
T *GetRight()
```

Same as *GetLeft*, except that the right end of the dequeue is returned.

See also: *PeekRight*

**IsEmpty**

```
int IsEmpty() const
```

Returns 1 if a dequeue has no elements; otherwise returns 0.

**IsFull**

```
int isFull() const
```

Returns 1 if a dequeue is full; otherwise returns 0.

**LastThat**

```
T *LastThat(CondFunc, void *args ) const;
```

Returns a pointer to the last object in the dequeue that satisfies a given condition. You supply a test function pointer, *f*, that returns true for a certain condition. You can pass arbitrary arguments via *args*. Returns 0 if no object in the array meets the condition. Note that *LastThat* creates its own internal iterator, so you can treat it as a "search" function.

See also: *FirstThat, ForEach*

**PeekLeft**

```
T *PeekLeft() const
```

Returns a pointer to the object at the left end (head) of the dequeue. The object stays in the dequeue.

See also: *GetLeft*

**PeekRight**

```
T *PeekRight() const
```

Returns the object at the right end (tail) of the dequeue. The object stays in the dequeue.

See also: *GetRight*

**PutLeft**

```
void PutLeft( T *t )
```

Adds (pushes) the given object pointer at the left end (head) of the dequeue.

**PutRight**

```
void PutRight( T *t )
```

Adds (pushes) the given object pointer at the right end (tail) of the dequeue.

## TMIDequeAsVectorIterator template — deques.h

Implements an iterator for the family of managed, indirect dequeues implemented as vectors. See *TMDequeAsVectorIterator* on page 385 for members.

### Public constructors

**Constructor**

```
TMIDequeAsVectorIterator( const TMIDequeAsVector<T,Alloc> &d )
```

Creates an object that iterates on *TMIDequeAsVector* objects.

## TIDequeAsVector template — deques.h

Implements an indirect dequeue of pointers to objects of type *T*, using a vector as the underlying implementation. See *TMIDequeAsVector* on page 387 for members.

### Public constructors

**Constructor**

```
TIDequeAsVector( unsigned sz = DEFAULT_DEQUE_SIZE ) :
TMIDequeAsVector<T,TStandardAllocator>(sz)
```

Constructs an indirect dequeue of *max* size.

## TIDequeAsVectorIterator template — deques.h

Implements an iterator for the family of indirect dequeues implemented as vectors. See *TMDequeAsVectorIterator* 385 for members.

### Public constructors

**Constructor**

```
TIDequeAsVectorIterator( const TIDequeAsVector<T> &d )
```

Constructs an object that iterates on *TIDequeAsVector* objects.

## TMDequeAsDoubleList template — deques.h

Implements a managed dequeue of objects of type *T*, using a double-linked list as the underlying implementation.

### Type definitions

**CondFunc**

```
typedef int ( *CondFunc)(const T &, void *);
```

Function type used as a parameter to *FirstThat* and *LastThat* member functions.

**IterFunc**

```
typedef void ( *IterFunc)(T &, void *);
```

Function type used as a parameter to *ForEach* member function.

### Public member functions

**FirstThat**

```
T *FirstThat(CondFunc, void *args ) const
```

Returns a pointer to the first object in the dequeue that satisfies a given condition. You supply a test-function pointer *f* that returns true for a certain condition. You can pass arbitrary arguments via *args*. Returns 0 if no object in the array meets the condition.

See also: *LastThat*

**Flush**

```
void Flush( int del )
```

Flushes the dequeue without destroying it. The fate of any objects removed depends on the current ownership status and the value of the *dt* argument.

**ForEach**

```
void ForEach(IterFunc, void *args )
```

Executes function *f* for each dequeue element. *ForEach* creates an internal iterator to execute the given function for each element in the array. The *args* argument lets you pass arbitrary data to this function.

**GetItemsInContainer** `int GetItemsInContainer() const`

Returns the number of items in the dequeue.

**GetLeft** `T GetLeft()`

Returns the object at the left end and removes it from the dequeue.

**GetRight** `T GetRight()`

Same as *GetLeft*, except that the right end of the dequeue is returned.

See also: *PeekRight*

**IsEmpty** `int IsEmpty() const`

Returns 1 if a dequeue has no elements; otherwise returns 0.

**IsFull** `int IsFull() const`

Returns 1 if a dequeue is full; otherwise returns 0.

**LastThat** `T *LastThat(CondFunc, void *args ) const`

Returns a pointer to the last object in the dequeue that satisfies a given condition. You supply a test function pointer, *f*, that returns true for a certain condition. You can pass arbitrary arguments via *args*. Returns 0 if no object in the array meets the condition. Note that *LastThat* creates its own internal iterator, so you can treat it as a "search" function.

See also: *FirstThat, ForEach*

**PeekLeft** `Const T& PeekLeft() const`

Returns a reference to the object at the left end (head) of the dequeue. The object stays in the dequeue.

See also: *GetLeft*

**PeekRight** `Const T& PeekRight() const`

Returns a reference to the object at the right end (tail) of the dequeue. The object stays in the dequeue.

See also: *GetRight*

**PutLeft** `void PutLeft( const T& t )`

Adds (pushes) the given object at the left end (head) of the dequeue.

**PutRight** `void PutRight( const T& t )`

Adds (pushes) the given object at the right end (tail) of the dequeue.

## TMDequeAsDoubleListIterator template — deques.h

Implements an iterator object for a double-list based deques. See *TMDoubleListIteratorImp* on page 404 for members.

### Public constructors

**Constructor**

```
TMDequeAsDoubleListIterator( const TMDequeAsDoubleList<T, Alloc> & s )
```

Constructs an object that iterates on *TMDequeAsDoubleList* objects.

## TDequeAsDoubleList template — deques.h

Implements a dequeue of objects of type *T*, using a double-linked list as the underlying implementation, and *TStandardAllocator* as its memory manager. See *TMDequeAsDoubleList* on page 390 for members.

## TDequeAsDoubleListIterator template — deques.h

Implements an iterator object for a double-list based dequeue.

### Public constructors

**Constructor**

```
TMDequeAsDoubleListIterator( const TMDequeAsDoubleList<T, Alloc> & s )
```

Constructs an object that iterates on *TDequeAsDoubleList* objects.

## TMIDequeAsDoubleList template — deques.h

Implements a managed dequeue of pointers to objects of type *T*, using a double-linked list as the underlying implementation.

### Type definitions

**CondFunc**

```
typedef int ( *CondFunc)(const T &, void *);
```

Function type used as a parameter to *FirstThat* and *LastThat* member functions.

**IterFunc**

```
typedef void ( *IterFunc)(T &, void *);
```

Function type used as a parameter to *ForEach* member function.

## Public member functions

**FirstThat**

```
T *FirstThat(CondFunc, void *args ) const
```

Returns a pointer to the first object in the dequeue that satisfies a given condition. You supply a test-function pointer *f* that returns true for a certain condition. You can pass arbitrary arguments via *args*. Returns 0 if no object in the array meets the condition.

See also: *LastThat*

**Flush**

```
void Flush( TShouldDelete::DeleteType dt = TShouldDelete::DefDelete )
```

Flushes the dequeue without destroying it. The fate of any objects removed depends on the current ownership status and the value of the *dt* argument.

**ForEach**

```
void ForEach(IterFunc, void *args )
```

Executes function *f* for each dequeue element. *ForEach* creates an internal iterator to execute the given function for each element in the array. The *args* argument lets you pass arbitrary data to this function.

**GetItemsInContainer**

```
int GetItemsInContainer() const
```

Returns the number of items in the dequeue.

**GetLeft**

```
T *GetLeft()
```

Returns a pointer to the object at the left end and removes it from the dequeue. Returns 0 if the dequeue is empty.

See also: *PeekLeft*

**GetRight**

```
T *GetRight()
```

Same as *GetLeft*, except that a pointer to the object at the right end of the dequeue is returned.

See also: *PeekRight*

**IsEmpty**

```
int IsEmpty() const
```

Returns 1 if the dequeue has no elements; otherwise returns 0.

**IsFull**

```
int IsFull() const
```

Returns 1 if the dequeue is full; otherwise returns 0.

**LastThat**

```
T *LastThat(CondFunc, void *args ) const
```

Returns a pointer to the last object in the dequeue that satisfies a given condition. You supply a test function pointer, *f*, that returns true for a certain condition. You can pass arbitrary arguments via *args*. Returns 0 if no object in the array meets the condition. Note that *LastThat* creates its own internal iterator, so you can treat it as a "search" function.

See also: *FirstThat*, *ForEach*

**PeekLeft**

```
T *PeekLeft() const
```

Returns a pointer to the object at the left end (head) of the dequeue. The object stays in the dequeue.

**PeekRight**

```
T *PeekRight() const
```

Returns the object at the right end (tail) of the dequeue. The object stays in the dequeue.

**PutLeft**

```
void PutLeft( T *t )
```

Adds (pushes) the given object pointer at the left end (head) of the dequeue.

**PutRight**

```
void PutRight( T *t )
```

Adds (pushes) the given object pointer at the right end (tail) of the dequeue.

## TMIDequeAsDoubleListIterator template — deques.h

Implements an iterator for the family of managed, indirect dequeues implemented as double lists. See *TMDoubleListIteratorImp* on page 404 for members.

### Public constructors

**Constructor**

```
TMIDequeAsDoubleListIterator( const TMIDequeAsDoubleList<T,Alloc> s )
```

Constructs an object that iterates on *TMIDequeAsDoubleList* objects.

## TIDequeAsDoubleList template — deques.h

Implements a dequeue of pointers to objects of type *T*, using a double-linked list as the underlying implementation. See *TMIDequeAsDoubleList* on page 392 for members.

## TIDequeAsDoubleListIterator template — deques.h

Implements an iterator for the family of indirect dequeues implemented as double lists. See *TMDoubleListIteratorImp* on page 404 for members.

### Public constructors

**Constructor**

```
TIDequeAsDoubleListIterator( const TIDequeAsDoubleList<T> & s )
```

Constructs an object that iterates on *TIDequeAsDoubleList* objects.

## TMDictionaryAsHashTable template — dict.h

Implements a managed dictionary using a hash table as the underlying FDS, and using the user-supplied storage allocator *A*. It assumes that *T* is one of the four types of associations, and that *T* has meaningful copy and == semantics as well as a default constructor.

### Protected data members

**HashTable**

```
TMHashTableImp<T,A> HashTable;
```

Implements the underlying hash table.

### Public constructors

**Constructor**

```
TMDictionaryAsHashTable( unsigned size = DEFAULT_HASH_TABLE_SIZE )
```

Constructs a dictionary with the specified *size*.

### Public member functions

**Add**

```
int Add( const T& t )
```

Adds item *t* if not already in the dictionary.

**Detach**

```
int Detach( const T& t, int del = 0 )
```

Removes item *t* from the dictionary, and deletes if *del* is 1. If *del* is 0 the item is not deleted.

**Find**

```
T * Find( constT& t )
```

Returns a pointer to item *t*.

**Flush**

```
void Flush( int del = 0 )
```

Removes all items from the dictionary. The items are deleted if *del* is 1. If *del* is 0 the items are not deleted.

**ForEach**

```
void ForEach( void ( *func)(T &, void *), void * args )
```

Creates an internal iterator that executes the given function *f* for each item in the container. The *args* argument lets you pass arbitrary data to this function.

**GetItemsInContainer**

```
inline unsigned GetItemsInContainer()
```

Returns the number of items in the dictionary.

**IsEmpty**

```
inline int IsEmpty()
```

Returns 1 if the dictionary is empty; otherwise returns 0.

## TMDictionaryAsHashTableIterator template — dict.h

Implements an iterator that traverses *TMDictionaryAsHashTable* objects, using the user-supplied storage allocator *A*.

### Public constructors

**Constructor**

```
TMDictionaryAsHashTableIterator(TMDictionaryAsHashTable<T,A> & t )
```

Constructs an iterator object that traverses a *TMDictionaryAsHashTable* container.

### Public member functions

**Current**

```
Const T& Current()
```

Returns the current object.

**Restart**

```
void Restart();
```

Restarts iteration from the beginning of the dictionary.

### Operators

**operator int**

```
operator int()
```

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

**operator ++**

```
Const T& operator ++ (int)
```

Moves to the next object, and returns the object that was current before the move (post-increment).

```
Const T& operator ++ ()
```

Moves to the next object, and returns the object that was current after the move (pre-increment).

## TDictionaryAsHashTable template — dict.h

Implements a dictionary objects of type *T*, using the system storage allocator *TStandardAllocator*. It assumes that *T* is one of the four types of associations, and that *T* has meaningful copy and == semantics as well as a default constructor. See *TMDictionaryAsHashTable* on page 395 for members.

### Public constructors

**Constructor**

```
TDictionaryAsHashTable( unsigned size = DEFAULT_HASH_TABLE_SIZE )
```

Constructs a dictionary with the specified *size*.

## TDictionaryAsHashTableIterator template — dict.h

Implements an iterator that traverses *TDictionaryAsHashTable* objects, using the system storage allocator *TStandardAllocator*.

### Public constructors

**Constructor**

```
TDictionaryAsHashTableIterator( TDictionaryAsHashTable<T> & t )
```

Constructs an iterator object that traverses a *TDictionaryAsHashTable* container.

## TMIDictionaryAsHashTable template — dict.h

Implements a managed indirect dictionary using a hash table as the underlying FDS, and using the user-supplied storage allocator *A*. It assumes that *T* is of class *TAssociation*.

### Public constructors

**Constructor**

```
TMIDictionaryAsHashTable( unsigned size = DEFAULT_HASH_TABLE_SIZE )
```

Constructs an indirect dictionary with the specified *size*.

### Public member functions

**Add**

```
int Add( T * t )
```

Adds a pointer to item *t* if not already in the dictionary.

**Detach**

```
int Detach( T * t, int del = 0 )
```

Removes the pointer to item *t* from the dictionary, and deletes if *del* is 1. If *del* is 0 the item is not deleted.

**Find**

```
T * Find( T * t )
```

Returns a pointer to item *t*.

**Flush**

```
void Flush( int del = 0 )
```

Removes all items from the dictionary. The item is deleted if *del* is 1. If *del* is 0 the item is not deleted.

**ForEach**

```
void ForEach( void ( *func)(T &, void *), void * args );
```

Creates an internal iterator that executes the given function *f* for each item in the container. The *args* argument lets you pass arbitrary data to this function.

**GetItemsInContainer**

```
inline unsigned GetItemsInContainer()
```

Returns the number of items in the dictionary.

**IsEmpty**

```
inline int IsEmpty()
```

Returns 1 if the dictionary is empty; otherwise returns 0.

## TMIDictionaryAsHashTableIterator template — dict.h

Implements an iterator that traverses *TMIDictionaryAsHashTable* objects, using the user-supplied storage allocator *A*.

### Public constructors

**Constructor**

```
TMIDictionaryAsHashTableIterator( TMIDictionaryAsHashTable<T,A> & t )
```

Constructs an iterator object that traverses a *TMIDictionaryAsHashTable* container.

### Public member functions

**Current**

```
T *Current()
```

Returns a pointer to the current object.

**Restart**

```
void Restart();
```

Restarts iteration from the beginning of the dictionary.

### Operators

**operator int**

```
operator int()
```

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

**operator ++**

```
T *operator ++ (int)
```

Moves to the next object, and returns a pointer to the object that was current before the move (post-increment).

```
T *operator ++ ()
```

Moves to the next object, and returns a pointer to the object that was current after the move (pre-increment).

## TIDictionaryAsHashTable template — dict.h

Implements an indirect dictionary using a hash table as the underlying FDS, and using the system storage allocator *TStandardAllocator*. It assumes that *T* is one of the four types of associations. See *TMIDictionaryAsHashTable* on page 398 for members.

### Public constructors

**Constructor**

```
TIDictionaryAsHashTable( unsigned size = DEFAULT_HASH_TABLE_SIZE )
```

Constructs an indirect dictionary with the specified *size*.

## TIDictionaryAsHashTableIterator template — dict.h

Implements an iterator that traverses *TIDictionaryAsHashTable* objects, using the user-supplied storage allocator *A*. See *TMIDictionaryAsHashTableIterator* on page 399 for members.

### Public constructors

**Constructor**

```
TIDictionaryAsHashTableIterator( TIDictionaryAsHashTable<T> & t )
```

Constructs an iterator object that traverses a *TIDictionaryAsHashTable* container.

## TDictionary template — dict.h

A simplified name for *TDictionaryAsHashTable*. See *TDictionaryAsHashTable* on page 397 for members.

## TDictionaryIterator template — dict.h

A simplified name for *TDictionaryAsHashTableIterator*. See *TDictionaryAsHashTableIterator* on page 397 for members.

### Public constructors

**Constructor**

```
TDictionaryIterator( const TDictionary<T> & a )
```

Constructs an iterator object that traverses a *TDictionary* container.

## TMDoubleListElement template — dlistimp.h

This class defines the nodes for double-list classes *TMDoubleListImp* and *TMIDoubleListImp*.

### Public data members

**data**

```
T data;
```

Data object contained in the double list.

**Next**

```
TMDoubleListElement<T> *Next;
```

A pointer to the next element in the double list.

**Prev**

```
TMDoubleListElement<T> *Prev;
```

A pointer to the previous element in the double list.

### Public constructors

**Constructor**

```
TMDoubleListElement();
```

Constructs a double-list element.

**Constructor**

```
TMDoubleListElement( T& t, TMDoubleListElement<T> *p )
```

Constructs a double-list element, and inserts after the object pointed to by *p*.

### Operators

**operator delete**

```
void operator delete( void * );
```

Deletes an object.

**operator new**

```
void *operator new( size_t sz );
```

Allocates a memory block of *sz* amount, and returns a pointer to the memory block.

# TMDoubleListImp template — dlistimp.h

Implements a managed, double-linked list of objects of type *T*. Assumes that *T* has meaningful copy semantics, operator ==, and a default constructor.

## Type definitions

**CondFunc**

```
typedef int ( *CondFunc)(const T &, void *);
```

Function type used as a parameter to *FirstThat* and *LastThat* member functions.

**IterFunc**

```
typedef void ( *IterFunc)(T &, void *);
```

Function type used as a parameter to *ForEach* member function.

## Public constructors

**Constructor**

```
TMDoubleListImp()
```

Constructs an empty, managed, double-linked list.

## Public member functions

**Add**

```
int Add( const T& t );
```

Add the given object at the beginning of the list.

**AddAtHead**

```
int AddAtHead( const T& t );
```

Add the given object at the beginning of the list.

**AddAtTail**

```
int AddAtTail( const T& );
```

Adds the given object at the end (tail) the list.

**Detach**

```
int Detach( const T&, int = 0 );
```

Removes the first occurrence of the given object encountered by searching from the beginning of the list. For direct containers the second argument is ignored. For indirect containers the *int* argument determines if the detached object is itself destroyed. See *TShouldDelete* on page 460 for details.

**FirstThat**

```
T *FirstThat( int ( *)(const T &, void *), void * ) const;
```

Returns a pointer to the first object in the double-list that satisfies a given condition. You supply a test-function pointer *f* that returns true for a certain condition. You can pass arbitrary arguments via *args*. Returns 0 if no object in the array meets the condition.

**Flush**

```
void Flush( int = 0 );
```

Removes all elements from the list without destroying the list. The value of *dt* determines whether the elements themselves are destroyed. By default, the ownership status of the array determines their fate, as explained in the *Detach* member function. You can also set *dt* to *Delete* and *NoDelete*.

**ForEach**

```
void ForEach(IterFunc, void * );
```

*ForEach* creates an internal iterator to execute the given function for each element in the array. The *args* argument lets you pass arbitrary data to this function.

**IsEmpty**

```
int IsEmpty() const
```

Returns 1 if array contains no elements; otherwise returns 0.

**LastThat**

```
T *LastThat( int ( *)(const T &, void *), void * ) const;
```

Returns a pointer to the last object in the double list that satisfies a given condition. You supply a test function pointer, *f*, that returns true for a certain condition. You can pass arbitrary arguments via *args*. Returns 0 if no object in the array meets the condition. Note that *LastThat* creates its own internal iterator, so you can treat it as a "search" function.

See also: *FirstThat*, *ForEach*

**PeekHead**

```
Const T& PeekHead() const
```

Returns a reference to the *Head* item in the double list, without removing it.

**PeekTail**

```
Const T& PeekTail() const
```

Returns a reference to the *Tail* item in the double list, without removing it.

## Protected data members

**Head,Tail**

```
TMDoubleListElement<T> Head, Tail;
```

The head and tail items of the double list.

## Protected member functions

**FindDetach**

```
virtual TMDoubleListElement<T> *FindDetach( const T& t )
```

Determines whether an object is in the list, and returns a pointer to its predecessor. Returns 0 if not found.

**FindPred**

```
virtual TMDoubleListElement<T> *FindPred( const T& );
```

Finds the element that would be followed by the parameter. The function does not check whether the parameter is actually there. This can be used for inserting (insert after returned element pointer).

## TMDoubleListIteratorImp template — dlistimp.h

Implements a double list iterator. This iterator works with any direct double-linked list. For indirect lists, see *TMIDoubleListIteratorImp* on page 409.

### Public constructors

**Constructor**

```
TMDoubleListIteratorImp( const TDoubleListImp<T> &l )
```

Constructs an iterator that traverses *TDoubleListImp* objects.

### Public member functions

**Current**

```
Const T& Current()
```

Returns the current object.

**Restart**

```
void Restart()
```

Restarts iteration from the beginning of the list.

### Operators

**operator int**

```
operator int()
```

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

**operator ++**

```
const T& operator ++ ( int )
```

Moves to the next object, and returns the object that was current before the move (post-increment).

```
const T& operator ++ ()
```

Moves to the next object, and returns the object that was current after the move (pre-increment).

**operator – –**

```
const T& operator -- ( int )
```

Moves to the previous object, and returns the object that was current before the move (post-decrement).

```
const T& operator -- ()
```

Moves to the previous object, and returns the object that was current after the move (pre-decrement).

## TDoubleListImp template — dlistimp.h

Implements a double-linked list of objects of type *T*, using *TStandardAllocator* for memory management. Assumes that *T* has meaningful copy semantics and a default constructor. See *TMDoubleListImp* on page 402 for members.

### Public constructors

**Constructor**

```
TDoubleListImp()
```

Constructs an empty double-linked list.

## TDoubleListIteratorImp template — dlistimp.h

Implements a double list iterator. This iterator works with any direct double-linked list. See *TMDoubleListIteratorImp* on page 404 for members.

### Public constructors

**Constructor**

```
TDoubleListIteratorImp( const TDoubleListImp<T> &l )
```

Constructs an iterator that traverses *TDoubleListImp* objects.

## TMSDoubleListImp template — dlistimp.h

Implements a managed, sorted, double-linked list of objects of type *T*. It assumes that *T* has meaningful copy semantics, a == operator, a < operator, and a default constructor. See *TMDoubleListImp* on page 402 for members.

### Protected member functions

In addition to the following member functions, *TMSDoubleListImp* inherits member functions from *TMDoubleListImp* (see page 402).

**FindDetach**

```
virtual TMDoubleListElement<T> *FindDetach( const T& );
```

Determines whether an object is in the list, and returns a pointer to its predecessor. Returns 0 if not found.

**FindPred**

```
virtual TMDoubleListElement<T> *FindPred( const T& );
```

Finds the element that would be followed by the parameter. The function does not check whether the parameter is actually there. This can be used for inserting (insert after returned element pointer).

## TMSDoubleListIteratorImp template — dlistimp.h

Implements a double list iterator. This iterator works with any direct double-linked list. See *TMDoubleListIteratorImp* on page 404 for members.

### Public constructors

**Constructor**

```
TMSDoubleListIteratorImp( const TMSDoubleListImp<T,Alloc> &l )
```

Constructs an iterator that traverses *TMSDoubleListImp* objects.

## TSDoubleListImp template — dlistimp.h

Implements a sorted, double-linked list of objects of type *T*. It assumes that *T* has meaningful copy semantics, a meaningful < operator, and a default constructor. See *TMSDoubleListImp* on page 406 for members.

## TSDoubleListIteratorImp template — dlistimp.h

Implements a double list iterator. This iterator works with any direct double-linked list. See *TMDoubleListIteratorImp* on page 404 for members.

### Public constructors

**Constructor**

```
TSDoubleListIteratorImp( const TSDoubleListImp<T> &l )
```

Constructs an iterator that traverses *TSDoubleListImp* objects.

## TMIDoubleListImp template — dlistimp.h

Implements a managed, double-linked list of pointers to objects of type *T*.The contained objects need a valid == operator. Since pointers always have meaningful copy semantics, this class can handle any type of object.

### Type definitions

**CondFunc**

```
typedef int ( *CondFunc)(const T &, void *);
```

Function type used as a parameter to *FirstThat* and *LastThat* member functions.

**IterFunc**

```
typedef void ( *IterFunc)(T &, void *);
```

Function type used as a parameter to *ForEach* member function.

### Public member functions

**Add**

```
int Add( T *t )
```

Adds an object pointer to the double list.

**AddAtHead**

```
int AddAtHead( T *t );
```

Add the given object at the beginning of the list.

**AddAtTail**

```
int AddAtTail( T *t )
```

Adds an object pointer to the tail of the double list.

**Detach**

```
int Detach( T *t, int del = 0 )
```

Removes the given object pointer from the list. The second argument specifies whether the object should be deleted. See *TShouldDelete* on page 460.

**DetachAtHead**

```
int DetachAtHead( int del = 0 )
```

Deletes the object pointer from the head of the list.

**DetachAtTail**

```
int DetachAtTail( int del = 0 )
```

Deletes the object pointer from the tail of the list.

**FirstThat**

```
T *FirstThat( int ( *)(const T &, void *), void * ) const;
```

Returns a pointer to the first object in the double list that satisfies a given condition. You supply a test-function pointer *f* that returns true for a certain condition. You can pass arbitrary arguments via *args*. Returns 0 if no object in the array meets the condition.

See also: *LastThat*

**Flush**

```
void Flush( int = 0 );
```

Removes all elements from the list without destroying the list. The value of *dt* determines whether the elements themselves are destroyed. By default, the ownership status of the array determines their fate, as explained in the *Detach* member function. You can also set *dt* to *Delete* and *NoDelete*.

**ForEach**

```
void ForEach(IterFunc, void * );
```

Executes function *f* for each double-list element. *ForEach* creates an internal iterator to execute the given function for each element in the array. The *args* argument lets you pass arbitrary data to this function.

**GetItemsInContainer**

```
unsigned GetItemsInContainer() const
```

Returns the number of items in the array.

**IsEmpty**

```
int IsEmpty() const
```

Returns 1 if array contains no elements; otherwise returns 0.

**LastThat**

```
T *LastThat( int ( *)(const T &, void *), void * ) const;
```

Returns a pointer to the last object in the list that satisfies a given condition. You supply a test function pointer, *f*, that returns true for a certain condition. You can pass arbitrary arguments via *args*. Returns 0 if no object in the array meets the condition. Note that *LastThat* creates its own internal iterator, so you can treat it as a "search" function.

See also: *FirstThat*, *ForEach*

**PeekHead**

```
T *PeekHead() const
```

Returns the object pointer at the *Head* of the list, without removing it.

**PeekTail**

```
T *PeekTail() const
```

Returns the object pointer at the *Tail* of the list, without removing it.

### Protected member functions

**FindPred**

```
virtual TDoubleListElement<void *> *FindPred( void * );
```

Finds the element that would be followed by the parameter. The function does not check whether the parameter is actually there. This can be used for inserting (insert after returned element pointer).

## TMIDoubleListIteratorImp template — dlistimp.h

Implements a double list iterator. This iterator works with any indirect double list. For direct lists, see *TMDoubleListIteratorImp* on page 404.

### Public constructors

**Constructor**

```
TMIDoubleListIteratorImp( const TMIDoubleListImp<T,Alloc> &l )
```

Constructs an object that iterates on *TIDoubleListImp* objects.

### Public member functions

**Current**

```
T *Current()
```

Returns the current object pointer.

**Restart**

```
void Restart()
```

Restarts iteration from the beginning of the list.

### Operators

**operator ++**

```
T *operator ++ (int)
```

Moves to the next object, and returns the object that was current before the move (post-increment).

```
T *operator ++ ()
```

Moves to the next object, and returns the object that was current after the move (pre-increment).

## TIDoubleListImp template — dlistimp.h

Implements a double-linked list of pointers to objects of type *T*, using *TStandardAllocator* for memory management. Since pointers always have meaningful copy semantics, this class can handle any type of object. See *TMIDoubleListImp* on page 407 for members.

## TIDoubleListIteratorImp template — dlistimp.h

Implements a double list iterator. This iterator works with any indirect double list. See *TMIDoubleListIteratorImp* on page 409 for members.

### Public constructors

**Constructor**

```
TIDoubleListIteratorImp( const TIDoubleListImp<T> &l )
```

Constructs an object that iterates on *TIDoubleListImp* objects.

## TMISDoubleListImp template — dlistimp.h

Implements a managed, sorted, double-linked list of pointers to objects of type *T*. Since pointers always have meaningful copy semantics, this class can handle any type of object.

### Protected member functions

In addition to the member function described here, *TMISDoubleListImp* inherits member functions (see *TMIDoubleListImp* on page 407).

**FindDetach**

```
virtual TMDoubleListElement<void *> *FindDetach( void * );
```

Determines whether an object is in the list, and returns a pointer to its predecessor.

## TMISDoubleListIteratorImp template — dlistimp.h

Implements a double list iterator. This iterator works with any indirect, sorted double list. See *TMIDoubleListIteratorImp* on page 409 for members.

### Public constructors

**Constructor**

```
TMISDoubleListIteratorImp( const TMISDoubleListImp<T,Alloc> &l )
```

Constructs an object that iterates on *TMISDoubleListImp* objects.

## TISDoubleListImp template — dlistimp.h

Implements a sorted, double-linked list of pointers to objects of type *T*, using *TStandardAllocator* for memory management. Since pointers always have meaningful copy semantics, this class can handle any type of object. See *TMIDoubleListImp* on page 407 for members.

## TISDoubleListIteratorImp template — dlistimp.h

Implements a double list iterator. This iterator works with any indirect, sorted double list. See *TMIDoubleListIteratorImp* on page 409 for members.

### Public constructors

**Constructor**

```
TISDoubleListIteratorImp( const TISDoubleListImp<T> &l )
```

Constructs an object that iterates on *TMISDoubleListImp* objects.

## TMHashTableImp template — hashimp.h

Implements a managed hash table of objects of type *T*, using the user-supplied storage allocator *A*. It assumes that *T* has meaningful copy and == semantics, as well as a default constructor.

### Public constructors and destructor

**Constructor**

```
TMHashTableImp( unsigned aPrime = DEFAULT_HASH_TABLE_SIZE )
```

Constructs a hash table.

**Destructor**

```
~TMHashTableImp()
```

Calls member function *Flush* to delete the container.

### Public member functions

**Add**

```
int Add( const T& t );
```

Adds item *t* to the hash table.

**Detach**

```
int Detach( const T& t, int del=0 );
```

Removes item *t* from the hash table. If *del* is set to 0, *t* is deleted; if *del* is set to 1, *t* is not deleted.

**Find**

```
T * Find( const T& t ) const
```

Returns a pointer to item *t*.

**Flush**

```
void Flush( int del = 0 )
```

Flushes all items in the hash table. The hash table is destroyed if *del* is nonzero.

**ForEach**

```
void ForEach(void ( *f)(T &, void *), void *args);
```

Creates an internal iterator that executes the given function *f* for each item in the container. The *args* argument lets you pass arbitrary data to this function.

**GetItemsInContainer**

```
unsigned GetItemsInContainer() const
```

Returns the number of items in the hash table.

**IsEmpty**

```
int IsEmpty() const
```

Returns 1 if the hash table is empty; otherwise returns 0.

## TMHashTableIteratorImp template — hashimp.h

Implements an iterator for traversing *TMHashTableImp* containers, using the user-supplied storage allocator *Alloc*.

### Public constructors and destructor

**Constructor**

```
TMHashTableIteratorImp( const TMHashTableImp<T,A> & h )
```

Constructs an iterator object that traverses a *TMHashTableImp* container.

**Destructor**

```
~TMHashTableIteratorImp()
```

Destroys the iterator.

### Public member functions

**Current**

```
Const T& Current()
```

Returns the current object.

**Restart**

```
void Restart();
```

Restarts iteration from the beginning of the hash table.

### Operators

**operator int**

```
operator int()
```

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

**operator ++**

```
Const T& operator ++ (int)
```

Moves to the next object, and returns the object that was current before the move (post-increment).

```
Const T& operator ++ ()
```

Moves to the next object, and returns the object that was current after the move (pre-increment).

## THashTableImp template — hashimp.h

Implements a hash table of objects of type *T*, using the system storage allocator *TStandardAllocator*. It assumes that *T* has meaningful copy and == semantics as well as a default constructor. See *TMHashTableImp* on page 411 for members.

### Public constructors

**Constructor**

```
THashTableImp( unsigned aPrime = DEFAULT_HASH_TABLE_SIZE )
```

Constructs a hash table that uses *TStandardAllocator* for memory management.

## THashTableIteratorImp template — hashimp.h

Implements an iterator for traversing *THashTableImp* containers. See *TMHashTableIteratorImp* on page 412 for members.

### Public constructors

**Constructor**

```
THashTableIteratorImp( const THashTableImp<T,A> & h )
```

Constructs an iterator object that traverses a *THashTableImp* container.

## TMIHashTableImp template — hashimp.h

Implements a managed hash table of pointers to objects of type *T*, using the user-supplied storage allocator *Alloc*.

### Public constructors

**Constructor**

```
TMIHashTableImp( unsigned aPrime = DEFAULT_HASH_TABLE_SIZE )
```

Constructs an indirect hash table.

### Public member functions

**Add**

```
int Add( T * t )
```

Adds a pointer to item *t* to the hash table.

**Detach**

```
int Detach( T * t, int del = 0 )
```

Removes a pointer to item *t* from the hash table. *t* is deleted if *del* is set 1, and not deleted if *del* is set to 0.

**Find**

```
T * Find( const T * t ) const
```

Returns a pointer to item *t*.

**Flush**

```
void Flush( int del = 0 )
```

Flushes all items in the hash table. The hash table is destroyed if *del* is nonzero.

**ForEach**

```
void ForEach(void ( *f)(T &, void *), void *args);
```

Creates an internal iterator that executes the given function *f* for each item in the container. The *args* argument lets you pass arbitrary data to this function.

**GetItemsInContainer**

```
unsigned GetItemsInContainer() const
```

Returns the number of items in the hash table.

**IsEmpty**

```
int IsEmpty() const
```

Returns 1 if the hash table is empty; otherwise returns 0.

## TMIHashTableIteratorImp template — hashimp.h

Implements an iterator for traversing *TMIHashTableImp* containers.

### Public constructors

**Constructor**

```
TMIHashTableIteratorImp( const TMIHashTableImp<T,A> & h )
```

Constructs an iterator object that traverses a *TMIHashTableImp* container.

### Public member functions

**Current**

```
T *Current()
```

Returns a pointer to the current object.

**Restart**

```
void Restart();
```

Restarts iteration from the beginning of the hash table.

### Operators

**operator int**

```
operator int()
```

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

**operator ++**

```
T *operator ++ (int)
```

Moves to the next object, and returns the object pointer that was current before the move (post-increment).

```
T *operator ++ ()
```

Moves to the next object, and returns the object pointer that was current after the move (pre-increment).

## TIHashTableImp template — hashimp.h

Implements a hash table of pointers to objects of type *T*, using the system storage allocator *TStandardAllocator*. See *TMIHashTableImp* on page 414 for members.

### Public constructors

**Constructor**

```
TIHashTableImp( unsigned aPrime = DEFAULT_HASH_TABLE_SIZE )
```

Constructs an indirect hash table that uses the system storage allocator.

## TIHashTableIteratorImp template — hashimp.h

Implements an iterator object that traverses *TIHashTableImp* containers, and uses the system memory allocator *TStandardAllocator*. See *TMIHashTableIteratorImp* on page 415 for members.

### Public constructors

**Constructor**

```
TIHashTableIteratorImp( const TIHashTableImp<T> & h )
```

## TMListElement template — listimp.h

This class defines the nodes for *TMListImp* and *TMIListImp* and related classes.

### Public data members

**data**

```
T Data;
```

Data object contained in the list.

**Next**

```
TMListElement<T,Alloc> *Next;
```

A pointer to the next element in the list.

### Public constructors

**Constructor**

```
TMListElement();
```

Constructs a list element.

**Constructor**

```
TMListElement( T& t, TMListElement<T,Alloc> *p )
```

Constructs a list element, and places it after the object at location *p*.

### Operators

**operator delete**

```
void operator delete( void * );
```

Deletes an object.

**operator new**

```
void *operator new( size_t sz );
```

Allocates a memory block of *sz* amount, and returns a pointer to the memory block.

## TMListImp template — listimp.h

Implements a managed list of objects of type *T*. *TMListImp* assumes that *T* has meaningful copy semantics, and a default constructor.

### Type definitions

**CondFunc**

```
typedef int ( *CondFunc)(const T &, void *);
```

Function type used as a parameter to *FirstThat* and *LastThat* member functions.

**IterFunc**

```
typedef void ( *IterFunc)(T &, void *);
```

Function type used as a parameter to *ForEach* member function.

### Public constructors

**Constructor**

```
TMListImp()
```

Constructs an empty list.

## Public member functions

**Add**

```
int Add( const T& t );
```

Adds an object to the list.

**Detach**

```
int Detach( const T&, int = 0 );
```

Removes the given object from the list. Returns 0 for failure, 1 for success in removing the object. The second argument specifies whether the object should be deleted. See *TShouldDelete* on page 460.

**FirstThat**

```
T *FirstThat( int ( *)(const T &, void *), void * ) const;
```

Returns a pointer to the first object in the list that satisfies a given condition. You supply a test-function pointer *f* that returns true for a certain condition. You can pass arbitrary arguments via *args*. Returns 0 if no object in the array meets the condition.

See also: *LastThat*

**Flush**

```
void Flush( int del = 0 );
```

Flushes the list without destroying it.

**ForEach**

```
int Detach( const T&, int = 0 );
void ForEach(IterFunc, void * );
```

Executes function *f* for list element. *ForEach* creates an internal iterator to execute the given function for each element in the array. The *args* argument lets you pass arbitrary data to this function.

**IsEmpty**

```
int IsEmpty() const
```

Returns 1 if the list has no elements; otherwise returns 0.

**LastThat**

```
T *LastThat( int ( *)(const T &, void *), void * ) const;
```

Returns a pointer to the last object in the list that satisfies a given condition. You supply a test function pointer, *f*, that returns true for a certain condition. You can pass arbitrary arguments via *args*. Returns 0 if no object in the array meets the condition. Note that *LastThat* creates its own internal iterator, so you can treat it as a "search" function.

See also: *FirstThat*, *ForEach*

**PeekHead**

```
Const T& PeekHead() const
```

Returns a reference to the *Head* item in the list, without removing it.

### Protected data members

**Head, Tail**

```
TMListElement<T,Alloc> Head, Tail;
```

The elements before the first and after the last elements in the list.

### Protected member functions

**FindDetach**

```
virtual TMListElement<T,Alloc> *FindDetach( const T& t )
```

Determines whether an object is in the list, and returns a pointer to its predecessor. Returns 0 if not found.

**FindPred**

```
virtual TMListElement<T,Alloc> *FindPred( const T& );
```

Finds the element that would be followed by the parameter. The function does not check whether the parameter is actually there. This can be used for inserting (insert after returned element pointer).

## TMListIteratorImp template — listimp.h

Implements a list iterator that works on direct, managed list. For indirect list iteration see *TMIListIteratorImp* on page 422.

### Public constructors

**Constructor**

```
TMListIteratorImp(const TMListImp<T,Alloc> &l)
```

Constructs an iterator that traverses *TMListImp* objects.

### Public member functions

**Current**

```
Const T& Current()
```

Returns the current object.

**Restart**

```
void Restart()
```

Restarts iteration from the beginning of the list.

### Operators

**operator int**

```
operator int();
```

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

**operator ++**

```
Const T& operator ++ ( int )
```

Moves to the next object, and returns the object that was current before the move (post-increment).

```
Const T& operator ++ ()
```

Moves to the next object, and returns the object that was current after the move (pre-increment).

## TListImp template — listimp.h

Implements a list of objects of type *T*. *TListImp* assumes that *T* has meaningful copy semantics, and a default constructor. See *TMListImp* on page 417 for members.

## TListIteratorImp template — listimp.h

Implements a list iterator that works on direct, managed list. See *TMListIteratorImp* on page 419 for members.

### Public constructors

**Constructor**

```
TListIteratorImp( const TMListImp<T,TStandardAllocator> &l )
```

Constructs an iterator that traverses *TListImp* objects.

## TMSListImp template — listimp.h

Implements a managed, sorted list of objects of type *T*. *TMSListImp* assumes that T has meaningful copy semantics, a meaningful < operator, and a default constructor. See *TMListImp* on page 417 for members.

## TMSListIteratorImp template — listimp.h

Implements a list iterator that works on direct, managed, sorted list. See *TMListIteratorImp* on page 419 for members.

### Public constructors

**Constructor**

```
TMSListIteratorImp( const TMSListImp<T,Alloc> &l )
```

Constructs an iterator that traverses *TMSListImp* objects.

## TSListImp template — listimp.h

Implements a sorted list of objects of type *T*, using *TStandardAllocator* for memory management. *TSListImp* assumes that T has meaningful copy semantics, a meaningful < operator, and a default constructor. See *TMListImp* on page 417 for members.

## TSListIteratorImp template — listimp.h

Implements a list iterator that works on direct, sorted list. See *TMListIteratorImp* on page 419 for members.

## TMIListImp template — listimp.h

Implements a managed list of pointers to objects of type *T*. Since pointers always have meaningful copy semantics, this class can handle any type of object.

### Type definitions

**CondFunc**

```
typedef int ( *CondFunc)(const T &, void *);
```

Function type used as a parameter to *FirstThat* and *LastThat* member functions.

**IterFunc**

```
typedef void ( *IterFunc)(T &, void *);
```

Function type used as a parameter to *ForEach* member function.

### Public member functions

**Add**

```
int Add( T *t );
```

Adds an object pointer to the list.

**Detach**

```
int Detach( T *t, int del = 0 )
```

Removes the given object pointer from the list. The second argument specifies whether the object should be deleted. See *TShouldDelete* on page 460.

**FirstThat**

```
T *FirstThat( int ( *)(const T &, void *), void * ) const;
```

Returns a pointer to the first object in the list that satisfies a given condition. You supply a test-function pointer *f* that returns true for a certain condition. You can pass arbitrary arguments via *args*. Returns 0 if no object in the array meets the condition.

See also: *LastThat*

**ForEach**

```
void ForEach(IterFunc, void * )
```

Executes function *f* for each list element. *ForEach* creates an internal iterator to execute the given function for each element in the array. The *args* argument lets you pass arbitrary data to this function.

**LastThat**

```
T *LastThat( int ( *)(const T &, void *), void * ) const;
```

Returns a pointer to the last object in the list that satisfies a given condition. You supply a test function pointer, *f*, that returns true for a certain condition. You can pass arbitrary arguments via *args*. Returns 0 if no object in the array meets the condition. Note that *LastThat* creates its own internal iterator, so you can treat it as a "search" function.

See also: *FirstThat, ForEach*

**PeekHead**

```
T *PeekHead() const
```

Returns the object pointer at the *Head* of the list, without removing it.

### Protected member functions

**FindPred**

```
virtual TMListElement<VoidPointer,Alloc> *FindPred( VoidPointer );
```

Finds the element that would be followed by the parameter. The function does not check whether the parameter is actually there. This can be used for inserting (insert after returned element pointer).

## TMIListIteratorImp template — listimp.h

Implements a list iterator that works with any managed indirect list. For direct lists, see *TMListIteratorImp* on page 419.

### Public constructors

**Constructor**

```
TMIListIteratorImp( const TMIListImp<VoidPointer,Alloc> &l )
```

Constructs an object that iterates on *TMIListImp* objects.

### Public member functions

**Current**

```
T *Current()
```

Returns the current object pointer.

**Restart**

```
void Restart()
```

Restarts iteration from the beginning of the list.

### Operators

**operator ++**

```
T *operator ++ (int)
```

Moves to the next object, and returns the object that was current before the move (post-increment).

```
T *operator ++ ()
```

Moves to the next object, and returns the object that was current after the move (pre-increment).

## TIListImp template — listimp.h

Implements a list of pointers to objects of type *T*. Since pointers always have meaningful copy semantics, this class can handle any type of object. See *TMIListImp* on page 421 for members.

## TIListIteratorImp template — listimp.h

Implements a list iterator that works with any indirect list. See *TMIListIteratorImp* on page 422 for members.

### Public constructors

**Constructor**

```
TIListIteratorImp( const TIListImp<T> &l )
```

Constructs an object that iterates on *TMIListImp* objects.

## TMISListImp template — listimp.h

Implements a managed sorted list of pointers to objects of type *T*. Since pointers always have meaningful copy semantics, this class can handle any type of object.

### Public member functions

In addition to the member functions described here, *TMISListImp* inherits other member functions from *TMIListImp* (see page 421).

**FindDetach**

```
virtual TMListElement<TVoidPointer,Alloc> *FindDetach(TVoidPointer);
```

Determines whether an object is in the list, and returns a pointer to its predecessor. Returns 0 if not found.

**FindPred**

```
virtual TMListElement<TVoidPointer,Alloc> *FindPred( TVoidPointer );
```

Finds the element that would be followed by the parameter. The function does not check whether the parameter is actually there. This can be used for inserting (insert after returned element pointer).

## TMISListIteratorImp template — listimp.h

Implements a list iterator that works with any managed indirect list. For direct lists, see *TMListIteratorImp* on page 419.

### Public constructors

**Constructor**

```
TMISListIteratorImp( const TMISListImp<T,Alloc> &l ) :
```

Constructs an object that iterates on *TMISListImp* objects.

## TISListImp template — listimp.h

Implements a sorted list of pointers to objects of type *T*, using *TStandardAllocator* for memory management. Since pointers always have meaningful copy semantics, this class can handle any type of object. See *TMISListImp* on page 424 for members.

## TISListIteratorImp template — listimp.h

Implements a list iterator that works with any indirect list. See *TMIListIteratorImp* on page 422 for members.

### Public constructors

**Constructor**

```
TISListIteratorImp( const TISListImp<T> &l )
```

Constructs an object that iterates on *TISListImp* objects.

## TMQueueAsVector template — queues.h

Implements a managed queue of objects of type *T*, using a vector as the underlying implementation. *TMQueueAsVector* assumes *T* has meaningful copy semantics, a < operator, and a default constructor.

### Public constructors

**Constructor**

```
TMQueueAsVector( unsigned sz = DEFAULT_QUEUE_SIZE )
```

Constructs a managed, vector-implemented queue, of *sz* size.

### Public member functions

**FirstThat**

```
T *FirstThat(CondFunc, void *args ) const;
```

Returns a pointer to the first object in the queue that satisfies a given condition. You supply a test-function pointer *f* that returns true for a certain condition. You can pass arbitrary arguments via *args*. Returns 0 if no object in the array meets the condition.

See also: *LastThat*

**Flush**

```
void Flush( TShouldDelete::DeleteType = TShouldDelete::DefDelete )
```

Flushes the queue without destroying it. The fate of any objects removed depends on the current ownership status and the value of the *dt* argument.

See also: *TShouldDelete::ownsElements*

**ForEach**

```
void ForEach(IterFunc, void *args );
```

Executes function *f* for each queue element. *ForEach* creates an internal iterator to execute the given function for each element in the array. The *args* argument lets you pass arbitrary data to this function.

**Get**

```
T Get()
```

Removes the object from the end (tail) of the queue. If the queue is empty, it returns 0. Otherwise the removed object is returned.

**GetItemsInContainer**

```
int GetItemsInContainer() const
```

Returns the number of items in the queue.

**IsEmpty**

```
int IsEmpty() const
```

Returns 1 if the queue has no elements; otherwise returns 0.

**IsFull**

```
int IsFull() const
```

Returns 1 if the queue is full; otherwise returns 0.

**LastThat**

```
T *LastThat(CondFunc, void *args ) const;
```

Returns a pointer to the last object in the queue that satisfies a given condition. You supply a test function pointer, *f*, that returns true for a certain condition. You can pass arbitrary arguments via *args*. Returns 0 if no object in the array meets the condition. Note that *LastThat* creates its own internal iterator, so you can treat it as a "search" function.

See also: *FirstThat*, *ForEach*

**Put**

```
void Put( T t )
```

Adds an object to (the tail of) a queue.

## TMQueueAsVectorIterator template — queues.h

Implements an iterator object for managed, vector-based queues. See *TMDequeAsVectorIterator* on page 385 for members.

### Public constructors

**Constructor**

```
TMQueueAsVectorIterator( const TMDequeAsVector<T,Alloc> &q )
```

Constructs an object that iterates on *TMQueueAsVector* objects.

## TQueueAsVector template — queues.h

See *TMQueueAsVector* on page 425 for members.

### Public constructors

**Constructor**

```
TQueueAsVector( unsigned sz = DEFAULT_QUEUE_SIZE )
```

Constructs a vector-implemented queue, of *sz* size.

## TQueueAsVectorIterator template — queues.h

Implements an iterator object for vector-based queues. See *TMDequeAsVectorIterator* on page 385 for members.

### Public constructors

**Constructor**

```
TQueueAsVectorIterator( const TQueueAsVector<T> &q )
```

Constructs an object that iterates on *TQueueAsVector* objects.

## TMIQueueAsVector template — queues.h

Implements a managed queue of pointers to objects of type *T*, using a vector as the underlying implementation.

### Public constructors

**Constructor**

```
TMIQueueAsVector( unsigned sz = DEFAULT_QUEUE_SIZE )
```

Constructs a managed, indirect queue, of *sz* size.

### Public member functions

**FirstThat**

```
T *FirstThat(CondFunc, void *args ) const;
```

Returns a pointer to the first object in the queue that satisfies a given condition. You supply a test-function pointer *f* that returns true for a certain condition. You can pass arbitrary arguments via *args*. Returns 0 if no object in the array meets the condition.

See also: *LastThat*

**Flush**

```
void Flush( TShouldDelete::DeleteType = TShouldDelete::DefDelete );
```

Flushes the queue without destroying it. The fate of any objects removed depends on the current ownership status and the value of the *dt* argument.

**ForEach**

```
void ForEach(IterFunc, void *args );
```

Executes function *f* for each queue element. *ForEach* creates an internal iterator to execute the given function for each element in the array. The *args* argument lets you pass arbitrary data to this function.

**Get**

```
T *Get()
```

Removes and returns the object pointer from the queue. If the queue is empty, it returns 0.

**GetItemsInContainer**

```
int GetItemsInContainer() const
```

Returns the number of items in the queue.

**IsEmpty**

```
int IsEmpty() const
```

Returns 1 if a queue has no elements; otherwise returns 0.

**IsFull**

```
int isFull() const
```

Returns 1 if a queue is full; otherwise returns 0.

**LastThat**

```
T *LastThat(CondFunc, void *args ) const;
```

Returns a pointer to the last object in the queue that satisfies a given condition. You supply a test function pointer, *f*, that returns true for a certain condition. You can pass arbitrary arguments via *args*. Returns 0 if no object in the array meets the condition. Note that *LastThat* creates its own internal iterator, so you can treat it as a "search" function.

See also: *FirstThat*, *ForEach*

**Put**

```
void Put( T *t )
```

Adds an object pointer to (the tail of) a queue.

## TMIQueueAsVectorIterator template — queues.h

Implements an iterator object for managed, indirect, vector-based queues.

### Public constructors

**Constructor**

```
TMIQueueAsVectorIterator( const TMIDequeAsVector<T,Alloc> &q )
```

Constructs an object that iterates on *TMIQueueAsVector* objects.

## TIQueueAsVector template — queues.h

Implements a queue of pointers to objects of type *T*, using a vector as the underlying implementation.

### Public constructors

**Constructor**

```
TIQueueAsVector( unsigned sz = DEFAULT_QUEUE_SIZE )
```

Constructs a indirect queue, of *sz* size.

## TIQueueAsVectorIterator template — queues.h

Implements an iterator object for indirect, vector-based queues. See *TMDequeAsVectorIterator* on page 385 for members.

### Public constructors

**Constructor**

```
TIQueueAsVectorIterator( const TIQueueAsVector<T> &q )
```

Constructs an object that iterates on *TIQueueAsVector* objects.

## TMQueueAsDoubleList template — queues.h

Implements a managed queue of objects of type *T*, using a double-linked list as the underlying implementation. See *TMDequeAsDoubleList* on page 390 for members.

### Public member functions

**FirstThat**

```
T *FirstThat(CondFunc, void *args ) const
```

Returns a pointer to the first object in the queue that satisfies a given condition. You supply a test-function pointer *f* that returns true for a certain condition. You can pass arbitrary arguments via *args*. Returns 0 if no object in the array meets the condition.

See also: *LastThat*

**Flush**

```
void Flush( int del )
```

Flushes objects from the queue. Flushes the queue without destroying it. The fate of any objects removed depends on the current ownership status and the value of the *dt* argument.

**ForEach**

```
void ForEach(IterFunc, void *args )
```

Executes function *f* for each queue element. *ForEach* creates an internal iterator to execute the given function for each element in the array. The *args* argument lets you pass arbitrary data to this function.

**Get**

```
T Get()
```

Removes the object from the end (tail) of the queue. If the queue is empty, it returns 0. Otherwise the removed object is returned.

**GetItemsInContainer**

```
int GetItemsInContainer() const
```

Returns the number of items in the queue.

**IsEmpty**

```
int IsEmpty() const
```

Returns 1 if a queue has no elements; otherwise returns 0.

**IsFull**

```
int IsFull() const
```

Returns 1 if a queue is full; otherwise returns 0.

**LastThat**

```
T *LastThat(CondFunc, void *args ) const
```

Returns a pointer to the last object in the queue that satisfies a given condition. You supply a test function pointer, *f*, that returns true for a certain condition. You can pass arbitrary arguments via *args*. Returns 0 if no object in the array meets the condition. Note that *LastThat* creates its own internal iterator, so you can treat it as a "search" function.

See also: *FirstThat*, *ForEach*

**Put**

```
void Put( T t )
```

Adds an object to (the tail of) a queue.

## TMQueueAsDoubleListIterator template — queues.h

Implements an iterator object for list-based queues. See *TMDequeAsDoubleListIterator* on page 392 for members.

### Public constructors

**Constructor**

```
TMQueueAsDoubleListIterator( const TMQueueAsDoubleList<T,Alloc> & q )
```

Constructs an object that iterates on *TMQueueAsDoubleList* objects.

## TQueueAsDoubleList template — queues.h

Implements a queue of objects of type *T*, using a double-linked list as the underlying implementation. See *TMQueueAsDoubleList* on page 429 for members.

## TQueueAsDoubleListIterator template — queues.h

Implements an iterator object for list-based queues. See *TMDequeAsDoubleListIterator* on page 392 for members.

### Public constructors

**Constructor**

```
TQueueAsDoubleListIterator( const TQueueAsDoubleList<T> &q )
```

Constructs an object that iterates on *TQueueAsDoubleList* objects.

## TMIQueueAsDoubleList template — queues.h

Implements a managed indirect queue of pointers to objects of type *T*, using a double-linked list as the underlying implementation.

### Public member functions

**FirstThat**

```
T *FirstThat(CondFunc, void *args ) const
```

Returns a pointer to the first object in the queue that satisfies a given condition. You supply a test-function pointer *f* that returns true for a certain

condition. You can pass arbitrary arguments via *args*. Returns 0 if no object in the array meets the condition.

See also: *LastThat*

**Flush**

```
void Flush( TShouldDelete::DeleteType dt = TShouldDelete::DefDelete )
```

Flushes the queue without destroying it. The fate of any objects removed depends on the current ownership status and the value of the *dt* argument.

**ForEach**

```
void ForEach(IterFunc, void *args )
```

Executes function *f* for each queue element. *ForEach* creates an internal iterator to execute the given function for each element in the array. The *args* argument lets you pass arbitrary data to this function.

**Get**

```
T *Get()
```

Removes and returns the object pointer from the queue. If the queue is empty, it returns 0.

**GetItemsInContainer**

```
int GetItemsInContainer() const
```

Returns the number of items in the queue.

**IsEmpty**

```
int IsEmpty() const
```

Returns 1 if the queue has no elements; otherwise returns 0.

**IsFull**

```
int IsFull() const
```

Returns 1 if the queue is full; otherwise returns 0.

**LastThat**

```
T *LastThat(CondFunc, void *args ) const
```

Returns a pointer to the last object in the dequeue that satisfies a given condition. You supply a test function pointer, *f*, that returns true for a certain condition. You can pass arbitrary arguments via *args*. Returns 0 if no object in the array meets the condition. Note that *LastThat* creates its own internal iterator, so you can treat it as a "search" function.

See also: *FirstThat*, *ForEach*

**Put**

```
void Put( T *t )
```

Adds an object pointer to (the tail of) a queue.

## TMIQueueAsDoubleListIterator template — queues.h

Implements an iterator object for indirect, list-based queues. See *TMIDequeAsDoubleListIterator* on page 394 for members.

### Public constructors

**Constructor**

```
TMIQueueAsDoubleListIterator( const TMIQueueAsDoubleList<T,Alloc> & q )
```

Constructs an object that iterates on *TMIQueueAsDoubleList* objects.

## TIQueueAsDoubleList template — queues.h

Implements an indirect queue of pointers to objects of type *T*, using a double-linked list as the underlying implementation. See *TMIQueueAsDoubleList* on page 431 for members.

## TIQueueAsDoubleListIterator template — queues.h

Implements an iterator object for indirect, list-based queues. See *TMIDequeAsDoubleListIterator* on page 394 for members.

### Public constructors

**Constructor**

```
TIQueueAsDoubleListIterator( const TIQueueAsDoubleList<T> & q )
```

Constructs an object that iterates on *TIQueueAsDoubleList* objects.

## TQueue template — queues.h

A simplified name for *TQueueAsVector*.

## TQueueIterator template — queues.h

A simplified name for *TQueueAsVectorIterator*.

## TMSetAsVector template — sets.h

Implements a managed set of objects of type *T*, using a vector as the underlying implementation. A set, unlike a bag, cannot contain duplicate items.

### Public constructors

**Constructor**

```
TMSetAsVector( unsigned sz = DEFAULT_SET_SIZE ) :
```

Constructs an empty set. *sz* represents the number of items the set can hold.

### Public member functions

In addition to the following member function, *TMSetAsVector* inherits member functions from *TMBagAsVector*. See *TMBagAsVector* on page 374 for members.

**Add**

```
int Add( const T& t );
```

Adds an object to the set.

## TMSetAsVectorIterator template — sets.h

Implements an iterator object to traverse *TMSetAsVector* objects. See *TMArrayAsVectorIterator* on page 359 for members.

### Public constructors

**Constructor**

```
TMSetAsVectorIterator( const TMSetAsVector<T,Alloc> &s ) :
```

Constructs an object that iterates on *TMSetAsVector* objects.

## TSetAsVector template — sets.h

Implements a set of objects of type *T*, using a vector as the underlying implementation. *TStandardAllocator* is used to manage memory. See *TMBagAsVector* on page 374 for members.

### Public constructors

**Constructor**

```
TSetAsVector( unsigned sz = DEFAULT_SET_SIZE ) :
```

Constructs an empty set. *sz* represents the number of items the set can hold.

## TSetAsVectorIterator template — sets.h

Implements an iterator object to traverse *TSetAsVector* objects. See *TMArrayAsVectorIterator* on page 359 for members.

### Public constructors

**Constructor**

```
TSetAsVectorIterator( const TSetAsVector<T> &s )
```

Constructs an object that iterates on *TMSetAsVector* objects.

## TMISetAsVector template — sets.h

Implements a managed set of pointers to objects of type *T*, using a vector as the underlying implementation. See *TMIBagAsVector* on page 376 for members.

### Public constructors

**Constructor**

```
TMISetAsVector( unsigned sz = DEFAULT_SET_SIZE ) :
```

Constructs an empty, managed, indirect set. *sz* represents the initial number of slots allocated.

### Public member functions

In addition to the following member function, *TMISetAsVector* inherits member functions from *TMIBagAsVector*. See *TMIBagAsVector* on page 376.

**Add**

```
int Add( T * );
```

Adds an object pointer to the set.

## TMISetAsVectorIterator template — sets.h

Implements an iterator object to traverse *TMISetAsVector* objects. See *TMIArrayAsVectorIterator* on page 364 for members.

### Public constructors

**Constructor** `TMISetAsVectorIterator( const TMISetAsVector<T,Alloc> &s )`

Constructs an object that iterates on *TMISetAsVector* objects.

## TISetAsVector template — sets.h

Implements a set of pointers to objects of type *T*, using a vector as the underlying implementation. See *TMIBagAsVector* on page 376 for members.

### Public constructors

**Constructor** `TISetAsVector( unsigned sz = DEFAULT_SET_SIZE )`

Constructs an empty, indirect bag. *sz* represents the initial number of slots allocated.

## TISetAsVectorIterator template — sets.h

Implements an iterator object to traverse *TISetAsVector* objects. See *TMIArrayAsVectorIterator* on page 364 for members.

### Public constructors

**Constructor** `TISetAsVectorIterator( const TISetAsVector<T> &s )`

Constructs an object that iterates on *TISetAsVector* objects.

## TSet template — sets.h

A simplified name for *TSetAsVector*.

## TSetIterator template — sets.h

A simplified name for *TSetAsVectorIterator*.

## TMStackAsVector template — stacks.h

Implements a managed stack of objects of type *T*, using a vector as the underlying implementation.

### Type definitions

**CondFunc**

```
typedef int ( *CondFunc)(const T &, void *);
```

Function type used as a parameter to *FirstThat* and *LastThat* member functions.

**IterFunc**

```
typedef void ( *IterFunc)(T &, void *);
```

Function type used as a parameter to *ForEach* member function.

### Public constructors

**Constructor**

```
TMStackAsVector( unsigned max = DEFAULT_STACK_SIZE )
```

Constructs a managed, vector-implemented stack, with *max* indicating the maximum stack size.

### Public member functions

**FirstThat**

```
T *FirstThat(CondFunc, void *args ) const
```

Returns a pointer to the first object in the stack that satisfies a given condition. You supply a test-function pointer *f* that returns true for a certain condition. You can pass arbitrary arguments via *args*. Returns 0 if no object in the array meets the condition.

See also: *LastThat*

**Flush**

```
void Flush( TShouldDelete::DeleteType = TShouldDelete::DefDelete )
```

Flushes the stack without destroying it. The fate of any objects removed depends on the current ownership status and the value of the *dt* argument.

See also: *TShouldDelete::ownsElements*

**ForEach**

```
void ForEach(IterFunc, void *args )
```

Executes function *f* for each stack element. *ForEach* creates an internal iterator to execute the given function for each element in the array. The *args* argument lets you pass arbitrary data to this function.

**GetItemsInContainer** `int GetItemsInContainer() const`

Returns the number of items in the stack.

**IsEmpty** `int IsEmpty() const`

Returns 1 if the stack has no elements; otherwise returns 0.

**IsFull** `int IsFull() const`

Returns 1 if the stack is full; otherwise returns 0.

**LastThat** `T *LastThat( int ( * f)(const T &, void *), void *args ) const`

Returns a pointer to the last object in the stack that satisfies a given condition. You supply a test function pointer, *f*, that returns true for a certain condition. You can pass arbitrary arguments via *args*. Returns 0 if no object in the array meets the condition. Note that *LastThat* creates its own internal iterator, so you can treat it as a "search" function.

See also: *FirstThat*, *ForEach*

**Pop** `T Pop()`

Removes the object from the top of the stack and returns the object. The fate of the popped object is determined by ownership. See *TShouldDelete* on page 460.

**Push** `void Push( const T& t )`

Pushes an object on the top of the stack.

**Top** `Const T& Top() const`

Returns but does not remove the object at the top of the stack.

## TMStackAsVectorIterator template — stacks.h

Implements an iterator object for managed, vector-based stacks. See *TMVectorIteratorImp* on page 447 for members.

### Public constructors

**Constructor** `TMStackAsVectorIterator( const TMStackAsVector<T,Alloc> & s ) :`

Constructs an object that iterates on *TMStackAsVector* objects.

## TStackAsVector template — stacks.h

Implements a stack of objects of type *T*, using a vector as the underlying implementation, and *TStandardAllocator* for memory management.

### Public constructors

**Constructor**

```
TStackAsVector( unsigned max = DEFAULT_STACK_SIZE )
```

Constructs a vector-implemented stack, with *max* indicating the maximum stack size.

## TStackAsVectorIterator template — stacks.h

Implements an iterator object for managed, vector-based stacks. See *TMVectorIteratorImp* on page 447 for members.

### Public constructors

**Constructor**

```
TStackAsVectorIterator( const TStackAsVector<T> & s ) :
```

Constructs an object that iterates on *TStackAsVector* objects.

## TMIStackAsVector template — stacks.h

*TMIStackAsVector* implements a managed stack of pointers to objects of type *T*, using a vector as the underlying implementation.

### Type definitions

**CondFunc**

```
typedef int ( *CondFunc)(const T &, void *);
```

Function type used as a parameter to *FirstThat* and *LastThat* member functions.

**IterFunc**

```
typedef void ( *IterFunc)(T &, void *);
```

Function type used as a parameter to *ForEach* member function.

## Public constructors

**Constructor**

```
TMIStackAsVector( unsigned max = DEFAULT_STACK_SIZE )
```

Constructs a managed, indirect, vector-implemented stack, with *max* indicating the maximum stack size.

## Public member functions

**FirstThat**

```
T *FirstThat(CondFunc, void *args ) const
```

Returns a pointer to the first object in the stack that satisfies a given condition. You supply a test-function pointer *f* that returns true for a certain condition. You can pass arbitrary arguments via *args*. Returns 0 if no object in the array meets the condition.

See also: *LastThat*

**Flush**

```
void Flush( TShouldDelete::DeleteType = TShouldDelete::DefDelete )
```

Flushes the stack without destroying it. The fate of any objects removed depends on the current ownership status and the value of the *dt* argument.

See also: *TShouldDelete::ownsElements*

**ForEach**

```
void ForEach(IterFunc, void *args )
```

Executes function *f* for each stack element. *ForEach* creates an internal iterator to execute the given function for each element in the array. The *args* argument lets you pass arbitrary data to this function.

**GetItemsInContainer**

```
int GetItemsInContainer() const
```

Returns the number of items in the stack.

**IsEmpty**

```
int IsEmpty() const
```

Returns 1 if the stack has no elements; otherwise returns 0.

**IsFull**

```
int IsFull() const
```

Returns 1 if the stack is full; otherwise returns 0.

**LastThat**

```
T *LastThat( CondFunc, void *args ) const
```

Returns a pointer to the last object in the stack that satisfies a given condition. You supply a test function pointer, *f*, that returns true for a certain condition. You can pass arbitrary arguments via *args*. Returns 0 if no object in the array meets the condition. Note that *LastThat* creates its own internal iterator, so you can treat it as a "search" function.

See also: *FirstThat*, *ForEach*

**Pop**

```
T *Pop()
```

Removes the object from the top of the stack and returns a pointer to the object. The fate of the popped object is determined by ownership. See *TShouldDelete* on page 460.

**Push**

```
void Push( T *t )
```

Pushes a pointer to an object on the top of the stack.

**Top**

```
T *Top() const
```

Returns but does not remove the object pointer at the top of the stack.

## TMIStackAsVectorIterator template — stacks.h

Implements an iterator object for managed, indirect, vector-based stacks. See *TMVectorIteratorImp* on page 447 for members.

### Public constructors

**Constructor**

```
TMIStackAsVectorIterator( const TMIStackAsVector<T,Alloc> & s )
```

Constructs an object that iterates on *TMIStackAsVector* objects.

## TIStackAsVector template — stacks.h

Implements an indirect stack of pointers to objects of type *T*, using a vector as the underlying implementation. See *TMIStackAsVector* on page 439 for members.

### Public constructors

**Constructor**

```
TIStackAsVector( unsigned max = DEFAULT_STACK_SIZE ) :
TMIStackAsVector<T,TStandardAllocator>( max )
```

Constructs an indirect, vector-implemented stack, with *max* indicating the maximum stack size.

## TIStackAsVectorIterator template — stacks.h

Implements an iterator object for indirect, vector-based stacks. See *TMIVectorIteratorImp* on page 455 for members.

### Public constructors

**Constructor**

```
TMIStackAsVectorIterator( const TMIStackAsVector<T,Alloc> & s )
```

Constructs an object that iterates on *TIStackAsVector* objects.

## TMStackAsList template — stacks.h

Implements a managed stack of objects of type *T*, using a list as the underlying implementation. See *TMStackAsVector* on page 437 for members.

## TMStackAsListIterator template — stacks.h

Implements an iterator object for managed, list-based stacks. See *TMListIteratorImp* on page 419 for members.

### Public constructors

**Constructor**

```
TMStackAsListIterator( const TMStackAsList<T,Alloc> & s ) :
TMListIteratorImp<T,Alloc>(s.Data)
```

Constructs an object that iterates on *TMStackAsList* objects.

## TStackAsList template — stacks.h

Implements a managed stack of objects of type *T*, using a list as the underlying implementation. See *TMStackAsVector* on page 437 for members.

## TStackAsListIterator template — stacks.h

Implements an iterator object for list-based stacks. See *TMVectorIteratorImp* on page 447 for members.

### Public constructors

**Constructor**

```
TStackAsListIterator( const TStackAsList<T> & s ) :
TMStackAsListIterator<T,TStandardAllocator>(s)
```

Constructs an object that iterates on *TIStackAsVector* objects.

## TMIStackAsList template — stacks.h

Implements a managed stack of pointers to objects of type *T*, using a linked list as the underlying implementation. See *TMIStackAsVector* on page 439 for members.

## TMIStackAsListIterator template — stacks.h

Implements an iterator object for managed, indirect, list-based stacks. See *TMIListIteratorImp* on page 422 for members.

### Public constructors

**Constructor**

```
TMIStackAsListIterator( const TMIStackAsList<T,Alloc> & s )
```

Constructs an object that iterates on *TMIStackAsList* objects.

## TIStackAsList template — stacks.h

Implements *TMIStackAsList* with the standard allocator *TStandardAllocator*. See *TMIStackAsVector* on page 439 for members.

## TIStackAsListIterator template — stacks.h

Implements an iterator object for indirect, list-based stacks. See *TMIVectorIteratorImp* on page 455 for members.

### Public constructors

**Constructor**

```
TIStackAsListIterator( const TIStackAsList<T> & s )
```

Constructs an object that iterates on *TIStackAsList* objects.

## TStack template — stacks.h

A simplified name for *TStackAsVector*.

## TStackIterator template — stacks.h

A simplified name for *TStackAsVectorIterator*.

## TMVectorImp template — vectimp.h

Implements a managed vector of objects of type *T*. *TMVectorImp* assumes that *T* has meaningful copy semantics, and a default constructor.

### Type definitions

**CondFunc**

```
typedef int ( *CondFunc)(const T &, void *);
```

Function type used as a parameter to *FirstThat* and *LastThat* member functions.

**IterFunc**

```
typedef void ( *IterFunc)(T &, void *);
```

Function type used as a parameter to *ForEach* member function.

### Public constructors

**Constructor**

```
TMVectorImp();
```

Constructs a vector with no entries.

**Constructor**

```
TMVectorImp( unsigned sz, unsigned = 0 );
```

Constructs a vector of *sz* objects, initialized by default to 0.

**Constructor**

```
TMVectorImp( const TMVectorImp<T,Alloc> & );
```

Constructs a vector copy.

## Public member functions

**FirstThat**

```
T *FirstThat(CondFunc, void *args ) const
```

Returns a pointer to the first object in the vector that satisfies a given condition. You supply a test-function pointer *f* that returns true for a certain condition. You can pass arbitrary arguments via *args*. Returns 0 if no object in the array meets the condition.

```
T *FirstThat(CondFunc, void *, unsigned, unsigned ) const;
```

This version of *FirstThat* allows you to specify a range to be searched. Returns a pointer to the first object in the vector that satisfies a given condition. You supply a test-function pointer *f* that returns true for a certain condition. You can pass arbitrary arguments via *args*. Returns 0 if no object in the array meets the condition.

See also: *LastThat*

**Flush**

```
void Flush( unsigned = 0, unsigned = UINT_MAX, unsigned = 0 );
```

Flushes the vector without destroying it. The fate of any objects removed depends on the current ownership status and the value of the first argument. A range to be flushed can be specified with the last two arguments.

See also: *TShouldDelete::ownsElements*

**ForEach**

```
void ForEach(IterFunc, void *args )
```

Returns a pointer to the first object in the vector that satisfies a given condition. *ForEach* creates an internal iterator to execute the given function for each element in the array. The *args* argument lets you pass arbitrary data to this function.

```
void ForEach(IterFunc, void *, unsigned, unsigned );
```

This version allows you to specify a range.

See also: *LastThat*

**GetDelta**

```
virtual unsigned GetDelta( ) const;
```

Returns the growth delta for the array.

**LastThat**

```
T *LastThat( CondFunc, void *args ) const
```

Returns a pointer to the last object in the vector that satisfies a given condition. You supply a test function pointer, *f*, that returns true for a certain condition. You can pass arbitrary arguments via *args*. Returns 0 if no

object in the array meets the condition. Note that *LastThat* creates its own internal iterator, so you can treat it as a "search" function.

```
T *LastThat( CondFunc, void *, unsigned, unsigned ) const;
```

This version allows you to specify a range.

See also: *FirstThat, ForEach*

**Limit**

```
unsigned Limit() const;
```

Returns the number of items that the vector can hold.

**Resize**

```
void Resize( unsigned sz, unsigned offset = 0 );
```

Creates a new vector of size *sz*. The existing vector is copied to the expanded vector, then deleted. In a vector of pointers the entries are zeroed. In an array of objects the default constructor is invoked for each unused element. *offset* is the location in the new vector where the first element of the old vector should be copied. This is needed when the vector has to be extended downward.

**Top**

```
virtual unsigned Top() const;
```

Returns the index of the current top element. For plain vectors *Top* returns *Lim*; for counted and sorted vectors *Top* returns the current insertion point.

## Operators

**operator [ ]**

```
T & operator [] ( unsigned index ) const
```

Returns a reference to the object at *index*.

**operator =**

```
const TMVectorImp<T,Alloc> & operator = ( const TMVectorImp<T,Alloc> & );
```

Provides the vector assignment operator.

## Protected data members

**Lim**

```
unsigned Lim;
```

*Lim* stores the upper limit for indexes into the vector.

## Protected member functions

**Zero**

```
virtual void Zero( unsigned, unsigned )
```

Provides for zeroing vector contents within the specified range.

## TMVectorIteratorImp template — vectimp.h

Implements a vector iterator that works with any direct, managed vector of objects of type *T*. For indirect vector iterators, see *TMIVectorIteratorImp* on page 455.

### Public constructors

**Constructor**

```
TMVectorIteratorImp( const TMVectorImp<T,Alloc> &v )
```

Creates an iterator object to traverse *TMVectorImp* objects.

**Constructor**

```
TMVectorIteratorImp( const TMVectorImp<T,Alloc> &v, unsigned start,
unsigned stop )
```

Creates an iterator object to traverse *TMVectorImp* objects. A range can be specified.

### Public member functions

**Current**

```
Const T& Current();
```

Returns the current object.

**Restart**

```
void Restart();
```

Restarts iteration over the whole vector.

```
void Restart( unsigned start, unsigned stop );
```

Restarts iteration over the given range.

### Operators

**operator ++**

```
Const T& operator ++(int);
```

Moves to the next object, and returns the object that was current before the move (post-increment).

```
Const T& operator ++();
```

Moves to the next object, and returns the object that was current after the move (pre-increment).

**operator int**

```
operator int();
```

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

## TVectorImp template — vectimp.h

Implements a vector of objects of type *T*. *TVectorImp* assumes that *T* has meaningful copy semantics, and a default constructor. See *TMVectorImp* on page 444 for members.

### Public constructors

**Constructor** `TVectorImp()`

Constructs a vector with no entries.

**Constructor** `TVectorImp( unsigned sz, unsigned = 0 )`

Constructs a vector of *sz* objects, initialized by default to 0.

**Constructor** `TVectorImp( const TVectorImp<T> &v )`

Constructs a vector copy.

## TVectorIteratorImp template — vectimp.h

Implements a vector iterator that works with any direct vector of objects of type *T*. See *TMVectorIteratorImp* on page 447 for members.

### Public constructors

**Constructor** `TVectorIteratorImp( const TVectorImp<T> &v )`

Creates an iterator object to traverse *TVectorImp* objects.

**Constructor** `TVectorIteratorImp( const TVectorImp<T> &v, unsigned start, unsigned stop )`

Creates an iterator object to traverse *TVectorImp* objects. A range can be specified.

## TMCVectorImp template — vectimp.h

Implements a managed, counted vector of objects of type *T*. *TMCVectorImp* assumes that *T* has meaningful copy semantics, and a default constructor.

### Public constructors

**Constructor**

```
TMCVectorImp();
```

Constructs a vector with no entries.

**Constructor**

```
TMCVectorImp( unsigned sz, unsigned = 0 );
```

Constructs a vector of *sz* objects, initialized by default to 0.

### Public member functions

In addition to the member functions described here, *TMCVectorImp* inherits member functions from *TMVectorImp* (see page 444).

**Add**

```
int Add( const T& t );
```

Adds an object to the vector and increments *Count_*.

**AddAt**

```
int AddAt( const T&, unsigned );
```

Adds an object to the vector at the specified location, and increments *Count_*.

**Count**

```
unsigned Count( ) const;
```

Returns *Count_*.

**Detach**

```
int Detach( unsigned, int dt = 0 );
int Detach( const T&, int dt = 0 );
```

Remove by specifying the object or its index. The first version removes the object at *loc;* the second version removes the first object that compares equal to the specified object. The value of *dt* and the current ownership setting determine whether the object itself will be deleted. *DeleteType* is defined in the base class *TShouldDelete* as `enum { NoDelete, DefDelete, Delete }`. The default value of *dt, NoDelete,* means that the object will not be deleted regardless of ownership. With *dt* set to *Delete,* the object will be deleted regardless of ownership. If *dt* is set to *DefDelete,* the object will be deleted only if the array owns its elements.

**Find**

```
virtual unsigned Find( const T& ) const;
```

Finds the specified object and returns the object's index; otherwise returns INT_MAX.

**GetDelta**

```
virtual unsigned GetDelta( ) const;
```

Returns *Delta*.

### Protected data members

In addition to the data members described here, *TMCVectorImp* inherits data members from *TMVectorImp* (see page 444).

**Count_**

```
unsigned Count_;
```

Maintains the number of objects in the vector.

**Delta**

```
unsigned Delta;
```

Specifies the size increment to be used when the vector grows.

### Protected member functions

**Top**

```
virtual unsigned Top( ) const
```

Returns *Count_*.

## TMCVectorIteratorImp template — vectimp.h

Implements a vector iterator that works with any direct, managed, counted vector of objects of type *T*. See *TMVectorIteratorImp* on page 447 for members.

### Public constructors

**Constructor**

```
TMCVectorIteratorImp( const TMCVectorImp<T,Alloc> &v )
```

Creates an iterator object to traverse *TMCVectorImp* objects.

**Constructor**

```
TMVectorIteratorImp( const TMCVectorImp<T,Alloc> &v, unsigned start,
unsigned stop )
```

Creates an iterator object to traverse *TMCVectorImp* objects. A range can be specified.

## TCVectorImp template — vectimp.h

Implements a counted vector of objects of type *T*. *TCVectorImp* assumes that *T* has meaningful copy semantics, and a default constructor. See *TMCVectorImp* on page 449 for members.

### Public constructors

**Constructor**

```
TCVectorImp();
```

Constructs a vector with no entries.

**Constructor**

```
MCVectorImp( unsigned sz, unsigned = 0 );
```

Constructs a vector of *sz* objects, initialized by default to 0.

## TCVectorIteratorImp template — vectimp.h

Implements a vector iterator that works with any direct, counted vector of objects of type *T*. See *TMCVectorIteratorImp* on page 450 for members.

### Public constructors

**Constructor**

```
TCVectorIteratorImp( const TCVectorImp<T> &v )
```

Creates an iterator object to traverse *TCVectorImp* objects.

**Constructor**

```
TCVectorIteratorImp( const TCVectorImp<T> &v, unsigned start, unsigned
stop )
```

Creates an iterator object to traverse *TCVectorImp* objects. A range can be specified.

## TMSVectorImp template — vectimp.h

Implements a managed, sorted vector of objects of type *T*. *TMSVectorImp* assumes that *T* has meaningful copy semantics, a meaningful < operator, and a default constructor. See *TMCVectorImp* on page 449 for members.

### Public constructors

**Constructor**

```
TMSVectorImp()
```

Constructs a vector with no entries.

**Constructor**

```
TMSVectorImp( unsigned sz, unsigned d = 0 )
```

Constructs a vector of *sz* objects, initialized by default to 0.

## TMSVectorIteratorImp template — vectimp.h

Implements a vector iterator that works with any direct, managed, sorted vector of objects of type *T*. See *TMVectorIteratorImp* on page 447 for members.

### Public constructors

**Constructor**

```
TMSVectorIteratorImp( const TMSVectorImp<T,Alloc> &v )
```

Creates an iterator object to traverse *TMSVectorImp* objects.

**Constructor**

```
TMSVectorIteratorImp( const TMSVectorImp<T,Alloc> &v, unsigned start,
unsigned stop )
```

Creates an iterator object to traverse *TMSVectorImp* objects. A range can be specified.

## TSVectorImp template — vectimp.h

Implements a sorted vector of objects of type *T*. *TMSVectorImp* assumes that *T* has meaningful copy semantics, a meaningful < operator, and a default constructor. See *TMCVectorImp* on page 449 for members.

### Public constructors

**Constructor**

```
TSVectorImp()
```

Constructs a vector with no entries.

**Constructor**

```
TSVectorImp( unsigned sz, unsigned d = 0 )
```

Constructs a vector of *sz* objects, initialized by default to 0.

## TSVectorIteratorImp template — vectimp.h

Implements a vector iterator that works with any direct, sorted vector of objects of type *T*. See *TMVectorIteratorImp* on page 447 for members.

### Public constructors

**Constructor**

```
TSVectorIteratorImp( const TSVectorImp<T> &v )
```

Creates an iterator object to traverse *TSVectorImp* objects.

**Constructor**

```
TSVectorIteratorImp( const TSVectorImp<T> &v, unsigned start, unsigned
stop )
```

Creates an iterator object to traverse *TSVectorImp* objects. A range can be specified.

## TMIVectorImp template — vectimp.h

Implements a managed vector of pointers to objects of type *T*. Since pointers always have meaningful copy semantics, this class can handle any type of object.

### Type definitions

**CondFunc**

```
typedef int ( *CondFunc)(const T &, void *);
```

Function type used as a parameter to *FirstThat* and *LastThat* member functions.

**IterFunc**

```
typedef void ( *IterFunc)(T &, void *);
```

Function type used as a parameter to *ForEach* member function.

### Public constructors

**Constructor**

```
TMIVectorImp( unsigned sz );
```

Constructs a managed vector of pointers to objects. *sz* represents the vector size.

## Public member functions

**FirstThat**

```
T *FirstThat(CondFunc, void *args ) const
```

Returns a pointer to the first object in the vector that satisfies a given condition. You supply a test-function pointer *f* that returns true for a certain condition. You can pass arbitrary arguments via *args*. Returns 0 if no object in the array meets the condition.

```
T *FirstThat( int ( *)(const T &, void *), void *, unsigned, unsigned )
const;
```

This version allows specifying a range to be searched. You supply a test-function pointer *f* that returns true for a certain condition. You can pass arbitrary arguments via *args*. Returns 0 if no object in the array meets the condition.

**Flush**

```
void Flush( unsigned = 0, unsigned = UINT_MAX, unsigned = 0 );
```

Flushes the vector without destroying it. The fate of any objects removed depends on the current ownership status and the value of the first argument. A range to be flushed can be specified with the last two arguments.

**ForEach**

```
void ForEach(IterFunc, void *args )
```

Returns a pointer to the first object in the vector that satisfies a given condition. See *TMArrayAsVector::FirstThat*.

```
void ForEach(IterFunc, void *, unsigned, unsigned );
```

This version allows specifying a range.

**GetDelta**

```
virtual unsigned GetDelta( ) const;
```

Returns the growth delta for the array.

**LastThat**

```
T *LastThat(CondFunc, void *args ) const
```

Returns a pointer to the last object in the vector that satisfies a given condition. See *TMArrayAsVector::LastThat*.

```
T *LastThat( CondFunc, void *, unsigned, unsigned ) const;
```

This version allows specifying a range.

**Limit**

```
unsigned Limit() const;
```

Returns the number of items that the vector can hold.

**Resize**

```
void Resize( unsigned sz, unsigned offset = 0 );
```

Creates a new vector of size *sz*. The existing vector is copied to the expanded vector, then deleted. In a vector of pointers the entries are zeroed. In an array of objects the default constructor is invoked for each unused element. *offset* is the location in the new vector where the first element of the old vector should be copied. This is needed when the vector has to be extended downward.

**Top**

```
virtual unsigned Top() const;
```

Returns the index of the current top element. For plain vectors *Top* returns *Lim*; for counted and sorted vectors *Top* returns the current insertion point.

**Zero**

```
virtual void Zero( unsigned, unsigned );
```

Provides for zeroing vector contents within the specified range.

### Operators

**operator [ ]**

```
T * & operator [] ( unsigned index )
T * & operator [] ( unsigned index ) const
```

Returns a reference to the object at *index*.

## TMIVectorIteratorImp template — vectimp.h

Implements a vector iterator that works with an indirect, managed vector.

### Public constructors

**Constructor**

```
TMIVectorIteratorImp( const TMIVectorImp<T,Alloc> &v )
```

Creates an iterator object to traverse *TMIVectorImp* objects.

**Constructor**

```
TMIVectorIteratorImp( const TMIVectorImp<T,Alloc> &v, unsigned l, unsigned u )
```

Creates an iterator object to traverse *TMIVectorImp* objects. A range can be specified.

### Public member functions

**Current**

```
T *Current();
```

Returns a pointer to the current object.

**Restart**

```
void Restart();
```

Restarts iteration over the whole vector.

```
void Restart( unsigned start, unsigned stop );
```

Restarts iteration over the given range.

### Operators

**operator ++**

```
Const T& operator ++(int);
```

Moves to the next object, and returns the object that was current before the move (post-increment).

```
Const T& operator ++();
```

Moves to the next object, and returns the object that was current after the move (pre-increment).

**operator int**

```
operator int();
```

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

## TIVectorImp template — vectimp.h

Implements a vector of pointers to objects of type *T*. Since pointers always have meaningful copy semantics, this class can handle any type of object. See *TMIVectorImp* on page 453 for members.

### Public constructors

**Constructor**

```
TIVectorImp( unsigned sz, unsigned d = 0 )
```

Constructs an indirect vector of *sz* size, with default initialization of 0.

## TIVectorIteratorImp template — vectimp.h

Implements a vector iterator that works with an indirect, managed vector. See *TMIVectorIteratorImp* on page 455 for members.

### Public constructors

**Constructor**

```
TIVectorIteratorImp( const TIVectorImp<T> &v )
```

Creates an iterator object to traverse *TIVectorImp* objects.

**Constructor**

```
TIVectorIteratorImp( const TIVectorImp<T> &v, unsigned l, unsigned u )
```

Creates an iterator object to traverse *TIVectorImp* objects. A range can be specified.

## TMICVectorImp template — vectimp.h

Implements a managed, counted vector of pointers to objects of type *T*. Since pointers always have meaningful copy semantics, this class can handle any type of object.

### Public constructors

**Constructor**

```
TMICVectorImp(unsigned sz, unsigned d = 0 )
```

Constructs a managed, counted vector of pointers to objects. *sz* represents the vector size. *d* represents the initialization value.

### Public member functions

In addition to the following member functions, *TMICVectorImp* inherits other member functions and operators from *TMIVectorImp* (see page 453).

**Add**

```
int Add( T *t );
```

Adds an object to the vector.

**Find**

```
unsigned Find( T *t ) const
```

Finds the specified object pointer, and returns its index.

### Protected member functions

**Find**

```
virtual unsigned Find( void * ) const;
```

Finds the specified pointer and returns its index.

## TMICVectorIteratorImp template — vectimp.h

Implements a vector iterator that works with an indirect, managed, counted vector. See *TMIVectorIteratorImp* on page 455 and *TMVectorIteratorImp* on page 447 for members.

### Public constructors

**Constructor**

```
TMICVectorIteratorImp( const TMICVectorImp<T,Alloc> &v )
```

Creates an iterator object to traverse *TMCIVectorImp* objects.

**Constructor**

```
TMICVectorIteratorImp( const TMICVectorImp<T,Alloc> &v, unsigned l,
unsigned u )
```

Creates an iterator object to traverse *TMICVectorImp* objects. A range can be specified.

## TICVectorImp template — vectimp.h

Implements a counted vector of pointers to objects of type *T*. Since pointers always have meaningful copy semantics, this class can handle any type of object. See *TMICVectorImp* on page 457 for members.

### Public constructors

**Constructor**

```
TICVectorImp( unsigned sz, unsigned d = 0 )
```

Constructs a counted vector of pointers to objects. *sz* represents the vector size. *d* represents the initialization value.

## TICVectorIteratorImp template — vectimp.h

Implements a vector iterator that works with an indirect, managed, counted vector. See *TMIVectorIteratorImp* on page 455 and *TMVectorIteratorImp* on page 447 for members.

### Public constructors

**Constructor**

```
TICVectorIteratorImp( const TICVectorImp<T> &v )
```

Creates an iterator object to traverse *TICVectorImp* objects.

**Constructor**

```
TICVectorIteratorImp( const TICVectorImp<T> &v, unsigned l, unsigned u )
```

Creates an iterator object to traverse *TICVectorImp* objects. A range can be specified.

## TMISVectorImp template — vectimp.h

Implements a managed, sorted vector of pointers to objects of type *T*. Since pointers always have meaningful copy semantics, this class can handle any type of object. See *TMICVectorImp* on page 457 for members.

### Public constructors

**Constructor**

```
TMISVectorImp( unsigned sz, unsigned d = 0 );
```

Constructs a managed, sorted vector of pointers to objects. *sz* represents the vector size. *d* represents the initialization value.

## TMISVectorIteratorImp template — vectimp.h

Implements a vector iterator that works with an indirect, managed, sorted vector. See *TMIVectorIteratorImp* on page 455 and *TMVectorIteratorImp* on page 447 for members.

### Public constructors

**Constructor**

```
TMISVectorIteratorImp( const TMISVectorImp<T,Alloc> &v )
```

Creates an iterator object to traverse *TMIVectorImp* objects.

**Constructor**

```
TMISVectorIteratorImp( const TMISVectorImp<T,Alloc> &v, unsigned l,
unsigned u )
```

Creates an iterator object to traverse *TMIVectorImp* objects. A range can be specified.

## TISVectorImp template — vectimp.h

Implements a sorted vector of pointers to objects of type *T*. Since pointers always have meaningful copy semantics, this class can handle any type of object. See *TMICVectorImp* on page 457 for members.

### Public constructors

**Constructor**

```
TISVectorImp( unsigned sz, unsigned d = 0 )
```

Constructs a managed, sorted vector of pointers to objects. *sz* represents the vector size. *d* represents the initialization value.

## TISVectorIteratorImp template — vectimp.h

Implements a vector iterator that works with an indirect, managed, sorted vector. See *TMIVectorIteratorImp* on page 455 and *TMVectorIteratorImp* on page 447 for members.

### Public constructors

**Constructor**

```
TISVectorIteratorImp( const TISVectorImp<T> &v )
```

Creates an iterator object to traverse *TISVectorImp* objects.

**Constructor**

```
TISVectorIteratorImp( const TISVectorImp<T> &v, unsigned l, unsigned u )
```

Creates an iterator object to traverse *TISVectorImp* objects. A range can be specified.

## TShouldDelete class — shddel.h

*TShouldDelete* maintains the ownership state of an indirect container. The fate of objects that are removed from a container can be made to depend on whether the container owns its elements or not. Similarly, when a container is destroyed, ownership can dictate the fate of contained objects that are still in scope. As a virtual base class, *TShouldDelete* provides ownership control for all containers classes. The member function *OwnsElements* can be used either to report or to change the ownership status of a container. The member function *DelObj* is used to determine if objects in containers should be deleted or not.

## Public data members

```
enum DeleteType { NoDelete, DefDelete, Delete };
```

Enumerates values to determine whether or not an object should be deleted upon removal from a container.

## Public constructors

**Constructor**

```
TShouldDelete( DeleteType dt = Delete )
```

Creates a *TShouldDelete* object. See member function *DelObj*.

## Public member functions

**OwnsElements**

```
int OwnsElements()
```

Returns 1 if the container owns its elements; otherwise returns 0.

```
void OwnsElements( int del )
```

Changes the ownership status as follows: if *del* is 0, ownership is turned off; otherwise ownership is turned on.

## Protected member functions

**DelObj**

```
int DelObj( DeleteType dt )
```

Tests the state of ownership and returns 1 if the contained objects should be deleted or 0 if the contained elements should not be deleted. The factors determining this are the current ownership state, and the value of *dt*, as shown in the following table.

| | delObj | |
|---|---|---|
| **ownsElements** | **No** | **Yes** |
| NoDelete | No | No |
| DefDelete | No | Yes |
| Delete | Yes | Yes |

*delObj* returns 1 if (*dt* is *Delete*) or (*dt* is *DefDelete* and the container currently owns its elements). Thus a *dt* of *NoDelete* returns 0 (don't delete) regardless of ownership; a *dt* of *Delete* return 1 (do delete) regardless of ownership; and a *dt* of *DefDelete* returns 1 (do delete) if the elements are owned, but a 0 (don't delete) if the objects are not owned.

CHAPTER 8

# The C++ mathematical classes

This chapter describes Borland C++ mathematics based on C++ classes. These mathematical operations are available only in C++ programs. However, a C++ program that uses any of these classes, the numerical types that the classes define, or any of the classes' **friend** and member functions can use any of ANSI C Standard mathematics routines.

There are two classes, *bcd* and *complex*, that construct numerical types. Along with these numerical types, each class defines the functions with which to carry out operations with their respective types (for example, converting to and from the *bcd* and *complex* type). Each class also overloads all necessary operators.

The mathematical classes are independent of any hierarchy. However, each class includes the iostream.h header file.

The portability for *bcd* and *complex* is as follows:

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | | ■ |

## bcd — bcd.h

The class constructors create binary coded decimals (BCD) from integers or floating-point numerical types. The **friend** function *real*, described on page 465, converts *bcd* numbers to **long double**.

Once you construct *bcd* numbers, you can freely mix them in expressions with **int**s, **double**s, and other numeric types. You can also use *bcd* numbers in any of the ANSI C Standard mathematical functions.

The following ANSI C math functions are overloaded to operate with *bcd* types:

```
friend bcd  abs(bcd &);
friend bcd  acos(bcd &);
friend bcd  asin(bcd &);
```

```
friend bcd  atan(bcd &);
friend bcd  cos(bcd &);
friend bcd  cosh(bcd &);
friend bcd  exp(bcd &);
friend bcd  log(bcd &);
friend bcd  log10(bcd &);
friend bcd  pow(bcd & base, bcd & expon);
friend bcd  sin(bcd &);
friend bcd  sinh(bcd &);
friend bcd  sqrt(bcd &);
friend bcd  tan(bcd &);
friend bcd  tanh(bcd &);
```

See the documentation of these functions in Chapter 3.

The *bcd* class also overloads the operators **+**, **–**, *****, **/**, **+=**, **–=**, ***=**, **/=**, **=**, **==**, and **!=**. These operators provide *bcd* arithmetic manipulation in the usual sense.

The operators **<<** and **>>** are overloaded for stream input and output of *bcd* numbers, as they are for other data types in iostream.h.

*bcd* numbers have about 17 decimal digits precision, and a range of about $1 \times 10^{-125}$ to $1 \times 10^{125}$.

The number is rounded according to the rules of banker's rounding, which means round to nearest whole number, with ties being rounded to an even digit.

## Public constructors

**Constructor**

```
bcd();
```

The default constructor. You typically use this to declare a variable of type *bcd*.

```
bcd i;       // Construct a bcd-type number.
bcd j = 37;  // Construct and initialize a bcd-type number.
```

**Constructor**

```
bcd(int x);
```

This constructor defines a *bcd* variable from an **int** variable or directly from an integer.

```
int i = 15;
bcd j = bcd(i);   // Initialize j with a previously declared type.
bcd k = bcd(12);  // Construct k from the integer provided.
```

The above example provides these variables:

```
i = 15   j = 15     k = 12
```

**Constructor**

```
bcd(unsigned int x);
```

This constructor defines a *bcd* variable from a variable that was previously declared to be an **unsigned int** type. An unsigned integer can be provided directly to the constructor.

**Constructor**

```
bcd(long x);
```

This constructor defines a *bcd* variable from an **long** variable or directly from a **long** value.

**Constructor**

```
bcd(unsigned long x);
```

This constructor defines a *bcd* variable from a variable that was previously declared to be an **unsigned long** type.

**Constructor**

```
bcd(double x, int decimals = Max);
```

This constructor defines a *bcd* variable from a variable that was previously declared to be a floating point **double** type. The constructor also creates a variable directly from a **double** value.

To specify a precision level (that is, the number of digits after the decimal point) that is different from the default, use the variable *decimals*; for example,

```
double x = 1.2345;  // Declare and initialize in the usual manner.
bcd y = bcd(x, 2);  // Create a bcd numerical type from x.
```

The precision level for *y* is set to 2. Therefore, *y* is initialized with 1.23.

**Constructor**

```
bcd(long double x, int decimals = Max);
```

This constructor defines a *bcd* variable from a variable that was previously declared to be a floating point **long double** type. Alternately, you can supply a **long double** value directly in the place of *x*.

To specify a precision level (that is, the number of digits after the decimal point) that is different from the default, use the variable *decimals*.

## Friend functions

**real**

```
long double real(bcd number)
```

You can use the *real* function to convert a binary coded decimal number back to a **long double**. See the *Programmer's Guide*, Chapter 2, for a discussion about arithmetic conversions.

# complex — complex.h

Creates *complex* numbers. Once you construct *complex* numbers, you can freely mix them in expressions with **int**s, **double**s, and other numeric types. You can also use *complex* numbers in any of the ANSI C Standard mathematical functions. The ANSI math functions are documented in Chapter 3.

The *complex* class also overloads the operators **+**, **–**, *****, **/**, **+=**, **–=**, ***=**, **/=**, **=**, **==**, and **!=**. These operators provide complex arithmetic manipulation in the usual sense.

The operators **<<** and **>>** are overloaded for stream input and output of *complex* numbers, as they are for other data types in iostream.h.

If you don't want to program in C++, but instead want to program in C, the only constructs available to you are **struct** *complex* and *cabs*, which give the absolute value of a complex number. Both of these alternates are defined in math.h.

## Public constructors

**Constructor**

```
complex();
```

The default constructor. You typically use this to declare a variable of type *complex*.

```
complex i;      // Construct a complex-type number.
complex j = 37; // Construct and initialize a complex-type number.
```

**Constructor**

```
complex(double real, double imag = 0);
```

Creates a *complex* numerical type out of a **double**. Upon construction, a real and an imaginary part are provided. The imaginary part is considered to be zero if *imag* is omitted.

## Friend functions

**abs**

```
friend double abs(complex& val);
```

Returns the absolute value of a complex number.

The complex version of *abs* returns a **double**. All other math functions return a *complex* type when *val* is *complex* type.

**acos**

```
friend complex acos(complex& z);
```

Calculates the arc cosine.

The complex inverse cosine is defined by

```
acos(z) = -i * log(z + i sqrt(1 - z²))
```

**arg**

```
double arg(complex x);
```

*arg* gives the angle, in radians, of the number in the complex plane.

The positive real axis has angle 0, and the positive imaginary axis has angle *pi*/2. If the argument passed to *arg* is *complex* 0 (zero), *arg* returns zero.

*arg*(*x*) returns *atan2*(*imag*(*x*), *real*(*x*)).

**asin**

```
friend complex asin(complex& z);
```

Calculates the arc sine.

The complex inverse sine is defined by

```
asin(z) = -i * log(i * z + sqrt(1 - z²))
```

**atan**

```
friend complex atan(complex& z);
```

Calculates the arc tangent.

The complex inverse tangent is defined by

```
atan(z) = -0.5 i log((1 + i z)/(1 - i z))
```

**conj**

```
complex conj(complex z);
```

Returns the complex conjugate of a complex number. `conj(z)` is the same as `complex(real(z), -imag(z))`.

**cos**

```
friend complex cos(complex& z);
```

Calculates the cosine of a value.

The complex cosine is defined by

```
cos(z) = ( exp(i * z) + exp(-i * z) ) / 2
```

**cosh**

```
friend complex cosh(complex& z);
```

Calculates the hyperbolic cosine of a value.

The complex hyperbolic cosine is defined by

```
cosh(z) = ( exp(z) + exp(-z) ) / 2
```

**exp**

```
friend complex exp(complex& y);
```

Calculates the exponential *e* to the *y*.

The complex exponential function is defined by

```
exp(x + y * i) = exp(x) (cos(y) + i * sin(y) )
```

**imag**

```
double imag(complex x);
```

Returns the imaginary part of a *complex* number.

The data associated to a complex number consists of two floating-point (**double**) numbers. *imag* returns the one considered to be the imaginary part.

**log**

```
friend complex log(complex& z);
```

Calculates the natural logarithm of z.

The complex natural logarithm is defined by

```
log(z) = log( abs(z) ) + i * arg(z)
```

**log10**

```
friend complex log10(complex& z);
```

Calculates $\log_{10}(z)$.

The complex common logarithm is defined by

```
log10(z) = log(z) / log(10)
```

**norm**

```
double norm(complex x);
```

Returns the square of the absolute value. *norm*(*x*) returns the magnitude *real*(*x*) * *real*(*x*) + *imag*(*x*) * *imag*(*x*).

*norm* can overflow if either the real or imaginary part is sufficiently large.

**polar**

```
complex polar(double mag, double angle = 0);
```

Returns a *complex* number with a given magnitude (absolute value) and angle.

*polar*(*mag, angle*) is the same as *complex*(*mag* * *cos*(*angle*), *mag* * *sin*(*angle*)).

**pow**

```
friend complex pow(complex& base, double expon);
friend complex pow(double base, complex& expon);
friend complex pow(complex& base, complex& expon);
```

Calculates *base* to the power of *expon*.

The complex *pow* is defined by

```
pow(base, expon) = exp(expon * log(base))
```

**real**

```
double real(complex x);
```

You can use the *real* function to convert a *complex* number back to a **long double**. The **friend** function returns the real part of a complex number or

converts a *complex* number back to **double**. The data associated to a complex number consists of two floating-point numbers. *real* returns the number considered to be the real part.

See the *Programmer's Guide*, Chapter 2, for a discussion about arithmetic conversions.

**sin**

```
friend complex sin(complex& z);
```

Calculates the trigonometric sine.

The complex sine is defined by

```
sin(z) = ( exp(i * z) - exp(-i * z) ) / (2 * i)
```

**sinh**

```
friend complex sinh(complex& z);
```

Calculates the hyperbolic sine.

The complex hyperbolic sine is defined by

```
sinh(z) = ( exp(z) - exp(-z) ) / 2
```

**sqrt**

```
friend complex sqrt(complex& x);
```

Calculates the positive square root.

For any *complex* number *x*, *sqrt*(*x*) gives the *complex* root whose *arg* is *arg*(*x*)/2.

The complex square root is defined by

```
sqrt(x) = sqrt(abs(x)) (cos( arg(x) / 2) + i * sin(arg(x)/2))
```

**tan**

```
friend complex tan(complex& z);
```

Calculates the trigonometric tangent.

The complex tangent is defined by

```
tan(z) = sin(z) / cos(z)
```

**tanh**

```
friend complex tanh(complex& z);
```

Calculates the hyperbolic tangent.

The complex hyperbolic tangent is defined by

```
tanh(z) = sinh(z) / cosh(z)
```

CHAPTER 9

# Class diagnostic macros

Borland provides a set of macros for debugging C++ code. These macros can be used with Windows and DOS and are located in checks.h. There are two types of macros, default and extended. The default macros are

- CHECK
- PRECONDITION
- TRACE
- WARN

The extended macros are

- CHECKX
- PRECONDITIONX
- TRACEX
- WARNX

The default macros provide straightforward value checking and message output. The extended macros let you create macro groups that you can selectively enable or disable. Extended macros also let you selectively enable or disable macros within a group based on a numeric threshold level.

Three preprocessor symbols control diagnostic macro expansion: __DEBUG, __TRACE, and __WARN. If one of these symbols is defined when compiling, then the corresponding macros expand and diagnostic code is generated. If none of these symbols is defined, then the macros do not expand and no diagnostic code is generated. These symbols can be defined on the command line using the **-D** switch, or by using `#define` statements within your code.

The diagnostic macros are enabled according to the following table:

| | __DEBUG=1 | __DEBUG=2 | __TRACE | __WARN |
|---|---|---|---|---|
| PRECONDITION | X | X | | |
| PRECONDITIONX | X | X | | |
| CHECK | | X | | |
| CHECKX | | X | | |
| TRACE | | | X | |
| TRACEX | | | X | |
| WARN | | | | X |
| WARNX | | | | X |

To create a diagnostic version of an executable, place the diagnostic macros at strategic points within the program code and compile with the appropriate preprocessor symbols defined. Diagnostic versions of the Borland class libraries are built in a similar manner.

The following sections describe the default and extended diagnostic macros, give examples of their use, and explain message output and run-time control.

## Default diagnostic macros — checks.h

### CHECK

`CHECK(<cond>)`

Outputs *<cond>* and throws an exception if *<cond>* equals 0. Use CHECK to perform value checking within a function.

### PRECONDITION

`PRECONDITION(<cond>)`

Outputs *<cond>* and throws an exception if *<cond>* equals 0. Use PRECONDITION on entry to a function to check the validity of the arguments and to do any other checking to determine if the function has been invoked correctly.

### TRACE

`TRACE(<msg>)`

Outputs *<msg>*. TRACE is used to output general messages that are not dependent on a particular condition.

### WARN

`WARN(<cond>,<msg>)`

Outputs *<msg>* if *<cond>* is nonzero. It is used to output conditional messages.

**Example** The following program illustrates the use of the default TRACE and WARN macros:

```
#include <checks.h>

int main()
{
   TRACE( "Hello World" );
   WARN( 5 != 5, "Math is broken!" );
   WARN( 5 != 7, "Math still works!" );

   return 0;
}
```

When the above code is compiled with __TRACE and __WARN defined, it produces the following output when run:

```
Trace PROG.C 5: [Def] Hello World
Warning PROG.C 7: [Def] Math still works!
```

The above output indicates that the message "Hello World" was output by the default TRACE macro on line 5 of PROG.C, and the message "Math still works!" was output by the default WARN macro on line 7 of PROG.C.

Default diagnostic macros expand to extended diagnostic macros with the group set to "Def" and the level set to 0. This "Def" group controls the behavior of the default macros and is initially enabled with a threshold level of 0.

## Extended diagnostic macros — checks.h

The extended macros CHECKX and PRECONDITIONX augment CHECK and PRECONDITION by letting you provide a message to be output when the condition fails.

The extended macros TRACEX and WARNX augment TRACE and WARN by providing a way to specify macro groups that can be independently enabled or disabled. TRACEX and WARNX require additional arguments that specify the group to which the macros belongs, and the threshold level at which the macro should be executed. The macro is executed only if the specified group is enabled and has a threshold level that is greater than or equal to the threshold-level argument used in the macro.

The following sections describe the extended diagnostic macros.

**CHECKX**

`CHECKX(<cond>,<msg>)`

Outputs <*msg*> and throws an exception if <*cond*> equals 0. Use CHECKX to perform value checking within a function.

**PRECONDITIONX**

`PRECONDITIONX(<cond>,<msg>)`

Outputs <*msg*> and throws an exception if <*cond*> equals 0. Use PRECONDITIONX on entry to a function to check the validity of the arguments and to do any other checking to determine if the function has been invoked correctly.

**TRACEX**

`TRACEX(<group>,<level>,<msg>)`

Trace only if <*group*> and <*level*> are enabled.

**WARNX**

`WARNX(<group>,<cond>,<level>,<msg>)`

Warn only if <*group*> and <*level*> are enabled.

When using TRACEX and WARNX you need to be able to create groups. The following three macros create diagnostic macro groups:

**DIAG_DECLARE_GROUP** `DIAG_DECLARE_GROUP(<name>)`

Declare a group named *<name>*.

**DIAG_DEFINE_GROUP** `DIAG_DEFINE_GROUP(<name>,<enabled>,<level>)`

Define a group named *<name>*.

**DIAG_CREATE_GROUP** `DIAG_CREATE_GROUP(<name>,<enabled>,<level>)`

Define and declare a group named *<name>*.

The following two macros manipulate groups:

**DIAG_ENABLE** `DIAG_ENABLE(<group>,<state>)`

Sets <group>'s enable flag to <state>.

**DIAG_ISENABLED** `DIAG_ISENABLED(<group>)`

Returns nonzero if *<group>* is enabled.

The following two macros manipulate levels:

**DIAG_SETLEVEL** `DIAG_SETLEVEL(<group>,<level>)`

Sets *<group>*'s threshold level to *<level>*.

**DIAG_GETLEVEL** `DIAG_GETLEVEL(<group>)`

Gets *<group>*'s threshold level.

Threshold levels are arbitrary numeric values that establish a threshold for enabling macros. A macro with a level greater than the group threshold level will not be executed. For example, if a group has a threshold level of 0 (the default value), all macros that belong to that group and have levels of 1 or greater are ignored.

**Example** The following PROG.C example defines two diagnostic groups, *Group1* and *Group2*, which are used as arguments to extended diagnostic macros:

```
#include <checks.h>

DIAG_CREATE_GROUP(Group1,1,0);
DIAG_CREATE_GROUP(Group2,1,0);

int main( int argc, char **argv )
{
   TRACE( "Always works, argc=" << argc );

   TRACEX( Group1, 0, "Hello" );
```

```
    TRACEX( Group2, 0, "Hello" );

    DIAG_DISABLE(Group1);

    TRACEX( Group1, 0, "Won't execute - group is disabled!" );
    TRACEX( Group2, 3, "Won't execute - level is too high!" );

    return 0;
}
```

When the above code is compiled with __TRACE defined and run, it produces the following output:

```
Trace PROG.C 8: [Def] Always works, argc=1
Trace PROG.C 10: [Group1] Hello
Trace PROG.C 11: [Group2] Hello
```

Note that the last two macros are not executed. In the first case, the group *Group1* is disabled. In the second case, the macro level exceeds *Group2*'s threshold level (set by default to 0).

## Macro message output

The CHECKX, PRECONDITIONX, TRACE, TRACEX, WARN, and WARNX macros take a *<msg>* argument that is conditionally inserted into an output stream. This means a sequence of objects can be inserted in the output stream (for example `TRACE( "Mouse @ " << x << "," << y );` ). The use of streams is extensible to different object types and allows for parameters within trace messages.

Diagnostic macro message output can be viewed while the program is running. If the target environment is Windows, the output is sent to the *OutputDebugString* function, and can be viewed with the DBWIN.EXE or OX.SYS utilities. If Turbo Debugger is running, the output will be sent to its log window. If the target environment is DOS, the output is sent to the standard error stream and can be easily redirected at the command line.

## Run-time macro control

Diagnostic groups can be controlled at run time by using the control macros described above within your program or by directly modifying the group information within the debugger.

This group information is contained in a template class named *TDiagGroup< TDiagGroupClass##Group >*, where *##Group* is the name of the

group. This class contains a static structure *Flags*, which in turn contains the enabled flag and the threshold level. For example, to enable the group *Group1*, you would set the variable *TDiagGroup<TDiagGroupClassGroup1>::Flags.Enabled* to 1.

CHAPTER 10

# Run-time support

This chapter provides a detailed description, in alphabetical order, of functions and classes that provide run-time support. Any class operators or member functions are listed immediately after the class constructor. See the *Programmer's Guide*, Chapter 4, for a discussion of how to use exception-handling keywords.

The portability for all classes and functions in this chapter is as follows:

| DOS | UNIX | Win 16 | Win 32 | ANSI C | ANSI C++ | OS/2 |
|---|---|---|---|---|---|---|
| ■ | | ■ | ■ | | ■ | |

## Bad_cast class — typeinfo.h

When **dynamic_cast** fails to make a cast to reference, the expression can throw *Bad_cast*. Note that when **dynamic_cast** fails to make a cast to pointer type, the result is the null pointer.

## Bad_typeid class — typeinfo.h

When the operand of **typeid** is a dereferenced 0 pointer, the **typeid** operator can throw *Bad_typeid*.

## set_new_handler function — new.h

```
typedef void (new * new_handler)() throw(xalloc);
new_handler set_new_handler(new_handler my_handler);
```

*set_new_handler* installs the function to be called when the global **operator new**() or **operator new**[]() cannot allocate the requested memory. By default the **new** operators throw an *xalloc* exception if memory cannot be allocated. You can change this default behavior by calling *set_new_handler* to set a

new handler. To retain the traditional version of **new**, which does not throw exceptions, you can use *set_new_handler*(0).

If **new** cannot allocate the requested memory, it calls the handler that was set by a previous call to *set_new_handler*. If there is no handler installed by *set_new_handler*, **new** returns 0. *my_handler* should specify the actions to be taken when **new** cannot satisfy a request for memory allocation. The *new_handler* type, defined in new.h, is a function that takes no arguments and returns **void**. A *new_handler* can throw an *xalloc* exception.

The user-defined *my_handler* should do one of the following:

- Return after freeing memory
- Throw an *xalloc* exception or an exception derived from *xalloc*
- Call *abort* or *exit* functions

If *my_handler* returns, then **new** will again attempt to satisfy the request.

Ideally, *my_handler* frees up memory and returns; **new** can then satisfy the request and the program can continue. However, if *my_handler* cannot provide memory for **new**, *my_handler* must throw an exception or terminate the program. Otherwise, an infinite loop will be created.

Preferably, you should overload **operator new()** and **operator new[]()** to take appropriate actions for your applications.

*set_new_handler* returns the old handler, if one has been registered.

The user-defined argument function, *my_handler*, should not return a value.

See also the description of *abort*, *exit*, and *_new_handler* (global variable).

## set_terminate function — except.h

```
typedef void (*terminate_function)();
terminate_function set_terminate(terminate_function t_func);
```

*set_terminate* lets you install a function that defines the program's termination behavior when a handler for the exception cannot be found. The actions are defined in *t_func*, which is declared to be a function of type *terminate_function*. A *terminate_function* type, defined in except.h, is a function that takes no arguments, and returns **void**.

By default, an exception for which no handler can be found results in the program calling the *terminate* function. This will normally result in a call to *abort*. The program then ends with the message `Abnormal program termination`. If you want some function other than *abort* to be called by the

*terminate* function, you should define your own *t_func* function. Your *t_func* function is installed by *set_terminate* as the termination function. The installation of *t_func* lets you implement any actions that are not taken by *abort*.

The previous function given to *set_terminate* will be the return value.

The definition of *t_func* must terminate the program. Such a user-defined function must not return to its caller, the *terminate* function. An attempt to return to the caller results in undefined program behavior. It is also an error for *t_func* to throw an exception.

See also the description of *abort*, *set_unexpected*, and *terminate*.

## set_unexpected function — except.h

```
typedef void ( * unexpected_function )();
unexpected_function set_unexpected(unexpected_function unexpected_func);
```

*set_unexpected* lets you install a function that defines the program's behavior when a function throws an exception not listed in its exception specification. The actions are defined in *unexpected_func*, which is declared to be a function of type *unexpected_function*. An *unexpected_function* type, defined in except.h, is a function that takes no arguments, and returns **void**.

By default, an unexpected exception causes *unexpected* to be called. If *unexpected_func* is defined, it is subsequently called by *unexpected*. Program control is then turned over to the user-defined *unexpected_func*. Otherwise, *terminate* is called.

The previous function given to *set_unexpected* will be the return value.

The definition of *unexpected_func* must not return to its caller, the *unexpected* function. An attempt to return to the caller results in undefined program behavior.

*unexpected_func* can also call *abort*, *exit*, or *terminate*.

See also the description of *abort*, *exit*, *set_terminate*, and *terminate*.

## terminate function — except.h

```
void terminate();
```

The function *terminate* can be called by *unexpected* or by the program when a handler for an exception cannot be found. The default action by *terminate*

is to call *abort*. Such a default action causes immediate program termination.

You can modify the way your program terminates when an exception is generated that is not listed in the exception specification. If you don't want the program to terminate with a call to *abort*, you can instead define a function to be called. Such a function (called a *terminate_function*) will be called by *terminate* if it is registered with *set_terminate*.

The function does not return.

See also the description of *abort* and *set_terminate*.

## Type_info class — typeinfo.h

Provides information about a type.

### Public constructor

**Constructor**

None.

Only a private constructor is provided. You cannot create *Type_info* objects. By declaring your objects to be **__rtti** types, or by using the **-RT** compiler switch, the compiler provides your objects with the elements of *Type_info*.

*Type_info* references are generated by the **typeid** operator. See Chapter 2 in the *Programmer's Guide* for a discussion of **typeid**.

### Operators

**operator ==**

```
int operator==(const Type_info &) const;
```

Provides comparison of *Typeinfo*s.

**operator !=**

```
int operator!=(const Type_info &) const;
```

Provides comparison of *Typeinfo*s.

### Public member functions

**before**

```
int before(const Type_info &);
```

Use this function to compare the lexical order of types. For example, to compare two types, ***T1*** and ***T2***, use the following syntax:

```
typeid( T1 ).before(typeid( T2 ));
```

The *before* function returns 0 or 1.

**fname**
**name**

```
const char* _ _far fname() const;
const char* name() const;
```

The functions, *fname* and *name*, perform identically. Use *fname* in large memory-model programs.

Each of the functions returns a printable string that identifies the type name of the operand to **typeid**. The space for the character string is overwritten on each call.

## unexpected function — except.h

```
void unexpected();
```

The *unexpected* function is called when a function throws an exception not listed in its exception specification. The program calls *unexpected*, which by default calls any user-defined function registered by *set_unexpected*. If no function is registered with *set_unexpected*, the *unexpected* function then calls *terminate*.

The *unexpected* function does not return. However, the function can throw an exception.

See also the description of *set_unexpected* and *terminate*.

## xalloc class — except.h

Reports an error on allocation request.

### Public constructors

**Constructor**

```
xalloc(const string &msg, size_t size);
```

The *xalloc* class has no default constructor. Every use of *xalloc* must define the message to be reported when a *size* allocation cannot be fulfilled. The *string* type is defined in cstring.h header file.

### Public member functions

**raise**

```
void raise() throw(xalloc);
```

Calling *raise* causes an *xalloc* to be thrown. In particular, it throws ***this**.

**requested**

```
size_t requested() const;
```

Returns the number of bytes that were requested for allocation.

## xmsg class — except.h

Reports a message related to an exception.

### Public constructor

**Constructor**

```
xmsg(string msg);
```

There is no default constructor for *xmsg*. Every *xmsg* object must have a *string* message explicitly defined. The *string* type is defined in cstring.h header file.

### Public member functions

**raise**

```
void raise() throw(xmsg);
```

Calling *raise* causes an *xmsg* to be thrown. In particular, it throws ***this**.

**why**

```
string why() const;
```

Reports the string used to construct an *xmsg*. Because every *xmsg* must have its message explicitly defined, every instance should have a unique message.

CHAPTER 11

# C++ utility classes

This chapter is a reference guide for the following classes, which are listed here with their associated header-file names:

- Date class (date.h)
- File classes (file.h)
- String classes (cstring.h)
- Threading classes (thread.h)
- Time classes (time.h)

The header files for these classes are found in \BC4\INCLUDE or \BC4\INCLUDE\CLASSLIB.

## TDate class — date.h

```
class  TDate
```

Class *TDate* represents a date. It has members that read, write, and store dates, and that convert dates to Gregorian calendar dates.

### Type definitions

**DayTy**

```
typedef unsigned DayTy;
```

Day type.

**HowToPrint**

```
enum HowToPrint{ Normal, Terse, Numbers, EuropeanNumbers, European };
```

Lists different print formats.

**JulTy**

```
typedef unsigned long JulTy;
```

Julian calendar type.

**MonthTy**

```
typedef unsigned MonthTy;
```

Month type.

**YearTy**

```
typedef unsigned YearTy;
```

Year type.

## Public constructors

**Constructor**

```
TDate();
```

Constructs a *TDate* object with the current date.

**Constructor**

```
TDate( DayTy day, YearTy year );
```

Constructs a *TDate* object with the given *day* and *year*. The base date for this computation is Dec. 31 of the previous year. If year == 0, it constructs a *TDate* with Jan. 1, 1901 as "day zero." For example, TDate(-1,0) = Dec. 31, 1900 and TDate(1,0) = Jan. 2, 1901.

**Constructor**

```
TDate( DayTy day, const char* month, YearTy year);
TDate( DayTy day, MonthTy month, YearTy year);
```

Constructs a *TDate* object for the given *day*, *month*, and *year*.

**Constructor**

```
TDate( istream& is );
```

Constructs a *TDate* object, reading the date from input stream *is*.

**Constructor**

```
TDate( const TTime& time);
```

Constructs a *TDate* object from *TTime* object *time*.

## Public member functions

**AsString**

```
string AsString() const;
```

Converts the *TDate* object to a *string* object.

**Between**

```
int Between( const TDate& d1, const TDate& d2 ) const;
```

Returns 1 if this *TDate* object is between *d1* and *d2*, inclusive.

**CompareTo**

```
int CompareTo( const TDate & ) const;
```

Returns 1 if the target *TDate* is greater than parameter *TDate*, –1 if the target is less than the parameter, and 0 if the dates are equal.

**Day**

```
DayTy Day() const;
```

Returns the day of the year (1-365).

**DayName**

```
const char *DayName( DayTy weekDayNumber );
```

Returns a string name for the day of the week, where Monday is 1 and Sunday is 7.

**DayOfMonth**

```
DayTy DayOfMonth() const;
```

Returns the day of the month (1-31).

**DayOfWeek**

```
DayTy DayOfWeek( const char* dayName );
```

Returns the number associated with a string naming the day of the week, where Monday is 1 and Sunday is 7.

**DaysInYear**

```
DayTy DaysInYear( YearTy );
```

Returns the number of days in the specified year (365 or 366).

**DayWithinMonth**

```
int DayWithinMonth( MonthTy, DayTy, YearTy );
```

Returns 1 if the given day is within the given month for the given year.

**FirstDayOfMonth**

```
DayTy FirstDayOfMonth() const;
```

Returns the number of the first day of the month for this *TDate*.

```
DayTy FirstDayOfMonth( MonthTy month) const;
```

Returns the number of the first day of a given month. Returns 0 if *month* is outside the range 1 through 12.

**Hash**

```
unsigned Hash() const;
```

Returns a hash value for the date.

**IndexOfMonth**

```
MonthTy IndexOfMonth( const char *monthName );
```

Returns the number (1-12) of the month *monthname*.

**IsValid**

```
int IsValid() const;
```

Returns 1 if this *TDate* is valid, 0 otherwise.

**Jday**

```
JulTy Jday( MonthTy, DayTy, YearTy );
```

Converts the given Gregorian calendar date to the corresponding Julian day number. Gregorian calendar started on Sep. 14, 1752. This function not valid before that date. Returns 0 if the date is invalid.

**Leap**

```
int Leap() const;
```

Returns 1 if this *TDate*'s year is a leap year, 0 otherwise.

**Max**

```
TDate Max( const TDate& dt ) const;
```

Compares this *TDate* with *dt* and returns the date with the greater Julian number.

**Min**

```
TDate Min( const TDate& dt ) const;
```

Compares this *TDate* with *dt* and returns the date with the lesser Julian number.

**Month**

```
MonthTy Month() const;
```

Returns the month number for this *TDate*.

**MonthName**

```
const char *MonthName( MonthTy monthNumber );
```

Returns the string name for the given *monthNumber* (1-12). Returns 0 for an invalid *monthNumber*.

**NameOfDay**

```
const char *NameOfDay() const;
```

Returns this *TDate*'s day string name.

**NameOfMonth**

```
const char *NameOfMonth() const;
```

Returns this *TDate*'s month string name.

**Previous**

```
TDate Previous( const char *dayName ) const;
```

Returns the *TDate* of the previous *dayName*.

```
TDate Previous( DayTy day ) const;
```

Returns the *TDate* of the previous *day*.

**SetPrintOption**

```
HowToPrint SetPrintOption( HowToPrint h );
```

Sets the print option for all *TDate* objects and returns the old setting. See *HowToPrint* in the "Type definition" section for this class.

**WeekDay**

```
DayTy WeekDay() const;
```

Returns 1 (Monday) through 7 (Sunday).

**Year**

```
YearTy Year() const;
```

Returns the year of this *TDate*.

## Protected member functions

**AssertIndexOfMonth**

```
static int AssertIndexOfMonth( MonthTy m );
```

Returns 1 if *m* is between 1 and 12 inclusive, otherwise returns 0.

**AssertWeekDayNumber**

```
static int AssertWeekDayNumber( DayTy d);
```

Returns 1 if *d* is between 1 and 7 inclusive, otherwise returns 0.

## Operators

**Operator <**

```
int operator < ( const TDate& date ) const;
```

Returns 1 if this *TDate* precedes *date*, otherwise returns 0.

**Operator <=**

```
int operator <= ( const TDate& date ) const;
```

Returns 1 if this *TDate* is less than or equal to *date*, otherwise returns 0.

**Operator >**

```
int operator >  ( const TDate& date ) const;
```

Returns 1 if this *TDate* is greater than *date*, otherwise returns 0.

**Operator >=**

```
int operator >= ( const TDate& date ) const;
```

Returns 1 if this *TDate* is greater than or equal to *date*, otherwise returns 0.

**Operator ==**

```
int operator == ( const TDate& date ) const;
```

Returns 1 if this *TDate* is equal to *date*, otherwise returns 0.

**Operator !=**

```
int operator != ( const TDate& date ) const;
```

Returns 1 if this *TDate* is not equal to *date*, otherwise returns 0.

**Operator –**

```
JulTy operator - ( const TDate& dt ) const;
```

Subtracts *dt* from this *TDate* and returns the difference.

**Operator +**

```
friend TDate operator + ( const TDate& dt, int dd );
friend TDate operator + ( int dd, const TDate& dt );
```

Returns a new *TDate* containing the sum of this *TDate* and *dd*.

**Operator –**

```
friend TDate operator - ( const TDate& dt, int dd );
```

Subtracts *dd* from this *TDate* and returns the difference.

**Operator ++**

```
void operator ++ ();
```

Increments this *TDate* by 1.

**Operator – –**

```
void operator -- ();
```

Decrements this *TDate* by 1.

**Operator +=**

```
void operator += ( int dd );
```

Adds *dd* to this *TDate*.

**Operator –=**

```
void operator -= ( int dd );
```

Subtracts *dd* from this *TDate*.

**Operator <<**

```
friend ostream& operator << ( ostream& os, const TDate& date );
```

Inserts *date* into output stream *os*.

**Operator >>**

```
friend istream& operator >> ( istream& is, TDate& date );
```

Extracts *date* from input stream *is*.

## TFileStatus structure — file.h

```
struct TFileStatus
{
   TTime createTime;
   TTime modifyTime;
   TTime accessTime;
   long size;
   uint8 attribute;
   char fullName[_MAX_PATH];
};
```

Describes a file record containing creation, modification, and access times; also provides the file size, attributes, and name.

See also: *TTime* class

## TFile class — file.h

```
class  TFile
```

Class *TFile* encapsulates standard file characteristics and operations.

### Public data members

**FileNull**

```
enum { FileNull };
```

Represents a null file handle.

**File flags**

```
enum{
   ReadOnly    = O_RDONLY,
   ReadWrite   = O_RDWR,
   WriteOnly   = O_WRONLY,
   Create      = O_CREAT | O_TRUNC,
   CreateExcl  = O_CREAT | O_EXCL,
   Append      = O_APPEND,
```

```
#if defined( __FLAT__ )
   Compat      = SH_COMPAT,
   DenyNone    = SH_DENYNONE,
#else
   DenyRead    = SH_DENYRD,
   DenyWrite   = SH_DENYWR,
#endif
   DenyRdWr    = SH_DENYRW,
   NoInherit   = O_NOINHERIT
   };
```

Enumerates file-translation modes and sharing capabilities. See the *open* and *sopen* functions in Chapter 3.

```
enum{
   PermRead    = S_IREAD,
   PermWrite   = S_IWRITE,
   PermRdWr    = S_IREAD | S_IWRITE
   };
```

Enumerates file read and write permissions. See the *creat* function in Chapter 3.

```
enum{
   Normal      = 0x00,
   RdOnly      = 0x01,
   Hidden      = 0x02,
   System      = 0x04,
   Volume      = 0x08,
   Directory   = 0x10,
   Archive     = 0x20
  };
```

Enumerates file types.

```
enum seek_dir
   {
   beg = 0,
   cur = 1,
   end = 2
   };
```

Enumerates file-pointer seek direction.

## Public constructors

**Constructor**

```
TFile();
```

Creates a *TFile* object with a file handle of *FileNull*.

**Constructor**

```
TFile( int handle );
```

Creates a *TFile* object with a file handle of *handle*.

**Constructor**

```
TFile( const TFile& file );
```

Creates a *TFile* object with the same file handle *file*.

**Constructor**

```
TFile( const char* name, uint16 access=ReadOnly, uint16
permission=PermRdWr );
```

Creates a *TFile* object and opens file *name* with the given attributes. The file is created if it doesn't exist.

## Public member functions

**Close**

```
int Close();
```

Closes the file. Returns nonzero if successful, 0 otherwise.

**Flush**

```
void Flush();
```

Performs any pending I/O functions.

**GetHandle**

```
int GetHandle() const;
```

Returns the file handle.

**GetStatus**

```
int GetStatus( TFileStatus& status ) const;
```

Fills *status* with the current file status. Returns nonzero if successful, 0 otherwise.

```
int GetStatus( const char *name, TFileStatus& status );
```

Fills *status* with the status for file *name*. Returns nonzero if successful, 0 otherwise.

**IsOpen**

```
int IsOpen() const;
```

Returns 1 if the file is open, 0 otherwise.

**Length**

```
long Length() const;
```

Returns the file length.

```
void Length( long newLen );
```

Resizes file to *newLen*.

**LockRange**

```
void LockRange( long position, uint32 count );
```

Locks *count* bytes, beginning at *position* of the associated file.

See also: *UnlockRange*

**Open**

```
int Open( const char* name, uint16 access, uint16 permission );
```

Opens file *name* with the given attributes. The file will be created if it doesn't exist. Returns 1 if successful, 0 otherwise.

**Position**

```
long Position() const;
```

Returns the current position of the file pointer. Returns –1 to indicate an error.

**Read**

```
int Read( void *buffer, int numBytes );
```

Reads *numBytes* from the file into *buffer*.

```
long Read( void huge *buffer, long numBytes );
```

Reads *numBytes* from the file into *buffer* (32-bit Windows version).

**Remove**

```
static void Remove( const char *name );
```

Removes file *name*. Returns 0 if successful, –1 if unsuccessful.

**Rename**

```
static void Rename( const char *oldName, const char *newName );
```

Renames file *oldName* to *newName*.

**Seek**

```
long Seek( long offset, int origin = beg );
```

Repositions the file pointer to *offset* bytes from the specified *origin*.

**SeekToBegin**

```
long SeekToBegin();
```

Repositions the file pointer to the beginning of the file.

**SeekToEnd**

```
long SeekToEnd();
```

Repositions the file pointer to the end of the file.

**SetStatus**

```
static int SetStatus( const char *name, const TFileStatus& status );
```

Sets file *name*'s status to *status*.

**UnlockRange**

```
void UnlockRange(long Position, uint32 count );
```

Unlocks the range at the given *Position*.

See also: *LockRange*

**Write**

```
int Write( const void *buffer, int numBytes );
```

Writes *numbytes* of *buffer* to the file.

```
long Write( const void huge *buffer, long numBytes );
```

Writes *numbytes* of *buffer* to the file (32-bit Windows version).

## String class — cstring.h

```
class  string
```

This class uses a technique called "copy-on-write." Multiple instances of a string can refer to the same piece of data so long as it is in a "read-only" situation. If a string writes to the data, a copy is automatically made if more than one string is referring to it.

### Type definitions

**StripType**

```
enum StripType { Leading, Trailing, Both };
```

Enumerates type of stripping. See *strip* in the "Public member functions" section for this class.

### Public constructors and destructor

**Constructor**

```
string();
```

The default constructor. Creates a string of length zero.

**Constructor**

```
string(const string _FAR &s);
```

Copy constructor. Creates a string that contains a copy of the contents of string *s*.

**Constructor**

```
string( const string _FAR &s, size_t n )
```

Creates a string containing a copy of the *n* bytes of string *s*.

**Constructor**

```
string(const char _FAR *cp);
```

Creates a string containing a copy of the bytes from the location pointed to by *cp* through the first 0 byte (conversion from *char**).

**Constructor**

```
string(const char _FAR *cp, size_t n);
```

Creates a string containing a copy of the *n* bytes beginning at the location pointed to by *cp*.

**Constructor**

```
string( char c )
```

Constructs a string containing the character c.

**Constructor**

```
string( char c, size_t n )
```

Constructs a string containing the character c repeated *n* times.

**Constructor**

```
string( signed char c )
```

Constructs a string containing the character c.

**Constructor**

```
string( signed char c, size_t n )
```

Constructs a string containing the character c repeated *n* times.

**Constructor**

```
string( unsigned char c )
```

Constructs a string containing the character c.

**Constructor**

```
string( unsigned char c, size_t n )
```

Constructs a string containing the character c repeated *n* times.

**Constructor**

```
string(const TSubString _FAR &ss);
```

Constructs a string from the substring *ss*.

**Constructor**

```
string( const char __far *cp )
string( const char __far *cp, size_t n )
```

Constructs strings for Windows small and medium memory models.

**Constructor**

```
string( HINSTANCE instance, UINT id, int len = 255 )
```

Windows version for constructing a string from a resource.

**Destructor**

```
~String();
```

Destroys the string and frees all resources allocated to this object.

## Public member functions

**ansi_to_oem**

```
void ansi_to_oem()
```

Converts the target string from the ANSI character set into the OEM-defined character set (Windows only).

**append**

```
string _FAR & append( const string _FAR &s )
```

Appends string *s* to the target string.

```
string _FAR & append( const string _FAR &s, size_t n )
```

Appends the first *n* characters of string *s* to the target string.

```
string _FAR & append( const char _FAR *cp, size_t n )
```

Appends the first *n* characters of the character array *cp* to the target string.

**assign**

```
string _FAR & assign( const string _FAR &s );
```

Assigns string *s* to target string.

See also: *operator* =

```
string _FAR & assign( const string _FAR &s, size_t n );
```

Assigns *n* characters of string *s* to target string.

See also: *operator* =

**compare**

```
int compare(const string _FAR &s);
```

Compares the target string to the string *s*. *compare* returns an integer less than, equal to, or greater than 0, depending on whether the target string is less than, equal to, or greater than *s*.

```
int compare(const string _FAR &s, size_t n );
```

Compares not more than *n* characters from the target string to the string *s*.

**contains**

```
int contains(const char _FAR * pat) const
```

Returns 1 if *pat* is found in the target string, 0 otherwise.

```
int contains(const string _FAR & s) const
```

Returns 1 if string *s* is found in the target string, 0 otherwise.

**copy**

```
size_t copy( char _FAR *cb, size_t n )
```

Copies at most *n* characters from the target string into the *char* array pointed to by *cb*. *copy* returns the number of characters copied.

```
size_t copy( char _FAR *cb, size_t n, size_t pos )
```

Copies at most *n* characters beginning at position *pos* from the target string into the *char* array pointed to by *cb*. *copy* returns the number of characters copied.

```
string copy() const throw( xalloc ).
```

Returns a distinct copy of the string.

**c_str**

```
const char _FAR *c_str() const
```

Returns a pointer to a zero-terminated character array that holds the same characters contained in the string. The returned pointer might point to the actual contents of the string, or it might point to an array that the string allocates for this function call. The effects of any direct modification to the contents of this array are undefined, and the results of accessing this array

after the execution of any non-**const** member function on the target string are undefined.

Conversions from a string object to a *char** are inherently dangerous, because they violate the class boundary and can lead to dangling pointers. For this reason class string does not have an implicit conversion to *char**, but provides *c_str* for use when this conversion is needed.

**find**

```
size_t find( const string _FAR &s )
```

Locates the first occurrence of the string *s* in the target string. If the string is found, it returns the position of the beginning of *s* within the target string. If the string *s* is not found, it returns *NPOS*.

```
size_t find( const string _FAR &s, size_t pos )
```

Locates the first occurrence of the string *s* in the target string, beginning at the position *pos*. If the string is found, it returns the position of the beginning of *s* within the target string. If the *s* is not found, it returns *NPOS* and does not change *pos*.

```
size_t find( const TRegexp _FAR &pat, size_t i = 0 )
```

Searches the string for patterns matching regular expression *pat* beginning at location *i*. It returns the position of the beginning of *pat* within the target string. If the *pat* is not found, it returns *NPOS* and does not change *pos*.

```
size_t find( const TRegexp _FAR &pat, size_t _FAR *ext, size_t i = 0 )
const;
```

Searches the string for patterns matching regular expression *pat* beginning at location *i*. Parameter *ext* returns the length of the matching string if found. It returns the position of the beginning of *pat* within the target string. If the *pat* is not found, it returns *NPOS* and does not change *pos*.

See also: *rfind*

**find_first_of**

```
size_t find_first_of( const string _FAR &s ) const
```

Locates the first occurrence in the target string of any character contained in string *s*. If the search is successful *find_first_of* returns the character location. If the search fails or if `pos > length()`, *find_first_of* returns 0.

```
size_t find_first_of( const string _FAR &s, size_t pos ) const
```

Locates the first occurrence in the target string of any character contained in string *s*. If the search is successful, *pos* is set to the position of that character within the target string, and *find_first_of* returns 1. If the search fails or if `pos > length()`, *find_first_of* returns 0.

**find_first_not_of**

```
size_t find_first_not_of( const string _FAR &s) const
```

Locates the first occurrence in the target string of any character not contained in string *s*. If the search is successful, *find_first_not_of* returns the character location. If the search fails or if `pos > length()`, *find_first_not_of* returns 0.

```
size_t find_first_not_of( const string _FAR &s, size_t pos ) const
```

Locates the first occurrence in the target string of any character not contained in string *s*. If the search is successful, *pos* is set to the position of that character within the target string, and *find_first_not_of* returns 1. If the search fails or if `pos > length()`, *find_first_not_of* returns 0.

**find_last_of**

```
size_t find_last_of( const string _FAR &s ) const
```

Locates the last occurrence in the target string of any character contained in string *s*. If the search is successful *find_last_of* returns the character location. If the search fails or if `pos > length()`, *find_last_of* returns 0.

```
size_t find_last_of( const string _FAR &s, size_t pos ) const
```

Locates the last occurrence in the target string of any character contained in string *s*. If the search is successful,*pos* is set to the position of that character within the target string, and *find_last_of* returns 1. If the search fails or if `pos > length()`, *find_last_of* returns 0.

**find_last_not_of**

```
size_t find_last_not_of( const string _FAR &s ) const
```

Locates the last occurrence in the target string of any character not contained in string *s*. If the search is successful *find_last_not_of* returns the character location. If the search fails or if `pos > length()`, *find_last_not_of* returns 0.

```
size_t find_last_not_of( const string _FAR &s, size_t pos ) const
```

Locates the last occurrence in the target string of any character not contained in string *s*. If the search is successful, *pos* is set to the position of that character within the target string, and *find_last_not_of* returns 1. If the search fails or if `pos > length()`, *find_last_not_of* returns 0.

**get_at**

```
char get_at( size_t pos ) const throw( outofrange );
```

Returns the character at the specified position. If `pos > length()-1`, an *outofrange* exception is thrown.

See also: *put_at*

**get_case_sensitive_flag**

```
static int get_case_sensitiveFlag()
```

Returns 0 if string comparisons are case sensitive, 1 if not.

**get_initial_capacity**

```
static unsigned get_initial_capacity()
```

Returns the number of characters that will fit in the string without resizing.

**get_max_waste** `static unsigned get_max_waste()`

After a string is resized, returns the amount of free space available.

**get_paranoid_check** `static int get_paranoid_check();`

Returns 1 if paranoid checking is enabled, 0 if not.

**get_resize_increment** `static unsigned get_resize_increment()`

Returns the string resizing increment.

**get_skipwhitespace_flag** `static int get_skipwhitespace_flag()`

Returns 1 if whitespace is skipped, 0 if not.

**hash** `unsigned hash() const;`

Returns a hash value.

**initial_capacity** `static size_t initial_capacity(size_t ic = 63);`

Sets initial string allocation capacity.

**insert** `string _FAR &insert( size_t pos, const string _FAR &s )`

Inserts string *s* at position *pos* in the target string. *insert* returns a reference to the resulting string.

`string _FAR &insert( size_t pos, const string _FAR &s, size_t n )`

Inserts *n* characters of string *s* at position *pos* in the target string. *insert* returns a reference to the resulting string.

**is_null** `int is_null() const`

Returns 1 if the string is empty, 0 otherwise.

**length** `unsigned length() const`

Returns the number of characters in the target string. Since null characters can be stored in a string, `length()` might be greater than `strlen(c_str())`.

**max_waste** `static size_t MaxWaste(size_t mw = 63);`

Sets the maximum empty space size and resizes the string.

**oem_to_ansi** `void oem_to_ansi()`

Windows function for converting the target string from the ANSI character set to the OEM-defined character set (Windows only).

**prepend** `string _FAR &prepend( const string _FAR &s )`

Prepends string *s* to the target string.

```
string _FAR &prepend( const string _FAR &s, size_t n )
```

Prepends the first *n* characters of string *s* to the target string.

```
string _FAR &prepend( const char _FAR *cp )
```

Prepends the character array *cp* to the target string.

```
string _FAR &prepend( const char _FAR *cp, size_t n )
```

Prepends the first *n* characters of the character array *cp* to the target string.

**put_at**

```
void put_at( size_t pos, char c ) throw( outofrange );
```

Replaces the character at *pos* with *c*. If `pos == length()`, *putAt* appends *c* to the target string. If `pos > length()` an *outofrange* exception is thrown.

**read_file**

```
istream _FAR &read_file(istream _FAR &is);
```

Reads from input stream *is* until an EOF or a null terminator is reached.

**read_line**

```
istream _FAR &read_line(istream _FAR &is);
```

Reads from input stream *is* until an EOF or a newline is reached.

**read_string**

```
istream _FAR &read_string(istream _FAR &is);
```

Reads from input stream *is* until an EOF or a null terminator is reached.

**read_to_delim**

```
istream _FAR &read_to_delim(istream _FAR &is, char delim='\n');
```

Reads from input stream *is* until an EOF or a *delim* is reached.

**read_token**

```
istream _FAR &read_token(istream _FAR &is);
```

Reads from input stream *is* until whitespace is reached. Note that this function skips any initial whitespace.

**rfind**

```
size_t rfind( const string _FAR &s )
```

Locates the last occurrence of the string *s* in the target string. If the string is found, it returns the position of the beginning of the string *s* within the target string. If *s* is not found, it returns *NPOS*.

```
size_t rfind( const string _FAR &s, size_t pos )
```

Locates the last occurrence of the string *s* that is not beyond the position *pos* in the target string. If the string is found, it returns the position of the beginning of *s* within the target string. If *s* is not found, it returns *NPOS* and does not change *pos*.

See also: *find*

**remove**

```
string _FAR &remove( size_t pos );
```

Removes the characters from *pos* to the end of the target string and returns a reference to the resulting string.

```
string _FAR &remove( size_t pos, size_t n )
```

Removes at most *n* characters from the target string beginning at *pos* and returns a reference to the resulting string.

**replace**

```
string _FAR &replace( size_t pos, size_t n, const string _FAR &s )
```

Removes at most *n* characters from the target string beginning at *pos*, and replaces them with a copy of the string *s*. *replace* returns a reference to the resulting string.

```
string _FAR &replace( size_t pos, size_t n1, const string _FAR &s, size_t
n2 )
```

Removes at most *n1* characters from the target string beginning at *pos*, and replaces them with the first *n2* characters of string *s*. *replace* returns a reference to the resulting string.

**reserve**

```
size_t reserve() const
```

Returns an implementation-dependent value that indicates the current internal storage size. The returned value is always greater than or equal to `length()`.

```
void reserve( size_t ic )
```

Suggests to the implementation that the target string might eventually require *ic* bytes of storage.

**resize**

```
void resize(size_t m);
```

Resizes the string to *m* characters, truncating or adding blanks as necessary.

**resize_increment**

```
static size_t resize_increment(size_t ri = 64);
```

Sets the resize increment for automatic resizing.

**set_case_sensitive**

```
static int set_case_sensitive(int tf = 1);
```

Sets case sensitivity. 1 is case sensitive; 0 is not case sensitive.

**set_paranoid_check**

```
static int set_paranoid_check(int ck = 1);
```

String searches use a hash value scheme to find the strings. There is a possibility that more than one string could hash to the same value. Calling *set_paranoid_check* with *ck* set to 1 forces checking the string found against the desired string with the C library function *strcmp*. When *set_paranoid_check* is called with *ck* set to 0, this final check isn't made.

**skip_whitespace**

```
static int skip_whitespace(int sk = 1);
```

Set to 1 to skip whitespace after a token read, 0 otherwise.

**strip**

```
TSubString strip( StripType s = Trailing, char c=' ');
```

Strips away *c* characters from the beginning, end, or both (beginning and end) of string *s*, depending on *StripType*.

**substr**

```
string substr( size_t pos ) const
```

Creates a string containing a copy of the characters from *pos* to the end of the target string.

```
string substr( size_t pos, size_t n ) const
```

Creates a string containing a copy of not more than *n* characters from *pos* to the end of the target string.

**substring**

```
TSubString substring( const char _FAR *cp )
```

Creates a *TSubString* object containing a copy of the characters pointed to by **cp*.

```
const TSubString substring( const char _FAR *cp ) const
```

Creates a *TSubString* object containing a copy of the characters pointed to by **cp*.

```
TSubString substring( const char _FAR *cp, size_t start )
```

Creates a *TSubString* object containing a copy of the characters pointed to by **cp*, starting at character *start*.

```
const TSubString substring( const char _FAR *cp, size_t start ) const
```

Creates a *TSubString* object containing a copy of the characters pointed to by **cp*, starting at character *start*.

**to_lower**

```
void to_lower();
```

Changes the string to lowercase.

**to_upper**

```
void to_upper();
```

Changes target string to uppercase.

## Protected member functions

**assert_element**

```
void assert_element( size_t pos ) const
```

Throws an *outofrange* exception if an invalid element is given.

**assert_index**

```
void assert_index( size_t pos ) const
```

Throws an *outofrange* exception if an invalid index is given.

**cow**

```
void cow();
```

Copy on write. Multiple instances of a string can refer to the same piece of data as long as it is in a read-only situation. If a string writes to the data, then *cow* (copy on write) is called to make a copy if more than one string is referring to it.

**valid_element**

```
int valid_element( size_t pos ) const
```

Returns 1 if *pos* is an element of the string, 0 otherwise.

**valid_index**

```
int valid_index( size_t pos ) const
```

Returns 1 if *pos* is a valid index of the string, 0 otherwise.

## Operators

**Operator =**

```
string _FAR & operator=(const string _FAR &s);
```

If the target string is the same object as the parameter passed to the assignment, the assignment operator does nothing. Otherwise it performs any actions necessary to free up resources allocated to the target string, then copies *s* into the target string.

**Operator +=**

```
string _FAR & operator += (const String _FAR &s)
```

Appends the contents of the string *s* to the target string.

```
string _FAR & operator+=(const char _FAR *cp);
```

Appends the contents of *cp* to the target string.

**Operator +**

```
friend String _Cdecl _FARFUNC operator+(const String _FAR &, const char
_FAR *cp);
```

Concatenates string *s* and *cp*.

**Operator []**

```
char _FAR & operator[](size_t pos);
```

Returns a reference to the character at position *pos*.

```
char operator[](size_t pos) const;
```

Returns the character at position *pos*.

**Operator ()**

```
char _FAR & operator()(size_t pos);
```

Returns a reference to the character at position *pos*.

```
TSubString operator()(size_t start, size_t len);
```

Returns the substring beginning at location *start* and spanning *len* bytes.

```
TSubString operator()(const TRegexp _FAR & re);
```

Returns the first occurrence of a substring matching regular expression *re*.

```
TSubString operator()(const TRegexp _FAR & re, size_t start);
```

Returns the first occurrence of a substring matching regular expression *re*, beginning at location *start*.

```
char operator()(size_t pos) const;
```

Returns the character at position *pos*.

```
const TSubString operator()(size_t start, size_t len) const;
```

Returns the substring beginning at location *start* and spanning *len* bytes.

```
const TSubString operator()(const TRegexp _FAR & pat) const;
```

Returns the first occurrence of a substring matching regular expression *re*.

```
const TSubString operator()(const TRegexp _FAR & pat, size_t start) const;
```

Returns the first occurrence of a substring matching regular expression *re*, beginning at location *start*.

**Operator ==**

```
friend int operator == ( const String _FAR &s1, const String _FAR &s2 );
```

Tests for equality of string *s1* and string *s2*. Two strings are equal if they have the same length, and if the same location in each string contains characters that compare equally. Operator == returns a 1 to indicate that the strings are equal, and a 0 to indicate that they are not equal.

```
friend int operator == ( const String _FAR &s, const char _FAR *cp );
friend int operator == ( const char _FAR *cp, const String _FAR &s );
```

Tests for equality of string *s1* and *char *cp*. The two are equal if they have the same length, and if the same location in each string contains characters that compare equally. Operator == returns a 1 to indicate that the strings are equal, and a 0 to indicate that they are not equal.

**Operator !=**

```
friend int operator != ( const String _FAR &s1, const String _FAR &s2 );
```

Tests for inequality of strings *s1* and *s2*. Two strings are equal if they have the same length, and if the same location in each string contains characters that compare equally. Operator != returns a 1 to indicate that the strings are not equal, and a 0 to indicate that they are equal.

```
friend int operator != ( const String _FAR &s, const char _FAR *cp );
friend int operator != ( const char _FAR *cp, const String _FAR &s );
```

Tests for inequality between string *s* and *char *cp*. The two are equal if they have the same length, and if the same location in each string contains the same character. Operator != returns a 1 to indicate that the strings are not equal, and a 0 to indicate that they are equal.

**Operator <**

```
friend int operator < ( const String _FAR &s1, const String _FAR &s2 );
```

Compares string *s1* to string *s2*. Returns 1 if string *s1* is less than *s2*, 0 otherwise.

```
friend int operator < ( const String _FAR &s, const char _FAR *cp );
friend int operator < ( const char _FAR *cp, const String _FAR &s );
```

Compares string *s1* to **cp2*. Returns 1 if the left side of the expression is less than the right side, 0 otherwise.

**Operator <=**

```
friend int operator <= ( const String _FAR &s1, const String _FAR &s2 );
```

Compares string *s1* to string *s2*. Returns 1 if string *s1* is less than or equal to *s2*, 0 otherwise.

```
friend int operator <= ( const String _FAR &s, const char _FAR *cp );
friend int operator <= ( const char _FAR *cp, const String _FAR &s );
```

Compares string *s1* to **cp*. Returns 1 if the left side of the expression is less than or equal to the right side, 0 otherwise.

**Operator >**

```
friend int operator > ( const String _FAR &s1, const String _FAR &s2 );
```

Compares string *s1* to string *s2*. Returns 1 if string *s1* is greater than *s2*, 0 otherwise.

```
friend int operator > ( const String _FAR &s, const char _FAR *cp );
friend int operator > ( const char _FAR *cp, const String _FAR &s );
```

Compares string *s1* to **cp2*. Returns 1 if the left side of the expression is greater than the right side, 0 otherwise.

**Operator >=**

```
friend int operator >= ( const String _FAR &s1, const String _FR &s2 );
```

Compares string *s1* to string *s2*. Returns 1 if string *s1* is greater than or equal to *s2*, 0 otherwise.

```
friend int operator >= ( const String _FAR &s, const char _FAR *cp );
friend int operator >= ( const char _FAR *cp, const String _FAR &s );
```

Compares string *s1* to **cp*. Returns 1 if the left side of the expression is greater than or equal to the right side, 0 otherwise.

**Operator >>**

```
friend ipstream _FAR & operator >> ( ipstream _FAR & is, string _FAR & str );
```

Extracts string *str* from input stream *is*.

## Related global operators and functions

**Operator >>**

```
istream _FAR & _Cdecl _FARFUNC operator>>(istream _FAR &is, string _FAR
&s);
```

Behaves the same as `operator>>(istream&, char *)` (see Chapter 5), and returns a reference to *is*.

**Operator <<**

```
ostream _FAR & _Cdecl _FARFUNC operator<<(ostream _FAR &os, const String
_FAR & s);
```

Behaves the same as `operator<<(ostream&,const char *)` (see Chapter 5) except that it does not terminate when it encounters a null character in the string. Returns a reference to *os*.

```
opstream _FAR& _Cdecl operator << ( opstream _FAR & os, const string _FAR
& str );
```

Inserts string *str* into persistent output stream *os*.

**Operator +**

```
string _Cdecl _FARFUNC operator + ( const char _FAR *cp, const string _FAR
& s);
```

Concatenates **cp* and string *s*.

```
string _Cdecl _FARFUNC operator + ( const string _FAR &s1, const string
_FAR &s2 );
```

Concatenates string *s1* and *s2*.

**getline**

```
istream _FAR & _Cdecl getline( istream _FAR &is, string _FAR &s );
```

Behaves the same as `istream::getline(chptr, NPOS)`, except that instead of storing into a *char* array, it stores into a *string*. *getline* returns a reference to *is*.

```
istream _FAR & _Cdecl getline( istream _FAR &is, string _FAR &s, char c );
```

Behaves the same as `istream::getline(cb, NPOS, c)`, except that instead of storing into a *char* array, it stores into a *string*. *getline* returns a reference to *is*.

**to_lower**

```
String  _Cdecl _FARFUNC to_lower(const string _FAR &s);
```

Changes string *s* to uppercase.

**to_upper**

```
String  _Cdecl _FARFUNC to_upper(const string _FAR &s);
```

Changes string *s* to lowercase.

# TSubString class — cstring.h

```
class  TSubString
```

Addresses selected substrings.

## Public member functions

**get_at**

```
char get_at( size_t pos ) const
```

Returns the character at the specified position. If `pos > length()-1`, an exception is thrown.

See also: *put_at*

**is_null**

```
int is_null() const
```

Returns 1 if the string is empty, 0 otherwise.

**length**

```
size_t length() const
```

Returns the substring length.

**put_at**

```
void put_at( size_t pos, char c )
```

Replaces the character at *pos* with *c*. If `pos == length()`, *putAt* appends *c* to the target string. If `pos > length()`, an exception is thrown.

**start**

```
int start() const
```

Returns the index of the starting character.

**to_lower**

```
void to_lower();
```

Changes the substring to lowercase.

**to_upper**

```
void to_upper();
```

Changes the substring to uppercase.

## Protected member functions

**assert_element**

```
int assert_element(size_t pos) const;
```

Returns 1 if *pos* represents a valid index into the substring, 0 otherwise.

## Operators

**Operator =**

```
TSubString _FAR & operator=(const string _FAR &s);
```

Copies *s* into the target substring.

**Operator ==**

```
int operator==(const char _FAR * cp) const;
```

Tests for equality between the target substring and **cp*. The two are equal if they have the same length, and if the same location in each string contains the same character. Operator == returns a 1 to indicate that the strings are equal, and a 0 to indicate that they are not equal.

```
int operator==(const string _FAR & s) const;
```

Tests for equality between the target substring and string *s*. Two are equal if they have the same length, and if the same location in each string contains the same character. Operator == returns a 1 to indicate that the strings are equal, and a 0 to indicate that they are not equal.

**Operator !=**

```
int operator!=(const char _FAR * cp) const
```

Tests for inequality between the target string and **cp*. Two strings are equal if they have the same length, and if the same location in each string contains the same character. Operator != returns a 1 to indicate that the strings are not equal, and a 0 to indicate that they are equal.

```
int operator!=(const string _FAR & s) const;
```

Tests for inequality between the target string and string *s*. Two strings are equal if they have the same length, and if the same location in each string contains the same character. Operator != returns a 1 to indicate that the strings are not equal, and a 0 to indicate that they are equal.

**Operator ()**

```
char _FAR & operator()(size_t pos);
```

Returns a reference to the character at position *pos*.

```
char operator()(size_t pos) const;
```

Returns the character at position *pos*.

**Operator []**

```
char _FAR & operator[](size_t pos);
```

Returns a reference to the character at position *pos*.

```
char operator[](size_t pos) const;
```

Returns the character at position *pos*.

**Operator !**

```
int operator!() const
```

Detects null substrings. Returns 1 if the substring is not null.

## TCriticalSection class — thread.h

```
class TCriticalSection
```

*TCriticalSection* provides a system-independent interface to critical sections in threads. *TCriticalSection* objects can be used in conjunction with *TCriticalSection::Lock* objects to guarantee that only one thread can be executing any of the code sections protected by the lock at any given time.

See also: *TCriticalSection::Lock*

### Constructors and destructor

**Constructor**

```
TCriticalSection();
```

Constructs a *TCriticalSection* object.

**Destructor**

```
~TCriticalSection();
```

Destroys a *TCriticalSection* object.

## TCriticalSection::Lock class — thread.h

```
class Lock
```

This nested class handles locking and unlocking critical sections. Here's an example:

```
TCriticalSection LockF;
void f()
{
   TCriticalSection::Lock(LockF);

   // critical processing here
}
```

Only one thread of execution will be allowed to execute the critical code inside function *f* at any one time.

### Public constructors and destructor

**Constructor**

```
Lock( const TCriticalSection& );
```

Requests a lock on the *TCriticalSection* object. If no *Lock* object in another thread holds a lock on that *TCriticalSection* object, the lock is allowed and

execution continues. If a Lock object in another thread holds a lock on that object, the requesting thread is blocked until the lock is released.

**Destructor**

```
~Lock();
```

Releases the lock.

## TMutex class — thread.h

*TMutex* provides a system-independent interface to critical sections in threads. *TMutex* objects can be used in conjunction with TMutex::Lock objects to guarantee that only one thread can be executing any of the code sections protected by the lock at any given time.

The differences between the classes *TCriticalSection* and *TMutex* are that a timeout can be specified when creating a *Lock* on a *TMutex* object, and that a *TMutex* object has a HANDLE that can be used outside the class. This mirrors the distinction made in Windows NT between a CRITICALSECTION and a Mutex. Under NT a *TCriticalSection* object is much faster than a *TMutex* object. Under operating systems that don't make this distinction a *TCriticalSection* object can use the same underlying implementation as a *TMutex,* losing the speed advantage that it has under NT.

### Public constructors and destructor

**Constructor**

```
TMutex();
```

Constructs a *TMutex* object.

**Destructor**

```
~TMutex();
```

Destroys a *TMutex* object.

### Operators

**HANDLE**

```
operator HANDLE() const;
```

Returns the handle of the underlying Windows NT semaphore object.

## TMutex::Lock class — thread.h

This nested class handles locking and unlocking *TMutex* objects.

### Public constructors

**Constructor**

```
Lock( const TMutex&, unsigned long timeOut = NoLimit );
```

Requests a lock on the *TMutex* object. If no *Lock* object in another thread holds a lock on that *TMutex* object, the lock is allowed and execution continues. If a Lock object in another thread holds a lock on that object, the requesting thread is blocked until the lock is released.

### Public member functions

**Release**

```
void Release();
```

Releases the lock on the *TMutex* object.

## TSync class — thread.h

*TSync* provides a system-independent interface for building classes that act like monitors—classes in which only one member function can execute on a particular instance at any one time. *TSync* uses *TCriticalSection*, has no public members, and can only be used as a base class. Here is an example of *TSync* in use:

```
class ThreadSafe : private TSync
{
public:
   void f();
   void g();
private:
   int i;
};

void ThreadSafe::f()
{
   Lock(this);
   if( i == 2 )
     i = 3;
}

void ThreadSafe::g()
{
   Lock(this);
   if( i == 3 )
     i = 2;
}
```

See also: class *TSync::Lock*

### Protected constructors

**Constructor**

```
TSync();
```

Default constructor.

**Constructor**

```
TSync( const TSync& );
```

Copy constructor. Does not copy the *TCriticalSection* object.

### Protected operators

**Operator =**

```
const TSync& operator = ( const TSync& s )
```

Assigns *s* to the target, and does not copy the *TCriticalSection* object.

## TSync::Lock class — thread.h

```
class Lock : private TCriticalSection::Lock
```

This nested class handles locking and unlocking critical sections.

### Public constructors and destructor

**Constructor**

```
Lock( const TSync *s );
```

Requests a lock on the critical section of the *TSync* object pointed to by *s*. If no other *Lock* object holds a lock on that *TCriticalSection* object, the lock is allowed and execution continues. If another Lock object holds a lock on that object, the requesting thread is blocked until the lock is released.

**Destructor**

```
~Lock();
```

Releases the lock.

## TThread class — thread.h

```
class TThread
```

*TThread* provides a system-independent interface to threads. Here is an example:

```
class TimerThread : private TThread
{
public:
   TimerThread() : Count(0) {}
private:
   unsigned long Run();
   int Count;
};

unsigned long TimerThread::Run()
{
   // loop 10 times
   while( Count++ < 10 )
   {
      Sleep(1000);    // delay 1 second
      cout << "Iteration " << Count << endl;
   }
   return 0L;
}

int main()
{
   TimerThread timer;
   timer.Start();
   Sleep( 20000 );     // delay 20 seconds
   return 0;
}
```

## Type definitions

**Status**

```
enum Status { Created, Running, Suspended, Finished, Invalid };
```

Describes the state of the thread, as follows:

- *Created*. The object has been created but its thread has not been started. The only valid transition from this state is to *Running*, which happens on a call to *Start*. In particular, a call to *Suspend* or *Resume* when the object is in this state is an error and will throw an exception.
- *Running*. The thread has been started successfully. There are two transitions from this state:
  - When the user calls *Suspend*, the object moves into the *Suspended* state.
  - When the thread exits, the object moves into the *Finished* state.

  Calling *Resume* on an object that is in the *Running* state is an error and will throw an exception.

- *Suspended*. The thread has been suspended by the user. Subsequent calls to *Suspend* nest, so there must be as many calls to *Resume* as there were to *Suspend* before the thread resumes execution.
- *Finished*. The thread has finished executing. There are no valid transitions out of this state. This is the only state from which it is legal to invoke the destructor for the object. Invoking the destructor when the object is in any other state is an error and will throw an exception.

## Protected constructors and destructor

**Constructor**

```
TThread();
```

Constructs an object of type *TThread*.

**Constructor**

```
TThread( const TThread& );
```

Copy constructor. Puts the target object into the *Created* state.

**Destructor**

```
virtual ~TThread();
```

Destroys the *TThread* object.

## Public member functions

**GetPriority**

```
int GetPriority() const;
```

Gets the thread priority.

See also: *SetPriority*

**GetStatus**

```
Status GetStatus() const;
```

Returns the current status of the thread. See data member *Status* for possible values.

**Resume**

```
unsigned long Resume();
```

Resumes execution of a suspended thread.

**SetPriority**

```
int SetPriority(int);
```

Sets the thread priority.

See also: *GetPriority*

**Start**

```
HANDLE Start();
```

Begins execution of the thread, and returns the thread handle.

**Suspend**

```
unsigned long Suspend();
```

Suspends execution of the thread.

**Terminate**

```
void Terminate();
```

Sets an internal flag that indicates that the thread should exit. The derived class can check the state of this flag by calling *ShouldTerminate*.

**TerminateAndWait**

```
void TerminateAndWait( unsigned long timeout = (unsigned long)(-1) );
```

Combines the behavior of *Terminate* and *WaitForExit*. Sets an internal flag that indicates that the thread should exit and blocks the calling thread until the internal thread exits or until the time specified by *timeout*, in milliseconds, expires. A *timeout* of –1 says to wait indefinitely.

**WaitForExit**

```
void WaitForExit( unsigned long timeout = (unsigned long)(-1) );
```

Blocks the calling thread until the internal thread exits or until the time specified by *timeout*, in milliseconds, expires. A *timeout* of –1 says wait indefinitely.

### Protected member functions

**ShouldTerminate**

```
int ShouldTerminate() const;
```

Returns a nonzero value to indicate that *Terminate* or *TerminateAndWait* has been called and that the thread will finish its processing and exit.

### Protected operators

**Operator =**

```
const TThread& operator = ( const TThread& );
```

The *TThread* assignment operator. The target object must be in either the *Created* or *Finished* state. If so, assignment puts the target object into the *Created* state. If the object is not in either state an exception will be thrown.

## TThread::TThreadError class — thread.h

```
class TThreadError
```

*TThreadError* defines the exceptions that are thrown when a threading error occurs.

## Type definitions

### ErrorType

```
enum ErrorType
   {
   SuspendBeforeRun,
   ResumeBeforeRun,
   ResumeDuringRun,
   SuspendAfterExit,
   ResumeAfterExit,
   CreationFailure,
   DestroyBeforeExit,
   AssignError
   };
```

Identifies the type of error that occurred. The following list explains each error type:

- *SuspendBeforeRun*. The user called *Suspend* on an object before calling *Start*.
- *ResumeBeforeRun*. The user called *Resume* on an object before calling *Start*.
- *ResumeDuringRun*. The user called *Resume* on a thread that was not suspended.
- *SuspendAfterExit*. The user called *Suspend* on an object whose thread had already exited.
- *ResumeAfterExit*. The user called *Resume* on an object whose thread had already exited.
- *CreationFailure*. The operating system was unable to create the thread.
- *DestroyBeforeExit*. The object's destructor was invoked before its thread had exited.
- *AssignError*. An attempt was made to assign to an object that was not in either the *Created* or *Finished* state.

## Public member functions

### GetErrorType

```
ErrorType GetErrorType() const;
```

Returns the *ErrorType* for the error that occurred.

## TTime type definitions — time.h

```
typedef unsigned HourTy;
typedef unsigned MinuteTy;
typedef unsigned SecondTy;
typedef unsigned long ClockTy;
```

Type definitions for hours, minutes, seconds, and seconds since January 1, 1901.

## TTime class — time.h

```
class  TTime
```

Class *TTime* encapsulates time functions and characteristics.

### Public constructors

**Constructor**

```
TTime();
```

Constructs a *TTime* object with the current time.

**Constructor**

```
TTime( ClockTy s );
```

Constructs a *TTime* object with the given *s* (seconds since January 1, 1901).

**Constructor**

```
TTime( HourTy h, MinuteTy m, SecondTy s = 0 );
```

Constructs a *TTime* object with the given time and today's date.

**Constructor**

```
TTime( const TDate&, HourTy h=0, MinuteTy m=0, SecondTy s=0 );
```

Constructs a *TTime* object with the given time and date.

### Public member functions

**AsString**

```
string AsString() const;
```

Returns a *string* object containing the time.

**BeginDST**

```
static TTime BeginDST( unsigned year );
```

Returns the start of daylight savings time for the given year.

**Between**

```
int Between( const TTime& a, const TTime& b ) const;
```

Returns 1 if the target date is between *TTimes a* and *b*, 0 otherwise.

**CompareTo**

```
int CompareTo( const TTime & ) const;
```

Compares *t* to this *TTime* object and returns 0 if the times are equal, 1 if *t* is earlier, and –1 if *t* is later.

**EndDST**

```
static TTime EndDST( unsigned year );
```

Returns the time when daylight savings time ends for the given year.

**Hash**

```
unsigned Hash() const;
```

Returns seconds since January 1, 1901.

**Hour**

```
HourTy Hour() const;
```

Returns the hour in local time.

**HourGMT**

```
HourTy HourGMT() const;
```

Returns the hour in Greenwich Mean Time.

**IsDST**

```
int IsDST() const;
```

Returns 1 if the time is in daylight savings time, 0 otherwise.

**IsValid**

```
int IsValid() const;
```

Returns 1 if this *TTime* object contains a valid time, 0 otherwise.

**Max**

```
TTime Max( const TTime& t ) const;
```

Returns either this *TTime* object or *t*, whichever is greater.

**Min**

```
TTime Min( const TTime& t ) const;
```

Returns either this *TTime* object or *t*, whichever is lesser.

**Minute**

```
MinuteTy Minute() const;
```

Returns the minute in local time.

**MinuteGMT**

```
MinuteTy MinuteGMT() const;
```

Returns the minute in Greenwich Mean Time.

**PrintDate**

```
static int PrintDate( int flag);
```

Set *flag* to 1 to print the date along with the time; set to 0 to not print the date. Returns the old setting.

**Second**

```
SecondTy Second() const;
```

Returns seconds.

**Seconds**

```
ClockTy Seconds() const;
```

Returns seconds since January 1, 1901.

## Protected member functions

**AssertDate**

```
static int AssertDate( const TDate& d );
```

Returns 1 if *d* is between the earliest valid date (*RefDate*) and the latest valid date (*MaxDate*).

## Protected data members

**RefDate**

```
static const TDate RefDate;
```

The minimum valid date for *TTime* objects: January 1, 1901.

**MaxDate**

```
static const TDate MaxDate;
```

The maximum valid date for *TTime* objects.

## Operators

**Operator <**

```
int operator <  ( const TTime& t ) const;
```

Returns 1 if the target time is less than time *t*, 0 otherwise.

**Operator <=**

```
int operator <= ( const TTime& t ) const;
```

Returns 1 if the target time is less than or equal to time *t*, 0 otherwise.

**Operator >**

```
int operator > ( const TTime& t ) const;
```

Returns 1 if the target time is greater than time *t*, 0 otherwise.

**Operator >=**

```
int operator >= ( const TTime& t ) const;
```

Returns 1 if the target time is greater than or equal to time *t*, 0 otherwise.

**Operator ==**

```
int operator == ( const TTime& t ) const;
```

Returns 1 if the target time is equal to time *t*, 0 otherwise.

**Operator !=**

```
int operator != ( const TTime& t ) const;
```

Returns 1 if the target time is not equal to time *t*, 0 otherwise.

**Operator ++**

```
void operator++();
```

Increments time by 1 second.

**Operator – –**

```
void operator--();
```

Decrements time by 1 second.

**Operator +=**

```
void operator+=(long s);
```

Adds *s* seconds to the time.

**Operator –=**

```
void operator-=(long s);
```

Subtracts *s* seconds from the time.

**Operator +**

```
friend TTime operator + ( const TTime& t, long s );
friend TTime operator + ( long s, const TTime& t );
```

Adds *s* seconds to time *t*.

**Operator –**

```
friend TTime operator - ( const TTime& t, long s );
friend TTime operator - ( long s, const TTime& t );
```

Performs subtraction, in seconds, between *s* and *t*.

**Operator <<**

```
friend ostream& operator << ( ostream& os, const TTime& t);
```

Inserts time *t* into output stream *os*.

```
friend opstream& operator << ( opstream& s, const TTime& d );
```

Inserts time *t* into persistent stream *s*.

**Operator >>**

```
friend ipstream& operator >> ( ipstream& s, TTime& d );
```

Extracts time *t* from persistent stream *s*.

# Index

## A

## B

## C

## D

## E

## F

## G

## H

## I

## J



## K



## L

## M

## N

## O



## P

## Q



## R

## S

## T

## U



## V

## W

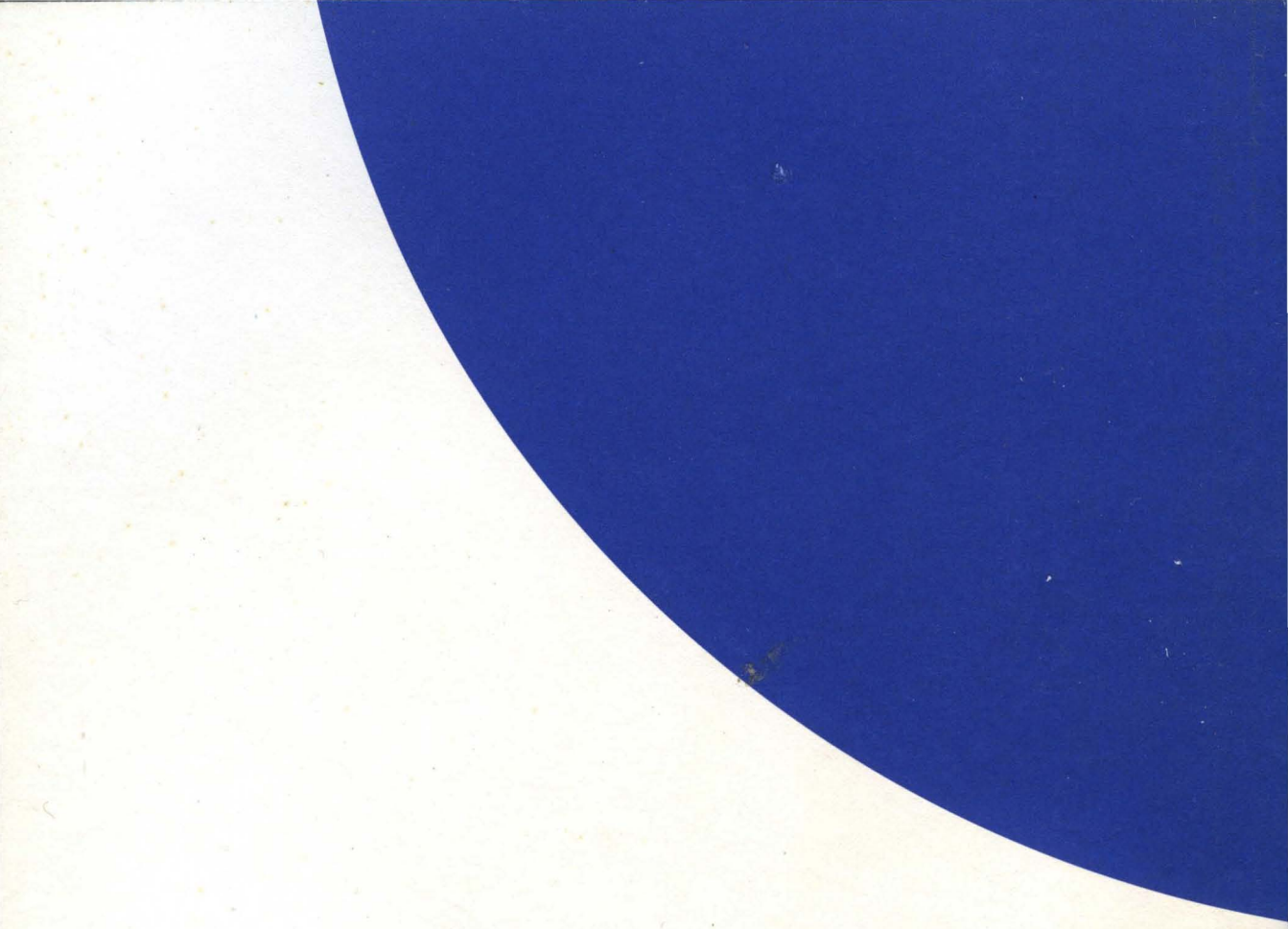


## X

## Y



## Z



**Borland**

Corporate Headquarters: 100 Borland Way, Scotts Valley, CA 95066-3249, (408) 431-1000. Offices in: Australia, Belgium, Canada, Denmark, France, Germany, Hong Kong, Italy, Japan, Korea, Latin America, Malaysia, Netherlands, New Zealand, Singapore, Spain, Sweden, Taiwan, and United Kingdom • Part # BCP1240WW21772 • BOR 6272