

Table of Contents

General C/C++.....	8
C/C++ Pre-processor Commands	9
#define	10
#error	11
#include	11
#line	11
#pragma	12
#if, #ifdef, #ifndef, #else, #elif, #endif	12
Predefined preprocessor variables	13
#, ##	14
#undef	14
C/C++ Keywords.....	15
asm	17
auto	17
bool	17
break	18
case	18
catch	18
char	19
class	19
const	20
const_cast	20
continue	20
default	21
delete	21
do	21
double	22
dynamic_cast	22
else	22
enum	23
explicit	24
export	24
extern	24
false	24
float	25
for	25
friend	26
goto.....	26
if	26
inline	27
int	27
long	27
mutable	28
namespace	28
new	29

operator	30
private	30
protected	31
public	31
register	31
reinterpret_cast	31
return	32
short	32
signed	32
sizeof	33
static	34
static_cast	34
struct	34
switch	35
template	36
this	36
throw	37
true	37
try	37
typedef	38
typeid	38
typename	38
union	39
unsigned	39
using	40
virtual	41
void	42
volatile	42
wchar_t	42
while	43
C++ Operator Precedence.....	44
C/C++ Data Types.....	46
Type Modifiers.....	46
Type Sizes and Ranges.....	47
ASCII Chart.....	48
Constant Escape Sequences.....	52
Standard C Library.....	53
Standard C I/O.....	54
clearerr	55
fclose	55
feof	55
ferror	56
fflush	56
fgetc	57
fgetpos	57
fgets	58
fopen	59

fprintf	60
fputc	60
fputs	61
fread	61
freopen	61
fscanf	62
fsetpos	63
ftell	63
fwrite	63
getc	64
getchar	64
gets	65
perror	65
printf	66
putc	68
putchar	68
puts	69
remove	69
rename	69
rewind	70
scanf	70
setbuf	72
setvbuf	72
sprintf	73
sscanf	74
tmpfile	74
tmpnam	74
ungetc	75
vprintf, vfprintf, and vsprintf	75
Standard C String and Character	76
atof	77
atoi	77
atol	78
isalnum	78
isalpha	79
isctrl	79
isdigit	80
isgraph	80
islower	81
isprint	81
ispunct	81
isspace	82
isupper	82
isxdigit	82
memchr	83
memcmp	83
memcpy	84

memmove	84
memset	85
strcat	86
strchr	86
strcmp	87
strcoll	87
strcpy	88
strncpy	88
strerror	88
strlen	89
strncat	89
strncmp	90
strncpy	90
strpbrk	91
strrchr	91
strstr	92
strtod	92
strtok	93
strtoul	94
strxfrm	94
tolower	94
Standard C Math	96
abs	97
acos	97
asin	98
atan	98
atan2	99
ceil	99
cos	100
cosh	100
div	101
exp	101
fabs	101
floor	102
fmod	102
frexp	102
labs	103
ldexp	103
ldiv	103
log	104
log10	104
modf	104
pow	105
sin	105
sinh	106
sqrt	106
tan	107

tanh	107
Standard C Date & Time	108
asctime	109
clock	109
ctime	110
difftime	110
gmtime	111
localtime	111
mktime	111
setlocale	112
time	114
Standard C Memory	115
calloc	116
free	116
malloc	117
realloc	117
Other Standard C Functions	118
abort	119
assert	119
atexit	119
bsearch	120
exit	120
getenv	121
longjmp	121
qsort	122
raise	123
rand	123
setjmp	124
signal	124
srand	126
system	126
va_arg	127
C++	129
C++ I/O	130
I/O Constructors	131
bad	132
clear	132
close	133
eof	133
fail	134
fill	134
flags	134
flush	135
gcount	135
get	136
getline	137
good	137

ignore	138
open	138
peek	139
precision	139
put	140
putback	140
rdstate	140
read	141
seekg	141
seekp	142
setf	142
sync_with_stdio	143
tellg	143
tellp	143
unsetf	144
width	144
write	145
C++ I/O Examples	146
Reading From Files.....	146
Checking For Errors.....	147
C++ Strings	148
String constructors	149
String operators	149
append	151
assign	152
at	153
begin	154
c_str	154
capacity	155
clear	155
compare	156
copy	157
data	157
empty	158
end	159
erase	160
find	161
find_first_not_of	162
find_first_of	163
find_last_not_of	164
find_last_of	165
getline	166
insert	167
length	167
max_size	168
push_back	168
rbegin	169

rend	169
replace	170
reserve	171
resize	171
rfind	172
size	172
substr	173
swap	174
C++ String Streams	175
String Stream Constructors	176
String Stream Operators	177
rdbuf	177
str	178

General C/C++

C/C++ Pre-processor Commands

<code>#, ##</code>	manipulate strings
<code>#define</code>	define variables
<code>#error</code>	display an error message
<code>#if, #ifdef, #ifndef, #else, #elif, #endif</code>	conditional operators
<code>#include</code>	insert the contents of another file
<code>#line</code>	set line and file information
<code>#pragma</code>	implementation specific command
<code>#undef</code>	used to undefine variables
Predefined preprocessor variables	miscellaneous preprocessor variables

#define

Syntax:

```
#define macro-name replacement-string
```

The `#define` command is used to make substitutions throughout the file in which it is located. In other words, `#define` causes the compiler to go through the file, replacing every occurrence of *macro-name* with *replacement-string*. The replacement string stops at the end of the line.

Here's a typical use for a `#define` (at least in C):

```
#define TRUE 1
#define FALSE 0
...
int done = 0;
while( done != TRUE ) {
    ...
}
```

Another feature of the `#define` command is that it can take arguments, making it rather useful as a pseudo-function creator. Consider the following code:

```
#define absolute_value( x ) ( ((x) < 0) ? -(x) : (x) )
...
int num = -1;
while( absolute_value( num ) ) {
    ...
}
```

It's generally a good idea to use extra parentheses when using complex macros. Notice that in the above example, the variable "x" is always within its own set of parentheses. This way, it will be evaluated in whole, before being compared to 0 or multiplied by -1. Also, the entire macro is surrounded by parentheses, to prevent it from being contaminated by other code. If you're not careful, you run the risk of having the compiler misinterpret your code.

Here is an example of how to use the `#define` command to create a general purpose incrementing for loop that prints out the integers 1 through 20:

```
#define count_up( v, low, high ) \
    for( (v) = (low); (v) <= (high); (v)++ )
...
int i;
count_up( i, 1, 20 ) {
    printf( "i is %d\n", i );
}
```

Related topics:

`#`, `##`
`#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, `#endif`
`#undef`

#error

Syntax:

```
#error message
```

The `#error` command simply causes the compiler to stop when it is encountered. When an `#error` is encountered, the compiler spits out the line number and whatever *message* is. This command is mostly used for debugging.

#include

Syntax:

```
#include <filename>  
#include "filename"
```

This command slurps in a file and inserts it at the current location. The main difference between the syntax of the two items is that if *filename* is enclosed in angled brackets, then the compiler searches for it somehow. If it is enclosed in quotes, then the compiler doesn't search very hard for the file.

While the behavior of these two searches is up to the compiler, usually the angled brackets means to search through the standard library directories, while the quotes indicate a search in the current directory. The spiffy new C++ `#include` commands don't need to map directly to filenames, at least not for the standard libraries. That's why you can get away with

```
#include <iostream>
```

and not have the compiler choke on you.

#line

Syntax:

```
#line line_number "filename"
```

The `#line` command is simply used to change the value of the `__LINE__` and `__FILE__` variables. The filename is optional. The `__LINE__` and `__FILE__` variables represent the current file and which line is being read. The command

```
#line 10 "main.cpp"
```

changes the current line number to 10, and the current file to "main.cpp".

#pragma

The `#pragma` command gives the programmer the ability to tell the compiler to do certain things. Since the `#pragma` command is implementation specific, uses vary from compiler to compiler. One option might be to trace program execution.

#if, #ifdef, #ifndef, #else, #elif, #endif

These commands give simple logic control to the compiler. As a file is being compiled, you can use these commands to cause certain lines of code to be included or not included.

```
#if expression
```

If the value of expression is true, then the code that immediately follows the command will be compiled.

```
#ifdef macro
```

If the *macro* has been defined by a `#define` statement, then the code immediately following the command will be compiled.

```
#ifndef macro
```

If the *macro* has not been defined by a `#define` statement, then the code immediately following the command will be compiled.

A few side notes: The command `#elif` is simply a horribly truncated way to say "elseif" and works like you think it would. You can also throw in a "defined" or "!defined" after an `#if` to get added functionality.

Example code:

Here's an example of all these:

```
#ifdef DEBUG
    cout << "This is the test version, i=" << i << endl;
#else
    cout << "This is the production version!" << endl;
#endif
```

You might notice how that second example could make debugging a lot easier than inserting and removing a million "cout"s in your code.

Related topics:

`#define`

Predefined preprocessor variables

Syntax:

```
__LINE__  
__FILE__  
__DATE__  
__TIME__  
__cplusplus  
__STDC__
```

The following variables can vary by compiler, but generally work:

- The `__LINE__` and `__FILE__` variables represent the current line and current file being processed.
 - The `__DATE__` variable contains the current date, in the form month/day/year. This is the date that the file was compiled, not necessarily the current date.
 - The `__TIME__` variable represents the current time, in the form hour:minute:second. This is the time that the file was compiled, not necessarily the current time.
 - The `__cplusplus` variable is only defined when compiling a C++ program. In some older compilers, this is also called `cplusplus`.
 - The `__STDC__` variable is defined when compiling a C program, and may also be defined when compiling C++.
-

#, ##

The `#` and `##` operators are used with the `#define` macro. Using `#` causes the first argument after the `#` to be returned as a string in quotes. Using `##` concatenates what's before the `##` with what's after it.

Example code:

For example, the command

```
#define to_string( s ) # s
```

will make the compiler turn this command

```
cout << to_string( Hello World! ) << endl;
```

into

```
cout << "Hello World!" << endl;
```

Here is an example of the `##` command:

```
#define concatenate( x, y ) x ## y  
...  
int xy = 10;  
...
```

This code will make the compiler turn

```
cout << concatenate( x, y ) << endl;
```

into

```
cout << xy << endl;
```

which will, of course, display '10' to standard output.

Related topics:

[#define](#)

#undef

The `#undef` command undefines a previously defined macro variable, such as a variable defined by a `#define`.

Related topics:

[#define](#)

C/C++ Keywords

asm	insert an assembly instruction
auto	declare a local variable
bool	declare a boolean variable
break	break out of a loop
case	a block of code in a switch statement
catch	handles exceptions from throw
char	declare a character variable
class	declare a class
const	declare immutable data or functions that do not change data
const_cast	cast from const variables
continue	bypass iterations of a loop
default	default handler in a case statement
delete	make memory available
do	looping construct
double	declare a double precision floating-point variable
dynamic_cast	perform runtime casts
else	alternate case for an if statement
enum	create enumeration types
explicit	only use constructors when they exactly match
export	allows template definitions to be separated from their declarations
extern	tell the compiler about variables defined elsewhere
false	the boolean value of false
float	declare a floating-point variable
for	looping construct
friend	grant non-member function access to private data
goto	jump to a different part of the program
if	execute code based off of the result of a test
inline	optimize calls to short functions
int	declare a integer variable
long	declare a long integer variable
mutable	override a const variable
namespace	partition the global namespace by defining a scope
new	allocate dynamic memory for a new variable
operator	create overloaded operator functions
private	declare private members of a class
protected	declare protected members of a class

public	declare public members of a class
register	request that a variable be optimized for speed
reinterpret_cast	change the type of a variable
return	return from a function
short	declare a short integer variable
signed	modify variable type declarations
sizeof	return the size of a variable or type
static	create permanent storage for a variable
static_cast	perform a nonpolymorphic cast
struct	define a new structure
switch	execute code based off of different possible values for a variable
template	create generic functions
this	a pointer to the current object
throw	throws an exception
true	the boolean value of true
try	execute code that can throw an exception
typedef	create a new type name from an existing type
typeid	describes an object
typename	declare a class or undefined type
union	a structure that assigns multiple variables to the same memory location
unsigned	declare an unsigned integer variable
using	import complete or partial namespaces into the current scope
virtual	create a function that can be overridden by a derived class
void	declare functions or data with no associated data type
volatile	warn the compiler about variables that can be modified unexpectedly
wchar_t	declare a wide-character variable
while	looping construct

asm

Syntax:

```
asm( "instruction" );
```

The `asm` command allows you to insert assembly language commands directly into your code. Various different compilers allow differing forms for this command, such as

```
asm {
    instruction-sequence
}
```

or

```
asm( instruction );
```

auto

The keyword `auto` is used to declare local variables, and is purely optional.

Related topics:

register

bool

The keyword `bool` is used to declare Boolean logic variables; that is, variables which can be either true or false.

For example, the following code declares a boolean variable called *done*, initializes it to false, and then loops until that variable is set to true.

```
bool done = false;
while( !done ) {
    ...
}
```

Also see the data types page.

Related topics:

<code>char</code>	<code>int</code>	<code>true</code>
<code>double</code>	<code>long</code>	<code>unsigned</code>
<code>false</code>	<code>short</code>	<code>wchar_t</code>
<code>float</code>	<code>signed</code>	

break

The `break` keyword is used to break out of a `do`, `for`, or `while` loop. It is also used to finish each clause of a `switch` statement, keeping the program from "falling through" to the next case in the code. An example:

```
while( x < 100 ) {
    if( x < 0 )
        break;
    cout << x << endl;
    x++;
}
```

A given `break` statement will break out of only the closest loop, no further. If you have a triply-nested `for` loop, for example, you might want to include extra logic or a `goto` statement to break out of the loop.

Related topics:

`continue`
`do`
`for`
`goto`
`switch`
`while`

case

The `case` keyword is used to test a variable against a certain value in a `switch` statement.

Related topics:

`default`
`switch`

catch

The `catch` statement handles exceptions generated by the `throw` statement.

Related topics:

`throw`
`try`

char

The `char` keyword is used to declare character variables. For more information about variable types, see the data types page.

Related topics:

bool
double
float
int
long
short
signed
unsigned
void
wchar_t

class

Syntax:

```
class class-name : inheritance-list {  
    private-members-list;  
    protected:  
    protected-members-list;  
    public:  
    public-members-list;  
} object-list;
```

The `class` keyword allows you to create new classes. *class-name* is the name of the class that you wish to create, and *inheritance-list* is an optional list of classes inherited by the new class. Members of the class are private by default, unless listed under either the protected or public labels. *object-list* can be used to immediately instantiate one or more instances of the class, and is also optional. For example:

```
class Date {  
    int Day;  
    int Month;  
    int Year;  
public:  
    void display();  
};
```

Related topics:

friend this
private typename
protected union
public virtual
struct

const

The `const` keyword can be used to tell the compiler that a certain variable should not be modified once it has been initialized.

It can also be used to declare functions of a class that do not alter any class data.

Related topics:

`const_cast`

`mutable`

const_cast

Syntax:

```
const_cast<type> (object);
```

The `const_cast` keyword can be used to remove the **const** or **volatile** property from some variable. The target data type must be the same as the source type, except (of course) that the target type doesn't have to be `const`.

Related topics:

`const`

`dynamic_cast`

`reinterpret_cast`

`static_cast`

continue

The `continue` statement can be used to bypass iterations of a given loop.

For example, the following code will display all of the numbers between 0 and 20 except 10:

```
for( int i = 0; i < 21; i++ ) {  
    if( i == 10 ) {  
        continue;  
    }  
    cout << i << " ";  
}
```

Related topics:

`break`

`do`

`for`

while

default

A default case in the switch statement.

Related topics:

case
switch

delete

Syntax:

```
delete p;  
delete[] pArray;
```

The delete operator frees the memory pointed to by *p*. The argument should have been previously allocated by a call to new. The second form of delete should be used to delete an array.

Related topics:

(Standard C Memory) free
(Standard C Memory) malloc
new

do

Syntax:

```
do {  
    statement-list;  
} while( condition );
```

The do construct evaluates the given *statement-list* repeatedly, until *condition* becomes false. Note that every do loop will evaluate its statement list at least once, because the terminating condition is tested at the end of the loop.

Related topics:

break
continue
for
while

double

The `double` keyword is used to declare double precision floating-point variables. Also see the data types page.

Related topics:

`bool`

`char`

`float`

`int`

`long`

`short`

`signed`

`unsigned`

`void`

`wchar_t`

dynamic_cast

Syntax:

```
dynamic_cast<type> (object);
```

The `dynamic_cast` keyword casts a datum from one type to another, performing a runtime check to ensure the validity of the cast. If you attempt to cast between incompatible types, the result of the cast will be **NULL**.

Related topics:

`const_cast`

`reinterpret_cast`

`static_cast`

else

The `else` keyword is used as an alternative case for the `if` statement.

Related topics:

`if`

enum

Syntax:

```
enum name {name-list} var-list;
```

The `enum` keyword is used to create an enumerated type named `name` that consists of the elements in *name-list*. The *var-list* argument is optional, and can be used to create instances of the type along with the declaration. For example, the following code creates an enumerated type for colors:

```
enum ColorT {red, orange, yellow, green, blue, indigo, violet};
...
ColorT c1 = indigo;
if( c1 == indigo ) {
    cout << "c1 is indigo" << endl;
}
```

In the above example, the effect of the enumeration is to introduce several new constants named *red*, *orange*, *yellow*, etc. By default, these constants are assigned consecutive integer values starting at zero. You can change the values of those constants, as shown by the next example:

```
enum ColorT { red = 10, blue = 15, green };
...
ColorT c = green;
cout << "c is " << c << endl;
```

When executed, the above code will display the following output:

```
c is 16
```

Note that the above examples will only work with C++ compilers. If you're working in regular C, you will need to specify the *enum* keyword whenever you create an instance of an enumerated type:

```
enum ColorT { red = 10, blue = 15, green };
...
enum ColorT c = green; // note the additional enum keyword
printf( "c is %d\n", c );
```

explicit

When a constructor is specified as explicit, no automatic conversion will be used with that constructor -- but parameters passed to the constructor may still be converted. For example:

```
struct foo {
    explicit foo( int a )
        : a_( a )
    { }
    int a_;
};
int bar( const foo & f ) {
    return f.a_;
}
bar( 1 ); // fails because an implicit conversion from int to foo
         // is forbidden by explicit.
bar( foo( 1 ) ); // works -- explicit call to explicit constructor.
bar( foo( 1.0 ) ); // works -- explicit call to explicit constructor
                  // with automatic conversion from float to int.
```

export

The export keyword is intended to allow definitions of C++ templates to be separated from their declarations. While officially part of the C++ standard, the export keyword is only supported by a few compilers (such as the Comeau C++ compiler) and is not supported by such mainstream compilers as GCC and Visual C++.

extern

The extern keyword is used to inform the compiler about variables declared outside of the current scope. Variables described by extern statements will not have any space allocated for them, as they should be properly defined elsewhere.

Extern statements are frequently used to allow data to span the scope of multiple files.

false

The Boolean value of "false".

Related topics:

bool
true

float

The `float` keyword is used to declare floating-point variables. Also see the data types page.

Related topics:

<code>bool</code>	<code>long</code>	<code>void</code>
<code>char</code>	<code>short</code>	<code>wchar_t</code>
<code>double</code>	<code>signed</code>	
<code>int</code>	<code>unsigned</code>	

for

Syntax:

```
for( initialization; test-condition; increment ) {
    statement-list;
}
```

The `for` construct is a general looping mechanism consisting of 4 parts:

1. the initialization, which consists of 0 or more comma-delimited variable initialization statements
2. the test-condition, which is evaluated to determine if the execution of the `for` loop will continue
3. the increment, which consists of 0 or more comma-delimited statements that increment variables
4. and the statement-list, which consists of 0 or more statements that will be executed each time the loop is executed.

For example:

```
for( int i = 0; i < 10; i++ ) {
    cout << "i is " << i << endl;
}
int j, k;
for( j = 0, k = 10;
     j < k;
     j++, k-- ) {
    cout << "j is " << j << " and k is " << k << endl;
}
for( ; ; ) {
    // loop forever!
}
```

Related topics:

<code>break</code>	<code>if</code>
<code>continue</code>	<code>while</code>
<code>do</code>	

inline

Syntax:

```
inline int functionA( int i ) {  
    ...  
}
```

The inline keyword requests that the compiler expand a given function in place, as opposed to inserting a call to that function. The inline keyword is a request, not a command, and the compiler is free to ignore it for whatever reason.

When a function declaration is included in a class definition, the compiler should try to automatically inline that function. No inline keyword is necessary in this case.

int

The int keyword is used to declare integer variables. Also see the data types page.

Related topics:

- bool
- char
- double
- float
- long
- short
- signed
- unsigned
- void
- wchar_t

long

The long keyword is a data type modifier that is used to declare long integer variables. For more information on long, see the data types page.

Related topics:

- bool
- char
- double
- float
- int
- short
- signed
- void

mutable

The mutable keyword overrides any enclosing const statement. A mutable member of a const object can be modified.

Related topics:

const

namespace

Syntax:

```
namespace name {  
  declaration-list;  
}
```

The namespace keyword allows you to create a new scope. The name is optional, and can be omitted to create an unnamed namespace. Once you create a namespace, you'll have to refer to it explicitly or use the using keyword.

Example code:

```
namespace CartoonNameSpace {  
  int HomersAge;  
  void incrementHomersAge() {  
    HomersAge++;  
  }  
}  
int main() {  
  ...  
  CartoonNameSpace::HomersAge = 39;  
  CartoonNameSpace::incrementHomersAge();  
  cout << CartoonNameSpace::HomersAge << endl;  
  ...  
}
```

Related topics:

using

new**Syntax:**

```

pointer = new type;
pointer = new type( initializer );
pointer = new type[size];
pointer = new( arg-list ) type...

```

The `new` operator (valid only in C++) allocates a new chunk of memory to hold a variable of type *type* and returns a pointer to that memory. An optional initializer can be used to initialize the memory. Allocating arrays can be accomplished by providing a *size* parameter in brackets.

The optional *arg-list* parameter can be used with any of the other formats to pass a variable number of arguments to an overloaded version of `new()`. For example, the following code shows how the `new()` function can be overloaded for a class and then passed arbitrary arguments:

```

class Base {
public:
    Base() { }
    void *operator new( unsigned int size, string str ) {
        cout << "Logging an allocation of " << size << " bytes for new
object '" << str << "'" << endl;
        return malloc( size );
    }
    int var;
    double var2;
};
...
Base* b = new ("Base instance 1") Base;

```

If an `int` is 4 bytes and a `double` is 8 bytes, the above code generates the following output when run:

```

Logging an allocation of 12 bytes for new object 'Base instance 1'

```

Related topics:

`delete`

(Standard C Memory) `free`

(Standard C Memory) `malloc`

operator

Syntax:

```
return-type class-name::operator#(parameter-list) {  
    ...  
}  
return-type operator#(parameter-list) {  
    ...  
}
```

The `operator` keyword is used to overload operators. The sharp sign (`#`) listed above in the syntax description represents the operator which will be overloaded. If part of a class, the *class-name* should be specified. For unary operators, *parameter-list* should be empty, and for binary operators, *parameter-list* should contain the operand on the right side of the operator (the operand on the left side is passed as `this`).

For the non-member operator overload function, the operand on the left side should be passed as the first parameter and the operand on the right side should be passed as the second parameter.

You cannot overload the `#`, `##`, `.`, `::`, `.*`, or `?` tokens.

Related topics:

[this](#)

private

Private data of a class can only be accessed by members of that class, except when `friend` is used. The `private` keyword can also be used to inherit a base class privately, which causes all public and protected members of the base class to become private members of the derived class.

Related topics:

[class](#)

[protected](#)

[public](#)

protected

Protected data are private to their own class but can be inherited by derived classes. The protected keyword can also be used as an inheritance specifier, which causes all public and protected members of the base class to become protected members of the derived class.

Related topics:

class
private
public

public

Public data in a class are accessible to everyone. The public keyword can also be used as an inheritance specifier, which causes all public and protected members of the base class to become public and protected members of the derived class.

Related topics:

class
private
protected

register

The register keyword requests that a variable be optimized for speed, and fell out of common use when computers became better at most code optimizations than humans.

Related topics:

auto

reinterpret_cast

Syntax:

```
reinterpret_cast<type> (object);
```

The reinterpret_cast operator changes one data type into another. It should be used to cast between incompatible pointer types.

Related topics:

const_cast
dynamic_cast
static_cast

return

Syntax:

```
return;  
return( value );
```

The return statement causes execution to jump from the current function to whatever function called the current function. An optional *value* can be returned. A function may have more than one return statement.

short

The short keyword is a data type modifier that is used to declare short integer variables. See the data types page.

Related topics:

bool
char
double
float
int
long
signed
unsigned
void
wchar_t

signed

The signed keyword is a data type modifier that is usually used to declare signed char variables. See the data types page.

Related topics:

bool
char
double
float
int
long
short
unsigned
void
wchar_t

sizeof

The `sizeof` operator is a compile-time operator that returns the size of the argument passed to it. The size is a multiple of the size of a `char`, which on many personal computers is 1 byte (or 8 bits). The number of bits in a `char` is stored in the `CHAR_BIT` constant defined in the `<climits>` header file.

For example, the following code uses `sizeof` to display the sizes of a number of variables:

```
struct EmployeeRecord {
    int ID;
    int age;
    double salary;
    EmployeeRecord* boss;
};
...
cout << "sizeof(int): " << sizeof(int) << endl
     << "sizeof(float): " << sizeof(float) << endl
     << "sizeof(double): " << sizeof(double) << endl
     << "sizeof(char): " << sizeof(char) << endl
     << "sizeof(EmployeeRecord): " << sizeof(EmployeeRecord) << endl;
int i;
float f;
double d;
char c;
EmployeeRecord er;
cout << "sizeof(i): " << sizeof(i) << endl
     << "sizeof(f): " << sizeof(f) << endl
     << "sizeof(d): " << sizeof(d) << endl
     << "sizeof(c): " << sizeof(c) << endl
     << "sizeof(er): " << sizeof(er) << endl;
```

On some machines, the above code displays this output:

```
sizeof(int): 4
sizeof(float): 4
sizeof(double): 8
sizeof(char): 1
sizeof(EmployeeRecord): 20
sizeof(i): 4
sizeof(f): 4
sizeof(d): 8
sizeof(c): 1
sizeof(er): 20
```

Note that `sizeof` can either take a variable type (such as **int**) or a variable name (such as **i** in the example above).

It is also important to note that the sizes of various types of variables can change depending on what system you're on. Check out a description of the C and C++ data types for more information.

The parentheses around the argument are not required if you are using `sizeof` with a variable type (e.g. `sizeof(int)`).

Related topics:
[C/C++ Data Types](#)

static

The static data type modifier is used to create permanent storage for variables. Static variables keep their value between function calls. When used in a class, all instantiations of that class share one copy of the variable.

static_cast

Syntax:

```
static_cast<type> (object);
```

The `static_cast` keyword can be used for any normal conversion between types. No runtime checks are performed.

Related topics:

`const_cast`

`dynamic_cast`

`reinterpret_cast`

struct

Syntax:

```
struct struct-name : inheritance-list {  
    public-members-list;  
    protected:  
    protected-members-list;  
    private:  
    private-members-list;  
} object-list;
```

Structs are like `classes`, except that by default members of a struct are public rather than private. In C, structs can only contain data and are not permitted to have inheritance lists. For example:

```
struct Date {  
    int Day;  
    int Month;  
    int Year;  
};
```

Related topics:

`class`

`union`

switch

Syntax:

```
switch( expression ) {  
    case A:  
        statement list;  
        break;  
    case B:  
        statement list;  
        break;  
    ...  
    case N:  
        statement list;  
        break;  
    default:  
        statement list;  
        break;  
}
```

The switch statement allows you to test an expression for many values, and is commonly used as a replacement for multiple if()...else if()...else if()... statements. break statements are required between each case statement, otherwise execution will "fall-through" to the next case statement. The default case is optional. If provided, it will match any case not explicitly covered by the preceding cases in the switch statement. For example:

```
char keystroke = getch();  
switch( keystroke ) {  
    case 'a':  
    case 'b':  
    case 'c':  
    case 'd':  
        KeyABCDPressed();  
        break;  
    case 'e':  
        KeyEPressed();  
        break;  
    default:  
        UnknownKeyPressed();  
        break;  
}
```

Related topics:

break
case
default
if

template

Syntax:

```
template <class data-type> return-type name( parameter-list ) {
    statement-list;
}
```

Templates are used to create generic functions and can operate on data without knowing the nature of that data. They accomplish this by using a placeholder data-type for which many other data types can be substituted.

Example code:

For example, the following code uses a template to define a generic swap function that can swap two variables of any type:

```
template<class X> void genericSwap( X &a, X &b ) {
    X tmp;
    tmp = a;
    a = b;
    b = tmp;
}
int main(void) {
    ...
    int num1 = 5;
    int num2 = 21;
    cout << "Before, num1 is " << num1 << " and num2 is " << num2 <<
endl;
    genericSwap( num1, num2 );
    cout << "After, num1 is " << num1 << " and num2 is " << num2 <<
endl;
    char c1 = 'a';
    char c2 = 'z';
    cout << "Before, c1 is " << c1 << " and c2 is " << c2 << endl;
    genericSwap( c1, c2 );
    cout << "After, c1 is " << c1 << " and c2 is " << c2 << endl;
    ...
    return( 0 );
}
```

Related topics:

[typename](#)

this

The `this` keyword is a pointer to the current object. All member functions of a class have a `this` pointer.

Related topics:

[class](#)

[operator](#)

throw

Syntax:

```
try {  
    statement list;  
}  
catch( typeA arg ) {  
    statement list;  
}  
catch( typeB arg ) {  
    statement list;  
}  
...  
catch( typeN arg ) {  
    statement list;  
}
```

The throw statement is part of the C++ mechanism for exception handling. This statement, together with the try and catch statements, the C++ exception handling system gives programmers an elegant mechanism for error recovery.

You will generally use a try block to execute potentially error-prone code. Somewhere in this code, a throw statement can be executed, which will cause execution to jump out of the try block and into one of the catch blocks. For example:

```
try {  
    cout << "Before throwing exception" << endl;  
    throw 42;  
    cout << "Shouldn't ever see this" << endl;  
}  
catch( int error ) {  
    cout << "Error: caught exception " << error << endl;  
}
```

Related topics:

catch
try

true

The Boolean value of "true".

Related topics:

bool
false

try

The try statement attempts to execute exception-generating code. See the throw statement for more details.

Related topics:

catch
throw

typedef

Syntax:

```
typedef existing-type new-type;
```

The typedef keyword allows you to create a new type from an existing type.

typeid

Syntax:

```
typeid( object );
```

The typeid operator returns a reference to a `type_info` object that describes `object`.

typename

The typename keyword can be used to describe an undefined type or in place of the class keyword in a template declaration.

Related topics:

class

template

union

Syntax:

```
union union-name {  
    public-members-list;  
    private:  
    private-members-list;  
} object-list;
```

A union is like a class, except that all members of a union share the same memory location and are by default public rather than private. For example:

```
union Data {  
    int i;  
    char c;  
};
```

Related topics:

class
struct

unsigned

The unsigned keyword is a data type modifier that is usually used to declare unsigned int variables. See the data types page.

Related topics:

bool
char
double
float
int
short
signed
void
wchar_t

using

The `using` keyword is used to import a namespace (or parts of a namespace) into the current scope.

Example code:

For example, the following code imports the entire *std* namespace into the current scope so that items within that namespace can be used without a preceding "std::".

```
using namespace std;
```

Alternatively, the next code snippet just imports a single element of the *std* namespace into the current namespace:

```
using std::cout;
```

Related topics:
[namespace](#)

virtual**Syntax:**

```
virtual return-type name( parameter-list );
virtual return-type name( parameter-list ) = 0;
```

The `virtual` keyword can be used to create virtual functions, which can be overridden by derived classes.

- A virtual function indicates that a function can be overridden in a subclass, and that the overridden function will actually be used.
- When a base object pointer points to a derived object that contains a virtual function, the decision about which version of that function to call is based on the type of object pointed to by the pointer, and this process happens at runtime.
- A base object can point to different derived objects and have different versions of the virtual function run.

If the function is specified as a pure virtual function (denoted by the `= 0`), it must be overridden by a derived class.

For example, the following code snippet shows how a child class can override a virtual method of its parent, and how a non-virtual method in the parent cannot be overridden:

```
class Base {
public:
    void nonVirtualFunc() {
        cout << "Base: non-virtual function" << endl;
    }
    virtual void virtualFunc() {
        cout << "Base: virtual function" << endl;
    }
};
class Child : public Base {
public:
    void nonVirtualFunc() {
        cout << "Child: non-virtual function" << endl;
    }
    void virtualFunc() {
        cout << "Child: virtual function" << endl;
    }
};
int main() {
    Base* basePointer = new Child();
    basePointer->nonVirtualFunc();
    basePointer->virtualFunc();
    return 0;
}
```

When run, the above code displays:

```
Base: non-virtual function
Child: virtual function
```

Related topics:

[class](#)

void

The void keyword is used to denote functions that return no value, or generic variables which can point to any type of data. Void can also be used to declare an empty parameter list. Also see the data types page.

Related topics:

char
double
float
int
long
short
signed
unsigned
wchar_t

volatile

The volatile keyword is an implementation-dependent modifier, used when declaring variables, which prevents the compiler from optimizing those variables. Volatile should be used with variables whose value can change in unexpected ways (i.e. through an interrupt), which could conflict with optimizations that the compiler might perform.

wchar_t

The keyword wchar_t is used to declare wide character variables. Also see the data types page.

Related topics:

bool
char
double
float
int
short
signed
unsigned
void

while

Syntax:

```
while( condition ) {  
    statement-list;  
}
```

The `while` keyword is used as a looping construct that will evaluate the *statement-list* as long as *condition* is true. Note that if the *condition* starts off as false, the *statement-list* will never be executed. (You can use a `do` loop to guarantee that the *statement-list* will be executed at least once.) For example:

```
bool done = false;  
while( !done ) {  
    ProcessData();  
    if( StopLooping() ) {  
        done = true;  
    }  
}
```

Related topics:

`break`

`continue`

`do`

`for`

`if`

C++ Operator Precedence

The operators at the top of this list are evaluated first.

Precedence	Operator	Description	Example	Associativity
1	::	Scoping operator	Class::age = 2;	none
	()	Grouping operator	(a + b) / 4;	
	[]	Array access	array[4] = 2;	
	ptr->	Member access from a pointer	ptr->age = 34;	
2	.	Member access from an object	obj.age = 34;	left to right
	++	Post-increment	for(i = 0; i < 10; i++) ...	
	--	Post-decrement	for(i = 10; i > 0; i--) ...	
	!	Logical negation	if(!done) ...	
	~	Bitwise complement	flags = ~flags;	
	++	Pre-increment	for(i = 0; i < 10; ++i) ...	
	--	Pre-decrement	for(i = 10; i > 0; --i) ...	
	-	Unary minus	int i = -1;	
	+	Unary plus	int i = +1;	
	*	Dereference	data = *ptr;	
3	&	Address of	address = &obj;	right to left
	(type)	Cast to a given type	int i = (int) floatNum;	
	sizeof	Return size in bytes	int size = sizeof(floatNum);	
4	->*	Member pointer selector	ptr->*var = 24;	left to right
	.*	Member object selector	obj.*var = 24;	
	*	Multiplication	int i = 2 * 4;	
5	/	Division	float f = 10 / 3;	left to right
	%	Modulus	int rem = 4 % 3;	
6	+	Addition	int i = 2 + 3;	left to right
	-	Subtraction	int i = 5 - 1;	
7	<<	Bitwise shift left	int flags = 33 << 1;	left to right
	>>	Bitwise shift right	int flags = 33 >> 1;	
	<	Comparison less-than	if(i < 42) ...	left to right
8	<=	Comparison less-than-or-equal-to	if(i <= 42) ...	
	>	Comparison greater-than	if(i > 42) ...	
	>=	Comparison greater-than-or-equal-to	if(i >= 42) ...	
9	==	Comparison equal-to	if(i == 42) ...	left to right
	!=	Comparison not-equal-to	if(i != 42) ...	
10	&	Bitwise AND	flags = flags & 42;	left to right

11	<code>^</code>	Bitwise exclusive OR	<code>flags = flags ^ 42;</code>	left to right
12	<code> </code>	Bitwise inclusive (normal) OR	<code>flags = flags 42;</code>	left to right
13	<code>&&</code>	Logical AND	<code>if(conditionA && conditionB) ...</code>	left to right
14	<code> </code>	Logical OR	<code>if(conditionA conditionB) ...</code>	left to right
15	<code>? :</code>	Ternary conditional (if-then-else)	<code>int i = (a > b) ? a : b;</code>	right to left
		Assignment operator		
	<code>=</code>	Increment and assign		
	<code>+=</code>	Decrement and assign	<code>int a = b;</code>	
	<code>-=</code>	Multiply and assign	<code>a += 3;</code>	
	<code>*=</code>	Divide and assign	<code>b -= 4;</code>	
	<code>/=</code>	Modulo and assign	<code>a *= 5;</code>	
16	<code>%=</code>	Bitwise AND and assign	<code>a /= 2;</code>	
	<code>&=</code>	Bitwise exclusive OR and assign	<code>a %= 3;</code>	right to left
	<code>^=</code>	Bitwise inclusive (normal) OR and assign	<code>flags &= new_flags;</code>	
	<code> =</code>	Bitwise shift left and assign	<code>flags ^= new_flags;</code>	
	<code><<=</code>	Bitwise shift right and assign	<code>flags = new_flags;</code>	
	<code>>>=</code>	Sequential evaluation operator	<code>flags <<= 2;</code>	
17	<code>,</code>	Bitwise shift right and assign	<code>flags >>= 2;</code>	
		Sequential evaluation operator	<code>for(i = 0, j = 0; i < 10; i++, j++) ...</code>	left to right

It is important to note that **there is no specified precedence** for the operation of changing a variable into a value. For example, consider the following code:

```
float x, result;
x = 1;
result = x / ++x;
```

The value of `result` is not guaranteed to be consistent across different compilers, because it is not clear whether the computer should change the variable `x` (the one that occurs on the left side of the division operator) before using it. Depending on which compiler you are using, the variable `result` can either be **1.0** or **0.5**. The bottom line is that you **should not use the same variable multiple times in a single expression when using operators with side effects**.

C/C++ Data Types

There are five data types for C: **void**, **int**, **float**, **double**, and **char**.

Type	Description
void	associated with no data type
int	integer
float	floating-point number
double	double precision floating-point number
char	character

C++ defines two more: **bool** and **wchar_t**.

Type	Description
bool	Boolean value, true or false
wchar_t	wide character

Type Modifiers

Several of these types can be modified using **signed**, **unsigned**, **short**, and **long**. When one of these type modifiers is used by itself, a data type of **int** is assumed. A complete list of possible data types follows:

bool
char
unsigned char
signed char
int
unsigned int
signed int
short int
unsigned short int
signed short int
long int
signed long int
unsigned long int
float
double
long double
wchar_t

Type Sizes and Ranges

The size and range of any data type is compiler and architecture dependent. The "cfloat" (or "float.h") header file often defines minimum and maximum values for the various data types. You can use the sizeof operator to determine the size of any data type, in bytes. However, many architectures implement data types of a standard size. **ints** and **floats** are often 32-bit, **chars** 8-bit, and **doubles** are usually 64-bit. **bools** are often implemented as 8-bit data types.

ASCII Chart

The following chart contains ASCII decimal, octal, hexadecimal and character codes for values from 0 to 127.

Decimal	Octal	Hex	Character	Description
0	0	00	NUL	
1	1	01	SOH	start of header
2	2	02	STX	start of text
3	3	03	ETX	end of text
4	4	04	EOT	end of transmission
5	5	05	ENQ	enquiry
6	6	06	ACK	acknowledge
7	7	07	BEL	bell
8	10	08	BS	backspace
9	11	09	HT	horizontal tab
10	12	0A	LF	line feed
11	13	0B	VT	vertical tab
12	14	0C	FF	form feed
13	15	0D	CR	carriage return
14	16	0E	SO	shift out
15	17	0F	SI	shift in
16	20	10	DLE	data link escape
17	21	11	DC1	no assignment, but usually XON
18	22	12	DC2	
19	23	13	DC3	no assignment, but usually XOFF
20	24	14	DC4	
21	25	15	NAK	negative acknowledge
22	26	16	SYN	synchronous idle
23	27	17	ETB	end of transmission block
24	30	18	CAN	cancel
25	31	19	EM	end of medium
26	32	1A	SUB	substitute
27	33	1B	ESC	escape
28	34	1C	FS	file separator
29	35	1D	GS	group separator
30	36	1E	RS	record separator
31	37	1F	US	unit separator
32	40	20	SPC	space

33	41	21	!	
34	42	22	"	
35	43	23	#	
36	44	24	\$	
37	45	25	%	
38	46	26	&	
39	47	27	'	
40	50	28	(
41	51	29)	
42	52	2A	*	
43	53	2B	+	
44	54	2C	,	
45	55	2D	-	
46	56	2E	.	
47	57	2F	/	
48	60	30	0	
49	61	31	1	
50	62	32	2	
51	63	33	3	
52	64	34	4	
53	65	35	5	
54	66	36	6	
55	67	37	7	
56	70	38	8	
57	71	39	9	
58	72	3A	:	
59	73	3B	;	
60	74	3C	<	
61	75	3D	=	
62	76	3E	>	
63	77	3F	?	
64	100	40	@	
65	101	41	A	
66	102	42	B	
67	103	43	C	
68	104	44	D	
69	105	45	E	
70	106	46	F	
71	107	47	G	

72	110	48	H	
73	111	49	I	
74	112	4A	J	
75	113	4B	K	
76	114	4C	L	
77	115	4D	M	
78	116	4E	N	
79	117	4F	O	
80	120	50	P	
81	121	51	Q	
82	122	52	R	
83	123	53	S	
84	124	54	T	
85	125	55	U	
86	126	56	V	
87	127	57	W	
88	130	58	X	
89	131	59	Y	
90	132	5A	Z	
91	133	5B	[
92	134	5C	\	
93	135	5D]	
94	136	5E	^	
95	137	5F	_	
96	140	60	`	
97	141	61	a	
98	142	62	b	
99	143	63	c	
100	144	64	d	
101	145	65	e	
102	146	66	f	
103	147	67	g	
104	150	68	h	
105	151	69	i	
106	152	6A	j	
107	153	6B	k	
108	154	6C	l	
109	155	6D	m	
110	156	6E	n	

111	157	6F	o	
112	160	70	p	
113	161	71	q	
114	162	72	r	
115	163	73	s	
116	164	74	t	
117	165	75	u	
118	166	76	v	
119	167	77	w	
120	170	78	x	
121	171	79	y	
122	172	7A	z	
123	173	7B	{	
124	174	7C		
125	175	7D	}	
126	176	7E	~	
127	177	7F	DEL	delete

Constant Escape Sequences

The following escape sequences can be used to define certain special characters within strings:

Escape Sequence	Description
\'	Single quote
\"	Double quote
\\	Backslash
\nnn	Octal number (nnn)
\0	Null character (really just the octal number zero)
\a	Audible bell
\b	Backspace
\f	Formfeed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\xnnn	Hexadecimal number (nnn)

An example of this is contained in the following code:

```
printf( "This\nis\na\ntest\n\nShe said, \"How are you?\"\n" );
```

which would display

```
This
is
a
test
She said, "How are you?"
```

Standard C Library

Standard C I/O

clearerr	clears errors
fclose	close a file
feof	true if at the end-of-file
ferror	checks for a file error
fflush	writes the contents of the output buffer
fgetc	get a character from a stream
fgetpos	get the file position indicator
fgets	get a string of characters from a stream
fopen	open a file
fprintf	print formatted output to a file
fputc	write a character to a file
fputs	write a string to a file
fread	read from a file
freopen	open an existing stream with a different name
fscanf	read formatted input from a file
fseek	move to a specific location in a file
fsetpos	move to a specific location in a file
ftell	returns the current file position indicator
fwrite	write to a file
getc	read a character from a file
getchar	read a character from stdin
gets	read a string from stdin
perror	displays a string version of the current error to stderr
printf	write formatted output to stdout
putc	write a character to a stream
putchar	write a character to stdout
puts	write a string to stdout
remove	erase a file
rename	rename a file
rewind	move the file position indicator to the beginning of a file
scanf	read formatted input from stdin
setbuf	set the buffer for a specific stream
setvbuf	set the buffer and size for a specific stream
sprintf	write formatted output to a buffer
sscanf	read formatted input from a buffer
tmpfile	return a pointer to a temporary file
tmpnam	return a unique filename
ungetc	puts a character back into a stream
vprintf, vfprintf, and vsprintf	write formatted output with variable argument lists

clearerr

Syntax:

```
#include <stdio.h>
void clearerr( FILE *stream );
```

The `clearerr` function resets the error flags and **EOF** indicator for the given *stream*. When an error occurs, you can use `perror()` to figure out which error actually occurred.

Related topics:

`feof`

`ferror`

`perror`

fclose

Syntax:

```
#include <stdio.h>
int fclose( FILE *stream );
```

The function `fclose()` closes the given file stream, deallocating any buffers associated with that stream. `fclose()` returns 0 upon success, and **EOF** otherwise.

Related topics:

`fflush`

`fopen`

`freopen`

`setbuf`

feof

Syntax:

```
#include <stdio.h>
int feof( FILE *stream );
```

The function `feof()` returns a nonzero value if the end of the given file *stream* has been reached.

Related topics:

`clearerr`

`perror`

`ferror`

`putc`

`getc`

error

Syntax:

```
#include <stdio.h>
int error( FILE *stream );
```

The `error()` function looks for errors with *stream*, returning zero if no errors have occurred, and non-zero if there is an error. In case of an error, use `perror()` to determine which error has occurred.

Related topics:

`clearerr`

`feof`

`perror`

fflush

Syntax:

```
#include <stdio.h>
int fflush( FILE *stream );
```

If the given file *stream* is an output stream, then `fflush()` causes the output buffer to be written to the file. If the given *stream* is of the input type, then `fflush()` causes the input buffer to be cleared. `fflush()` is useful when debugging, if a program segfaults before it has a chance to write output to the screen. Calling `fflush(stdout)` directly after debugging output will ensure that your output is displayed at the correct time.

```
printf( "Before first call\n" );
fflush( stdout );
shady_function();
printf( "Before second call\n" );
fflush( stdout );
dangerous_dereference();
```

Related topics:

`fclose`

`fopen`

`fread`

`fwrite`

`getc`

`putc`

fgetc

Syntax:

```
#include <stdio.h>
int fgetc( FILE *stream );
```

The `fgetc()` function returns the next character from *stream*, or **EOF** if the end of file is reached or if there is an error.

Related topics:

- fopen
- fputc
- fread
- fwrite
- getc
- getchar
- gets
- putc

fgetpos

Syntax:

```
#include <stdio.h>
int fgetpos( FILE *stream, fpos_t *position );
```

The `fgetpos()` function stores the file position indicator of the given file *stream* in the given *position* variable. The position variable is of type `fpos_t` (which is defined in `stdio.h`) and is an object that can hold every possible position in a `FILE`. `fgetpos()` returns zero upon success, and a non-zero value upon failure.

Related topics:

- fseek
- fsetpos
- ftell

fgets

Syntax:

```
#include <stdio.h>
char *fgets( char *str, int num, FILE *stream );
```

The function `fgets()` reads up to *num* - 1 characters from the given file *stream* and dumps them into *str*. The string that `fgets()` produces is always **NULL**-terminated. `fgets()` will stop when it reaches the end of a line, in which case *str* will contain that newline character. Otherwise, `fgets()` will stop when it reaches *num* - 1 characters or encounters the **EOF** character. `fgets()` returns *str* on success, and **NULL** on an error.

Related topics:

fputs
fscanf
gets
scanf

fopen

Syntax:

```
#include <stdio.h>
FILE *fopen( const char *fname, const char *mode );
```

The `fopen()` function opens a file indicated by *fname* and returns a stream associated with that file. If there is an error, `fopen()` returns **NULL**. *mode* is used to determine how the file will be treated (i.e. for input, output, etc)

Mode	Meaning
"r"	Open a text file for reading
"w"	Create a text file for writing
"a"	Append to a text file
"rb"	Open a binary file for reading
"wb"	Create a binary file for writing
"ab"	Append to a binary file
"r+"	Open a text file for read/write
"w+"	Create a text file for read/write
"a+"	Open a text file for read/write
"rb+"	Open a binary file for read/write
"wb+"	Create a binary file for read/write
"ab+"	Open a binary file for read/write

An example:

```
int ch;
FILE *input = fopen( "stuff", "r" );
ch = getc( input );
```

Related topics:

fclose
fflush
fgetc
fputc
fread
freopen
fseek
fwrite
getc
getchar
setbuf

fprintf

Syntax:

```
#include <stdio.h>
int fprintf( FILE *stream, const char *format, ... );
```

The `fprintf()` function sends information (the arguments) according to the specified *format* to the file indicated by *stream*. `fprintf()` works just like `printf()` as far as the format goes. The return value of `fprintf()` is the number of characters outputted, or a negative number if an error occurs. An example:

```
char name[20] = "Mary";
FILE *out;
out = fopen( "output.txt", "w" );
if( out != NULL )
    fprintf( out, "Hello %s\n", name );
```

Related topics:

- fputc
- fputs
- fscanf
- printf
- sprintf

fputc

Syntax:

```
#include <stdio.h>
int fputc( int ch, FILE *stream );
```

The function `fputc()` writes the given character *ch* to the given output *stream*. The return value is the character, unless there is an error, in which case the return value is **EOF**.

Related topics:

- fgetc
- fopen
- fprintf
- fread
- fwrite
- getc
- getchar
- putc

fputs

Syntax:

```
#include <stdio.h>
int fputs( const char *str, FILE *stream );
```

The `fputs()` function writes an array of characters pointed to by *str* to the given output *stream*. The return value is non-negative on success, and **EOF** on failure.

Related topics:

<code>fgets</code>	<code>gets</code>
<code>fprintf</code>	<code>puts</code>
<code>fscanf</code>	

fread

Syntax:

```
#include <stdio.h>
int fread( void *buffer, size_t size, size_t num, FILE *stream );
```

The function `fread()` reads *num* number of objects (where each object is *size* bytes) and places them into the array pointed to by *buffer*. The data comes from the given input *stream*. The return value of the function is the number of things read. You can use `feof()` or `ferror()` to figure out if an error occurs.

Related topics:

<code>fflush</code>	<code>fputc</code>	<code>getc</code>
<code>fgetc</code>	<code>fscanf</code>	
<code>fopen</code>	<code>fwrite</code>	

freopen

Syntax:

```
#include <stdio.h>
FILE *freopen( const char *fname, const char *mode, FILE *stream );
```

The `freopen()` function is used to reassign an existing *stream* to a different file and mode. After a call to this function, the given file *stream* will refer to *fname* with access given by *mode*. The return value of `freopen()` is the new stream, or **NULL** if there is an error.

Related topics:

<code>fclose</code>	<code>fopen</code>
---------------------	--------------------

fscanf

Syntax:

```
#include <stdio.h>
int fscanf( FILE *stream, const char *format, ... );
```

The function `fscanf()` reads data from the given file *stream* in a manner exactly like `scanf()`. The return value of `fscanf()` is the number of variables that are actually assigned values, or **EOF** if no assignments could be made.

Related topics:

fgets
 fprintf
 fputs
 fread
 fwrite
 scanf
 sscanf

fseek

Syntax:

```
#include <stdio.h>
int fseek( FILE *stream, long offset, int origin );
```

The function `fseek()` sets the file position data for the given *stream*. The origin value should have one of the following values (defined in `stdio.h`):

Name	Explanation
SEEK_SET	Seek from the start of the file
SEEK_CUR	Seek from the current location
SEEK_END	Seek from the end of the file

`fseek()` returns zero upon success, non-zero on failure. You can use `fseek()` to move beyond a file, but not before the beginning. Using `fseek()` clears the **EOF** flag associated with that stream.

Related topics:

fgetpos
 fopen
 fsetpos
 ftell
 rewind

fsetpos

Syntax:

```
#include <stdio.h>
int fsetpos( FILE *stream, const fpos_t *position );
```

The `fsetpos()` function moves the file position indicator for the given *stream* to a location specified by the *position* object. `fpos_t` is defined in `stdio.h`. The return value for `fsetpos()` is zero upon success, non-zero on failure.

Related topics:

fgetpos
fseek
ftell

ftell

Syntax:

```
#include <stdio.h>
long ftell( FILE *stream );
```

The `ftell()` function returns the current file position for *stream*, or -1 if an error occurs.

Related topics:

fgetpos
fseek
fsetpos

fwrite

Syntax:

```
#include <stdio.h>
int fwrite( const void *buffer, size_t size, size_t count, FILE
*stream );
```

The `fwrite()` function writes, from the array *buffer*, *count* objects of size *size* to *stream*. The return value is the number of objects written.

Related topics:

fflush	fputc	getc
fgetc	fread	
fopen	fscanf	

getc

Syntax:

```
#include <stdio.h>
int getc( FILE *stream );
```

The `getc()` function returns the next character from *stream*, or **EOF** if the end of file is reached. `getc()` is identical to `fgetc()`. For example:

```
int ch;
FILE *input = fopen( "stuff", "r" );
ch = getc( input );
while( ch != EOF ) {
    printf( "%c", ch );
    ch = getc( input );
}
```

Related topics:

- feof
- fflush
- fgetc
- fopen
- fputc
- fread
- fwrite
- putc
- ungetc

getchar

Syntax:

```
#include <stdio.h>
int getchar( void );
```

The `getchar()` function returns the next character from **stdin**, or **EOF** if the end of file is reached.

Related topics:

- fgetc
- fopen
- fputc
- putc

gets

Syntax:

```
#include <stdio.h>
char *gets( char *str );
```

The `gets()` function reads characters from **stdin** and loads them into *str*, until a newline or **EOF** is reached. The newline character is translated into a null termination. The return value of `gets()` is the read-in string, or **NULL** if there is an error.

Note that `gets()` does not perform bounds checking, and thus risks overrunning *str*. For a similar (and safer) function that includes bounds checking, see `fgets()`.

Related topics:

[fgets](#)
[fgets](#)
[fputs](#)
[puts](#)

perror

Syntax:

```
#include <stdio.h>
void perror( const char *str );
```

The `perror()` function prints *str* and an implementation-defined error message corresponding to the global variable *errno*. For example:

```
char* input_filename = "not_found.txt";
FILE* input = fopen( input_filename, "r" );
if( input == NULL ) {
    char error_msg[255];
    sprintf( error_msg, "Error opening file '%s'", input_filename );
    perror( error_msg );
    exit( -1 );
}
```

The the file called *not_found.txt* is not found, this code will produce the following output:

```
Error opening file 'not_found.txt': No such file or directory
```

Related topics:

[clearerr](#)
[feof](#)
[ferror](#)

printf

Syntax:

```
#include <stdio.h>
int printf( const char *format, ... );
```

The `printf()` function prints output to **stdout**, according to *format* and other arguments passed to `printf()`. The string *format* consists of two types of items - characters that will be printed to the screen, and format commands that define how the other arguments to `printf()` are displayed. Basically, you specify a format string that has text in it, as well as "special" characters that map to the other arguments of `printf()`. For example, this code

```
char name[20] = "Bob";
int age = 21;
printf( "Hello %s, you are %d years old\n", name, age );
```

displays the following output:

```
Hello Bob, you are 21 years old
```

The `%s` means, "insert the first argument, a string, right here." The `%d` indicates that the second argument (an integer) should be placed there. There are different `%`-codes for different variable types, as well as options to limit the length of the variables and whatnot.

Code	Format
<code>%c</code>	character
<code>%d</code>	signed integers
<code>%i</code>	signed integers
<code>%e</code>	scientific notation, with a lowercase "e"
<code>%E</code>	scientific notation, with a uppercase "E"
<code>%f</code>	floating point
<code>%g</code>	use <code>%e</code> or <code>%f</code> , whichever is shorter
<code>%G</code>	use <code>%E</code> or <code>%f</code> , whichever is shorter
<code>%o</code>	octal
<code>%s</code>	a string of characters
<code>%u</code>	unsigned integer
<code>%x</code>	unsigned hexadecimal, with lowercase letters
<code>%X</code>	unsigned hexadecimal, with uppercase letters
<code>%p</code>	a pointer
<code>%n</code>	the argument shall be a pointer to an integer into which is placed the number of characters written so far
<code>%%</code>	a <code>'%'</code> sign

An integer placed between a `%` sign and the format command acts as a minimum field width specifier, and pads the output with spaces or zeros to make it long enough. If you want to pad with zeros, place a zero before the minimum field width specifier:

```
%012d
```

You can also include a precision modifier, in the form of a `.N` where `N` is some number, before the format command:

```
%012.4d
```

The precision modifier has different meanings depending on the format command being used:

- With `%e`, `%E`, and `%f`, the precision modifier lets you specify the number of decimal places desired. For example, `%12.6f` will display a floating number at least 12 digits wide, with six decimal places.
- With `%g` and `%G`, the precision modifier determines the maximum number of significant digits displayed.
- With `%s`, the precision modifier simply acts as a maximum field length, to complement the minimum field length that precedes the period.

All of `printf()`'s output is right-justified, unless you place a minus sign right after the `%` sign. For example,

```
%-12.4f
```

will display a floating point number with a minimum of 12 characters, 4 decimal places, and left justified. You may modify the `%d`, `%i`, `%o`, `%u`, and `%x` type specifiers with the letter `l` and the letter `h` to specify long and short data types (e.g. `%hd` means a short integer). The `%e`, `%f`, and `%g` type specifiers can have the letter `l` before them to indicate that a double follows. The `%g`, `%f`, and `%e` type specifiers can be preceded with the character `#` to ensure that the decimal point will be present, even if there are no decimal digits. The use of the `#` character with the `%x` type specifier indicates that the hexadecimal number should be printed with the `'0x'` prefix. The use of the `#` character with the `%o` type specifier indicates that the octal value should be displayed with a `0` prefix.

Inserting a plus sign `'+'` into the type specifier will force positive values to be preceded by a `'+'` sign. Putting a space character `' '` there will force positive values to be preceded by a single space character.

You can also include constant escape sequences in the output string.

The return value of `printf()` is the number of characters printed, or a negative number if an error occurred.

Related topics:

`fprintf`
`puts`
`scanf`
`sprintf`

putc

Syntax:

```
#include <stdio.h>
int putc( int ch, FILE *stream );
```

The `putc()` function writes the character *ch* to *stream*. The return value is the character written, or **EOF** if there is an error. For example:

```
int ch;
FILE *input, *output;
input = fopen( "tmp.c", "r" );
output = fopen( "tmpCopy.c", "w" );
ch = getc( input );
while( ch != EOF ) {
    putc( ch, output );
    ch = getc( input );
}
fclose( input );
fclose( output );
```

generates a copy of the file `tmp.c` called `tmpCopy.c`.

Related topics:

<code>feof</code>	<code>fputc</code>	<code>putc</code>
<code>fflush</code>	<code>getc</code>	<code>puts</code>
<code>fgetc</code>	<code>getchar</code>	

putchar

Syntax:

```
#include <stdio.h>
int putchar( int ch );
```

The `putchar()` function writes *ch* to **stdout**. The code

```
putchar( ch );
```

is the same as

```
putc( ch, stdout );
```

The return value of `putchar()` is the written character, or **EOF** if there is an error.

Related topics:

`putc`

puts

Syntax:

```
#include <stdio.h>
int puts( char *str );
```

The function `puts()` writes *str* to **stdout**. `puts()` returns non-negative on success, or **EOF** on failure.

Related topics:

`fputs`
`gets`
`printf`
`putc`

remove

Syntax:

```
#include <stdio.h>
int remove( const char *fname );
```

The `remove()` function erases the file specified by *fname*. The return value of `remove()` is zero upon success, and non-zero if there is an error.

Related topics:

`rename`

rename

Syntax:

```
#include <stdio.h>
int rename( const char *oldfname, const char *newfname );
```

The function `rename()` changes the name of the file *oldfname* to *newfname*. The return value of `rename()` is zero upon success, non-zero on error.

Related topics:

`remove`

rewind

Syntax:

```
#include <stdio.h>
void rewind( FILE *stream );
```

The function `rewind()` moves the file position indicator to the beginning of the specified *stream*, also clearing the error and **EOF** flags associated with that stream.

Related topics:

fseek

scanf

Syntax:

```
#include <stdio.h>
int scanf( const char *format, ... );
```

The `scanf()` function reads input from **stdin**, according to the given *format*, and stores the data in the other arguments. It works a lot like `printf()`. The *format* string consists of control characters, whitespace characters, and non-whitespace characters. The control characters are preceded by a % sign, and are as follows:

Control Character	Explanation
%c	a single character
%d	a decimal integer
%i	an integer
%e, %f, %g	a floating-point number
%lf	a double
%o	an octal number
%s	a string
%x	a hexadecimal number
%p	a pointer
%n	an integer equal to the number of characters read so far
%u	an unsigned integer
%[]	a set of characters
%%	a percent sign

`scanf()` reads the input, matching the characters from *format*. When a control character is read, it puts the value in the next variable. Whitespace (tabs, spaces, etc) are skipped.

Non-whitespace characters are matched to the input, then discarded. If a number comes between the % sign and the control character, then only that many characters will be converted into the variable. If scanf() encounters a set of characters, denoted by the %[] control character, then any characters found within the brackets are read into the variable. The return value of scanf() is the number of variables that were successfully assigned values, or **EOF** if there is an error.

Example code:

This code snippet uses scanf() to read an int, float, and a double from the user. Note that the variable arguments to scanf() are passed in by address, as denoted by the ampersand (&) preceding each variable:

```
int i;
float f;
double d;
printf( "Enter an integer: " );
scanf( "%d", &i );
printf( "Enter a float: " );
scanf( "%f", &f );
printf( "Enter a double: " );
scanf( "%lf", &d );
printf( "You entered %d, %f, and %f\n", i, f, d );
```

Related topics:

fgets

fscanf

printf

sscanf

setbuf

Syntax:

```
#include <stdio.h>
void setbuf( FILE *stream, char *buffer );
```

The `setbuf()` function sets *stream* to use *buffer*, or, if *buffer* is null, turns off buffering. If a non-standard buffer size is used, it should be BUFSIZ characters long.

Related topics:

`fclose`

`fopen`

`setvbuf`

setvbuf

Syntax:

```
#include <stdio.h>
int setvbuf( FILE *stream, char *buffer, int mode, size_t size );
```

The function `setvbuf()` sets the buffer for *stream* to be *buffer*, with a size of *size*. *mode* can be:

- `_IOFBF`, which indicates full buffering
- `_IOLBF`, which means line buffering
- `_IONBF`, which means no buffering

Related topics:

`setbuf`

sprintf

Syntax:

```
#include <stdio.h>
int sprintf( char *buffer, const char *format, ... );
```

The `sprintf()` function is just like `printf()`, except that the output is sent to *buffer*. The return value is the number of characters written. For example:

```
char string[50];
int file_number = 0;
sprintf( string, "file.%d", file_number );
file_number++;
output_file = fopen( string, "w" );
```

Note that `sprintf()` does the opposite of a function like `atoi()` -- where `atoi()` converts a string into a number, `sprintf()` can be used to convert a number into a string.

For example, the following code uses `sprintf()` to convert an integer into a string of characters:

```
char result[100];
int num = 24;
sprintf( result, "%d", num );
```

This code is similar, except that it converts a floating-point number into an array of characters:

```
char result[100];
float fnum = 3.14159;
sprintf( result, "%f", fnum );
```

Related topics:

(Standard C String and Character) `atof`

(Standard C String and Character) `atoi`

(Standard C String and Character) `atol`

`fprintf`

`printf`

sscanf

Syntax:

```
#include <stdio.h>
int sscanf( const char *buffer, const char *format, ... );
```

The function `sscanf()` is just like `scanf()`, except that the input is read from *buffer*.

Related topics:

[fscanf](#)

[scanf](#)

tmpfile

Syntax:

```
#include <stdio.h>
FILE *tmpfile( void );
```

The function `tmpfile()` opens a temporary file with an unique filename and returns a pointer to that file. If there is an error, null is returned.

Related topics:

[tmpnam](#)

tmpnam

Syntax:

```
#include <stdio.h>
char *tmpnam( char *name );
```

The `tmpnam()` function creates an unique filename and stores it in *name*. `tmpnam()` can be called up to **TMP_MAX** times.

Related topics:

[tmpfile](#)

ungetc

Syntax:

```
#include <stdio.h>
int ungetc( int ch, FILE *stream );
```

The function `ungetc()` puts the character *ch* back in *stream*.

Related topics:

`getc`

(C++ I/O) `putback`

vprintf, vfprintf, and vsprintf

Syntax:

```
#include <stdarg.h>
#include <stdio.h>
int vprintf( char *format, va_list arg_ptr );
int vfprintf( FILE *stream, const char *format, va_list arg_ptr );
int vsprintf( char *buffer, char *format, va_list arg_ptr );
```

These functions are very much like `printf()`, `fprintf()`, and `sprintf()`. The difference is that the argument list is a pointer to a list of arguments. **va_list** is defined in `stdarg.h`, and is also used by (Other Standard C Functions) `va_arg()`. For example:

```
void error( char *fmt, ... ) {
    va_list args;
    va_start( args, fmt );
    fprintf( stderr, "Error: " );
    vfprintf( stderr, fmt, args );
    fprintf( stderr, "\n" );
    va_end( args );
    exit( 1 );
}
```

Standard C String and Character

atof	converts a string to a double
atoi	converts a string to an integer
atol	converts a string to a long
isalnum	true if a character is alphanumeric
isalpha	true if a character is alphabetic
iscntrl	true if a character is a control character
isdigit	true if a character is a digit
isgraph	true if a character is a graphical character
islower	true if a character is lowercase
isprint	true if a character is a printing character
ispunct	true if a character is punctuation
isspace	true if a character is a space character
isupper	true if a character is an uppercase character
isxdigit	true if a character is a hexadecimal character
memchr	searches an array for the first occurrence of a character
memcmp	compares two buffers
memcpy	copies one buffer to another
memmove	moves one buffer to another
memset	fills a buffer with a character
strcat	concatenates two strings
strchr	finds the first occurrence of a character in a string
strcmp	compares two strings
strcoll	compares two strings in accordance to the current locale
strcpy	copies one string to another
strcspn	searches one string for any characters in another
strerror	returns a text version of a given error code
strlen	returns the length of a given string
strncat	concatenates a certain amount of characters of two strings
strncmp	compares a certain amount of characters of two strings
strncpy	copies a certain amount of characters from one string to another
strpbrk	finds the first location of any character in one string, in another string
strrchr	finds the last occurrence of a character in a string
strspn	returns the length of a substring of characters of a string
strstr	finds the first occurrence of a substring of characters
strtod	converts a string to a double
strtok	finds the next token in a string
strtol	converts a string to a long
strtoul	converts a string to an unsigned long
strxfrm	converts a substring so that it can be used by string comparison functions
tolower	converts a character to lowercase
toupper	converts a character to uppercase

atof

Syntax:

```
#include <stdlib.h>
double atof( const char *str );
```

The function `atof()` converts *str* into a double, then returns that value. *str* must start with a valid number, but can be terminated with any non-numerical character, other than "E" or "e". For example,

```
x = atof( "42.0is_the_answer" );
```

results in *x* being set to 42.0.

Related topics:

<code>atoi</code>	(Standard C I/O) <code>sprintf</code>
<code>atol</code>	<code>strtod</code>

atoi

Syntax:

```
#include <stdlib.h>
int atoi( const char *str );
```

The `atoi()` function converts *str* into an integer, and returns that integer. *str* should start with whitespace or some sort of number, and `atoi()` will stop reading from *str* as soon as a non-numerical character has been read. For example:

```
int i;
i = atoi( "512" );
i = atoi( "512.035" );
i = atoi( " 512.035" );
i = atoi( " 512+34" );
i = atoi( " 512 bottles of beer on the wall" );
```

All five of the above assignments to the variable *i* would result in it being set to 512.

If the conversion cannot be performed, then `atoi()` will return zero:

```
int i = atoi( " does not work: 512" ); // results in i == 0
```

You can use `sprintf()` to convert a number into a string.

Related topics:

<code>atof</code>	<code>atol</code>	(Standard C I/O) <code>sprintf</code>
-------------------	-------------------	---------------------------------------

atol

Syntax:

```
#include <stdlib.h>
long atol( const char *str );
```

The function `atol()` converts *str* into a long, then returns that value. `atol()` will read from *str* until it finds any character that should not be in a long. The resulting truncated value is then converted and returned. For example,

```
x = atol( "1024.0001" );
```

results in `x` being set to 1024L.

Related topics:

[atof](#)

[atoi](#)

(Standard C I/O) [sprintf](#)

[strtol](#)

isalnum

Syntax:

```
#include <ctype.h>
int isalnum( int ch );
```

The function `isalnum()` returns non-zero if its argument is a numeric digit or a letter of the alphabet. Otherwise, zero is returned.

```
char c;
scanf( "%c", &c );
if( isalnum(c) )
    printf( "You entered the alphanumeric character %c\n", c );
```

Related topics:

[isalpha](#)

[isctrl](#)

[isdigit](#)

[isgraph](#)

[isprint](#)

[ispunct](#)

[isspace](#)

[isxdigit](#)

isalpha

Syntax:

```
#include <ctype.h>
int isalpha( int ch );
```

The function `isalpha()` returns non-zero if its argument is a letter of the alphabet. Otherwise, zero is returned.

```
char c;
scanf( "%c", &c );
if( isalpha(c) )
    printf( "You entered a letter of the alphabet\n" );
```

Related topics:

isalnum
isctrl
isdigit
isgraph
isprint
ispunct
isspace
isxdigit

isctrl

Syntax:

```
#include <ctype.h>
int isctrl( int ch );
```

The `isctrl()` function returns non-zero if its argument is a control character (between 0 and 0x1F or equal to 0x7F). Otherwise, zero is returned.

Related topics:

isalnum
isalpha
isdigit
isgraph
isprint
ispunct
isspace
isxdigit

isdigit

Syntax:

```
#include <ctype.h>
int isdigit( int ch );
```

The function `isdigit()` returns non-zero if its argument is a digit between 0 and 9. Otherwise, zero is returned.

```
char c;
scanf( "%c", &c );
if( isdigit(c) )
    printf( "You entered the digit %c\n", c );
```

Related topics:

isalnum
isalpha
isctrl
isgraph
isprint
ispunct
isspace
isxdigit

isgraph

Syntax:

```
#include <ctype.h>
int isgraph( int ch );
```

The function `isgraph()` returns non-zero if its argument is any printable character other than a space (if you can see the character, then `isgraph()` will return a non-zero value). Otherwise, zero is returned.

Related topics:

isalnum
isalpha
isctrl
isdigit
isprint
ispunct
isspace
isxdigit

islower

Syntax:

```
#include <ctype.h>
int islower( int ch );
```

The `islower()` function returns non-zero if its argument is a lowercase letter. Otherwise, zero is returned.

Related topics:

isupper

isprint

Syntax:

```
#include <ctype.h>
int isprint( int ch );
```

The function `isprint()` returns non-zero if its argument is a printable character (including a space). Otherwise, zero is returned.

Related topics:

isalnum

isdigit

isspace

isalpha

isgraph

isctrl

ispunct

ispunct

Syntax:

```
#include <ctype.h>
int ispunct( int ch );
```

The `ispunct()` function returns non-zero if its argument is a printing character but neither alphanumeric nor a space. Otherwise, zero is returned.

Related topics:

isalnum

isdigit

isspace

isalpha

isgraph

isxdigit

isctrl

isprint

isspace

Syntax:

```
#include <ctype.h>
int isspace( int ch );
```

The `isspace()` function returns non-zero if its argument is some sort of space (i.e. single space, tab, vertical tab, form feed, carriage return, or newline). Otherwise, zero is returned.

Related topics:

isalnum	isdigit	ispunct
isalpha	isgraph	isxdigit
isctrl	isprint	

isupper

Syntax:

```
#include <ctype.h>
int isupper( int ch );
```

The `isupper()` function returns non-zero if its argument is an uppercase letter. Otherwise, zero is returned.

Related topics:

islower
tolower

isxdigit

Syntax:

```
#include <ctype.h>
int isxdigit( int ch );
```

The function `isxdigit()` returns non-zero if its argument is a hexadecimal digit (i.e. A-F, a-f, or 0-9). Otherwise, zero is returned.

Related topics:

isalnum	isdigit	isspace
isalpha	isgraph	
isctrl	isprint	

memchr

Syntax:

```
#include <string.h>
void *memchr( const void *buffer, int ch, size_t count );
```

The `memchr()` function looks for the first occurrence of *ch* within *count* characters in the array pointed to by *buffer*. The return value points to the location of the first occurrence of *ch*, or **NULL** if *ch* isn't found. For example:

```
char names[] = "Alan Bob Chris X Dave";
if( memchr( names, 'X', strlen( names ) ) == NULL )
    printf( "Didn't find an X\n" );
else
    printf( "Found an X\n" );
```

Related topics:

memcmp
memcpy
strstr

memcmp

Syntax:

```
#include <string.h>
int memcmp( const void *buffer1, const void *buffer2, size_t count );
```

The function `memcmp()` compares the first *count* characters of *buffer1* and *buffer2*. The return values are as follows:

Value	Explanation
less than 0	buffer1 is less than buffer2
equal to 0	buffer1 is equal to buffer2
greater than 0	buffer1 is greater than buffer2

Related topics:

memchr
memcpy
memset
strcmp

memcpy

Syntax:

```
#include <string.h>
void *memcpy( void *to, const void *from, size_t count );
```

The function `memcpy()` copies *count* characters from the array *from* to the array *to*. The return value of `memcpy()` is *to*. The behavior of `memcpy()` is undefined if *to* and *from* overlap.

Related topics:

`memchr`

`memcmp`

`memmove`

`memset`

`strcpy`

`strlen`

`strncpy`

memmove

Syntax:

```
#include <string.h>
void *memmove( void *to, const void *from, size_t count );
```

The `memmove()` function is identical to `memcpy()`, except that it works even if *to* and *from* overlap.

Related topics:

`memcpy`

`memset`

memset

Syntax:

```
#include <string.h>
void* memset( void* buffer, int ch, size_t count );
```

The function `memset()` copies *ch* into the first *count* characters of *buffer*, and returns *buffer*. `memset()` is useful for initializing a section of memory to some value. For example, this command:

```
memset( the_array, '\0', sizeof(the_array) );
```

...is a very efficient way to set all values of `the_array` to zero.

The table below compares two different methods for initializing an array of characters: a for-loop versus `memset()`. As the size of the data being initialized increases, `memset()` clearly gets the job done much more quickly:

Input size	Initialized with a for-loop	Initialized with memset()
1000	0.016	0.017
10000	0.055	0.013
100000	0.443	0.029
1000000	4.337	0.291

Related topics:

`memcmp`

`memcpy`

`memmove`

strcat

Syntax:

```
#include <string.h>
char *strcat( char *str1, const char *str2 );
```

The `strcat()` function concatenates *str2* onto the end of *str1*, and returns *str1*. For example:

```
printf( "Enter your name: " );
scanf( "%s", name );
title = strcat( name, " the Great" );
printf( "Hello, %s\n", title );
```

Note that `strcat()` does not perform bounds checking, and thus risks overrunning *str1* or *str2*. For a similar (and safer) function that includes bounds checking, see `strncat()`.

Related topics:

- `strchr`
- `strcmp`
- `strcpy`
- `strncat`

Another set of related (but non-standard) functions are `strlcpy` and `strlcat`.

strchr

Syntax:

```
#include <string.h>
char *strchr( const char *str, int ch );
```

The function `strchr()` returns a pointer to the first occurrence of *ch* in *str*, or **NULL** if *ch* is not found.

Related topics:

- `strcat`
- `strcmp`
- `strcpy`
- `strlen`
- `strncat`
- `strncmp`
- `strncpy`
- `strpbrk`
- `strspn`
- `strstr`
- `strtok`

strcmp

Syntax:

```
#include <string.h>
int strcmp( const char *str1, const char *str2 );
```

The function `strcmp()` compares *str1* and *str2*, then returns:

Return value	Explanation
less than 0	"str1" is less than "str2"
equal to 0	"str1" is equal to "str2"
greater than 0	"str1" is greater than "str2"

For example:

```
printf( "Enter your name: " );
scanf( "%s", name );
if( strcmp( name, "Mary" ) == 0 ) {
    printf( "Hello, Dr. Mary!\n" );
}
```

Note that if *str1* or *str2* are missing a null-termination character, then `strcmp()` may not produce valid results. For a similar (and safer) function that includes explicit bounds checking, see `strncmp()`.

Related topics:

memcmp	strcoll	strncmp
strcat	strcpy	strxfrm
strchr	strlen	

strcoll

Syntax:

```
#include <string.h>
int strcoll( const char *str1, const char *str2 );
```

The `strcoll()` function compares *str1* and *str2*, much like `strcmp()`. However, `strcoll()` performs the comparison using the locale specified by the (Standard C Date & Time) `setlocale()` function.

Related topics:

(Standard C Date & Time) `setlocale`
`strcmp`
`strxfrm`

strcpy

Syntax:

```
#include <string.h>
char *strcpy( char *to, const char *from );
```

The `strcpy()` function copies characters in the string *from* to the string *to*, including the null termination. The return value is *to*.

Note that `strcpy()` does not perform bounds checking, and thus risks overrunning *from* or *to*. For a similar (and safer) function that includes bounds checking, see `strncpy()`.

Related topics:

<code>memcpy</code>	<code>strcmp</code>
<code>strcat</code>	<code>strncmp</code>
<code>strchr</code>	<code>strncpy</code>

Another set of related (but non-standard) functions are `strncpy` and `strlcat`.

strcspn

Syntax:

```
#include <string.h>
size_t strcspn( const char *str1, const char *str2 );
```

The function `strcspn()` returns the index of the first character in *str1* that matches any of the characters in *str2*.

Related topics:

`strpbrk`
`strchr`
`strstr`
`strtok`

strerror

Syntax:

```
#include <string.h>
char *strerror( int num );
```

The function `strerror()` returns an implementation defined string corresponding to *num*.

strlen

Syntax:

```
#include <string.h>
size_t strlen( char *str );
```

The `strlen()` function returns the length of *str* (determined by the number of characters before null termination).

Related topics:

`memcpy`

`strchr`

`strcmp`

`strncmp`

strncat

Syntax:

```
#include <string.h>
char *strncat( char *str1, const char *str2, size_t count );
```

The function `strncat()` concatenates at most *count* characters of *str2* onto *str1*, adding a null termination. The resulting string is returned.

Related topics:

`strcat`

`strchr`

`strncmp`

`strncpy`

Another set of related (but non-standard) functions are `strlcpy` and `strlcat`.

strncmp

Syntax:

```
#include <string.h>
int strncmp( const char *str1, const char *str2, size_t count );
```

The `strncmp()` function compares at most *count* characters of *str1* and *str2*. The return value is as follows:

Return value	Explanation
less than 0	"str1" is less than "str2"
equal to 0	"str1" is equal to "str2"
greater than 0	"str1" is greater than str2"

If there are less than *count* characters in either string, then the comparison will stop after the first null termination is encountered.

Related topics:

[strchr](#)
[strcmp](#)
[strcpy](#)
[strlen](#)
[strncat](#)
[strncpy](#)

strncpy

Syntax:

```
#include <string.h>
char *strncpy( char *to, const char *from, size_t count );
```

The `strncpy()` function copies at most *count* characters of *from* to the string *to*. If *from* has less than *count* characters, the remainder is padded with `'\0'` characters. The return value is the resulting string.

Related topics:

[memcpy](#)
[strchr](#)
[strcpy](#)
[strncat](#)
[strncmp](#)

Another set of related (but non-standard) functions are `strncpy` and `strlcat`.

strpbrk

Syntax:

```
#include <string.h>
char* strpbrk( const char* str1, const char* str2 );
```

The function `strpbrk()` returns a pointer to the first occurrence in *str1* of any character in *str2*, or **NULL** if no such characters are present.

Related topics:

(C++ Algorithms) <code>find_first_of</code>	<code>strspn</code>
<code>strchr</code>	<code>strstr</code>
<code>strcspn</code>	<code>strtok</code>
<code>strrchr</code>	

strrchr

Syntax:

```
#include <string.h>
char *strrchr( const char *str, int ch );
```

The function `strrchr()` returns a pointer to the last occurrence of *ch* in *str*, or **NULL** if no match is found.

Related topics:

<code>strcspn</code>	<code>strstr</code>
<code>strpbrk</code>	<code>strtok</code>
<code>strspn</code>	

strspn

Syntax:

```
#include <string.h>
size_t strspn( const char *str1, const char *str2 );
```

The `strspn()` function returns the index of the first character in *str1* that doesn't match any character in *str2*.

Related topics:

<code>strchr</code>	<code>strstr</code>
<code>strpbrk</code>	<code>strtok</code>
<code>strrchr</code>	

strstr

Syntax:

```
#include <string.h>
char *strstr( const char *str1, const char *str2 );
```

The function `strstr()` returns a pointer to the first occurrence of *str2* in *str1*, or **NULL** if no match is found. If the length of *str2* is zero, then `strstr()` will simply return *str1*.

For example, the following code checks for the existence of one string within another string:

```
char* str1 = "this is a string of characters";
char* str2 = "a string";
char* result = strstr( str1, str2 );
if( result == NULL ) printf( "Could not find '%s' in '%s'\n", str2,
str1 );
else printf( "Found a substring: '%s'\n", result );
```

When run, the above code displays this output:

```
Found a substring: 'a string of characters'
```

Related topics:

memchr
strchr
strcspn
strpbrk
strchr
strspn
strtok

strtod

Syntax:

```
#include <stdlib.h>
double strtod( const char *start, char **end );
```

The function `strtod()` returns whatever it encounters first in *start* as a double. *end* is set to point at whatever is left in *start* after that double. If overflow occurs, `strtod()` returns either **HUGE_VAL** or **-HUGE_VAL**.

Related topics:

atof

strtok

Syntax:

```
#include <string.h>
char *strtok( char *str1, const char *str2 );
```

The `strtok()` function returns a pointer to the next "token" in *str1*, where *str2* contains the delimiters that determine the token. `strtok()` returns **NULL** if no token is found. In order to convert a string to tokens, the first call to `strtok()` should have *str1* point to the string to be tokenized. All calls after this should have *str1* be **NULL**.

For example:

```
char str[] = "now # is the time for all # good men to come to the #
aid of their country";
char delims[] = "#";
char *result = NULL;
result = strtok( str, delims );
while( result != NULL ) {
    printf( "result is \"%s\"\n", result );
    result = strtok( NULL, delims );
}
```

The above code will display the following output:

```
result is "now "
result is " is the time for all "
result is " good men to come to the "
result is " aid of their country"
```

Related topics:

strchr

strpbrk

strspn

strcspn

strchr

strstr

strtol

Syntax:

```
#include <stdlib.h>
long strtol( const char *start, char **end, int base );
```

The `strtol()` function returns whatever it encounters first in *start* as a long, doing the conversion to *base* if necessary. *end* is set to point to whatever is left in *start* after the long. If the result can not be represented by a long, then `strtol()` returns either **LONG_MAX** or **LONG_MIN**. Zero is returned upon error.

Related topics:

atol

strtoul

strtoul

Syntax:

```
#include <stdlib.h>
unsigned long strtoul( const char *start, char **end, int base );
```

The function `strtoul()` behaves exactly like `strtol()`, except that it returns an unsigned long rather than a mere long.

Related topics:

`strtol`

strxfrm

Syntax:

```
#include <string.h>
size_t strxfrm( char *str1, const char *str2, size_t num );
```

The `strxfrm()` function manipulates the first *num* characters of *str2* and stores them in *str1*. The result is such that if a `strcoll()` is performed on *str1* and the old *str2*, you will get the same result as with a `strcmp()`.

Related topics:

`strcmp`

`strcoll`

tolower

Syntax:

```
#include <ctype.h>
int tolower( int ch );
```

The function `tolower()` returns the lowercase version of the character *ch*.

Related topics:

`isupper`

`toupper`

toupper

Syntax:

```
#include <ctype.h>
int toupper( int ch );
```

The `toupper()` function returns the uppercase version of the character *ch*.

Related topics:

`tolower`

Standard C Math

abs	absolute value
acos	arc cosine
asin	arc sine
atan	arc tangent
atan2	arc tangent, using signs to determine quadrants
ceil	the smallest integer not less than a certain value
cos	cosine
cosh	hyperbolic cosine
div	returns the quotient and remainder of a division
exp	returns "e" raised to a given power
fabs	absolute value for floating-point numbers
floor	returns the largest integer not greater than a given value
fmod	returns the remainder of a division
frexp	decomposes a number into scientific notation
labs	absolute value for long integers
ldexp	computes a number in scientific notation
ldiv	returns the quotient and remainder of a division, in long integer form
log	natural logarithm (to base e)
log10	common logarithm (to base 10)
modf	decomposes a number into integer and fractional parts
pow	returns a given number raised to another number
sin	sine
sinh	hyperbolic sine
sqrt	square root
tan	tangent
tanh	hyperbolic tangent

abs

Syntax:

```
#include <stdlib.h>
int abs( int num );
```

The `abs()` function returns the absolute value of *num*. For example:

```
int magic_number = 10;
cout << "Enter a guess: ";
cin >> x;
cout << "Your guess was " << abs( magic_number - x ) << " away from the magic number"
```

Related topics:

[fabs](#)

[labs](#)

acos

Syntax:

```
#include <math.h>
double acos( double arg );
```

The `acos()` function returns the arc cosine of *arg*, which will be in the range $[0, \pi]$. *arg* should be between -1 and 1. If *arg* is outside this range, `acos()` returns NAN and raises a floating-point exception.

Related topics:

[asin](#)

[atan](#)

[atan2](#)

[cos](#)

[cosh](#)

[sin](#)

[sinh](#)

[tan](#)

[tanh](#)

asin

Syntax:

```
#include <math.h>
double asin( double arg );
```

The `asin()` function returns the arc sine of *arg*, which will be in the range $[-\pi/2, +\pi/2]$. *arg* should be between -1 and 1. If *arg* is outside this range, `asin()` returns NAN and raises a floating-point exception.

Related topics:

acos
atan
atan2
cos
cosh
sin
sinh
tan
tanh

atan

Syntax:

```
#include <math.h>
double atan( double arg );
```

The function `atan()` returns the arc tangent of *arg*, which will be in the range $[-\pi/2, +\pi/2]$.

Related topics:

acos
asin
atan2
cos
cosh
sin
sinh
tan
tanh

atan2

Syntax:

```
#include <math.h>
double atan2( double y, double x );
```

The `atan2()` function computes the arc tangent of y/x , using the signs of the arguments to compute the quadrant of the return value.

Note the order of the arguments passed to this function.

Related topics:

acos
asin
atan
cos
cosh
sin
sinh
tan
tanh

ceil

Syntax:

```
#include <math.h>
double ceil( double num );
```

The `ceil()` function returns the smallest integer no less than *num*. For example,

```
y = 6.04;
x = ceil( y );
```

would set *x* to 7.0.

Related topics:

floor
fmod

cos

Syntax:

```
#include <math.h>
double cos( double arg );
```

The `cos()` function returns the cosine of *arg*, where *arg* is expressed in radians. The return value of `cos()` is in the range $[-1,1]$. If *arg* is infinite, `cos()` will return NAN and raise a floating-point exception.

Related topics:

acos

asin

atan

atan2

cosh

sin

sinh

tan

tanh

cosh

Syntax:

```
#include <math.h>
double cosh( double arg );
```

The function `cosh()` returns the hyperbolic cosine of *arg*.

Related topics:

acos

asin

atan

atan2

cos

sin

sinh

tan

tanh

div

Syntax:

```
#include <stdlib.h>


```

The function `div()` returns the quotient and remainder of the operation *numerator* / *denominator*. The `div_t` structure is defined in `stdlib.h`, and has at least:

```
int quot;    // The quotient
int rem;     // The remainder
```

For example, the following code displays the quotient and remainder of `x/y`:

```
div_t temp;
temp = div( x, y );
printf( "%d divided by %d yields %d with a remainder of %d\n",
        x, y, temp.quot, temp.rem );
```

Related topics:

ldiv

exp

Syntax:

```
#include <math.h>
double exp( double arg );
```

The `exp()` function returns e (2.7182818) raised to the *arg*th power.

Related topics:

log

pow

sqrt

fabs

Syntax:

```
#include <math.h>
double fabs( double arg );
```

The function `fabs()` returns the absolute value of *arg*.

Related topics:

abs

fmod

labs

floor

Syntax:

```
#include <math.h>
double floor( double arg );
```

The function `floor()` returns the largest integer not greater than *arg*. For example,

```
y = 6.04;
x = floor( y );
```

would result in `x` being set to 6.0.

Related topics:

ceil
fmod

fmod

Syntax:

```
#include <math.h>
double fmod( double x, double y );
```

The `fmod()` function returns the remainder of x/y .

Related topics:

ceil
fabs
floor

frexp

Syntax:

```
#include <math.h>
double frexp( double num, int* exp );
```

The function `frexp()` is used to decompose *num* into two parts: a mantissa between 0.5 and 1 (returned by the function) and an exponent returned as *exp*. Scientific notation works like this:

```
num = mantissa * (2 ^ exp)
```

Related topics:

ldexp
modf

labs

Syntax:

```
#include <stdlib.h>
long labs( long num );
```

The function `labs()` returns the absolute value of *num*.

Related topics:

[abs](#)

[fabs](#)

ldexp

Syntax:

```
#include <math.h>
double ldexp( double num, int exp );
```

The `ldexp()` function returns $num * (2 ^ exp)$. And get this: if an overflow occurs, **HUGE_VAL** is returned.

Related topics:

[frexp](#)

[modf](#)

ldiv

Syntax:

```
#include <stdlib.h>
ldiv_t ldiv( long numerator, long denominator );
```

Testing: [adiv_t](#), [div_t](#), [ldiv_t](#).

The `ldiv()` function returns the quotient and remainder of the operation *numerator / denominator*. The `ldiv_t` structure is defined in `stdlib.h` and has at least:

```
long quot; // the quotient
long rem;  // the remainder
```

Related topics:

[div](#)

log

Syntax:

```
#include <math.h>
double log( double num );
```

The function `log()` returns the natural (base e) logarithm of *num*. There's a domain error if *num* is negative, a range error if *num* is zero.

In order to calculate the logarithm of x to an arbitrary base b , you can use:

```
double answer = log(x) / log(b);
```

Related topics:

[exp](#)
[log10](#)
[pow](#)
[sqrt](#)

log10

Syntax:

```
#include <math.h>
double log10( double num );
```

The `log10()` function returns the base 10 (or common) logarithm for *num*. There's a domain error if *num* is negative, a range error if *num* is zero.

Related topics:

[log](#)

modf

Syntax:

```
#include <math.h>
double modf( double num, double *i );
```

The function `modf()` splits *num* into its integer and fraction parts. It returns the fractional part and loads the integer part into *i*.

Related topics:

[frexp](#)
[ldexp](#)

pow

Syntax:

```
#include <math.h>
double pow( double base, double exp );
```

The `pow()` function returns *base* raised to the *exp*th power. There's a domain error if *base* is zero and *exp* is less than or equal to zero. There's also a domain error if *base* is negative and *exp* is not an integer. There's a range error if an overflow occurs.

Related topics:

exp
log
sqrt

sin

Syntax:

```
#include <math.h>
double sin( double arg );
```

The function `sin()` returns the sine of *arg*, where *arg* is given in radians. The return value of `sin()` will be in the range $[-1,1]$. If *arg* is infinite, `sin()` will return NAN and raise a floating-point exception.

Related topics:

acos
asin
atan
atan2
cos
cosh
sinh
tan
tanh

sinh

Syntax:

```
#include <math.h>
double sinh( double arg );
```

The function `sinh()` returns the hyperbolic sine of *arg*.

Related topics:

`acos`

`asin`

`atan`

`atan2`

`cos`

`cosh`

`sin`

`tan`

`tanh`

sqrt

Syntax:

```
#include <math.h>
double sqrt( double num );
```

The `sqrt()` function returns the square root of *num*. If *num* is negative, a domain error occurs.

Related topics:

`exp`

`log`

`pow`

tan

Syntax:

```
#include <math.h>
double tan( double arg );
```

The `tan()` function returns the tangent of *arg*, where *arg* is given in radians. If *arg* is infinite, `tan()` will return NAN and raise a floating-point exception.

Related topics:

acos
asin
atan
atan2
cos
cosh
sin
sinh
tanh

tanh

Syntax:

```
#include <math.h>
double tanh( double arg );
```

The function `tanh()` returns the hyperbolic tangent of *arg*.

Related topics:

acos
asin
atan
atan2
cos
cosh
sin
sinh
tan

Standard C Date & Time

asctime	a textual version of the time
clock	returns the amount of time that the program has been running
ctime	returns a specifically formatted version of the time
difftime	the difference between two times
gmtime	returns a pointer to the current Greenwich Mean Time
localtime	returns a pointer to the current time
mktime	returns the calendar version of a given time
setlocale	sets the current locale
strftime	returns individual elements of the date and time
time	returns the current calendar time of the system

asctime

Syntax:

```
#include <time.h>
char *asctime( const struct tm *ptr );
```

The function `asctime()` converts the time in the struct 'ptr' to a character string of the following format:

```
day month date hours:minutes:seconds year
```

An example:

```
Mon Jun 26 12:03:53 2000
```

Related topics:

- clock
- ctime
- difftime
- gmtime
- localtime
- mktime
- time

clock

Syntax:

```
#include <time.h>
clock_t clock( void );
```

The `clock()` function returns the processor time since the program started, or -1 if that information is unavailable. To convert the return value to seconds, divide it by `CLOCKS_PER_SEC`. (Note: if your compiler is POSIX compliant, then `CLOCKS_PER_SEC` is always defined as 1000000.)

Related topics:

- asctime
- ctime
- time

ctime

Syntax:

```
#include <time.h>
char *ctime( const time_t *time );
```

The `ctime()` function converts the calendar time `time` to local time of the format:

```
day month date hours:minutes:seconds year
```

using `ctime()` is equivalent to

```
asctime( localtime( tp ) );
```

Related topics:

- [asctime](#)
- [clock](#)
- [gmtime](#)
- [localtime](#)
- [mktime](#)
- [time](#)

difftime

Syntax:

```
#include <time.h>
double difftime( time_t time2, time_t time1 );
```

The function `difftime()` returns `time2 - time1`, in seconds.

Related topics:

- [asctime](#)
- [gmtime](#)
- [localtime](#)
- [time](#)

gmtime

Syntax:

```
#include <time.h>
struct tm *gmtime( const time_t *time );
```

The `gmtime()` function returns the given *time* in Coordinated Universal Time (usually Greenwich mean time), unless it's not supported by the system, in which case **NULL** is returned. Watch out for static return.

Related topics:

<code>asctime</code>	<code>localtime</code>	<code>time</code>
<code>ctime</code>	<code>mktime</code>	
<code>difftime</code>	<code>strftime</code>	

localtime

Syntax:

```
#include <time.h>
struct tm *localtime( const time_t *time );
```

The function `localtime()` converts calendar time `time` into local time. Watch out for the static return.

Related topics:

<code>asctime</code>	<code>difftime</code>	<code>strftime</code>
<code>ctime</code>	<code>gmtime</code>	<code>time</code>

mktime

Syntax:

```
#include <time.h>
time_t mktime( struct tm *time );
```

The `mktime()` function converts the local time in *time* to calendar time, and returns it. If there is an error, -1 is returned.

Related topics:

<code>asctime</code>	<code>gmtime</code>
<code>ctime</code>	<code>time</code>

setlocale

Syntax:

```
#include <locale.h>
char *setlocale( int category, const char * locale );
```

The `setlocale()` function is used to set and retrieve the current locale. If *locale* is **NULL**, the current locale is returned. Otherwise, *locale* is used to set the locale for the given *category*.

category can have the following values:

Value	Description
LC_ALL	All of the locale
LC_TIME	Date and time formatting
LC_NUMERIC	Number formatting
LC_COLLATE	String collation and regular expression matching
LC_CTYPE	Regular expression matching, conversion, case-sensitive comparison, wide character functions, and character classification.
LC_MONETARY	For monetary formatting
LC_MESSAGES	For natural language messages

Related topics:

(Standard C String and Character) `strcoll`**strftime**

Syntax:

```
#include <time.h>
size_t strftime( char *str, size_t maxsize, const char *fmt, struct
tm *time );
```

The function `strftime()` formats date and time information from *time* to a format specified by *fmt*, then stores the result in *str* (up to *maxsize* characters). Certain codes may be used in *fmt* to specify different types of time:

Code	Meaning
%a	abbreviated weekday name (e.g. Fri)
%A	full weekday name (e.g. Friday)
%b	abbreviated month name (e.g. Oct)
%B	full month name (e.g. October)
%c	the standard date and time string

%d	day of the month, as a number (1-31)
%H	hour, 24 hour format (0-23)
%I	hour, 12 hour format (1-12)
%j	day of the year, as a number (1-366)
%m	month as a number (1-12). Note: some versions of Microsoft Visual C++ may use values that range from 0-11.
%M	minute as a number (0-59)
%p	locale's equivalent of AM or PM
%S	second as a number (0-59)
%U	week of the year, (0-53), where week 1 has the first Sunday
%w	weekday as a decimal (0-6), where Sunday is 0
%W	week of the year, (0-53), where week 1 has the first Monday
%x	standard date string
%X	standard time string
%y	year in decimal, without the century (0-99)
%Y	year in decimal, with the century
%Z	time zone name
%%	a percent sign

The `strftime()` function returns the number of characters put into *str*, or zero if an error occurs.

Related topics:

gmtime
localtime
time

time

Syntax:

```
#include <time.h>
time_t time( time_t *time );
```

The function `time()` returns the current time, or -1 if there is an error. If the argument 'time' is given, then the current time is stored in 'time'.

Related topics:

asctime

clock

ctime

difftime

gmtime

localtime

mktime

(Other Standard C Functions) srand

strftime

Standard C Memory

<code>calloc</code>	allocates and clears a two-dimensional chunk of memory
<code>free</code>	returns previously allocated memory to the operating system
<code>malloc</code>	allocates memory
<code>realloc</code>	changes the size of previously allocated memory

calloc

Syntax:

```
#include <stdlib.h>
void* calloc( size_t num, size_t size );
```

The `calloc()` function returns a pointer to space for an array of *num* objects, each of size *size*. The newly allocated memory is initialized to zero.

`calloc()` returns **NULL** if there is an error.

Related topics:

[free](#)

[malloc](#)

[realloc](#)

free

Syntax:

```
#include <stdlib.h>
void free( void* ptr );
```

The `free()` function deallocates the space pointed to by *ptr*, freeing it up for future use. *ptr* must have been used in a previous call to `malloc()`, `calloc()`, or `realloc()`. An example:

```
typedef struct data_type {
    int age;
    char name[20];
} data;
data *willy;
willy = (data*) malloc( sizeof(*willy) );
...
free( willy );
```

Related topics:

[calloc](#)

[\(C/C++ Keywords\) delete](#)

[malloc](#)

[\(C/C++ Keywords\) new](#)

[realloc](#)

malloc

Syntax:

```
#include <stdlib.h>
void *malloc( size_t size );
```

The function `malloc()` returns a pointer to a chunk of memory of size *size*, or **NULL** if there is an error. The memory pointed to will be on the heap, not the stack, so make sure to free it when you are done with it. An example:

```
typedef struct data_type {
    int age;
    char name[20];
} data;
data *bob;
bob = (data*) malloc( sizeof(data) );
if( bob != NULL ) {
    bob->age = 22;
    strcpy( bob->name, "Robert" );
    printf( "%s is %d years old\n", bob->name, bob->age );
}
free( bob );
```

Related topics:

`calloc`

(C/C++ Keywords) `delete`

`free`

(C/C++ Keywords) `new`

`realloc`

realloc

Syntax:

```
#include <stdlib.h>
void *realloc( void *ptr, size_t size );
```

The `realloc()` function changes the size of the object pointed to by `ptr` to the given size. `size` can be any size, larger or smaller than the original. The return value is a pointer to the new space, or **NULL** if there is an error.

Related topics:

`calloc`

`free`

`malloc`

Other Standard C Functions

abort	stops the program
assert	stops the program if an expression isn't true
atexit	sets a function to be called when the program exits
bsearch	perform a binary search
exit	stop the program
getenv	get environment information about a variable
longjmp	start execution at a certain point in the program
qsort	perform a quicksort
raise	send a signal to the program
rand	returns a pseudorandom number
setjmp	set execution to start at a certain point
signal	register a function as a signal handler
srand	initialize the random number generator
system	perform a system call
va_arg	use variable length parameter lists

abort

Syntax:

```
#include <stdlib.h>
void abort( void );
```

The function `abort()` terminates the current program. Depending on the implementation, the return value can indicate failure.

Related topics:

assert
atexit
exit

assert

Syntax:

```
#include <assert.h>
assert( exp );
```

The `assert()` macro is used to test for errors. If *exp* evaluates to zero, `assert()` writes information to **stderr** and exits the program. If the macro `NDEBUG` is defined, the `assert()` macros will be ignored.

Related topics:

abort

atexit

Syntax:

```
#include <stdlib.h>
int atexit( void (*func)(void) );
```

The function `atexit()` causes the function pointed to by *func* to be called when the program terminates. You can make multiple calls to `atexit()` (at least 32, depending on your compiler) and those functions will be called in reverse order of their establishment. The return value of `atexit()` is zero upon success, and non-zero on failure.

Related topics:

abort
exit

bsearch

Syntax:

```
#include <stdlib.h>
void *bsearch( const void *key, const void *buf, size_t num, size_t
size, int (*compare)(const void *, const void *) );
```

The `bsearch()` function searches `buf[0]` to `buf[num-1]` for an item that matches `key`, using a binary search. The function `compare` should return negative if its first argument is less than its second, zero if equal, and positive if greater. The items in the array `buf` should be in ascending order. The return value of `bsearch()` is a pointer to the matching item, or `NULL` if none is found.

Related topics:

qsort

exit

Syntax:

```
#include <stdlib.h>
void exit( int exit_code );
```

The `exit()` function stops the program. `exit_code` is passed on to be the return value of the program, where usually zero indicates success and non-zero indicates an error.

Related topics:

abort
atexit
system

getenv

Syntax:

```
#include <stdlib.h>
char *getenv( const char *name );
```

The function `getenv()` returns environmental information associated with *name*, and is very implementation dependent. **NULL** is returned if no information about *name* is available.

Related topics:

system

longjmp

Syntax:

```
#include <setjmp.h>
void longjmp( jmp_buf envbuf, int status );
```

The function `longjmp()` causes the program to start executing code at the point of the last call to `setjmp()`. *envbuf* is usually set through a call to `setjmp()`. *status* becomes the return value of `setjmp()` and can be used to figure out where `longjmp()` came from. *status* should not be set to zero.

Related topics:

setjmp

qsort

Syntax:

```
#include <stdlib.h>
void qsort( void *buf, size_t num, size_t size, int (*compare)(const
void *, const void *) );
```

The `qsort()` function sorts *buf* (which contains *num* items, each of size *size*) using Quicksort. The *compare* function is used to compare the items in *buf*. *compare* should return negative if the first argument is less than the second, zero if they are equal, and positive if the first argument is greater than the second. `qsort()` sorts *buf* in ascending order.

Example code:

For example, the following bit of code uses `qsort()` to sort an array of integers:

```
int compare_ints( const void* a, const void* b ) {
    int* arg1 = (int*) a;
    int* arg2 = (int*) b;
    if( *arg1 < *arg2 ) return -1;
    else if( *arg1 == *arg2 ) return 0;
    else return 1;
}
int array[] = { -2, 99, 0, -743, 2, 3, 4 };
int array_size = 7;
...
printf( "Before sorting: " );
for( int i = 0; i < array_size; i++ ) {
    printf( "%d ", array[i] );
}
printf( "\n" );
qsort( array, array_size, sizeof(int), compare_ints );
printf( "After sorting: " );
for( int i = 0; i < array_size; i++ ) {
    printf( "%d ", array[i] );
}
printf( "\n" );
```

When run, this code displays the following output:

```
Before sorting: -2 99 0 -743 2 3 4
After sorting: -743 -2 0 2 3 4 99
```

Related topics:

[bsearch](#)

[\(C++ Algorithms\) sort](#)

raise

Syntax:

```
#include <signal.h>
int raise( int signal );
```

The raise() function sends the specified *signal* to the program. Some signals:

Signal	Meaning
SIGABRT	Termination error
SIGFPE	Floating pointer error
SIGILL	Bad instruction
SIGINT	User presed CTRL-C
SIGSEGV	Illegal memory access
SIGTERM	Terminate program

The return value is zero upon success, nonzero on failure.

Related topics:

signal

rand

Syntax:

```
#include <stdlib.h>
int rand( void );
```

The function rand() returns a pseudorandom integer between zero and RAND_MAX. An example:

```
srand( time(NULL) );
for( i = 0; i < 10; i++)
    printf( "Random number #%d: %d\n", i, rand() );
```

Related topics:

srand

setjmp

Syntax:

```
#include <setjmp.h>
int setjmp( jmp_buf envbuf );
```

The `setjmp()` function saves the system stack in *envbuf* for use by a later call to `longjmp()`. When you first call `setjmp()`, its return value is zero. Later, when you call `longjmp()`, the second argument of `longjmp()` is what the return value of `setjmp()` will be. Confused? Read about `longjmp()`.

Related topics:

[longjmp](#)

signal

Syntax:

```
#include <signal.h>
void ( *signal( int signal, void (* func) (int)) ) (int);
```

The `signal()` function sets *func* to be called when *signal* is received by your program. *func* can be a custom signal handler, or one of these macros (defined in `signal.h`):

Macro	Explanation
<code>SIG_DFL</code>	default signal handling
<code>SIG_IGN</code>	ignore the signal

Some basic signals that you can attach a signal handler to are:

Signal	Description
<code>SIGTERM</code>	Generic stop signal that can be caught.
<code>SIGINT</code>	Interrupt program, normally ctrl-c.
<code>SIGQUIT</code>	Interrupt program, similar to <code>SIGINT</code> .
<code>SIGKILL</code>	Stops the program. Cannot be caught.
<code>SIGHUP</code>	Reports a disconnected terminal.

The return value of `signal()` is the address of the previously defined function for this signal, or `SIG_ERR` if there is an error.

Example code:

The following example uses the `signal()` function to call an arbitrary number of functions when the user aborts the program. The functions are stored in a vector, and a single

"clean-up" function calls each function in that vector of functions when the program is aborted:

```
void f1() {
    cout << "calling f1()..." << endl;
}
void f2() {
    cout << "calling f2()..." << endl;
}
typedef void(*endFunc)(void);
vector<endFunc> endFuncs;
void cleanUp( int dummy ) {
    for( unsigned int i = 0; i < endFuncs.size(); i++ ) {
        endFunc f = endFuncs.at(i);
        (*f)();
    }
    exit(-1);
}
int main() {
    // connect various signals to our clean-up function
    signal( SIGTERM, cleanUp );
    signal( SIGINT, cleanUp );
    signal( SIGQUIT, cleanUp );
    signal( SIGHUP, cleanUp );
    // add two specific clean-up functions to a list of functions
    endFuncs.push_back( f1 );
    endFuncs.push_back( f2 );
    // loop until the user breaks
    while( 1 );
    return 0;
}
```

Related topics:

[raise](#)

srand

Syntax:

```
#include <stdlib.h>
void srand( unsigned seed );
```

The function `srand()` is used to seed the random sequence generated by `rand()`. For any given *seed*, `rand()` will generate a specific "random" sequence over and over again.

```
srand( time(NULL) );
for( i = 0; i < 10; i++ )
    printf( "Random number #%d: %d\n", i, rand() );
```

Related topics:

[rand](#)

(Standard C Date & Time) [time](#)

system

Syntax:

```
#include <stdlib.h>
int system( const char *command );
```

The `system()` function runs the given *command* by passing it to the default command interpreter.

The return value is usually zero if the command executed without errors. If *command* is `NULL`, `system()` will test to see if there is a command interpreter available. Non-zero will be returned if there is a command interpreter available, zero if not.

Related topics:

[exit](#)

[getenv](#)

va_arg

Syntax:

```
#include <stdarg.h>
type va_arg( va_list argptr, type );
void va_end( va_list argptr );
void va_start( va_list argptr, last_parm );
```

The `va_arg()` macros are used to pass a variable number of arguments to a function.

1. First, you must have a call to `va_start()` passing a valid **va_list** and the mandatory first argument of the function. This first argument can be anything; one way to use it is to have it be an integer describing the number of parameters being passed.
2. Next, you call `va_arg()` passing the **va_list** and the type of the argument to be returned. The return value of `va_arg()` is the current parameter.
3. Repeat calls to `va_arg()` for however many arguments you have.
4. Finally, a call to `va_end()` passing the **va_list** is necessary for proper cleanup.

For example:

```
int sum( int num, ... ) {
    int answer = 0;
    va_list argptr;
    va_start( argptr, num );
    for( ; num > 0; num-- ) {
        answer += va_arg( argptr, int );
    }
    va_end( argptr );
    return( answer );
}

int main( void ) {
    int answer = sum( 4, 4, 3, 2, 1 );
    printf( "The answer is %d\n", answer );
    return( 0 );
}
```

This code displays 10, which is 4+3+2+1.

Here is another example of variable argument function, which is a simple printing function:

```
void my_printf( char *format, ... ) {
    va_list argptr;
    va_start( argptr, format );
    while( *format != '\0' ) {
        // string
        if( *format == 's' ) {
            char* s = va_arg( argptr, char* );
            printf( "Printing a string: %s\n", s );
        }
        // character
        else if( *format == 'c' ) {
```

```
        char c = (char) va_arg( argptr, int );
        printf( "Printing a character: %c\n", c );
        break;
    }
    // integer
    else if( *format == 'd' ) {
        int d = va_arg( argptr, int );
        printf( "Printing an integer: %d\n", d );
    }
    *format++;
}
va_end( argptr );
}

int main( void ) {
    my_printf( "sdc", "This is a string", 29, 'X' );
    return( 0 );
}
```

This code displays the following output when run:

```
Printing a string: This is a string
Printing an integer: 29
Printing a character: X
```


C++

C++ I/O

The `<iostream>` library automatically defines a few standard objects:

- `cout`, an object of the `ostream` class, which displays data to the standard output device.
- `cerr`, another object of the `ostream` class that writes unbuffered output to the standard error device.
- `clog`, like `cerr`, but uses buffered output.
- `cin`, an object of the `istream` class that reads data from the standard input device.

The `<fstream>` library allows programmers to do file input and output with the `ifstream` and `ofstream` classes.

C++ programmers can also do input and output from strings by using the `String Stream` class.

Some of the behavior of the C++ I/O streams (precision, justification, etc) may be modified by manipulating various io stream format flags.

<code>bad</code>	true if an error occurred
<code>clear</code>	clear and set status flags
<code>close</code>	close a stream
<code>eof</code>	true if at the end-of-file
<code>fail</code>	true if an error occurred
<code>fill</code>	manipulate the default fill character
<code>flags</code>	access or manipulate io stream format flags
<code>flush</code>	empty the buffer
<code>gcount</code>	number of characters read during last input
<code>get</code>	read characters
<code>getline</code>	read a line of characters
<code>good</code>	true if no errors have occurred
<code>ignore</code>	read and discard characters
<code>open</code>	open a new stream
<code>peek</code>	check the next input character
<code>precision</code>	manipulate the precision of a stream
<code>put</code>	write characters
<code>putback</code>	return characters to a stream
<code>rdstate</code>	returns the state flags of the stream
<code>read</code>	read data into a buffer
<code>seekg</code>	perform random access on an input stream
<code>seekp</code>	perform random access on output streams
<code>setf</code>	set format flags
<code>sync_with_stdio</code>	synchronize with standard I/O
<code>tellg</code>	read input stream pointers
<code>tellp</code>	read output stream pointers
<code>unsetf</code>	clear io stream format flags
<code>width</code>	access and manipulate the minimum field width
<code>write</code>	write characters

I/O Constructors

Syntax:

```
#include <fstream>
fstream( const char *filename, openmode mode );
ifstream( const char *filename, openmode mode );
ofstream( const char *filename, openmode mode );
```

The `fstream`, `ifstream`, and `ofstream` objects are used to do file I/O. The optional *mode* defines how the file is to be opened, according to the io stream mode flags. The optional *filename* specifies the file to be opened and associated with the stream.

Input and output file streams can be used in a similar manner to C++ predefined I/O streams, `cin` and `cout`.

Example code:

The following code reads input data and appends the result to an output file.

```
ifstream fin( "/tmp/data.txt" );
ofstream fout( "/tmp/results.txt", ios::app );
while( fin >> temp )
    fout << temp + 2 << endl;
fin.close();
fout.close();
```

Related topics:

[close](#)

[open](#)

bad

Syntax:

```
#include <fstream>
bool bad();
```

The `bad()` function returns true if a fatal error with the current stream has occurred, false otherwise.

Related topics:

<code>eof</code>	<code>good</code>
<code>fail</code>	<code>rdstate</code>

clear

Syntax:

```
#include <fstream>
void clear( iostate flags = ios::goodbit );
```

The function `clear()` does two things:

- it clears all io stream state flags associated with the current stream,
- and sets the flags denoted by *flags*

The *flags* argument defaults to `ios::goodbit`, which means that by default, all flags will be cleared and `ios::goodbit` will be set.

For example, the following code uses the `clear()` function to reset the flags of an output file stream, after an attempt is made to read from that output stream:

```
fstream outputFile( "output.txt", fstream::out );
// try to read from the output stream; this shouldn't work
int val;
outputFile >> val;
if( outputFile.fail() ) {
    cout << "Error reading from the output stream" << endl;
    // reset the flags associated with the stream
    outputFile.clear();
}

for( int i = 0; i < 10; i++ ) {
    outputFile << i << " ";
}
outputFile << endl;
```

Related topics:

<code>eof</code>	<code>good</code>
<code>fail</code>	<code>rdstate</code>

close

Syntax:

```
#include <fstream>
void close();
```

The `close()` function closes the associated file stream.

Related topics:
I/O Constructors
open

eof

Syntax:

```
#include <fstream>
bool eof();
```

The function `eof()` returns true if the end of the associated input file has been reached, false otherwise.

For example, the following code reads data from an input stream *in* and writes it to an output stream *out*, using `eof()` at the end to check if an error occurred:

```
char buf[BUFSIZE];
do {
    in.read( buf, BUFSIZE );
    std::streamsize n = in.gcount();
    out.write( buf, n );
} while( in.good() );
if( in.bad() || !in.eof() ) {
    // fatal error occurred
}
in.close();
```

Related topics:

bad
clear
fail
good
rdstate

fail

Syntax:

```
#include <fstream>
bool fail();
```

The fail() function returns true if an error has occurred with the current stream, false otherwise.

Related topics:

bad	eof	rdstate
clear	good	

fill

Syntax:

```
#include <fstream>
char fill();
char fill( char ch );
```

The function fill() either returns the current fill character, or sets the current fill character to *ch*.

The fill character is defined as the character that is used for padding when a number is smaller than the specified width(). The default fill character is the space character.

Related topics:

precision
width

flags

Syntax:

```
#include <fstream>
fmtflags flags();
fmtflags flags( fmtflags f );
```

The flags() function either returns the io stream format flags for the current stream, or sets the flags for the current stream to be *f*.

Related topics:

setf
unsetf

flush

Syntax:

```
#include <fstream>
ostream& flush();
```

The `flush()` function causes the buffer for the current output stream to be actually written out to the attached device.

This function is useful for printing out debugging information, because sometimes programs abort before they have a chance to write their output buffers to the screen. Judicious use of `flush()` can ensure that all of your debugging statements actually get printed.

Related topics:

[put](#)
[write](#)

gcount

Syntax:

```
#include <fstream>
streamsize gcount();
```

The function `gcount()` is used with input streams, and returns the number of characters read by the last input operation.

Related topics:

[get](#)
[getline](#)
[read](#)

get

Syntax:

```
#include <fstream>
int get();
istream& get( char& ch );
istream& get( char* buffer, streamsize num );
istream& get( char* buffer, streamsize num, char delim );
istream& get( streambuf& buffer );
istream& get( streambuf& buffer, char delim );
```

The `get()` function is used with input streams, and either:

- reads a character and returns that value,
- reads a character and stores it as *ch*,
- reads characters into *buffer* until *num* - 1 characters have been read, or **EOF** or newline encountered,
- reads characters into *buffer* until *num* - 1 characters have been read, or **EOF** or the *delim* character encountered (*delim* is not read until next time),
- reads characters into buffer until a newline or **EOF** is encountered,
- or reads characters into buffer until a newline, **EOF**, or *delim* character is encountered (again, *delim* isn't read until the next `get()`).

For example, the following code displays the contents of a file called `temp.txt`, character by character:

```
char ch;
ifstream fin( "temp.txt" );
while( fin.get(ch) )
    cout << ch;
fin.close();
```

Related topics:

gcount

getline

(C++ Strings) getline

ignore

peek

put

read

getline

Syntax:

```
#include <fstream>
istream& getline( char* buffer, streamsize num );
istream& getline( char* buffer, streamsize num, char delim );
```

The `getline()` function is used with input streams, and reads characters into *buffer* until either:

- *num* - 1 characters have been read,
- a newline is encountered,
- an **EOF** is encountered,
- or, optionally, until the character *delim* is read. The *delim* character is not put into *buffer*.

For example, the following code uses the `getline` function to display the first 100 characters from each line of a text file:

```
ifstream fin("tmp.dat");
int MAX_LENGTH = 100;
char line[MAX_LENGTH];
while( fin.getline(line, MAX_LENGTH) ) {
    cout << "read line: " << line << endl;
}
```

If you'd like to read lines from a file into strings instead of character arrays, consider using the `string getline` function.

Those using a Microsoft compiler may find that `getline()` reads an extra character, and should consult the documentation on the Microsoft `getline` bug.

Related topics:

<code>gcount</code>	(C++ Strings) <code>getline</code>	<code>read</code>
<code>get</code>	<code>ignore</code>	

good

Syntax:

```
#include <fstream>
bool good();
```

The function `good()` returns true if no errors have occurred with the current stream, false otherwise.

Related topics:

<code>bad</code>	<code>eof</code>	<code>rdstate</code>
<code>clear</code>	<code>fail</code>	

ignore

Syntax:

```
#include <fstream>
istream& ignore( streamsize num=1, int delim=EOF );
```

The `ignore()` function is used with input streams. It reads and throws away characters until *num* characters have been read (where *num* defaults to 1) or until the character *delim* is read (where *delim* defaults to **EOF**).

The `ignore()` function can sometimes be useful when using the `getline()` function together with the `>>` operator. For example, if you read some input that is followed by a newline using the `>>` operator, the newline will remain in the input as the next thing to be read. Since `getline()` will by default stop reading input when it reaches a newline, a subsequent call to `getline()` will return an empty string. In this case, the `ignore()` function could be called before `getline()` to "throw away" the newline.

Related topics:

[get](#)
[getline](#)

open

Syntax:

```
#include <fstream>
void open( const char *filename );
void open( const char *filename, openmode mode = default_mode );
```

The function `open()` is used with file streams. It opens *filename* and associates it with the current stream. The optional io stream mode flag *mode* defaults to `ios::in` for `ifstream`, `ios::out` for `ofstream`, and `ios::in|ios::out` for `fstream`.

If `open()` fails, the resulting stream will evaluate to false when used in a Boolean expression. For example:

```
ifstream inputStream;
inputStream.open("file.txt");
if( !inputStream ) {
    cerr << "Error opening input stream" << endl;
    return;
}
```

Related topics:

[I/O Constructors](#)
[close](#)

peek

Syntax:

```
#include <fstream>
int peek();
```

The function `peek()` is used with input streams, and returns the next character in the stream or **EOF** if the end of file is read. `peek()` does not remove the character from the stream.

Related topics:

get
putback

precision

Syntax:

```
#include <fstream>
streamsize precision();
streamsize precision( streamsize p );
```

The `precision()` function either sets or returns the current number of digits that is displayed for floating-point variables.

For example, the following code sets the precision of the `cout` stream to 5:

```
float num = 314.15926535;
cout.precision( 5 );
cout << num;
```

This code displays the following output:

```
314.16
```

Related topics:

fill
width

put

Syntax:

```
#include <fstream>
ostream& put( char ch );
```

The function `put()` is used with output streams, and writes the character *ch* to the stream.

Related topics:

flush

get

write

putback

Syntax:

```
#include <fstream>
istream& putback( char ch );
```

The `putback()` function is used with input streams, and returns the previously-read character *ch* to the input stream.

Related topics:

peek

(Standard C I/O) ungetc

rdstate

Syntax:

```
#include <fstream>
iostate rdstate();
```

The `rdstate()` function returns the io stream state flags of the current stream.

Related topics:

bad

clear

eof

fail

good

read

Syntax:

```
#include <fstream>
istream& read( char* buffer, streamsize num );
```

The function `read()` is used with input streams, and reads *num* bytes from the stream before placing them in *buffer*. If **EOF** is encountered, `read()` stops, leaving however many bytes it put into *buffer* as they are.

For example:

```
struct {
    int height;
    int width;
} rectangle;
input_file.read( (char *)(&rectangle), sizeof(rectangle) );
if( input_file.bad() ) {
    cerr << "Error reading data" << endl;
    exit( 0 );
}
```

Related topics:

gcount
get
getline
write

seekg

Syntax:

```
#include <fstream>
istream& seekg( off_type offset, ios::seekdir origin );
istream& seekg( pos_type position );
```

The function `seekg()` is used with input streams, and it repositions the "get" pointer for the current stream to *offset* bytes away from *origin*, or places the "get" pointer at *position*.

Related topics:

seekp
tellg
tellp

seekp

Syntax:

```
#include <fstream>
ostream& seekp( off_type offset, ios::seekdir origin );
ostream& seekp( pos_type position );
```

The `seekp()` function is used with output streams, but is otherwise very similar to `seekg()`.

Related topics:

seekg

tellg

tellp

setf

Syntax:

```
#include <fstream>
fmtflags setf( fmtflags flags );
fmtflags setf( fmtflags flags, fmtflags needed );
```

The function `setf()` sets the io stream format flags of the current stream to *flags*. The optional *needed* argument specifies that only the flags that are in both *flags* and *needed* should be set. The return value is the previous configuration of io stream format flags.

For example:

```
int number = 0x3FF;
cout.setf( ios::dec );
cout << "Decimal: " << number << endl;
cout.unsetf( ios::dec );
cout.setf( ios::hex );
cout << "Hexadecimal: " << number << endl;
```

Note that the preceding code is functionally identical to:

```
int number = 0x3FF;
cout << "Decimal: " << number << endl << hex << "Hexadecimal: " <<
number << dec << endl;
```

thanks to io stream manipulators.

Related topics:

flags

unsetf

sync_with_stdio

Syntax:

```
#include <fstream>
static bool sync_with_stdio( bool sync=true );
```

The `sync_with_stdio()` function allows you to turn on and off the ability for the C++ I/O system to work with the C I/O system.

tellg

Syntax:

```
#include <fstream>
pos_type tellg();
```

The `tellg()` function is used with input streams, and returns the current "get" position of the pointer in the stream.

Related topics:

[seekg](#)

[seekp](#)

[tellp](#)

tellp

Syntax:

```
#include <fstream>
pos_type tellp();
```

The `tellp()` function is used with output streams, and returns the current "put" position of the pointer in the stream.

For example, the following code displays the file pointer as it writes to a stream:

```
string s("In Xanadu did Kubla Khan...");
ofstream fout("output.txt");
for( int i=0; i < s.length(); i++ ) {
    cout << "File pointer: " << fout.tellp();
    fout.put( s[i] );
    cout << " " << s[i] << endl;
}
fout.close();
```

Related topics:

[seekg](#)

[seekp](#)

[tellg](#)

unsetf

Syntax:

```
#include <fstream>
void unsetf( fmtflags flags );
```

The function `unsetf()` uses *flags* to clear the io stream format flags associated with the current stream.

Related topics:

flags
setf

width

Syntax:

```
#include <fstream>
int width();
int width( int w );
```

The function `width()` returns the current width, which is defined as the minimum number of characters to display with each output. The optional argument *w* can be used to set the width.

For example:

```
cout.width( 5 );
cout << "2";
```

displays

```
    2
```

(that's four spaces followed by a '2')

Related topics:

fill
precision

write

Syntax:

```
#include <fstream>
ostream& write( const char* buffer, streamsize num );
```

The `write()` function is used with output streams, and writes *num* bytes from *buffer* to the current output stream.

Related topics:

flush

put

read

C++ I/O Examples

Reading From Files

Assume that we have a file named *data.txt* that contains this text:

```
Fry: One Jillion dollars.
[Everyone gasps.]
Auctioneer: Sir, that's not a number.
[Everyone gasps.]
```

We could use this code to read data from the file, word by word:

```
ifstream fin("data.txt");
string s;
while( fin >> s ) {
    cout << "Read from file: " << s << endl;
}
```

When used in this manner, we'll get space-delimited bits of text from the file:

```
Read from file: Fry:
Read from file: One
Read from file: Jillion
Read from file: dollars.
Read from file: [Everyone
Read from file: gasps.]
Read from file: Auctioneer:
Read from file: Sir,
Read from file: that's
Read from file: not
Read from file: a
Read from file: number.
Read from file: [Everyone
Read from file: gasps.]
```

Note that in the previous example, all of the whitespace that separated words (including newlines) was lost. If we were interested in preserving whitespace, we could read the file in line-by-line using the I/O `getline()` function.

```
ifstream fin("data.txt");
const int LINE_LENGTH = 100;
char str[LINE_LENGTH];
while( fin.getline(str,LINE_LENGTH) ) {
    cout << "Read from file: " << str << endl;
}
```

Reading line-by-line produces the following output:

```
Read from file: Fry: One Jillion dollars.
Read from file: [Everyone gasps.]
Read from file: Auctioneer: Sir, that's not a number.
Read from file: [Everyone gasps.]
```

If you want to avoid reading into character arrays, you can use the C++ string `getline()` function to read lines into strings:

```
ifstream fin("data.txt");
string s;
while( getline(fin,s) ) {
    cout << "Read from file: " << s << endl;
}
```

Checking For Errors

Simply evaluating an I/O object in a boolean context will return false if any errors have occurred:

```
string filename = "data.txt";
ifstream fin( filename.c_str() );
if( !fin ) {
    cout << "Error opening " << filename << " for input" << endl;
    exit(-1);
}
```

C++ Strings

String constructors	create strings from arrays of characters and other strings
String operators	concatenate strings, assign strings, use strings for I/O, compare strings
append	append characters and strings onto a string
assign	give a string values from strings of characters and other C++ strings
at	returns the character at a specific location
begin	returns an iterator to the beginning of the string
c_str	returns a non-modifiable standard C character array version of the string
capacity	returns the number of characters that the string can hold
clear	removes all characters from the string
compare	compares two strings
copy	copies characters from a string into an array
data	returns a pointer to the first character of a string
empty	true if the string has no characters
end	returns an iterator just past the last character of a string
erase	removes characters from a string
find	find characters in the string
find_first_not_of	find first absence of characters
find_first_of	find first occurrence of characters
find_last_not_of	find last absence of characters
find_last_of	find last occurrence of characters
getline	read data from an I/O stream into a string
insert	insert characters into a string
length	returns the length of the string
max_size	returns the maximum number of characters that the string can hold
push_back	add a character to the end of the string
rbegin	returns a reverse_iterator to the end of the string
rend	returns a reverse_iterator to the beginning of the string
replace	replace characters in the string
reserve	sets the minimum capacity of the string
resize	change the size of the string
rfind	find the last occurrence of a substring
size	returns the number of items in the string
substr	returns a certain substring
swap	swap the contents of this string with another

String constructors

Syntax:

```
#include <string>
string();
string( const string& s );
string( size_type length, const char& ch );
string( const char* str );
string( const char* str, size_type length );
string( const string& str, size_type index, size_type length );
string( input_iterator start, input_iterator end );
~string();
```

The string constructors create a new string containing:

- nothing; an empty string,
- a copy of the given string *s*,
- *length* copies of *ch*,
- a duplicate of *str* (optionally up to *length* characters long),
- a substring of *str* starting at *index* and *length* characters long
- a string of characters *ss* denoted by the *start* and *end* iterators

For example,

```
string str1( 5, 'c' );
string str2( "Now is the time..." );
string str3( str2, 11, 4 );
cout << str1 << endl;
cout << str2 << endl;
cout << str3 << endl;
```

displays

```
ccccc
Now is the time...
time
```

The string constructors usually run in linear time, except the empty constructor, which runs in constant time.

String operators

Syntax:

```
#include <string>
bool operator==(const string& c1, const string& c2);
bool operator!=(const string& c1, const string& c2);
bool operator<(const string& c1, const string& c2);
bool operator>(const string& c1, const string& c2);
bool operator<=(const string& c1, const string& c2);
```

```

bool operator>=(const string& c1, const string& c2);
string operator+(const string& s1, const string& s2 );
string operator+(const char* s, const string& s2 );
string operator+( char c, const string& s2 );
string operator+( const string& s1, const char* s );
string operator+( const string& s1, char c );
ostream& operator<<( ostream& os, const string& s );
istream& operator>>( istream& is, string& s );
string& operator=( const string& s );
string& operator=( const char* s );
string& operator=( char ch );
char& operator[]( size_type index );

```

C++ strings can be compared and assigned with the standard comparison operators: ==, !=, <=, >=, <, >, and =. Performing a comparison or assigning one string to another takes linear time.

Two strings are equal if:

1. Their size is the same, and
2. Each member in location *i* in one string is equal to the the member in location *i* in the other string.

Comparisons among strings are done lexicographically.

In addition to the normal container operators, strings can also be concatenated with the + operator and fed to the C++ I/O stream classes with the << and >> operators.

For example, the following code concatenates two strings and displays the result:

```

string s1 = "Now is the time...";
string s2 = "for all good men...";
string s3 = s1 + s2;
cout << "s3 is " << s3 << endl;

```

Futhermore, strings can be assigned values that are other strings, character arrays, or even single characters. The following code is perfectly valid:

```

char ch = 'N';
string s;
s = ch;

```

Individual characters of a string can be examined with the [] operator, which runs in constant time.

Related topics:

c_str
compare
data

append

Syntax:

```
#include <string>
string& append( const string& str );
string& append( const char* str );
string& append( const string& str, size_type index, size_type len );
string& append( const char* str, size_type num );
string& append( size_type num, char ch );
string& append( input_iterator start, input_iterator end );
```

The append() function either:

- appends *str* on to the end of the current string,
- appends a substring of *str* starting at *index* that is *len* characters long on to the end of the current string,
- appends *num* characters of *str* on to the end of the current string,
- appends *num* repetitions of *ch* on to the end of the current string,
- or appends the sequence denoted by *start* and *end* on to the end of the current string.

For example, the following code uses append() to add 10 copies of the '!' character to a string:

```
string str = "Hello World";
str.append( 10, '!' );
cout << str << endl;
```

That code displays:

```
Hello World!!!!!!!!!!!!
```

In the next example, append() is used to concatenate a substring of one string onto another string:

```
string str1 = "Eventually I stopped caring...";
string str2 = "but that was the '80s so nobody noticed.";
str1.append( str2, 25, 15 );
cout << "str1 is " << str1 << endl;
```

When run, the above code displays:

```
str1 is Eventually I stopped caring...nobody noticed.
```

assign

Syntax:

```

#include <string>
void assign( size_type num, const char& val );
void assign( input_iterator start, input_iterator end );
string& assign( const string& str );
string& assign( const char* str );
string& assign( const char* str, size_type num );
string& assign( const string& str, size_type index, size_type len );
string& assign( size_type num, const char& ch );

```

The default `assign()` function gives the current string the values from *start* to *end*, or gives it *num* copies of *val*.

In addition to the normal (C++ Lists) `assign()` functionality that all C++ containers have, strings possess an `assign()` function that also allows them to:

- assign *str* to the current string,
- assign the first *num* characters of *str* to the current string,
- assign a substring of *str* starting at *index* that is *len* characters long to the current string,

For example, the following code:

```

string str1, str2 = "War and Peace";
str1.assign( str2, 4, 3 );
cout << str1 << endl;

```

displays

and

This function will destroy the previous contents of the string.

Related topics:
(C++ Lists) `assign`

at

Syntax:

```
#include <string>
TYPE& at( size_type loc );
const TYPE& at( size_type loc ) const;
```

The `at()` function returns a reference to the element in the string at index *loc*. The `at()` function is safer than the `[]` operator, because it won't let you reference items outside the bounds of the string.

For example, consider the following code:

```
vector<int> v( 5, 1 );
for( int i = 0; i < 10; i++ ) {
    cout << "Element " << i << " is " << v[i] << endl;
}
```

This code overruns the end of the vector, producing potentially dangerous results. The following code would be much safer:

```
vector<int> v( 5, 1 );
for( int i = 0; i < 10; i++ ) {
    cout << "Element " << i << " is " << v.at(i) << endl;
}
```

Instead of attempting to read garbage values from memory, the `at()` function will realize that it is about to overrun the vector and will throw an exception.

Related topics:

(C++ Multimaps) [Multimap operators](#)

(C++ Double-ended Queues) [Container operators](#)

begin

Syntax:

```
#include <string>
iterator begin();
const_iterator begin() const;
```

The function `begin()` returns an iterator to the first element of the string. `begin()` should run in constant time.

For example, the following code uses `begin()` to initialize an iterator that is used to traverse a list:

```
// Create a list of characters
list<char> charList;
for( int i=0; i < 10; i++ ) {
    charList.push_front( i + 65 );
}
// Display the list
list<char>::iterator theIterator;
for( theIterator = charList.begin(); theIterator != charList.end();
theIterator++ ) {
    cout << *theIterator;
}
```

Related topics:

[end](#)

[rbegin](#)

[rend](#)

c_str

Syntax:

```
#include <string>
const char* c_str();
```

The function `c_str()` returns a `const` pointer to a regular C string, identical to the current string. The returned string is null-terminated.

Note that since the returned pointer is of type `const`, the character data that `c_str()` returns **cannot be modified**. Furthermore, you do not need to call `free()` or `delete` on this pointer.

Related topics:

[String operators](#)

[data](#)

capacity

Syntax:

```
#include <string>
size_type capacity() const;
```

The `capacity()` function returns the number of elements that the string can hold before it will need to allocate more space.

For example, the following code uses two different methods to set the capacity of two vectors. One method passes an argument to the constructor that suggests an initial size, the other method calls the `reserve` function to achieve a similar goal:

```
vector<int> v1(10);
cout << "The capacity of v1 is " << v1.capacity() << endl;
vector<int> v2;
v2.reserve(20);
cout << "The capacity of v2 is " << v2.capacity() << endl;
```

When run, the above code produces the following output:

```
The capacity of v1 is 10
The capacity of v2 is 20
```

C++ containers are designed to grow in size dynamically. This frees the programmer from having to worry about storing an arbitrary number of elements in a container. However, sometimes the programmer can improve the performance of her program by giving hints to the compiler about the size of the containers that the program will use. These hints come in the form of the `reserve()` function and the constructor used in the above example, which tell the compiler how large the container is expected to get.

The `capacity()` function runs in constant time.

Related topics:

[reserve](#)

[resize](#)

[size](#)

clear

Syntax:

```
#include <string>
void clear();
```

The function `clear()` deletes all of the elements in the string. `clear()` runs in linear time.

Related topics:

(C++ Lists) [erase](#)

compare

Syntax:

```

#include <string>
int compare( const string& str );
int compare( const char* str );
int compare( size_type index, size_type length, const string& str );
int compare( size_type index, size_type length, const string& str,
size_type index2,
size_type length2 );
int compare( size_type index, size_type length, const char* str,
size_type length2 );

```

The `compare()` function either compares *str* to the current string in a variety of ways, returning

Return Value	Case
less than zero	this < str
zero	this == str
greater than zero	this > str

The various functions either:

- compare *str* to the current string,
- compare *str* to a substring of the current string, starting at *index* for *length* characters,
- compare a substring of *str* to a substring of the current string, where *index2* and *length2* refer to *str* and *index* and *length* refer to the current string,
- or compare a substring of *str* to a substring of the current string, where the substring of *str* begins at zero and is *length2* characters long, and the substring of the current string begins at *index* and is *length* characters long.

For example, the following code uses `compare()` to compare four strings with each other:

```

string names[] = {"Homer", "Marge", "3-eyed fish", "inanimate carbon
rod"};
for( int i = 0; i < 4; i++ ) {
    for( int j = 0; j < 4; j++ ) {
        cout << names[i].compare( names[j] ) << " ";
    }
    cout << endl;
}

```

Data from the above code was used to generate this table, which shows how the various strings compare to each other:

	Homer	Marge	3-eyed fish	inanimate carbon rod
"Homer".compare(x)	0	-1	1	-1
"Marge".compare(x)	1	0	1	-1
"3-eyed fish".compare(x)	-1	-1	0	-1
"inanimate carbon rod".compare(x)	1	1	1	0

Related topics:

String operators

copy

Syntax:

```
#include <string>
size_type copy( char* str, size_type num, size_type index = 0 );
```

The `copy()` function copies *num* characters of the current string (starting at *index* if it's specified, 0 otherwise) into *str*.

The return value of `copy()` is the number of characters copied.

For example, the following code uses `copy()` to extract a substring of a string into an array of characters:

```
char buf[30];
memset( buf, '\0', 30 );
string str = "Trying is the first step towards failure.";
str.copy( buf, 24 );
cout << buf << endl;
```

When run, this code displays:

```
Trying is the first step
```

Note that before calling `copy()`, we first call (Standard C String and Character) `memset()` to fill the destination array with copies of the **NULL** character. This step is included to make sure that the resulting array of characters is **NULL**-terminated.

Related topics:

[substr](#)

data

Syntax:

```
#include <string>
const char *data();
```

The function `data()` returns a pointer to the first character in the current string.

Related topics:

[String operators](#)

[c_str](#)

empty

Syntax:

```
#include <string>
bool empty() const;
```

The `empty()` function returns true if the string has no elements, false otherwise.

For example:

```
string s1;
string s2("");
string s3("This is a string");
cout.setf(ios::boolalpha);
cout << s1.empty() << endl;
cout << s2.empty() << endl;
cout << s3.empty() << endl;
```

When run, this code produces the following output:

```
true
true
false
```

Related topics:

[size](#)

end

Syntax:

```
#include <string>
iterator end();
const_iterator end() const;
```

The `end()` function returns an iterator just past the end of the string.

Note that before you can access the last element of the string using an iterator that you get from a call to `end()`, you'll have to decrement the iterator first.

For example, the following code uses `begin()` and `end()` to iterate through all of the members of a vector:

```
vector<int> v1( 5, 789 );
vector<int>::iterator it;
for( it = v1.begin(); it != v1.end(); it++ ) {
    cout << *it << endl;
}
```

The iterator is initialized with a call to `begin()`. After the body of the loop has been executed, the iterator is incremented and tested to see if it is equal to the result of calling `end()`. Since `end()` returns an iterator pointing to an element just after the last element of the vector, the loop will only stop once all of the elements of the vector have been displayed.

`end()` runs in constant time.

Related topics:

[begin](#)
[rbegin](#)
[rend](#)

erase

Syntax:

```
#include <string>
iterator erase( iterator loc );
iterator erase( iterator start, iterator end );
string& erase( size_type index = 0, size_type num = npos );
```

The `erase()` function either:

- removes the character pointed to by *loc*, returning an iterator to the next character,
- removes the characters between *start* and *end* (including the one at *start* but not the one at *end*), returning an iterator to the character after the last character removed,
- or removes *num* characters from the current string, starting at *index*, and returns **this*.

The parameters *index* and *num* have default values, which means that `erase()` can be called with just *index* to erase all characters after *index* or with no arguments to erase all characters.

For example:

```
string s("So, you like donuts, eh? Well, have all the donuts in the
world!");
cout << "The original string is '" << s << "'" << endl;
s.erase( 50, 14 );
cout << "Now the string is '" << s << "'" << endl;
s.erase( 24 );
cout << "Now the string is '" << s << "'" << endl;
s.erase();
cout << "Now the string is '" << s << "'" << endl;
```

will display

```
The original string is 'So, you like donuts, eh? Well, have all the
donuts in the world!'
Now the string is 'So, you like donuts, eh? Well, have all the
donuts'
Now the string is 'So, you like donuts, eh?'
Now the string is ''
```

`erase()` runs in linear time.

Related topics:

[insert](#)

find

Syntax:

```
#include <string>
size_type find( const string& str, size_type index );
size_type find( const char* str, size_type index );
size_type find( const char* str, size_type index, size_type length );
size_type find( char ch, size_type index );
```

The function find() either:

- returns the first occurrence of *str* within the current string, starting at *index*, string::npos if nothing is found,
- if the *length* parameter is given, then find() returns the first occurrence of the first *length* characters of *str* within the current string, starting at *index*, string::npos if nothing is found,
- or returns the index of the first occurrence *ch* within the current string, starting at *index*, string::npos if nothing is found.

For example:

```
string str1( "Alpha Beta Gamma Delta" );
string::size_type loc = str1.find( "Omega", 0 );
if( loc != string::npos ) {
    cout << "Found Omega at " << loc << endl;
} else {
    cout << "Didn't find Omega" << endl;
}
```

Related topics:

[find_first_not_of](#)

[find_first_of](#)

[find_last_not_of](#)

[find_last_of](#)

[rfind](#)

find_first_not_of

Syntax:

```

#include <string>
size_type find_first_not_of( const string& str, size_type index = 0 );
size_type find_first_not_of( const char* str, size_type index = 0 );
size_type find_first_not_of( const char* str, size_type index,
size_type num );
size_type find_first_not_of( char ch, size_type index = 0 );

```

The `find_first_not_of()` function either:

- returns the index of the first character within the current string that does not match any character in *str*, beginning the search at *index*, `string::npos` if nothing is found,
- searches the current string, beginning at *index*, for any character that does not match the first *num* characters in *str*, returning the index in the current string of the first character found that meets this criteria, otherwise returning `string::npos`,
- or returns the index of the first occurrence of a character that does not match *ch* in the current string, starting the search at *index*, `string::npos` if nothing is found.

For example, the following code searches a string of text for the first character that is not a lower-case character, space, comma, or hyphen:

```

string lower_case = "abcdefghijklmnopqrstuvwxyz ,-";
string str = "this is the lower-case part, AND THIS IS THE UPPER-CASE PART";
cout << "first non-lower-case letter in str at: " <<
str.find_first_not_of(lower_case) << endl;

```

When run, `find_first_not_of()` finds the first upper-case letter in *str* at index 29 and displays this output:

```

first non-lower-case letter in str at: 29

```

Related topics:

[find](#)
[find_first_not_of](#)
[find_first_of](#)
[find_last_not_of](#)
[find_last_of](#)
[rfind](#)

find_first_of

Syntax:

```
#include <string>
size_type find_first_of( const string &str, size_type index = 0 );
size_type find_first_of( const char* str, size_type index = 0 );
size_type find_first_of( const char* str, size_type index, size_type
num );
size_type find_first_of( char ch, size_type index = 0 );
```

The `find_first_of()` function either:

- returns the index of the first character within the current string that matches any character in *str*, beginning the search at *index*, `string::npos` if nothing is found,
- searches the current string, beginning at *index*, for any of the first *num* characters in *str*, returning the index in the current string of the first character found, or `string::npos` if no characters match,
- or returns the index of the first occurrence of *ch* in the current string, starting the search at *index*, `string::npos` if nothing is found.

Related topics:

[find](#)

[find_first_not_of](#)

[find_last_not_of](#)

[find_last_of](#)

[rfind](#)

find_last_not_of

Syntax:

```

#include <string>
size_type find_last_not_of( const string& str, size_type index = npos
);
size_type find_last_not_of( const char* str, size_type index = npos);
size_type find_last_not_of( const char* str, size_type index,
size_type num );
size_type find_last_not_of( char ch, size_type index = npos );

```

The `find_last_not_of()` function either:

- returns the index of the last character within the current string that does not match any character in *str*, doing a reverse search from *index*, `string::npos` if nothing is found,
- does a reverse search in the current string, beginning at *index*, for any character that does not match the first *num* characters in *str*, returning the index in the current string of the first character found that meets this criteria, otherwise returning `string::npos`,
- or returns the index of the last occurrence of a character that does not match *ch* in the current string, doing a reverse search from *index*, `string::npos` if nothing is found.

For example, the following code searches for the last non-lower-case character in a mixed string of characters:

```

string lower_case = "abcdefghijklmnopqrstuvwxy";
string str = "abcdefgABCDEFGHijklmnop";
cout << "last non-lower-case letter in str at: " <<
str.find_last_not_of(lower_case) << endl;

```

This code displays the following output:

```
last non-lower-case letter in str at: 13
```

Related topics:

[find](#)

[find_first_not_of](#)

[find_first_of](#)

[find_last_of](#)

[rfind](#)

find_last_of

Syntax:

```
#include <string>
size_type find_last_of( const string& str, size_type index = npos );
size_type find_last_of( const char* str, size_type index = npos );
size_type find_last_of( const char* str, size_type index, size_type
num );
size_type find_last_of( char ch, size_type index = npos );
```

The `find_last_of()` function either:

- does a reverse search from *index*, returning the index of the first character within the current string that matches any character in *str*, or `string::npos` if nothing is found,
- does a reverse search in the current string, beginning at *index*, for any of the first *num* characters in *str*, returning the index in the current string of the first character found, or `string::npos` if no characters match,
- or does a reverse search from *index*, returning the index of the first occurrence of *ch* in the current string, `string::npos` if nothing is found.

Related topics:

[find](#)

[find_first_not_of](#)

[find_first_of](#)

[find_last_not_of](#)

[rfind](#)

getline

Syntax:

```
#include <string>
istream& getline( istream& is, string& s, char delimiter = '\n' );
```

The C++ string class defines the global function `getline()` to read strings from an I/O stream. The `getline()` function, which is not part of the string class, reads a line from *is* and stores it into *s*. If a character *delimiter* is specified, then `getline()` will use *delimiter* to decide when to stop reading data.

For example, the following code reads a line of text from **stdin** and displays it to **stdout**:

```
string s;
getline( cin, s );
cout << "You entered " << s << endl;
```

After getting a line of data in a string, you may find that string streams are useful in extracting data from that string. For example, the following code reads numbers from standard input, ignoring any "commented" lines that begin with double slashes:

```
// expects either space-delimited numbers or lines that start with
// two forward slashes (//)
string s;
while( getline(cin,s) ) {
    if( s.size() >= 2 && s[0] == '/' && s[1] == '/' ) {
        cout << " ignoring comment: " << s << endl;
    } else {
        istringstream ss(s);
        double d;
        while( ss >> d ) {
            cout << " got a number: " << d << endl;
        }
    }
}
```

When run with a user supplying input, the above code might produce this output:

```
// test
ignoring comment: // test
23.3 -1 3.14159
got a number: 23.3
got a number: -1
got a number: 3.14159
// next batch
ignoring comment: // next batch
1 2 3 4 5
got a number: 1
got a number: 2
got a number: 3
got a number: 4
got a number: 5
50
got a number: 50
```

Related topics:

(C++ I/O) [get](#)

(C++ I/O) [getline](#)

[string streams](#)

insert

Syntax:

```

#include <string>
iterator insert( iterator i, const char& ch );
string& insert( size_type index, const string& str );
string& insert( size_type index, const char* str );
string& insert( size_type index1, const string& str, size_type
index2, size_type num );
string& insert( size_type index, const char* str, size_type num );
string& insert( size_type index, size_type num, char ch );
void insert( iterator i, size_type num, const char& ch );
void insert( iterator i, iterator start, iterator end );

```

The very multi-purpose `insert()` function either:

- inserts *ch* before the character denoted by *i*,
- inserts *str* into the current string, at location *index*,
- inserts a substring of *str* (starting at *index2* and *num* characters long) into the current string, at location *index1*,
- inserts *num* characters of *str* into the current string, at location *index*,
- inserts *num* copies of *ch* into the current string, at location *index*,
- inserts *num* copies of *ch* into the current string, before the character denoted by *i*,
- or inserts the characters denoted by *start* and *end* into the current string, before the character specified by *i*.

Related topics:

erase

replace

length

Syntax:

```

#include <string>
size_type length() const;

```

The `length()` function returns the number of elements in the current string, performing the same role as the `size()` function.

Related topics:

size

max_size

Syntax:

```
#include <string>
size_type max_size() const;
```

The `max_size()` function returns the maximum number of elements that the string can hold. The `max_size()` function should not be confused with the `size()` or `capacity()` functions, which return the number of elements currently in the string and the the number of elements that the string will be able to hold before more memory will have to be allocated, respectively.

Related topics:

[size](#)

push_back

Syntax:

```
#include <string>
void push_back( const TYPE& val );
```

The `push_back()` function appends *val* to the end of the string.

For example, the following code puts 10 integers into a list:

```
list<int> the_list;
for( int i = 0; i < 10; i++ )
    the_list.push_back( i );
```

When displayed, the resulting list would look like this:

```
0 1 2 3 4 5 6 7 8 9
```

`push_back()` runs in constant time.

Related topics:

[\(C++ Lists\) assign](#)

[\(C++ Lists\) insert](#)

[\(C++ Lists\) pop_back](#)

[\(C++ Lists\) push_front](#)

rbegin

Syntax:

```
#include <string>
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
```

The `rbegin()` function returns a `reverse_iterator` to the end of the current string.

`rbegin()` runs in constant time.

Related topics:

[begin](#)

[end](#)

[rend](#)

rend

Syntax:

```
#include <string>
reverse_iterator rend();
const_reverse_iterator rend() const;
```

The function `rend()` returns a `reverse_iterator` to the beginning of the current string.

`rend()` runs in constant time.

Related topics:

[begin](#)

[end](#)

[rbegin](#)

replace

Syntax:

```

#include <string>
string& replace( size_type index, size_type num, const string& str );
string& replace( size_type index1, size_type num1, const string& str,
size_type index2, size_type num2 );
string& replace( size_type index, size_type num, const char* str );
string& replace( size_type index, size_type num1, const char* str,
size_type num2 );
string& replace( size_type index, size_type num1, size_type num2,
char ch );
string& replace( iterator start, iterator end, const string& str );
string& replace( iterator start, iterator end, const char* str );
string& replace( iterator start, iterator end, const char* str,
size_type num );
string& replace( iterator start, iterator end, size_type num, char ch
);

```

The function `replace()` either:

- replaces characters of the current string with up to *num* characters from *str*, beginning at *index*,
- replaces up to *num1* characters of the current string (starting at *index1*) with up to *num2* characters from *str* beginning at *index2*,
- replaces up to *num* characters of the current string with characters from *str*, beginning at *index* in *str*,
- replaces up to *num1* characters in the current string (beginning at *index1*) with *num2* characters from *str* beginning at *index2*,
- replaces up to *num1* characters in the current string (beginning at *index*) with *num2* copies of *ch*,
- replaces the characters in the current string from *start* to *end* with *str*,
- replaces characters in the current string from *start* to *end* with *num* characters from *str*,
- or replaces the characters in the current string from *start* to *end* with *num* copies of *ch*.

For example, the following code displays the string "They say he carved it himself...find your soul-mate, Homer."

```

string s = "They say he carved it himself...from a BIGGER spoon";
string s2 = "find your soul-mate, Homer.";
s.replace( 32, s2.length(), s2 );
cout << s << endl;

```

Related topics:

insert

reserve

Syntax:

```
#include <string>
void reserve( size_type size );
```

The `reserve()` function sets the capacity of the string to at least *size*.

`reserve()` runs in linear time.

Related topics:

capacity

resize

Syntax:

```
#include <string>
void resize( size_type size, const TYPE& val = TYPE() );
```

The function `resize()` changes the size of the string to *size*. If *val* is specified then any newly-created elements will be initialized to have a value of *val*.

This function runs in linear time.

Related topics:

(C++ Multimaps) Multimap constructors & destructors

capacity

size

rfind

Syntax:

```

#include <string>
size_type rfind( const string& str, size_type index );
size_type rfind( const char* str, size_type index );
size_type rfind( const char* str, size_type index, size_type num );
size_type rfind( char ch, size_type index );

```

The rfind() function either:

- returns the location of the first occurrence of *str* in the current string, doing a reverse search from *index*, string::npos if nothing is found,
- returns the location of the first occurrence of *str* in the current string, doing a reverse search from *index*, searching at most *num* characters, string::npos if nothing is found,
- or returns the location of the first occurrence of *ch* in the current string, doing a reverse search from *index*, string::npos if nothing is found.

For example, in the following code, the first call to rfind() returns string::npos, because the target word is not within the first 8 characters of the string. However, the second call returns 9, because the target word is within 20 characters of the beginning of the string.

```

int loc;
string s = "My cat's breath smells like cat food.";
loc = s.rfind( "breath", 8 );
cout << "The word breath is at index " << loc << endl;
loc = s.rfind( "breath", 20 );
cout << "The word breath is at index " << loc << endl;

```

Related topics:

find	find_last_not_of
find_first_not_of	find_last_of
find_first_of	

size

Syntax:

```

#include <string>
size_type size() const;

```

The size() function returns the number of elements in the current string.

Related topics:

capacity	length	resize
empty	max_size	

substr

Syntax:

```
#include <string>
string substr( size_type index, size_type length = npos );
```

The `substr()` function returns a substring of the current string, starting at *index*, and *length* characters long. If *length* is omitted, it will default to `string::npos`, and the `substr()` function will simply return the remainder of the string starting at *index*.

For example:

```
string s("What we have here is a failure to communicate");
string sub = s.substr(21);
cout << "The original string is " << s << endl;
cout << "The substring is " << sub << endl;
```

displays

```
The original string is What we have here is a failure to communicate
The substring is a failure to communicate
```

Related topics:

[copy](#)

swap

Syntax:

```
#include <string>
void swap( container& from );
```

The `swap()` function exchanges the elements of the current string with those of *from*. This function operates in constant time.

For example, the following code uses the `swap()` function to exchange the values of two strings:

```
string first( "This comes first" );
string second( "And this is second" );
first.swap( second );
cout << first << endl;
cout << second << endl;
```

The above code displays:

```
And this is second
This comes first
```

Related topics:
(C++ Lists) [splice](#)

C++ String Streams

String streams are similar to the `<iostream>` and `<fstream>` libraries, except that string streams allow you to perform I/O on strings instead of streams. The `<sstream>` library provides functionality similar to `scanf()` and `sprintf()` in the standard C library. Three main classes are available in `<sstream>`:

- `stringstream` - allows input and output
- `istringstream` - allows input only
- `ostringstream` - allows output only

String streams are actually subclasses of `iostreams`, so **all of the functions available for `iostreams` are also available for `stringstream`**. See the C++ I/O functions for more information.

Constructors	create new string streams
Operators	read from and write to string streams
<code>rdbuf</code>	get the buffer for a string stream
<code>str</code>	get or set the stream's string

String Stream Constructors

Syntax:

```
#include <sstream>
stringstream()
stringstream( openmode mode )
stringstream( string s, openmode mode )
ostringstream()
ostringstream( openmode mode )
ostringstream( string s, openmode mode )
istringstream()
istringstream( openmode mode )
istringstream( string s, openmode mode )
```

The `stringstream`, `ostringstream`, and `istringstream` objects are used for input and output to a string. They behave in a manner similar to `fstream`, `ofstream` and `ifstream` objects.

The optional *mode* parameter defines how the file is to be opened, according to the io stream mode flags.

An `ostringstream` object can be used to write to a string. This is similar to the C `sprintf()` function. For example:

```
ostringstream s1;
int i = 22;
s1 << "Hello " << i << endl;
string s2 = s1.str();
cout << s2;
```

An `istringstream` object can be used to read from a string. This is similar to the C `sscanf()` function. For example:

```
istringstream stream1;
string string1 = "25";
stream1.str(string1);
int i;
stream1 >> i;
cout << i << endl; // displays 25
```

You can also specify the input string in the `istringstream` constructor as in this example:

```
string string1 = "25";
istringstream stream1(string1);
int i;
stream1 >> i;
cout << i << endl; // displays 25
```

A `stringstream` object can be used for both input and output to a string like an `fstream` object.

Related topics:
C++ I/O Streams

String Stream Operators

Syntax:

```
#include <sstream>
operator<<
operator>>
```

Like C++ I/O Streams, the simplest way to use string streams is to take advantage of the overloaded << and >> operators.

The << operator inserts data into the stream. For example:

```
stream1 << "hello" << i;
```

This example inserts the string "hello" and the variable *i* into *stream1*. In contrast, the >> operator extracts data out of a string stream:

```
stream1 >> i;
```

This code reads a value from *stream1* and assigns the variable *i* that value.

Related topics:
C++ I/O Streams

rdbuf

Syntax:

```
#include <sstream>
stringbuf* rdbuf();
```

The *rdbuf()* function returns a pointer to the string buffer for the current string stream.

Related topics:
[str\(\)](#)
C++ I/O Streams

str**Syntax:**

```
#include <sstream>
void str( string s );
string str();
```

The function *str()* can be used in two ways. First, it can be used to get a copy of the string that is being manipulated by the current stream string. This is most useful with output strings. For example:

```
ostringstream stream1;
stream1 << "Testing!" << endl;
cout << stream1.str();
```

Second, *str()* can be used to copy a string into the stream. This is most useful with input strings. For example:

```
istringstream stream1;
string string1 = "25";
stream1.str(string1);
```

str(), along with *clear()*, is also handy when you need to clear the stream so that it can be reused:

```
istringstream stream1;
float num;
// use it once
string string1 = "25 1 3.235\n1111111\n222222";
stream1.str(string1);
while( stream1 >> num ) cout << "num: " << num << endl; // displays
numbers, one per line
// use the same string stream again with clear() and str()
string string2 = "1 2 3 4 5 6 7 8 9 10";
stream1.clear();
stream1.str(string2);
while( stream1 >> num ) cout << "num: " << num << endl; // displays
numbers, one per line
```

Related topics:

rdbuf()

C++ I/O Streams