C/C++ Reference

General C/C++

- Pre-processor commands
- Operator Precedence
- Escape Sequences
- ASCII Chart
- Data Types
- Keywords

Standard C Library

- Standard C I/O
- Standard C String & Character
- Standard C Math
- Standard C Date & Time
- Standard C Memory
- Other standard C functions

All C Functions

C++

- C++ I/O
- C++ Strings
- C++ String Streams
- Miscellaneous C++

C++ Standard Template Library

- About the Standard Template Library
- Iterators
- C++ Algorithms
- C++ Vectors
- C++ Double-Ended Queues
- C++ Lists
- C++ Priority Queues
- C++ Queues

- C++ Stacks
- C++ Sets
- C++ Multisets
- C++ Maps
- C++ Multimaps
- C++ Bitsets

All C++ Functions

Questions? Check out the FAQ, look at these other language references, or contact us.

Last modified on 6/23/2007 by Nate Kohl, with help from a lot of people.

All C Functions

#, ##	manipulate strings
#define	define variables
#error	display an error message
#if, #ifdef, #ifndef, #else, #elif, #endif	conditional operators
#include	insert the contents of another file
#line	set line and file information
#pragma	implementation specific command
#undef	used to undefine variables
Predefined preprocessor variables	miscellaneous preprocessor variables
abort	stops the program
abs	absolute value
acos	arc cosine
asctime	a textual version of the time
asin	arc sine
assert	stops the program if an expression isn't true
atan	arc tangent
atan2	arc tangent, using signs to determine quadrants
atexit	sets a function to be called when the program exits
atof	converts a string to a double
atoi	converts a string to an integer
atol	converts a string to a long
bsearch	perform a binary search
calloc	allocates and clears a two-dimensional chunk of memory
ceil	the smallest integer not less than a certain value
clearerr	clears errors
clock	returns the amount of time that the program has been running
cos	cosine
cosh	hyperbolic cosine
ctime	returns a specifically formatted version of the time
difftime	the difference between two times
div	returns the quotient and remainder of a division
exit	stop the program
exp	returns "e" raised to a given power
fabs	absolute value for floating-point numbers
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
floor	returns the largest integer not greater than a given value
fmod	returns the remainder of a division
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
free	returns previously allocated memory to the operating system
freopen	open an existing stream with a different name
frexp	decomposes a number into scientific notation
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**
getenv	get enviornment information about a variable
gets	read a string from **STDIN**
gmtime	returns a pointer to the current Greenwich Mean Time
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 hexidecimal character
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
localtime	returns a pointer to the current time
log	natural logarithm
log10	natural logarithm, in base 10
longjmp	start execution at a certain point in the program
malloc	allocates memory
memchr	searches an array for the first occurance 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
mktime	returns the calendar version of a given time
modf	decomposes a number into integer and fractional parts
perror	displays a string version of the current error to **STDERR**
pow	returns a given number raised to another number
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**
qsort	perform a quicksort
raise	send a signal to the program
rand	returns a pseudorandom number
realloc	changes the size of previously allocated memory
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
setjmp	set execution to start at a certain point
setlocale	sets the current locale
setvbuf	set the buffer and size for a specific stream
signal	register a function as a signal handler
sin	sine
sinh	hyperbolic sine
sprintf	write formatted output to a buffer
sqrt	square root
srand	initialize the random number generator
sscanf	read formatted input from a buffer
strcat	concatenates two strings
strchr	finds the first occurance 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
strftime	returns individual elements of the date and time
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 occurance of a character in a string
strspn	returns the length of a substring of characters of a string
strstr	finds the first occurance 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
system	perform a system call
tan	tangent
tanh	hyperbolic tangent
time	returns the current calendar time of the system
tmpfile	return a pointer to a temporary file
tmpnam	return a unique filename
tolower	converts a character to lowercase
toupper	converts a character to uppercase
ungetc	puts a character back into a stream
va_arg	use variable length parameter lists
vprintf, vfprintf, and vsprintf	write formatted output with variable argument lists

All C++ Functions

Bitset Constructors (C++ Bitsets)	create new bitsets
Bitset Operators (C++ Bitsets)	compare and assign bitsets
Vector constructors	create vectors and initialize them with some data
Container constructors (C++ Double-ended Queues)	create containers and initialize them with some data
Container constructors (C++ Lists)	create containers and initialize them with some data
Container constructors & destructors (C++ Sets)	default methods to allocate, copy, and deallocate containers
Container constructors & destructors (C++ Multisets)	default methods to allocate, copy, and deallocate multisets
Map constructors & destructors (C++ Maps)	default methods to allocate, copy, and deallocate maps
Multimap constructors & destructors (C++ Multimaps)	default methods to allocate, copy, and deallocate containers
Container operators (C++ Lists)	assign and compare containers
Container operators (C++ Sets)	assign and compare containers
Container operators (C++ Multisets)	assign and compare containers
Multimap operators (C++ Multimaps)	assign and compare containers
Vector operators	compare, assign, and access elements of a vector
Container operators (C++ Double-ended Queues)	compare, assign, and access elements of a container
I/O Constructors (C++ I/O)	constructors
Map operators (C++ Maps)	assign, compare, and access elements of a map
Priority queue constructors (C++ Priority Queues)	construct a new priority queue
Queue constructor (C++ Queues)	construct a new queue
Stack constructors (C++ Stacks)	construct a new stack
String constructors (C++ Strings)	create strings from arrays of characters and other strings
String operators (C++ Strings)	concatenate strings, assign strings, use strings for I/O, compare strings
accumulate (C++ Algorithms)	sum up a range of elements
adjacent_difference (C++ Algorithms)	compute the differences between adjacent elements in a range
adjacent_find (C++ Algorithms)	finds two items that are adjacent to eachother
any (C++ Bitsets)	true if any bits are set
append (C++ Strings)	append characters and strings onto a string
assign (C++ Vectors)	assign elements to a container
assign (C++ Double-ended Queues)	assign elements to a container
assign (C++ Lists)	assign elements to a container
assign (C++ Strings)	give a string values from strings of

compare (C++ Strings)	compares two strings
copy (C++ Strings)	copies characters from a string into an array
copy (C++ Algorithms)	copy some range of elements to a new location
copy_backward (C++ Algorithms)	copy a range of elements in backwards order
copy_n (C++ Algorithms)	copy N elements
count (C++ Sets)	returns the number of elements matching a certain key
count (C++ Multisets)	returns the number of elements matching a certain key
count (C++ Maps)	returns the number of elements matching a certain key
count (C++ Multimaps)	returns the number of elements matching a certain key
count (C++ Bitsets)	returns the number of set bits
count (C++ Algorithms)	return the number of elements matching a given value
count_if (C++ Algorithms)	return the number of elements for which a predicate is true
data (C++ Strings)	returns a pointer to the first character of a string
empty (C++ Strings)	true if the container has no elements
empty (C++ Vectors)	true if the container has no elements
empty (C++ Double-ended Queues)	true if the container has no elements
empty (C++ Lists)	true if the container has no elements
empty (C++ Sets)	true if the container has no elements
empty (C++ Multisets)	true if the container has no elements
empty (C++ Maps)	true if the container has no elements
empty (C++ Multimaps)	true if the container has no elements
empty (C++ Stacks)	true if the container has no elements
empty (C++ Queues)	true if the container has no elements
empty (C++ Priority Queues)	true if the container has no elements
end (C++ Strings)	returns an iterator just past the last element of a container
end (C++ Vectors)	returns an iterator just past the last element of a container
end (C++ Double-ended Queues)	returns an iterator just past the last element of a container
end (C++ Lists)	returns an iterator just past the last element of a container
end (C++ Sets)	returns an iterator just past the last element of a container
end (C++ Multisets)	returns an iterator just past the last element of a container
end (C++ Maps)	returns an iterator just past the last element of a container
end (C++ Multimaps)	returns an iterator just past the last

	element of a container
eof (C++ I/O)	true if at the end-of-file
equal (C++ Algorithms)	determine if two sets of elements are the same
equal_range (C++ Sets)	returns iterators to the first and just past the last elements matching a specific key
equal_range (C++ Multisets)	returns iterators to the first and just past the last elements matching a specific key
equal_range (C++ Maps)	returns iterators to the first and just past the last elements matching a specific key
equal_range (C++ Multimaps)	returns iterators to the first and just past the last elements matching a specific key
equal_range (C++ Algorithms)	search for a range of elements that are all equal to a certain element
erase (C++ Strings)	removes elements from a string
erase (C++ Vectors)	removes elements from a container
erase (C++ Double-ended Queues)	removes elements from a container
erase (C++ Lists)	removes elements from a container
erase (C++ Sets)	removes elements from a container
erase (C++ Multisets)	removes elements from a container
erase (C++ Maps)	removes elements from a container
erase (C++ Multimaps)	removes elements from a container
fail (C++ I/O)	true if an error occurred
fill (C++ I/O)	manipulate the default fill character
fill (C++ Algorithms)	assign a range of elements a certain value
fill_n (C++ Algorithms)	assign a value to some number of elements
find (C++ Algorithms)	find a value in a given range
find (C++ Sets)	returns an iterator to specific elements
find (C++ Multisets)	returns an iterator to specific elements
find (C++ Maps)	returns an iterator to specific elements
find (C++ Multimaps)	returns an iterator to specific elements
find (C++ Strings)	find characters in the string
find_end (C++ Algorithms)	find the last sequence of elements in a certain range
find_first_not_of (C++ Strings)	find first absence of characters
find_first_of (C++ Strings)	find first occurrence of characters
find_first_of (C++ Algorithms)	search for any one of a set of elements
find_if (C++ Algorithms)	find the first element for which a certain predicate is true
find_last_not_of (C++ Strings)	find last absence of characters
find_last_of (C++ Strings)	find last occurrence of characters
flags (C++ I/O)	access or manipulate io stream format flag s
flip (C++ Bitsets)	reverses the bitset
flush (C++ I/O)	empty the buffer
for_each (C++ Algorithms)	apply a function to a range of elements
front (C++ Vectors)	returns a reference to the first element of a

	container
front (C++ Double-ended Queues)	returns a reference to the first element of a container
front (C++ Lists)	returns a reference to the first element of a container
front (C++ Queues)	returns a reference to the first element of a container
gcount (C++ I/O)	number of characters read during last input
generate (C++ Algorithms)	saves the result of a function in a range
generate_n (C++ Algorithms)	saves the result of N applications of a function
get (C++ I/O)	read characters
getline (C++ I/O)	read a line of characters
getline (C++ Strings)	read data from an I/O stream into a string
good (C++ I/O)	true if no errors have occurred
ignore (C++ I/O)	read and discard characters
includes (C++ Algorithms)	returns true if one set is a subset of another
inner_product (C++ Algorithms)	compute the inner product of two ranges of elements
inplace_merge (C++ Algorithms)	merge two ordered ranges in-place
insert (C++ Strings)	insert characters into a string
insert (C++ Vectors)	inserts elements into the container
insert (C++ Double-ended Queues)	inserts elements into the container
insert (C++ Lists)	inserts elements into the container
insert (C++ Sets)	insert items into a container
insert (C++ Multisets)	inserts items into a container
insert (C++ Multimaps)	inserts items into a container
insert (C++ Maps)	insert items into a container
is_heap (C++ Algorithms)	returns true if a given range is a heap
is_sorted (C++ Algorithms)	returns true if a range is sorted in ascending order
iter_swap (C++ Algorithms)	swaps the elements pointed to by two iterators
key_comp (C++ Sets)	returns the function that compares keys
key_comp (C++ Multisets)	returns the function that compares keys
key_comp (C++ Maps)	returns the function that compares keys
key_comp (C++ Multimaps)	returns the function that compares keys
length (C++ Strings)	returns the length of the string
lexicographical_compare (C++ Algorithms)	returns true if one range is lexicographically less than another
lexicographical_compare_3way (C++ Algorithms)	determines if one range is lexicographically less than or greater than another
lower_bound (C++ Sets)	returns an iterator to the first element greater than or equal to a certain value
lower_bound (C++ Multisets)	returns an iterator to the first element

	greater than or equal to a certain value
lower_bound (C++ Maps)	returns an iterator to the first element greater than or equal to a certain value
lower_bound (C++ Multimaps)	returns an iterator to the first element greater than or equal to a certain value
lower_bound (C++ Algorithms)	search for the first place that a value can be inserted while preserving order
make_heap (C++ Algorithms)	creates a heap out of a range of elements
max (C++ Algorithms)	returns the larger of two elements
max_element (C++ Algorithms)	returns the largest element in a range
max_size (C++ Strings)	returns the maximum number of elements that the container can hold
max_size (C++ Vectors)	returns the maximum number of elements that the container can hold
max_size (C++ Double-ended Queues)	returns the maximum number of elements that the container can hold
max_size (C++ Lists)	returns the maximum number of elements that the container can hold
max_size (C++ Sets)	returns the maximum number of elements that the container can hold
max_size (C++ Multisets)	returns the maximum number of elements that the container can hold
max_size (C++ Maps)	returns the maximum number of elements that the container can hold
max_size (C++ Multimaps)	returns the maximum number of elements that the container can hold
merge (C++ Lists)	merge two lists
merge (C++ Algorithms)	merge two sorted ranges
min (C++ Algorithms)	returns the smaller of two elements
min_element (C++ Algorithms)	returns the smallest element in a range
mismatch (C++ Algorithms)	finds the first position where two ranges differ
next_permutation (C++ Algorithms)	generates the next greater lexicographic permutation of a range of elements
none (C++ Bitsets)	true if no bits are set
nth_element (C++ Algorithms)	put one element in its sorted location and make sure that no elements to its left are greater than any elements to its right
open (C++ I/O)	create an input stream
partial_sort (C++ Algorithms)	sort the first N elements of a range
partial_sort_copy (C++ Algorithms)	copy and partially sort a range of elements
partial_sum (C++ Algorithms)	compute the partial sum of a range of elements
partition (C++ Algorithms)	divide a range of elements into two groups
peek (C++ I/O)	check the next input character
pop (C++ Stacks)	removes the top element of a container
pop (C++ Queues)	removes the top element of a container
pop (C++ Priority Queues)	removes the top element of a container

set (C++ Bitsets)	sets bits
set_difference (C++ Algorithms)	computes the difference between two sets
set_intersection (C++ Algorithms)	computes the intersection of two sets
set_symmetric_difference (C++ Algorithms)	computes the symmetric difference between two sets
set_union (C++ Algorithms)	computes the union of two sets
setf (C++ I/O)	set format flags
size (C++ Strings)	returns the number of items in the container
size (C++ Vectors)	returns the number of items in the container
size (C++ Double-ended Queues)	returns the number of items in the container
size (C++ Lists)	returns the number of items in the container
size (C++ Sets)	returns the number of items in the container
size (C++ Multisets)	returns the number of items in the container
size (C++ Maps)	returns the number of items in the container
size (C++ Multimaps)	returns the number of items in the container
size (C++ Stacks)	returns the number of items in the container
size (C++ Queues)	returns the number of items in the container
size (C++ Priority Queues)	returns the number of items in the container
size (C++ Bitsets)	number of bits that the bitset can hold
sort (C++ Lists)	sorts a list into ascending order
sort (C++ Algorithms)	sort a range into ascending order
sort_heap (C++ Algorithms)	turns a heap into a sorted range of elements
splice (C++ Lists)	merge two lists in constant time
stable_partition (C++ Algorithms)	divide elements into two groups while preserving their relative order
stable_sort (C++ Algorithms)	sort a range of elements while preserving order between equal elements
substr (C++ Strings)	returns a certain substring
swap (C++ Strings)	swap the contents of this container with another
swap (C++ Vectors)	swap the contents of this container with another
swap (C++ Double-ended Queues)	swap the contents of this container with another
swap (C++ Lists)	swap the contents of this container with another
swap (C++ Sets)	swap the contents of this container with

	another
swap (C++ Multisets)	swap the contents of this container with another
swap (C++ Maps)	swap the contents of this container with another
swap (C++ Multimaps)	swap the contents of this container with another
swap (C++ Algorithms)	swap the values of two objects
swap_ranges (C++ Algorithms)	swaps two ranges of elements
sync_with_stdio (C++ I/O)	synchronize with standard I/O
tellg (C++ I/O)	read input stream pointers
tellp (C++ I/O)	read output stream pointers
test (C++ Bitsets)	returns the value of a given bit
to_string (C++ Bitsets)	string representation of the bitset
to_ulong (C++ Bitsets)	returns an integer representation of the bitset
top (C++ Stacks)	returns the top element of the container
top (C++ Priority Queues)	returns the top element of the container
transform (C++ Algorithms)	applies a function to a range of elements
unique (C++ Lists)	removes consecutive duplicate elements
unique (C++ Algorithms)	remove consecutive duplicate elements in a range
unique_copy (C++ Algorithms)	create a copy of some range of elements that contains no consecutive duplicates
unsetf (C++ I/O)	clear io stream format flags
upper_bound (C++ Sets)	returns an iterator to the first element greater than a certain value
upper_bound (C++ Multisets)	returns an iterator to the first element greater than a certain value
upper_bound (C++ Maps)	returns an iterator to the first element greater than a certain value
upper_bound (C++ Multimaps)	returns an iterator to the first element greater than a certain value
upper_bound (C++ Algorithms)	searches for the last possible location to insert an element into an ordered range
value_comp (C++ Sets)	returns the function that compares values
value_comp (C++ Multisets)	returns the function that compares values
value_comp (C++ Maps)	returns the function that compares values
value_comp (C++ Multimaps)	returns the function that compares values
width (C++ I/O)	access and manipulate the minimum field width
write (C++ I/O)	write characters

Frequently Asked Questions

Can I get a copy of this site?

We do provide a downloadable archived version of cppreference.com. If you're interested in getting archived versions of websites in general, you might want to check out utilities like GNU's wget (Windows version here).

Can I translate this site to some other language?

Sure, that would be great! All that we would ask is that you include a link back to this site so that people know where to get the most up-to-date content.

Who is this site meant for?

There are no "Introduction to Programming" tutorials here. This site is meant to be used by more-or-less experienced C++ programmers, who have a good idea of what they want to do and simply need to look up the syntax. If you're interested in learning C/C++, try one of these sites:

- How C Programming Works
- C Programming
- C++ Language Tutorial

Does this site contain a complete and definitive list of C/C++ functions?

Few things in life are absolute. If you don't find what you are looking for here, don't assume that it doesn't exist. Do a search on Google for it.

Some of the examples on this site don't work on my system. What's going on?

Most of the code on this site was compiled under Linux (Red Hat, Debian, or Ubuntu) with the GNU Compiler Collection. Since this site is merely a reference for the Standard C and C++ specification , not every compiler will support every function listed here. For example,

- Header files change like mad. To include the necessary support for C++ Vectors, you might have to use any of these:

```
#include <vector>
#include <Vector>
#include <vector.h>
```

  (according to the spec, the first of those should work, and the compiler should know enough to use it to reference the real vector header file.)
- Another header file issue is that newer compilers can use a more platform-independent commands to include standard C libraries. For example, you might be able to use

```
#include <cstdio>
```

  instead of

```
#include <stdio.h>
```

- All of the code on this site assumes that the correct namespace has been designated. If your compiler is a little old, then you might be able to get away with using simple statements like:

```
cout << "hello world!";
```

  However, newer compilers require that you either use

```
std::cout << "hello world!";
```

or declare what namespace to use with the "using namespace" command.

- Certain popular compilers (like the one shipped with Microsoft's Visual C++) have added alternative or additional functionality to the C++ Standard Template Library. For example, the MFC in Visual C++ provides you with the string type "CString", which has string functionality but is not part of the C++ STL.

...The list goes on and on. In other words, individual results may vary.

You've got an error in this site.

If you find any errors in this reference, please feel free to contact us -- feedback and code examples are always welcome.

What's up with this site?

Think of it as a community service, for geeks.

Huge thanks to all these people for sending in bug fixes and suggestions on how to improve the site:

Alex Vinokur - Ted Felix

A.J.M. van den Berg - Adrian Pfisterer - Alex Wilson - Alexandre Kostine - Andre - Andre Gillibert - Andrew L Roth - Annamalai Gurusami - Art Stamness - Arvid Norberg - Benjamin Lee Hansen - Brian Higgins - Brian T Stadler - Carl - Cedric Blaser - Chip Lemon - Chris Frey - Chris H - Chris Rimmer - Chris Yate - Christian Foerg - Christoffer Nyborg - Christoph Otto - Christoph Vogelbusch - Claudio Alberto Andreoni - Colin Hirsch - Damian (doublenegative) - Dan Stronger - Daniel Fish - Daniel Goering - Daniel Lorch - Darsh Ranjan - Dave Schuyler - Dave T - David E Freitas - Davoud Taghawi-Nejad - Deepak Goyal - Devin Pratt - Diggory Hardy - Drew Dormann - Dzu Nguyen - E.Guadalupe - Edgardo Rossetto - Eirik Stangeland - Emmanuel Viaud - Enrique Pineda - Eric Kinser - Erik Aas - Erik Wikstrom - Florian Schaper - Fred Ma - Frederik Hertzum - Gerhard Grossauer - guiliano - Guillaume Bouchez - Hasan Amjad - Henning Diedrich - Henrik Huttunen - Henrik Mattsson - Iain Staffell - Imre Pentek - JP (Pete) Donnell - James Bliese - James Dennett - James Jones - Jan - Jann Poppinga - Jari Karppinen - Jeff Bowden - Jeff Dwork - Jeroen Missinne - Jonathan Dent - Jonathan Kleid - Joseph Bruni - Joshua Haberman - Joshua R. Warr - Justin M. Lee - Katherine Haines - Keith Knapp - Ken Sedgwick - Kien Nguyen - Kiyoshi Aman - Kuang-che Wu - Kwan Ting Chan - Kurt McKee - Leor Zolman - Lucas Fisher - Mael Herz - Magnus Kulke - Manish Malik - Martin - Martin Milata - Martin Richardt - Martijn van de Giessen - Matthias Britsch - Matthias Hofmann - Matthias Neeracher - Mike Angstadt - Mike Clarke - Mike Ekoka - Mike Jennings - Milan Mimica - Moonrie - Nadia De Bode - Nate Silva - Neelesh Bodas - Nick Gianakas - Nicolas Boichat - Olivier Ricou - Onur Tugcu - Osku Salerma - Patrick Spendrin - Paul Fee - Paul L. Tomlinson - Philip Dunstan - Phillip Lee - Piers Daniell - Ralf Denzer - Randall Rathbun - Rex Kerr - Rob Larkins - Rodrigo Cesar Dias - Roger D Pack - Romans Kasperovics - Salman Mahbub - Selim T. Erdogan - Sergio Martinez - Shibukawa Yoshiki - Simon Perkins - snlee - Stefan Suffa - Stefan Voegel - Steve Ward - Tarjei Knapstad - Tetra - Thomas Volk - Tiaan van Aardt - Tom (prkchp) - Tor Husab - Tyler Cole - Vegard Nossum - Victor Rachels - Vijay S. - William Charles Deich IV - William Dye - William K. Austad - Wouter Lievens - XenteX

Thank you!

Links

Here are some links to other language references:

- C++ (Dinkumware)
- C++ Language and Library
- Java 1.5 (Sun)
- MySQL
- Perl
- Python
- Ruby
- Tcl
- Visual C++ STL (Microsoft)

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.

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

2	()
	[]
	->
	.
	++
	--
		Grouping operator
		Array access
		Member access from a pointer
		Member access from an object
		Post-increment
		Post-decrement
			(a + b) / 4;
			array[4] = 2;
			ptr->age = 34;
			obj.age = 34;
			for(i = 0; i < 10; i++) ...
			for(i = 10; i > 0; i--) ...
				left to right

3	!
	~
	++
	--
	-
	+
	*
	&
	(type)
	sizeof
		Logical negation
		Bitwise complement
		Pre-increment
		Pre-decrement
		Unary minus
		Unary plus
		Dereference
		Address of
		Cast to a given type
		Return size in bytes
			if(!done) ...
			flags = ~flags;
			for(i = 0; i < 10; ++i) ...
			for(i = 10; i > 0; --i) ...
			int i = -1;
			int i = +1;
			data = *ptr;
			address = &obj;
			int i = (int) floatNum;
			int size = sizeof(floatNum);
				right to left

4	->* .*	Member pointer selector Member object selector	ptr->*var = 24; obj.*var = 24;	left to right

5	*
	/
	%
		Multiplication
		Division
		Modulus
			int i = 2 * 4;
			float f = 10 / 3;
			int rem = 4 % 3;
				left to right

6	+ -	Addition Subtraction	int i = 2 + 3; int i = 5 - 1;	left to right
7	<< >>	Bitwise shift left Bitwise shift right	int flags = 33 << 1; int flags = 33 >> 1;	left to right

8	<
	<=
	>
	>=
		Comparison less-than
		Comparison less-than-or-equal-to
			if(i < 42) ...
			if(i <= 42) ...
			if(i > 42) ...
			if(i >= 42) ...
				left to right

		Comparison greater-than Comparison geater-than-or-equal-to		
9	== !=	Comparison equal-to Comparison not-equal-to	if(i == 42) ... if(i != 42) ...	left to right
10	&	Bitwise AND	flags = flags & 42;	left to right
11	^	Bitwise exclusive OR	flags = flags ^ 42;	left to right
12	|	Bitwise inclusive (normal) OR	flags = flags | 42;	left to right
13	&&	Logical AND	if(conditionA && conditionB) ...	left to right
14	||	Logical OR	if(conditionA || conditionB) ...	left to right
15	? :	Ternary conditional (if-then-else)	int i = (a > b) ? a : b;	right to left

16	=
	+=
	-=
	*=
	/=
	%=
	&=
	^=
	|=
	<<=
	>>=
		Assignment operator
		Increment and assign
		Decrement and assign
		Multiply and assign
		Divide and assign
		Modulo and assign
		Bitwise AND and assign
		Bitwise exclusive OR and assign
		Bitwise inclusive (normal) OR and assign
		Bitwise shift left and assign
		Bitwise shift right and assign
			int a = b;
			a += 3;
			b -= 4;
			a *= 5;
			a /= 2;
			a %= 3;
			flags &= new_flags;
			flags ^= new_flags;
			flags |= new_flags;
			flags <<= 2;
			flags >>= 2;
				right to left

17	,	Sequential	for(i = 0, j = 0; i <	left to right

		evaluation operator	10; i++, j++) ...	

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.**

Watch out.

This function returns a variable that is statically located, and therefore overwritten each time this function is called. If you want to save the return value of this function, you should manually save it elsewhere.

Of course, when you save it elsewhere, you should make sure to actually copy the value(s) of this variable to another location. If the return value is a struct, you should make a new struct, then copy over the members of the struct.

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?"
```

C++ I/O Flags

Format flags

C++ defines some format flags for standard input and output, which can be manipulated with the flags(), setf(), and unsetf() functions. For example,

```
cout.setf(ios::left);
```

turns on left justification for all output directed to **cout**.

Flag	Meaning
boolalpha	Boolean values can be input/output using the words "true" and "false".
dec	Numeric values are displayed in decimal.
fixed	Display floating point values using normal notation (as opposed to scientific).
hex	Numeric values are displayed in hexidecimal.
internal	If a numeric value is padded to fill a field, spaces are inserted between the sign and base character.
left	Output is left justified.
oct	Numeric values are displayed in octal.
right	Output is right justified.
scientific	Display floating point values using scientific notation.
showbase	Display the base of all numeric values.
showpoint	Display a decimal and extra zeros, even when not needed.
showpos	Display a leading plus sign before positive numeric values.
skipws	Discard whitespace characters (spaces, tabs, newlines) when reading from a stream.
unitbuf	Flush the buffer after each insertion.
uppercase	Display the "e" of scientific notation and the "x" of hexidecimal notation as capital letters.

Manipulators

You can also manipulate flags indirectly, using the following *manipulators*. Most programmers are familiar with the **endl** manipulator, which might give you an idea of how manipulators are used. For example, to set the *dec* flag, you might use the following command:

```
cout << dec;
```

Manipulators defined in <iostream>

Manipulator	Description	Input	Output
boolalpha	Turns on the boolalpha flag	X	X
dec	Turns on the dec flag	X	X
endl	Output a newline character, flush the stream		X
ends	Output a null character		X
fixed	Turns on the fixed flag		X
flush	Flushes the stream		X
hex	Turns on the hex flag	X	X
internal	Turns on the internal flag		X
left	Turns on the left flag		X
noboolalpha	Turns off the boolalpha flag	X	X
noshowbase	Turns off the showbase flag		X

noshowpoint	Turns off the showpoint flag		X
noshowpos	Turns off the showpos flag		X
noskipws	Turns off the skipws flag	X	
nounitbuf	Turns off the unitbuf flag		X
nouppercase	Turns off the uppercase flag		X
oct	Turns on the oct flag	X	X
right	Turns on the right flag		X
scientific	Turns on the scientific flag		X
showbase	Turns on the showbase flag		X
showpoint	Turns on the showpoint flag		X
showpos	Turns on the showpos flag		X
skipws	Turns on the skipws flag	X	
unitbuf	Turns on the unitbuf flag		X
uppercase	Turns on the uppercase flag		X
ws	Skip any leading whitespace	X	

Manipulators defined in <iomanip>

Manipulator	**Description**	**Input**	**Output**
resetiosflags(long f)	Turn off the flags specified by *f*	X	X
setbase(int base)	Sets the number base to *base*		X
setfill(int ch)	Sets the fill character to *ch*		X
setiosflags(long f)	Turn on the flags specified by *f*	X	X
setprecision(int p)	Sets the number of digits of precision		X
setw(int w)	Sets the field width to *w*		X

State flags

The I/O stream state flags tell you the current state of an I/O stream. The flags are:

Flag	**Meaning**
badbit	a fatal error has occurred
eofbit	EOF has been found
failbit	a nonfatal error has occurred
goodbit	no errors have occurred

Mode flags

The I/O stream mode flags allow you to access files in different ways. The flags are:

Mode	**Meaning**
ios::app	append output
ios::ate	seek to EOF when opened
ios::binary	open the file in binary mode
ios::in	open the file for reading
ios::out	open the file for writing
ios::trunc	overwrite the existing file

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 seperator
29	35	1D	GS	group seperator
30	36	1E	RS	record seperator
31	37	1F	US	unit seperator
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

Complexity

There are different measurements of the speed of any given algorithm. Given an input size of **N**, they can be described as follows:

Name	Speed	Description	Formula
exponential time	slow	takes an amount of time proportional to a constant raised to the **N**th power	K^**N**
polynomial time	fast	takes an amount of time proportional to **N** raised to some constant power	**N**^K
linear time	faster	takes an amount of time directly proportional to **N**	K * **N**
logarithmic time	much faster	takes an amount of time proportional to the logarithm of **N**	K * log(**N**)
constant time	fastest	takes a fixed amount of time, no matter how large the input is	K

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.

Here are some examples of what you can do with C++ I/O.

Display all entries for C++ I/O on one page, or view entries individually:

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

tellp	read output stream pointers
unsetf	clear io stream format flags
width	access and manipulate the minimum field width
write	write characters

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:

eof
fail
good
rdstate

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

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);
}
```

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:

eof
fail
good
rdstate

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.

Example code:

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:

eof
fail
good
rdstate

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
clear
eof
good
rdstate

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:
gcount
get
(C++ Strings) getline
ignore
read

good

Syntax:

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

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

Related topics:

bad
clear
eof
fail
rdstate

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++ Strings

Display all entries for C++ Strings on one page, or view entries individually:

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 an element 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 elements that the string can hold
clear	removes all elements 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 elements
end	returns an iterator just past the last element of a string
erase	removes elements 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 elements that the string can hold
push_back	add an element 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

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* repititions 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.
```

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 characterss 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 these normal (C++ Multimaps) Multimap 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++ Multimaps) Multimap operators
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* repititions 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 deafult 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 eachother:

```
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 eachother:

	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 hypen:

```
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 = "abcdefghijklmnopqrstuvwxyz";
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_first_not_of
find_first_of
find_last_not_of
find_last_of

size

Syntax:

```
#include <string>
size_type size() const;
```

The size() function returns the number of elements in the current string.

Related topics:

capacity
empty
length
max_size
resize

substr

Syntax:

```
#include <string>
string substr( size_type index, size_type num = npos );
```

The substr() function returns a substring of the current string, starting at *index*, and *num* characters long. If *num* 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 sscanf() 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.

Display all entries for C++ String Streams on one page, or view entries individually:

Constructors	create new string streams
Operators	read from and write to string streams
rdbuf	get the buffer for a string stream
str	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 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

value_comp

Syntax:

```
#include <set>
value_compare value_comp() const;
```

The value_comp() function returns the function that compares values.

value_comp() runs in constant time.

Related topics:

key_comp

Miscellaneous C++

Display all entries for Miscellaneous C++ on one page, or view entries individually:

auto_ptr	create pointers that automatically destroy objects

auto_ptr

Syntax:

```
#include <memory>
auto_ptr<class TYPE> name
```

The auto_ptr class allows the programmer to create pointers that point to other objects. When auto_ptr pointers are destroyed, the objects to which they point are also destroyed.

The auto_ptr class supports normal pointer operations like =, *, and ->, as well as two functions TYPE* get() and TYPE* release(). The get() function returns a pointer to the object that the auto_ptr points to. The release() function acts similarily to the get() function, but also relieves the auto_ptr of its memory destruction duties. When an auto_ptr that has been released goes out of scope, it will not call the destructor of the object that it points to.

Warning: It is generally a **bad idea** to put auto_ptr objects inside C++ STL containers. C++ containers can do funny things with the data inside them, including frequent reallocation (when being copied, for instance). Since calling the destructor of an auto_ptr object will free up the memory associated with that object, any C++ container reallocation will cause any auto_ptr objects to become invalid.

Example code:

```
#include <memory>
using namespace std;

class MyClass {
public:
  MyClass() {} // nothing
  ~MyClass() {} // nothing
  void myFunc() {} // nothing
};

int main() {
  auto_ptr<MyClass> ptr1(new MyClass), ptr2;

  ptr2 = ptr1;
  ptr2->myFunc();

  MyClass* ptr = ptr2.get();

  ptr->myFunc();

  return 0;
}
```

auto_ptr

Syntax:

```
#include <memory>
auto_ptr<class TYPE> name
```

The auto_ptr class allows the programmer to create pointers that point to other objects. When auto_ptr pointers are destroyed, the objects to which they point are also destroyed.

The auto_ptr class supports normal pointer operations like =, *, and ->, as well as two functions TYPE* get() and TYPE* release(). The get() function returns a pointer to the object that the auto_ptr points to. The release() function acts similarily to the get() function, but also relieves the auto_ptr of its memory destruction duties. When an auto_ptr that has been released goes out of scope, it will not call the destructor of the object that it points to.

Warning: It is generally a **bad idea** to put auto_ptr objects inside C++ STL containers. C++ containers can do funny things with the data inside them, including frequent reallocation (when being copied, for instance). Since calling the destructor of an auto_ptr object will free up the memory associated with that object, any C++ container reallocation will cause any auto_ptr objects to become invalid.

Example code:

```
#include <memory>
using namespace std;

class MyClass {
public:
  MyClass() {} // nothing
  ~MyClass() {} // nothing
  void myFunc() {} // nothing
};

int main() {
  auto_ptr<MyClass> ptr1(new MyClass), ptr2;

  ptr2 = ptr1;
  ptr2->myFunc();

  MyClass* ptr = ptr2.get();

  ptr->myFunc();

  return 0;
}
```

Standard C I/O

Display all entries for Standard C I/O on one page, or view entries individually:

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

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
ferror
getc
perror
putc

ferror

Syntax:

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

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

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:
fgets
fprintf
fscanf
gets
puts

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:

fflush
fgetc
fopen
fputc
fscanf
fwrite
getc

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:
fclose
fopen

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
fgetc
fopen
fputc
fread
fscanf
getc

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:
fgetc
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
%c	character
%d	signed integers
%i	signed integers
%e	scientific notation, with a lowercase "e"
%E	scientific notation, with a uppercase "E"
%f	floating point
%g	use %e or %f, whichever is shorter
%G	use %E or %f, whichever is shorter
%o	octal
%s	a string of characters
%u	unsigned integer
%x	unsigned hexadecimal, with lowercase letters
%X	unsigned hexadecimal, with uppercase letters
%p	a pointer
%n	the argument shall be a pointer to an integer into which is placed the number of characters written so far
%%	a '%' 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 modifer simply acts as a maximumfield 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 hexidecimal 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:

feof
fflush
fgetc
fputc
getc
getchar
putchar
puts

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
%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 repeatedly uses scanf() to read integers and floats from the user. Note that the variable arguments to scanf() are passed in by reference, as denoted by the ampersand (&) preceding each variable:

```
int i;
float f;

while( 1 ) {
  printf( "Enter an integer: " );
  scanf( "%d", &i );
```

```
  printf( "Enter a float: " );
  scanf( "%f", &f );

  printf( "You entered %d and then %f\n", i, f );
}
```

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 tempfile() 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

Display all entries for Standard C String and Character on one page, or view entries individually:

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 hexidecimal character
memchr	searches an array for the first occurance 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 occurance 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 occurance of a character in a string
strspn	returns the length of a substring of characters of a string
strstr	finds the first occurance 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:
atoi
atol
(Standard C I/O) sprintf
strtod

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:
atoi
atol
(Standard C I/O) sprintf
strtod

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:
atof
atol
(Standard C I/O) sprintf

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
iscntrl
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
iscntrl
isdigit
isgraph
isprint
ispunct
isspace
isxdigit

iscntrl

Syntax:

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

The iscntrl() 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
iscntrl
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
iscntrl
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
isalpha
iscntrl
isdigit
isgraph
ispunct
isspace

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
isalpha
iscntrl
isdigit
isgraph
isprint
isspace
isxdigit

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
isalpha
iscntrl
isdigit
isgraph
isprint
ispunct
isxdigit

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 hexidecimal digit (i.e. A-F, a-f, or 0-9). Otherwise, zero is returned.

Related topics:

isalnum
isalpha
iscntrl
isdigit
isgraph
ispunct
isspace

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 intializing 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 occurence 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
strcat
strchr
strcoll
strcpy
strlen
strncmp
strxfrm

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:

memcpy
strcat
strchr
strcmp
strncmp
strncpy

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

strpbrk

Syntax:

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

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

Related topics:
(C++ Algorithms) find_first_of
strchr
strcspn
strrchr
strspn
strstr
strtok

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:

strcspn
strpbrk
strspn
strstr
strtok

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:
strchr
strpbrk
strrchr
strstr
strtok

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
strrchr
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
strcspn
strpbrk
strrchr
strspn
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

Display all entries for Standard C Math on one page, or view entries individually:

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." << endl;
```

Related topics:

fabs
labs

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." << endl;
```

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>
div_t div( int numerator, int denominator );
```

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: **a**div_t, **div_t**, **l**div_t.

The ldiv() function returns the quotient and remainder of the operation *numerator* / *denominator*. The **l**div_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

Display all entries for Standard C Date & Time on one page, or view entries individually:

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

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:

asctime
ctime
difftime
localtime
mktime
strftime
time

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:

asctime
ctime
difftime
gmtime
strftime
time

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:
asctime
ctime
gmtime
time

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

Display all entries for Standard C Memory on one page, or view entries individually:

calloc	allocates and clears a two-dimensional chunk of memory
free	returns previously allocated memory to the operating system
malloc	allocates memory
realloc	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

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

Display all entries for Other Standard C Functions on one page, or view entries individually:

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 enviornment 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

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 recieved by your program. *func* can be a custom signal handler, or one of these macros (defined in signal.h):

Macro	Explanation
SIG_DFL	default signal handling
SIG_IGN	ignore the signal

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

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

The return value of signal() is the address of the previously defined function for this signal, or SIG_ERR is 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++ Standard Template Library

The C++ STL (Standard Template Library) is a generic collection of class templates and algorithms that allow programmers to easily implement standard data structures like queues, lists, and stacks.

The C++ STL provides programmers with the following constructs, grouped into three categories:

- Sequences
  - C++ Vectors
  - C++ Lists
  - C++ Double-Ended Queues
- Container Adapters
  - C++ Stacks
  - C++ Queues
  - C++ Priority Queues
- Associative Containers
  - C++ Bitsets
  - C++ Maps
  - C++ Multimaps
  - C++ Sets
  - C++ Multisets

The idea behind the C++ STL is that the hard part of using complex data structures has already been completed. If a programmer would like to use a stack of integers, all that she has to do is use this code:

```
stack<int> myStack;
```

With minimal effort, she can now push() and pop() integers onto this stack. Through the magic of C++ Templates, she could specify any data type, not just integers. The STL Stack class will provide generic functionality of a stack, regardless of the data in the stack.

C++ Containers

The C++ Containers (vectors, lists, etc.) are generic vessels capable of holding many different types of data. For example, the following statement creates a vector of integers:

```
vector<int> v;
```

Containers can hold standard objects (like the **int** in the above example) as well as custom objects, as long as the objects in the container meet a few requirements:

- The object must have a default constructor,
- an accessible destructor, and
- an accessible assignment operator.

When describing the functions associated with these various containers, this website defines the word **TYPE** to be the object type that the container holds. For example, in the above statement, **TYPE** would be **int**. Similarily, when referring to containers associated with pairs of data (map for example) **key_type** and **value_type** are used to refer to the key and value types for that container.

C++ Iterators

Iterators are used to access members of the container classes, and can be used in a similar manner to pointers. For example, one might use an iterator to step through the elements of a vector. There are several different types of iterators:

Iterator	Description
input_iterator	Read values with forward movement. These can be incremented, compared, and dereferenced.
output_iterator	Write values with forward movement. These can be incremented and dereferenced.
forward_iterator	Read or write values with forward movement. These combine the functionality of input and output iterators with the ability to store the iterators value.
bidirectional_iterator	Read and write values with forward and backward movement. These are like the forward iterators, but you can increment and decrement them.
random_iterator	Read and write values with random access. These are the most powerful iterators, combining the functionality of bidirectional iterators with the ability to do pointer arithmetic and pointer comparisons.
reverse_iterator	Either a random iterator or a bidirectional iterator that moves in reverse direction.

Each of the container classes is associated with a type of iterator, and each of the STL algorithms uses a certain type of iterator. For example, vectors are associated with **random-access iterators**, which means that they can use algorithms that require random access. Since random-access iterators encompass all of the characteristics of the other iterators, vectors can use algorithms designed for other iterators as well.

The following code creates and uses an iterator with a vector:

```
vector<int> the_vector;
vector<int>::iterator the_iterator;
for( int i=0; i < 10; i++ )
  the_vector.push_back(i);
int total = 0;
the_iterator = the_vector.begin();
while( the_iterator != the_vector.end() ) {
  total += *the_iterator;
  the_iterator++;
}
cout << "Total=" << total << endl;
```

Notice that you can access the elements of the container by dereferencing the iterator.

C++ Algorithms

Display all entries for C++ Algorithms on one page, or view entries individually:

accumulate	sum up a range of elements
adjacent_difference	compute the differences between adjacent elements in a range
adjacent_find	finds two items that are adjacent to eachother
binary_search	determine if an element exists in a certain range
copy	copy some range of elements to a new location
copy_backward	copy a range of elements in backwards order
copy_n	copy N elements
count	return the number of elements matching a given value
count_if	return the number of elements for which a predicate is true
equal	determine if two sets of elements are the same
equal_range	search for a range of elements that are all equal to a certain element
fill	assign a range of elements a certain value
fill_n	assign a value to some number of elements
find	find a value in a given range
find_end	find the last sequence of elements in a certain range
find_first_of	search for any one of a set of elements
find_if	find the first element for which a certain predicate is true
for_each	apply a function to a range of elements
generate	saves the result of a function in a range
generate_n	saves the result of N applications of a function
includes	returns true if one set is a subset of another
inner_product	compute the inner product of two ranges of elements
inplace_merge	merge two ordered ranges in-place
is_heap	returns true if a given range is a heap
is_sorted	returns true if a range is sorted in ascending order
iter_swap	swaps the elements pointed to by two iterators
lexicographical_compare	returns true if one range is lexicographically less than another
lexicographical_compare_3way	determines if one range is lexicographically less than or greater than another
lower_bound	search for the first place that a value can be inserted while preserving order
make_heap	creates a heap out of a range of elements
max	returns the larger of two elements
max_element	returns the largest element in a range

merge	merge two sorted ranges
min	returns the smaller of two elements
min_element	returns the smallest element in a range
mismatch	finds the first position where two ranges differ
next_permutation	generates the next greater lexicographic permutation of a range of elements
nth_element	put one element in its sorted location and make sure that no elements to its left are greater than any elements to its right
partial_sort	sort the first N elements of a range
partial_sort_copy	copy and partially sort a range of elements
partial_sum	compute the partial sum of a range of elements
partition	divide a range of elements into two groups
pop_heap	remove the largest element from a heap
prev_permutation	generates the next smaller lexicographic permutation of a range of elements
push_heap	add an element to a heap
random_sample	randomly copy elements from one range to another
random_sample_n	sample N random elements from a range
random_shuffle	randomly re-order elements in some range
remove	remove elements equal to certain value
remove_copy	copy a range of elements omitting those that match a certian value
remove_copy_if	create a copy of a range of elements, omitting any for which a predicate is true
remove_if	remove all elements for which a predicate is true
replace	replace every occurrence of some value in a range with another value
replace_copy	copy a range, replacing certain elements with new ones
replace_copy_if	copy a range of elements, replacing those for which a predicate is true
replace_if	change the values of elements for which a predicate is true
reverse	reverse elements in some range
reverse_copy	create a copy of a range that is reversed
rotate	move the elements in some range to the left by some amount
rotate_copy	copy and rotate a range of elements
search	search for a range of elements
search_n	search for N consecutive copies of an element in some range
set_difference	computes the difference between two sets
set_intersection	computes the intersection of two sets
set_symmetric_difference	computes the symmetric difference between two sets

set_union	computes the union of two sets
sort	sort a range into ascending order
sort_heap	turns a heap into a sorted range of elements
stable_partition	divide elements into two groups while preserving their relative order
stable_sort	sort a range of elements while preserving order between equal elements
swap	swap the values of two objects
swap_ranges	swaps two ranges of elements
transform	applies a function to a range of elements
unique	remove consecutive duplicate elements in a range
unique_copy	create a copy of some range of elements that contains no consecutive duplicates
upper_bound	searches for the last possible location to insert an element into an ordered range

accumulate

Syntax:

```
#include <numeric>
TYPE accumulate( iterator start, iterator end, TYPE val );
TYPE accumulate( iterator start, iterator end, TYPE val, BinaryFunction f );
```

The accummulate() function computes the sum of *val* and all of the elements in the range [*start*,*end*).

If the binary function *f* if specified, it is used instead of the + operator to perform the summation.

accumulate() runs in linear time.

Related topics:

adjacent_difference
count
inner_product
partial_sum

accumulate

Syntax:

```
#include <numeric>
TYPE accumulate( iterator start, iterator end, TYPE val );
TYPE accumulate( iterator start, iterator end, TYPE val, BinaryFunction f );
```

The accummulate() function computes the sum of *val* and all of the elements in the range [*start*,*end*).

If the binary function *f* if specified, it is used instead of the + operator to perform the summation.

accumulate() runs in linear time.

Related topics:

adjacent_difference
count
inner_product
partial_sum

adjacent_difference

Syntax:

```
#include <numeric>
iterator adjacent_difference( iterator start, iterator end, iterator result );
iterator adjacent_difference( iterator start, iterator end, iterator result, BinaryFunction f );
```

The adjacent_difference() function calculates the differences between adjacent elements in the range [*start*,*end*) and stores the result starting at *result*.

If a binary function *f* is given, it is used instead of the - operator to compute the differences.

adjacent_difference() runs in linear time.

Related topics:
accumulate
count
inner_product
partial_sum

adjacent_find

Syntax:

```
#include <algorithm>
iterator adjacent_find( iterator start, iterator end );
iterator adjacent_find( iterator start, iterator end, BinPred pr );
```

The adjacent_find() function searches between *start* and *end* for two consecutive identical elements. If the binary predicate *pr* is specified, then it is used to test whether two elements are the same or not.

The return value is an iterator that points to the first of the two elements that are found. If no matching elements are found, the returned iterator points to *end*.

For example, the following code creates a vector containing the integers between 0 and 10 with 7 appearing twice in a row. adjacent_find() is then used to find the location of the pair of 7's:

```
vector<int> v1;
for( int i = 0; i < 10; i++ ) {
  v1.push_back(i);
  // add a duplicate 7 into v1
  if( i == 7 ) {
    v1.push_back(i);

  }
}

vector<int>::iterator result;
result = adjacent_find( v1.begin(), v1.end() );

if( result == v1.end() ) {
  cout << "Did not find adjacent elements in v1" << endl;
}

else {
  cout << "Found matching adjacent elements starting at " << *result << endl;
}
```

Related topics:

find
find_end
find_first_of
find_if
unique
unique_copy

binary_search

Syntax:

```
#include <algorithm>
bool binary_search( iterator start, iterator end, const TYPE& val );
bool binary_search( iterator start, iterator end, const TYPE& val, Comp f );
```

The binary_search() function searches from *start* to *end* for *val*. The elements between *start* and *end* that are searched should be in ascending order as defined by the < operator. Note that a binary search **will not work** unless the elements being searched are in order.

If *val* is found, binary_search() returns true, otherwise false.

If the function *f* is specified, then it is used to compare elements.

For example, the following code uses binary_search() to determine if the integers 0-9 are in an array of integers:

```
int nums[] = { -242, -1, 0, 5, 8, 9, 11 };
int start = 0;
int end = 7;

for( int i = 0; i < 10; i++ ) {
  if( binary_search( nums+start, nums+end, i ) ) {
    cout << "nums[] contains " << i << endl;
  } else {
    cout << "nums[] DOES NOT contain " << i << endl;
  }
}
```

When run, this code displays the following output:

```
nums[] contains 0
nums[] DOES NOT contain 1
nums[] DOES NOT contain 2
nums[] DOES NOT contain 3
nums[] DOES NOT contain 4
nums[] contains 5
nums[] DOES NOT contain 6
nums[] DOES NOT contain 7
nums[] contains 8
nums[] contains 9
```

Related topics:

equal_range
is_sorted
lower_bound
partial_sort
partial_sort_copy
sort
stable_sort
upper_bound

copy

Syntax:

```
#include <algorithm>
iterator copy( iterator start, iterator end, iterator dest );
```

The copy() function copies the elements between *start* and *end* to *dest*. In other words, after copy() has run,

```
*dest == *start
*(dest+1) == *(start+1)
*(dest+2) == *(start+2)
...
*(dest+N) == *(start+N)
```

The return value is an iterator to the last element copied. copy() runs in linear time.

For example, the following code uses copy() to copy the contents of one vector to another:

```
vector<int> from_vector;
for( int i = 0; i < 10; i++ ) {
  from_vector.push_back( i );
}

vector<int> to_vector(10);

copy( from_vector.begin(), from_vector.end(), to_vector.begin() );


cout << "to_vector contains: ";
for( unsigned int i = 0; i < to_vector.size(); i++ ) {
  cout << to_vector[i] << " ";
}
cout << endl;
```

Related topics:

copy_backward
copy_n
generate
remove_copy
swap
transform

copy_backward

Syntax:

```
#include <algorithm>
iterator copy_backward( iterator start, iterator end, iterator dest );
```

copy_backward() is similar to (C++ Strings) copy(), in that both functions copy elements from *start* to *end* to *dest*. The copy_backward() function , however, starts depositing elements at *dest* and then works backwards, such that:

```
*(dest-1) == *(end-1)
*(dest-2) == *(end-2)
*(dest-3) == *(end-3)
...
*(dest-N) == *(end-N)
```

The following code uses copy_backward() to copy 10 integers into the end of an empty vector:

```
vector<int> from_vector;
for( int i = 0; i < 10; i++ ) {
  from_vector.push_back( i );
}

vector<int> to_vector(15);

copy_backward( from_vector.begin(), from_vector.end(), to_vector.end() );


cout << "to_vector contains: ";
for( unsigned int i = 0; i < to_vector.size(); i++ ) {
  cout << to_vector[i] << " ";
}
cout << endl;
```

The above code produces the following output:

```
to_vector contains: 0 0 0 0 0 0 1 2 3 4 5 6 7 8 9
```

Related topics:

copy
copy_n
swap

copy_n

Syntax:

```
#include <algorithm>
iterator copy_n( iterator from, size_t num, iterator to );
```

The copy_n() function copies *num* elements starting at *from* to the destination pointed at by *to*. To put it another way, copy_n() performs *num* assignments and duplicates a subrange.

The return value of copy_n() is an iterator that points to the last element that was copied, i.e. (to + num).

This function runs in linear time.

Related topics:
copy
copy_backward
swap

count

Syntax:

```
#include <algorithm>
size_t count( iterator start, iterator end, const TYPE& val );
```

The count() function returns the number of elements between *start* and *end* that match *val*.

For example, the following code uses count() to determine how many integers in a vector match a target value:

```
vector<int> v;
for( int i = 0; i < 10; i++ ) {
  v.push_back( i );
}

int target_value = 3;
int num_items = count( v.begin(), v.end(), target_value );

cout << "v contains " << num_items << " items matching " << target_value <<
endl;
```

The above code displays the following output:

```
v contains 1 items matching 3
```

Related topics:

accumulate
adjacent_difference
count_if
inner_product
partial_sum

count_if

Syntax:

```
#include <algorithm>
size_t count_if( iterator start, iterator end, UnaryPred p );
```

The count_if() function returns the number of elements between *start* and *end* for which the predicate *p* returns true.

For example, the following code uses count_if() with a predicate that returns true for the integer 3 to count the number of items in an array that are equal to 3:

```
int nums[] = { 0, 1, 2, 3, 4, 5, 9, 3, 13 };
int start = 0;
int end = 9;

int target_value = 3;
int num_items = count_if( nums+start,
                   nums+end,
                   bind2nd(equal_to<int>(), target_value) );

cout << "nums[] contains " << num_items << " items matching " << target_value
<< endl;
```

When run, the above code displays the following output:

```
nums[] contains 2 items matching 3
```

Related topics:
count

equal

Syntax:

```
#include <algorithm>
bool equal( iterator start1, iterator end1, iterator start2 );
bool equal( iterator start1, iterator end1, iterator start2, BinPred p );
```

The equal() function returns true if the elements in two ranges are the same. The first range of elements are those between *start1* and *end1*. The second range of elements has the same size as the first range but starts at *start2*.

If the binary predicate *p* is specified, then it is used instead of == to compare each pair of elements.

For example, the following code uses equal() to compare two vectors of integers:

```
vector<int> v1;
for( int i = 0; i < 10; i++ ) {
  v1.push_back( i );
}

vector<int> v2;
for( int i = 0; i < 10; i++ ) {
  v2.push_back( i );
}

if( equal( v1.begin(), v1.end(), v2.begin() ) ) {
  cout << "v1 and v2 are equal" << endl;
} else {
  cout << "v1 and v2 are NOT equal" << endl;
}
```

Related topics:

find_if
lexicographical_compare
mismatch
search

equal_range

Syntax:

```
#include <algorithm>
```

```
pair<iterator,iterator> equal_range( iterator first, iterator last, const
TYPE& val );
```

```
pair<iterator,iterator> equal_range( iterator first, iterator last, const
TYPE& val, CompFn comp );
```

The equal_range() function returns the range of elements between *first* and *last* that are equal to *val*. This function assumes that the elements between *first* and *last* are in order according to *comp*, if it is specified, or the < operator otherwise.

equal_range() can be thought of as a combination of the lower_bound() and `upper_bound1`() functions, since the first of the pair of iterators that it returns is what lower_bound() returns and the second iterator in the pair is what `upper_bound1`() returns.

For example, the following code uses equal_range() to determine all of the possible places that the number 8 can be inserted into an ordered vector of integers such that the existing ordering is preserved:

```
vector<int> nums;
nums.push_back( -242 );
nums.push_back( -1 );
nums.push_back( 0 );
nums.push_back( 5 );
nums.push_back( 8 );
nums.push_back( 8 );
nums.push_back( 11 );

pair<vector<int>::iterator, vector<int>::iterator> result;
int new_val = 8;

result = equal_range( nums.begin(), nums.end(), new_val );

cout << "The first place that " << new_val << " could be inserted is before "
     << *result.first << ", and the last place that it could be inserted is
before "
     << *result.second << endl;
```

The above code produces the following output:

```
The first place that 8 could be inserted is before 8,
and the last place that it could be inserted is before 11
```

Related topics:

binary_search

lower_bound

upper_bound

fill

Syntax:

```
#include <algorithm>
#include <algorithm>
void fill( iterator start, iterator end, const TYPE& val );
```

The function fill() assigns *val* to all of the elements between *start* and *end*.

For example, the following code uses fill() to set all of the elements of a vector of integers to -1:

```
vector<int> v1;
for( int i = 0; i < 10; i++ ) {
  v1.push_back( i );
}

cout << "Before, v1 is: ";
for( unsigned int i = 0; i < v1.size(); i++ ) {
  cout << v1[i] << " ";
}
cout << endl;

fill( v1.begin(), v1.end(), -1 );

cout << "After, v1 is: ";
for( unsigned int i = 0; i < v1.size(); i++ ) {
  cout << v1[i] << " ";
}
cout << endl;
```

When run, the above code displays:

```
Before, v1 is: 0 1 2 3 4 5 6 7 8 9
After, v1 is: -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
```

Related topics:

fill_n
generate
transform

fill_n

Syntax:

```
#include <algorithm>
```

```
#include <algorithm>
iterator fill_n( iterator start, size_t n, const TYPE& val );
```

The fill_n() function is similar to (C++ I/O) fill(). Instead of assigning *val* to a range of elements, however, fill_n() assigns *val* to the first *n* elements starting at *start*.

For example, the following code uses fill_n() to assign -1 to the first half of a vector of integers:

```
vector<int> v1;
for( int i = 0; i < 10; i++ ) {
  v1.push_back( i );
}

cout << "Before, v1 is: ";
for( unsigned int i = 0; i < v1.size(); i++ ) {
  cout << v1[i] << " ";
}
cout << endl;

fill_n( v1.begin(), v1.size()/2, -1 );

cout << "After, v1 is: ";
for( unsigned int i = 0; i < v1.size(); i++ ) {
  cout << v1[i] << " ";
}
cout << endl;
```

When run, this code displays:

```
Before, v1 is: 0 1 2 3 4 5 6 7 8 9
After, v1 is: -1 -1 -1 -1 -1 5 6 7 8 9
```

Related topics:

fill

find

Syntax:

```
#include <algorithm>
iterator find( iterator start, iterator end, const TYPE& val );
```

The find() algorithm looks for an element matching *val* between *start* and *end*. If an element matching *val* is found, the return value is an iterator that points to that element. Otherwise, the return value is an iterator that points to *end*.

For example, the following code uses find() to search a vector of integers for the number 3:

```
int num_to_find = 3;

vector<int> v1;
for( int i = 0; i < 10; i++ ) {
  v1.push_back(i);
}

vector<int>::iterator result;
result = find( v1.begin(), v1.end(), num_to_find );

if( result == v1.end() ) {
  cout << "Did not find any element matching " << num_to_find << endl;
}

else {
  cout << "Found a matching element: " << *result << endl;
}
```

In the next example, shown below, the find() function is used on an array of integers. This example shows how the C++ Algorithms can be used to manipulate arrays and pointers in the same manner that they manipulate containers and iterators:

```
int nums[] = { 3, 1, 4, 1, 5, 9 };

int num_to_find = 5;
int start = 0;
int end = 2;
int* result = find( nums + start, nums + end, num_to_find );

if( result == nums + end ) {
  cout << "Did not find any number matching " << num_to_find << endl;
} else {
  cout << "Found a matching number: " << *result << endl;
}
```

Related topics:

adjacent_find
find_end
find_first_of
find_if
mismatch
search

find_end

Syntax:

```
#include <algorithm>
iterator find_end( iterator start, iterator end, iterator seq_start, iterator seq_end );
iterator find_end( iterator start, iterator end, iterator seq_start, iterator seq_end, BinPred bp );
```

The find_end() function searches for the sequence of elements denoted by *seq_start* and *seq_end*. If such a sequence if found between *start* and *end*, an iterator to the first element of the last found sequence is returned. If no such sequence is found, an iterator pointing to *end* is returned.

If the binary predicate *bp* is specified, then it is used to when elements match.

For example, the following code uses find_end() to search for two different sequences of numbers. The the first chunk of code, the last occurence of "1 2 3" is found. In the second chunk of code, the sequence that is being searched for is not found:

```
int nums[] = { 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4 };
int* result;
int start = 0;
int end = 11;

int target1[] = { 1, 2, 3 };
result = find_end( nums + start, nums + end, target1 + 0, target1 + 2 );
if( *result == nums[end] ) {
  cout << "Did not find any subsequence matching { 1, 2, 3 }" << endl;
} else {
  cout << "The last matching subsequence is at: " << *result << endl;
}

int target2[] = { 3, 2, 3 };
result = find_end( nums + start, nums + end, target2 + 0, target2 + 2 );
if( *result == nums[end] ) {
  cout << "Did not find any subsequence matching { 3, 2, 3 }" << endl;
} else {
  cout << "The last matching subsequence is at: " << *result << endl;
}
```

Related topics:

adjacent_find
find
find_first_of
find_if
search_n

find_first_of

Syntax:

```
#include <algorithm>
iterator find_first_of( iterator start, iterator end, iterator find_start, iterator find_end );
iterator find_first_of( iterator start, iterator end, iterator find_start, iterator find_end, BinPred bp );
```

The find_first_of() function searches for the first occurence of any element between *find_start* and *find_end*. The data that are searched are those between *start* and *end*.

If any element between *find_start* and *find_end* is found, an iterator pointing to that element is returned. Otherwise, an iterator pointing to *end* is returned.

For example, the following code searches for a 9, 4, or 7 in an array of integers:

```
int nums[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int* result;
int start = 0;
int end = 10;

int targets[] = { 9, 4, 7 };
result = find_first_of( nums + start, nums + end, targets + 0, targets + 2 );
if( *result == nums[end] ) {
  cout << "Did not find any of { 9, 4, 7 }" << endl;
} else {
  cout << "Found a matching target: " << *result << endl;
}
```

Related topics:

adjacent_find
find
find_end
find_if
(Standard C String and Character) strpbrk

find_if

Syntax:

```
#include <algorithm>
iterator find_if( iterator start, iterator end, UnPred up );
```

The find_if() function searches for the first element between *start* and *end* for which the unary predicate *up* returns true.

If such an element is found, an iterator pointing to that element is returned. Otherwise, an iterator pointing to *end* is returned.

For example, the following code uses find_if() and a "greater-than-zero" unary predicate to the first positive, non-zero number in a list of numbers:

```
int nums[] = { 0, -1, -2, -3, -4, 342, -5 };
int* result;
int start = 0;
int end = 7;

result = find_if( nums + start, nums + end, bind2nd(greater<int>(), 0));
if( *result == nums[end] ) {
  cout << "Did not find any number greater than zero" << endl;
} else {
  cout << "Found a positive non-zero number: " << *result << endl;
}
```

Related topics:

adjacent_find
equal
find
find_end
find_first_of
search_n

for_each

Syntax:

```
#include <algorithm>
UnaryFunction for_each( iterator start, iterator end, UnaryFunction f );
```

The for_each() algorithm applies the function *f* to each of the elements between *start* and *end*. The return value of for_each() is *f*.

For example, the following code snippets define a unary function then use it to increment all of the elements of an array:

```
template<class TYPE> struct increment : public unary_function<TYPE, void> {
  void operator() (TYPE& x) {
    x++;
  }
};

...

int nums[] = {3, 4, 2, 9, 15, 267};
const int N = 6;

cout << "Before, nums[] is: ";
for( int i = 0; i < N; i++ ) {
  cout << nums[i] << " ";
}
cout << endl;

for_each( nums, nums + N, increment<int>() );

cout << "After, nums[] is: ";
for( int i = 0; i < N; i++ ) {
  cout << nums[i] << " ";
}
cout << endl;
```

The above code displays the following output:

```
Before, nums[] is: 3 4 2 9 15 267
After, nums[] is: 4 5 3 10 16 268
```

generate

Syntax:

```
#include <algorithm>
void generate( iterator start, iterator end, Generator g );
```

The generate() function runs the Generator function object *g* a number of times, saving the result of each execution in the range [*start*,*end*).

Related topics:

copy
fill
generate_n
transform

generate_n

Syntax:

```
#include <algorithm>
iterator generate_n( iterator result, size_t num, Generator g );
```

The generate_n() function runs the Generator function object *g* *num* times, saving the result of each execution in *result*, (*result*+1), etc.

Related topics:
generate

includes

Syntax:

```
#include <algorithm>
bool includes( iterator start1, iterator end1, iterator start2, iterator end2 );
bool includes( iterator start1, iterator end1, iterator start2, iterator end2, StrictWeakOrdering cmp );
```

The includes() algorithm returns true if every element in [*start2*,*end2*) is also in [*start1*,*end1*). Both of the given ranges must be sorted in ascending order.

By default, the < operator is used to compare elements. If the strict weak ordering function object *cmp* is given, then it is used instead.

includes() runs in linear time.

Related topics:
set_difference
set_intersection
set_symmetric_difference
set_union

inner_product

Syntax:

```
#include <numeric>

TYPE inner_product( iterator start1, iterator end1, iterator start2, TYPE val );

TYPE inner_product( iterator start1, iterator end1, iterator start2, TYPE val, BinaryFunction f1, BinaryFunction f2 );
```

The inner_product() function computes the inner product of [*start1*,*end1*) and a range of the same size starting at *start2*.

inner_product() runs in linear time.

Related topics:
accumulate
adjacent_difference
count
partial_sum

inplace_merge

Syntax:

```
#include <algorithm>
inline void inplace_merge( iterator start, iterator middle, iterator end );
inline void inplace_merge( iterator start, iterator middle, iterator end,
StrictWeakOrdering cmp );
```

The inplace_merge() function is similar to the merge() function, but instead of creating a new sorted range of elements, inplace_merge() alters the existing ranges to perform the merge in-place.

Related topics:
merge

is_heap

Syntax:

```
#include <algorithm>
bool is_heap( iterator start, iterator end );
bool is_heap( iterator start, iterator end, StrictWeakOrdering cmp );
```

The is_heap() function returns true if the given range [*start*,*end*) is a heap.

If the strict weak ordering comparison function object *cmp* is given, then it is used instead of the < operator to compare elements.

is_heap() runs in linear time.

Related topics:
make_heap
pop_heap
push_heap
sort_heap

is_sorted

Syntax:

```
#include <algorithm>
bool is_sorted( iterator start, iterator end );
bool is_sorted( iterator start, iterator end, StrictWeakOrdering cmp );
```

The is_sorted() algorithm returns true if the elements in the range [*start*,*end*) are sorted in ascending order.

By default, the < operator is used to compare elements. If the strict weak order function object *cmp* is given, then it is used instead.

is_sorted() runs in linear time.

Related topics:
binary_search
partial_sort
partial_sort_copy
sort
stable_sort

iter_swap

Syntax:

```
#include <algorithm>
inline void iter_swap( iterator a, iterator b );
```

A call to iter_swap() exchanges the values of two elements exactly as a call to

```
swap( *a, *b );
```

would.

Related topics:
swap
swap_ranges

lexicographical_compare

Syntax:

```
#include <algorithm>
bool lexicographical_compare( iterator start1, iterator end1, iterator start2, iterator end2 );
bool lexicographical_compare( iterator start1, iterator end1, iterator start2, iterator end2, BinPred p );
```

The lexicographical_compare() function returns true if the range of elements [*start1*,*end1*) is lexicographically less than the range of elements [*start2*,*end2*).

If you're confused about what lexicographic means, it might help to know that dictionaries are ordered lexicographically.

lexicographical_compare() runs in linear time.

Related topics:
equal
lexicographical_compare_3way
mismatch
search

lexicographical_compare_3way

Syntax:

```
#include <algorithm>
int lexicographical_compare_3way( iterator start1, iterator end1, iterator start2, iterator end2 );
```

The lexicographical_compare_3way() function compares the first range, defined by [*start1*,*end1*) to the second range, defined by [*start2*,*end2*).

If the first range is lexicographically less than the second range, this function returns a negative number. If the first range is lexicographically greater than the second, a positive number is returned. Zero is returned if neither range is lexicographically greater than the other.

lexicographical_compare_3way() runs in linear time.

Related topics:
lexicographical_compare

lower_bound

Syntax:

```
#include <algorithm>
iterator lower_bound( iterator first, iterator last,  const TYPE& val );
iterator lower_bound( iterator first, iterator last, const TYPE& val,
CompFn f );
```

The lower_bound() function is a type of binary_search(). This function searches for the first place that *val* can be inserted into the ordered range defined by *first* and *last* that will not mess up the existing ordering.

The return value of lower_bound() is an iterator that points to the location where *val* can be safely inserted. Unless the comparison function *f* is specified, the < operator is used for ordering.

For example, the following code uses lower_bound() to insert the number 7 into an ordered vector of integers:

```
vector<int> nums;
nums.push_back( -242 );
nums.push_back( -1 );
nums.push_back( 0 );
nums.push_back( 5 );
nums.push_back( 8 );
nums.push_back( 8 );
nums.push_back( 11 );

cout << "Before nums is: ";
for( unsigned int i = 0; i < nums.size(); i++ ) {
  cout << nums[i] << " ";
}
cout << endl;

vector<int>::iterator result;
int new_val = 7;

result = lower_bound( nums.begin(), nums.end(), new_val );

nums.insert( result, new_val );

cout << "After, nums is: ";
for( unsigned int i = 0; i < nums.size(); i++ ) {
  cout << nums[i] << " ";
}
cout << endl;
```

The above code produces the following output:

```
Before nums is: -242 -1 0 5 8 8 11
After, nums is: -242 -1 0 5 7 8 8 11
```

Related topics:
binary_search
equal_range

make_heap

Syntax:

```
#include <algorithm>
void make_heap( iterator start, iterator end );
void make_heap( iterator start, iterator end, StrictWeakOrdering cmp );
```

The make_heap() function turns the given range of elements [*start*,*end*) into a heap.

If the strict weak ordering comparison function object *cmp* is given, then it is used instead of the < operator to compare elements.

make_heap() runs in linear time.

Related topics:
is_heap
pop_heap
push_heap
sort_heap

max

Syntax:

```
#include <algorithm>
const TYPE& max( const TYPE& x, const TYPE& y );
const TYPE& max( const TYPE& x, const TYPE& y, BinPred p );
```

The max() function returns the greater of *x* and *y*.

If the binary predicate *p* is given, then it will be used instead of the < operator to compare the two elements.

Example code:

For example, the following code snippet displays various uses of the max() function:

```
cout << "Max of 1 and 9999 is " << max( 1, 9999) << endl;
cout << "Max of 'a' and 'b' is " << max( 'a', 'b') << endl;
cout << "Max of 3.14159 and 2.71828 is " << max( 3.14159, 2.71828) << endl;
```

When run, this code displays:

```
Max of 1 and 9999 is 9999
Max of 'a' and 'b' is b
Max of 3.14159 and 2.71828 is 3.14159
```

Related topics:

max_element
min
min_element

max_element

Syntax:

```
#include <algorithm>
iterator max_element( iterator start, iterator end );
iterator max_element( iterator start, iterator end, BinPred p );
```

The max_element() function returns an iterator to the largest element in the range [*start*,*end*).

If the binary predicate *p* is given, then it will be used instead of the < operator to determine the largest element.

Example code:

For example, the following code uses the max_element() function to determine the largest integer in an array and the largest character in a vector of characters:

```
int array[] = { 3, 1, 4, 1, 5, 9 };
unsigned int array_size = 6;
cout << "Max element in array is " << *max_element( array, array+array_size)
<< endl;

vector<char> v;
v.push_back('a'); v.push_back('b'); v.push_back('c'); v.push_back('d');
cout << "Max element in the vector v is " << *max_element( v.begin(), v.end()
) << endl;
```

When run, the above code displays this output:

```
Max element in array is 9
Max element in the vector v is d
```

Related topics:

max
min
min_element

merge

Syntax:

```
  #include <algorithm>
  iterator merge( iterator start1, iterator end1, iterator start2, iterator end2, iterator result );
  iterator merge( iterator start1, iterator end1, iterator start2, iterator end2, iterator result, StrictWeakOrdering cmp );
```

The merge() function combines two sorted ranges [*start1*,*end1*) and [*start2*,*end2*) into a single sorted range, stored starting at *result*. The return value of this function is an iterator to the end of the merged range.

If the strict weak ordering function object *cmp* is given, then it is used in place of the < operator to perform comparisons between elements.

merge() runs in linear time.

Related topics:
inplace_merge
set_union
sort

min

Syntax:

```
#include <algorithm>
const TYPE& min( const TYPE& x, const TYPE& y );
const TYPE& min( const TYPE& x, const TYPE& y, BinPred p );
```

The min() function, unsurprisingly, returns the smaller of *x* and *y*.

By default, the < operator is used to compare the two elements. If the binary predicate *p* is given, it will be used instead.

Related topics:
max
max_element
min_element

min_element

Syntax:

```
#include <algorithm>
iterator min_element( iterator start, iterator end );
iterator min_element( iterator start, iterator end, BinPred p );
```

The min_element() function returns an iterator to the smallest element in the range [*start*,*end*).

If the binary predicate *p* is given, then it will be used instead of the < operator to determine the smallest element.

Related topics:
max
max_element
min

mismatch

Syntax:

```
#include <algorithm>
pair <iterator1,iterator2> mismatch( iterator start1, iterator end1, iterator start2 );
pair <iterator1,iterator2> mismatch( iterator start1, iterator end1, iterator start2, BinPred p );
```

The mismatch() function compares the elements in the range defined by [*start1*,*end1*) to the elements in a range of the same size starting at *start2*. The return value of mismatch() is the first location where the two ranges differ.

If the optional binary predicate *p* is given, then it is used to compare elements from the two ranges.

The mismatch() algorithm runs in linear time.

Related topics:
equal
find
lexicographical_compare
search

next_permutation

Syntax:

```
#include <algorithm>
bool next_permutation( iterator start, iterator end );
bool next_permutation( iterator start, iterator end, StrictWeakOrdering cmp );
```

The next_permutation() function attempts to transform the given range of elements [*start*,*end*) into the next lexicographically greater permutation of elements. If it succeeds, it returns true, otherwise, it returns false.

If a strict weak ordering function object *cmp* is provided, it is used in lieu of the < operator when comparing elements.

Related topics:
prev_permutation
random_sample
random_sample_n
random_shuffle

nth_element

Syntax:

```
#include <algorithm>
void nth_element( iterator start, iterator middle, iterator end );
void nth_element( iterator start, iterator middle, iterator end,
StrictWeakOrdering cmp );
```

The nth_element() function semi-sorts the range of elements defined by [*start*,*end*). It puts the element that *middle* points to in the place that it would be if the entire range was sorted, and it makes sure that none of the elements before that element are greater than any of the elements that come after that element.

nth_element() runs in linear time on average.

Related topics:
partial_sort

partial_sort

Syntax:

```
#include <algorithm>
void partial_sort( iterator start, iterator middle, iterator end );
void partial_sort( iterator start, iterator middle, iterator end,
StrictWeakOrdering cmp );
```

The partial_sort() function arranges the first N elements of the range [*start*,*end*) in ascending order. N is defined as the number of elements between *start* and *middle*.

By default, the < operator is used to compare two elements. If the strict weak ordering comparison function *cmp* is given, it is used instead.

Related topics:
binary_search
is_sorted
nth_element
partial_sort_copy
sort
stable_sort

partial_sort_copy

Syntax:

```
#include <algorithm>
iterator partial_sort_copy( iterator start, iterator end, iterator result_start, iterator result_end );
iterator partial_sort_copy( iterator start, iterator end, iterator result_start, iterator result_end, StrictWeakOrdering cmp );
```

The partial_sort_copy() algorithm behaves like partial_sort(), except that instead of partially sorting the range in-place, a copy of the range is created and the sorting takes place in the copy. The initial range is defined by [*start*,*end*) and the location of the copy is defined by [*result_start*, *result_end*).

partial_sort_copy() returns an iterator to the end of the copied, partially-sorted range of elements.

Related topics:
binary_search
is_sorted
partial_sort
sort
stable_sort

partial_sum

Syntax:

```
#include <numeric>
iterator partial_sum( iterator start, iterator end, iterator result );
iterator partial_sum( iterator start, iterator end, iterator result, BinOp
p );
```

The partial_sum() function calculates the partial sum of a range defined by [*start*,*end*), storing the output at *result*.

- *start* is assigned to **result*, the sum of **start* and *(*start* + 1) is assigned to *(*result* + 1), etc.

partial_sum() runs in linear time.

Related topics:
accumulate
adjacent_difference
count
inner_product

partition

Syntax:

```
#include <algorithm>
iterator partition( iterator start, iterator end, Predicate p );
```

The partition() algorithm re-orders the elements in [*start*,*end*) such that the elements for which the predicate *p* returns true come before the elements for which *p* returns false.

In other words, partition() uses *p* to divide the elements into two groups.

The return value of partition() is an iterator to the first element for which *p* returns false.

parition() runs in linear time.

Related topics:
stable_partition

pop_heap

Syntax:

```
#include <algorithm>
void pop_heap( iterator start, iterator end );
void pop_heap( iterator start, iterator end, StrictWeakOrdering cmp );
```

The pop_heap() function removes the larges element (defined as the element at the front of the heap) from the given heap.

If the strict weak ordering comparison function object *cmp* is given, then it is used instead of the < operator to compare elements.

pop_heap() runs in logarithmic time.

Related topics:
is_heap
make_heap
push_heap
sort_heap

prev_permutation

Syntax:

```
#include <algorithm>
bool prev_permutation( iterator start, iterator end );
bool prev_permutation( iterator start, iterator end, StrictWeakOrdering cmp );
```

The prev_permutation() function attempts to transform the given range of elements [*start*,*end*) into the next lexicographically smaller permutation of elements. If it succeeds, it returns true, otherwise, it returns false.

If a strict weak ordering function object *cmp* is provided, it is used instead of the < operator when comparing elements.

Related topics:
next_permutation
random_sample
random_sample_n
random_shuffle

push_heap

Syntax:

```
#include <algorithm>
void push_heap( iterator start, iterator end );
void push_heap( iterator start, iterator end, StrictWeakOrdering cmp );
```

The push_heap() function adds an element (defined as the last element before *end*) to a heap (defined as the range of elements between [*start*,"end-1).

If the strict weak ordering comparison function object *cmp* is given, then it is used instead of the < operator to compare elements.

push_heap() runs in logarithmic time.

Related topics:
is_heap
make_heap
pop_heap
sort_heap

random_sample

Syntax:

```
#include <algorithm>
iterator random_sample( iterator start1, iterator end1, iterator start2, iterator end2 );
iterator random_sample( iterator start1, iterator end1, iterator start2, iterator end2, RandomNumberGenerator& rnd );
```

The random_sample() algorithm randomly copies elements from [*start1*,*end1*) to [*start2*,*end2*). Elements are chosen with uniform probability and elements from the input range will appear at most once in the output range.

If a random number generator function object *rnd* is supplied, then it will be used instead of an internal random number generator.

The return value of random_sample() is an iterator to the end of the output range.

random_sample() runs in linear time.

Related topics:
next_permutation
prev_permutation
random_sample_n
random_shuffle

random_sample_n

Syntax:

```
#include <algorithm>
iterator random_sample_n( iterator start, iterator end, iterator result, size_t N );
iterator random_sample_n( iterator start, iterator end, iterator result, size_t N, RandomNumberGenerator& rnd );
```

The random_sample_n() algorithm randomly copies *N* elements from [*start*,*end*) to *result*. Elements are chosen with uniform probability and elements from the input range will appear at most once in the output range. **Element order is preserved** from the input range to the output range.

If a random number generator function object *rnd* is supplied, then it will be used instead of an internal random number generator.

The return value of random_sample_n() is an iterator to the end of the output range.

random_sample_n() runs in linear time.

Related topics:
next_permutation
prev_permutation
random_sample
random_shuffle

random_shuffle

Syntax:

```
#include <algorithm>
void random_shuffle( iterator start, iterator end );
void random_shuffle( iterator start, iterator end, RandomNumberGenerator& rnd );
```

The random_shuffle() function randomly re-orders the elements in the range [*start*,*end*). If a random number generator function object *rnd* is supplied, it will be used instead of an internal random nunber generator.

Related topics:

next_permutation
prev_permutation
random_sample
random_sample_n

remove

Syntax:

```
#include <algorithm>
iterator remove( iterator start, iterator end, const TYPE& val );
```

The remove() algorithm removes all of the elements in the range [*start*,*end*) that are equal to *val*.

The return value of this function is an iterator to the last element of the new sequence that should contain no elements equal to *val*.

The remove() function runs in linear time.

Related topics:

remove_copy
remove_copy_if
remove_if
unique
unique_copy

remove_copy

Syntax:

```
#include <algorithm>
iterator remove_copy( iterator start, iterator end, iterator result, const TYPE& val );
```

The remove_copy() algorithm copies the range [*start*,*end*) to *result* but omits any elements that are equal to *val*.

remove_copy() returns an iterator to the end of the new range, and runs in linear time.

Related topics:
copy
remove
remove_copy_if
remove_if

remove_copy_if

Syntax:

```
#include <algorithm>
iterator remove_copy_if( iterator start, iterator end, iterator result, Predicate p );
```

The remove_copy_if() function copies the range of elements [*start*,*end*) to *result*, omitting any elements for which the predicate function *p* returns true.

The return value of remove_copy_if() is an iterator the end of the new range.

remove_copy_if() runs in linear time.

Related topics:
remove
remove_copy
remove_if

remove_if

Syntax:

```
#include <algorithm>
iterator remove_if( iterator start, iterator end, Predicate p );
```

The remove_if() function removes all elements in the range [*start*,*end*) for which the predicate *p* returns true.

The return value of this function is an iterator to the last element of the pruned range.

remove_if() runs in linear time.

Related topics:
remove
remove_copy
remove_copy_if

replace

Syntax:

```
#include <algorithm>
void replace( iterator start, iterator end, const TYPE& old_value, const TYPE& new_value );
```

The replace() function sets every element in the range [*start*,*end*) that is equal to *old_value* to have *new_value* instead.

replace() runs in linear time.

Related topics:
replace_copy
replace_copy_if
replace_if

replace_copy

Syntax:

```
#include <algorithm>
iterator replace_copy( iterator start, iterator end, iterator result, const TYPE& old_value, const TYPE& new_value );
```

The replace_copy() function copies the elements in the range [*start*,*end*) to the destination *result*. Any elements in the range that are equal to *old_value* are replaced with *new_value*.

Related topics:
replace

replace_copy_if

Syntax:

```
#include <algorithm>
iterator replace_copy_if( iterator start, iterator end, iterator result,
Predicate p, const TYPE& new_value );
```

The replace_copy_if() function copies the elements in the range [*start*,*end*) to the destination *result*. Any elements for which the predicate *p* is true are replaced with *new_value*.

Related topics:
replace

replace_if

Syntax:

```
#include <algorithm>
void replace_if( iterator start, iterator end, Predicate p, const TYPE& new_value );
```

The replace_if() function assigns every element in the range [*start*,*end*) for which the predicate function *p* returns true the value of *new_value*.

This function runs in linear time.

Related topics:
replace

reverse

Syntax:

```
#include <algorithm>
void reverse( iterator start, iterator end );
```

The reverse() algorithm reverses the order of elements in the range [*start*,*end*).

Related topics:
reverse_copy

reverse_copy

Syntax:

```
#include <algorithm>
iterator reverse_copy( iterator start, iterator end, iterator result );
```

The reverse_copy() algorithm copies the elements in the range [*start*,*end*) to *result* such that the elements in the new range are in reverse order.

The return value of the reverse_copy() function is an iterator the end of the new range.

Related topics:
reverse

rotate

Syntax:

```
#include <algorithm>
inline iterator rotate( iterator start, iterator middle, iterator end );
```

The rotate() algorithm moves the elements in the range [*start*,*end*) such that the *middle* element is now where *start* used to be, (*middle*+1) is now at (*start*+1), etc.

The return value of rotate() is an iterator to *start* + (*end*-*middle*).

rotate() runs in linear time.

Related topics:
rotate_copy

rotate_copy

Syntax:

```
#include <algorithm>
iterator rotate_copy( iterator start, iterator middle, iterator end, iterator result );
```

The rotate_copy() algorithm is similar to the rotate() algorithm, except that the range of elements is copied to *result* before being rotated.

Related topics:
rotate

search

Syntax:

```
#include <algorithm>
iterator search( iterator start1, iterator end1, iterator start2, iterator end2 );
iterator search( iterator start1, iterator end1, iterator start2, iterator end2, BinPred p );
```

The search() algorithm looks for the elements [*start2*,*end2*) in the range [*start1*,*end1*). If the optional binary predicate *p* is provided, then it is used to perform comparisons between elements.

If search() finds a matching subrange, then it returns an iterator to the beginning of that matching subrange. If no match is found, an iterator pointing to *end1* is returned.

In the worst case, search() runs in quadratic time, on average, it runs in linear time.

Related topics:
equal
find
lexicographical_compare
mismatch
search_n

search_n

Syntax:

```
#include <algorithm>
iterator search_n( iterator start, iterator end, size_t num, const TYPE& val );
iterator search_n( iterator start, iterator end, size_t num, const TYPE& val, BinPred p );
```

The search_n() function looks for *num* occurances of *val* in the range [*start*,*end*).

If *num* consecutive copies of *val* are found, search_n() returns an iterator to the beginning of that sequence. Otherwise it returns an iterator to *end*.

If the optional binary predicate *p* is given, then it is used to perform comparisons between elements.

This function runs in linear time.

Related topics:
find_end
find_if
search

set_difference

Syntax:

```
#include <algorithm>
iterator set_difference( iterator start1, iterator end1, iterator start2, iterator end2, iterator result );
iterator set_difference( iterator start1, iterator end1, iterator start2, iterator end2, iterator result, StrictWeakOrdering cmp );
```

The set_difference() algorithm computes the difference between two sets defined by [*start1*,*end1*) and [*start2*,*end2*) and stores the difference starting at *result*.

Both of the sets, given as ranges, must be sorted in ascending order.

The return value of set_difference() is an iterator to the end of the result range.

If the strict weak ordering comparison function object *cmp* is not specified, set_difference() will use the < operator to compare elements.

Related topics:
includes
set_intersection
set_symmetric_difference
set_union

set_intersection

Syntax:

```
#include <algorithm>
iterator set_intersection( iterator start1, iterator end1, iterator start2, iterator end2, iterator result );
iterator set_intersection( iterator start1, iterator end1, iterator start2, iterator end2, iterator result, StrictWeakOrdering cmp );
```

The set_intersection() algorithm computes the intersection of the two sets defined by [*start1*,*end1*) and [*start2*,*end2*) and stores the intersection starting at *result*.

Both of the sets, given as ranges, must be sorted in ascending order.

The return value of set_intersection() is an iterator to the end of the intersection range.

If the strict weak ordering comparison function object *cmp* is not specified, set_intersection() will use the < operator to compare elements.

Related topics:
includes
set_difference
set_symmetric_difference
set_union

set_symmetric_difference

Syntax:

```
#include <algorithm>
iterator set_symmetric_difference( iterator start1, iterator end1, iterator start2, iterator end2, iterator result );
iterator set_symmetric_difference( iterator start1, iterator end1, iterator start2, iterator end2, iterator result, StrictWeakOrdering cmp );
```

The set_symmetric_difference() algorithm computes the symmetric difference of the two sets defined by [*start1*,*end1*) and [*start2*,*end2*) and stores the difference starting at *result*.

Both of the sets, given as ranges, must be sorted in ascending order.

The return value of set_symmetric_difference() is an iterator to the end of the result range.

If the strict weak ordering comparison function object *cmp* is not specified, set_symmetric_difference() will use the < operator to compare elements.

Related topics:
includes
set_difference
set_intersection
set_union

set_union

Syntax:

```
  #include <algorithm>
  iterator set_union( iterator start1, iterator end1, iterator start2,
iterator end2, iterator result );
  iterator set_union( iterator start1, iterator end1, iterator start2,
iterator end2, iterator result, StrictWeakOrdering cmp );
```

The set_union() algorithm computes the union of the two ranges [*start1*,*end1*) and [*start2*,*end2*) and stores it starting at *result*.

The return value of set_union() is an iterator to the end of the union range.

set_union() runs in linear time.

Related topics:
includes
merge
set_difference
set_intersection
set_symmetric_difference

sort

Syntax:

```
#include <algorithm>
void sort( iterator start, iterator end );
void sort( iterator start, iterator end, StrictWeakOrdering cmp );
```

The sort() algorithm sorts the elements in the range [*start*,*end*) into ascending order. If two elements are equal, there is no guarantee what order they will be in.

If the strict weak ordering function object *cmp* is given, then it will be used to compare two objects instead of the < operator.

The algorithm behind sort() is the *introsort* algorithm. sort() runs in O(N log(N)) time (average and worst case) which is faster than polynomial time but slower than linear time.

Example code:

For example, the following code sorts a vector of integers into ascending order:

```
vector<int> v;
v.push_back( 23 );
v.push_back( -1 );
v.push_back( 9999 );
v.push_back( 0 );
v.push_back( 4 );

cout << "Before sorting: ";
for( unsigned int i = 0; i < v.size(); i++ ) {
  cout << v[i] << " ";
}
cout << endl;

sort( v.begin(), v.end() );

cout << "After sorting: ";
for( unsigned int i = 0; i < v.size(); i++ ) {
  cout << v[i] << " ";
}
cout << endl;
```

When run, the above code displays this output:

```
Before sorting: 23 -1 9999 0 4
After sorting: -1 0 4 23 9999
```

Alternatively, the following code uses the sort() function to sort a normal array of integers, and displays the same output as the previous example:

```
int array[] = { 23, -1, 9999, 0, 4 };
unsigned int array_size = 5;

cout << "Before sorting: ";
for( unsigned int i = 0; i < array_size; i++ ) {
  cout << array[i] << " ";
}
```

```
cout << endl;

sort( array, array + array_size );

cout << "After sorting: ";
for( unsigned int i = 0; i < array_size; i++ ) {
  cout << array[i] << " ";
}
cout << endl;
```

This next example shows how to use sort() with a user-specified comparison function. The function **cmp** is defined to do the opposite of the < operator. When sort() is called with **cmp** used as the comparison function, the result is a list sorted in descending, rather than ascending, order:

```

bool cmp( int a, int b ) {
  return a > b;
}

...

vector<int> v;
for( int i = 0; i < 10; i++ ) {
  v.push_back(i);
}

cout << "Before: ";
for( int i = 0; i < 10; i++ ) {
  cout << v[i] << " ";
}
cout << endl;

sort( v.begin(), v.end(), cmp );

cout << "After: ";
for( int i = 0; i < 10; i++ ) {
  cout << v[i] << " ";
}
cout << endl;
```

Related topics:
binary_search
is_sorted
merge
partial_sort
partial_sort_copy
stable_sort
(Other Standard C Functions) qsort

sort_heap

Syntax:

```
#include <algorithm>
void sort_heap( iterator start, iterator end );
void sort_heap( iterator start, iterator end, StrictWeakOrdering cmp );
```

The sort_heap() function turns the heap defined by [*start*,*end*) into a sorted range.

If the strict weak ordering comparison function object *cmp* is given, then it is used instead of the < operator to compare elements.

Related topics:
is_heap
make_heap
pop_heap
push_heap

stable_partition

Syntax:

```
#include <algorithm>
iterator stable_partition( iterator start, iterator end, Predicate p );
```

The stable_partition() function behaves similarily to partition(). The difference between the two algorithms is that stable_partition() will preserve the initial ordering of the elements in the two groups.

Related topics:
partition

stable_sort

Syntax:

```
#include <algorithm>
void stable_sort( iterator start, iterator end );
void stable_sort( iterator start, iterator end, StrictWeakOrdering cmp );
```

The stable_sort() algorithm is like the sort() algorithm, in that it sorts a range of elements into ascending order. Unlike sort(), however, stable_sort() will preserve the original ordering of elements that are equal to eachother.

This functionality comes at a small cost, however, as stable_sort() takes a few more comparisons that sort() in the worst case: N (log N)^2 instead of N log N.

Related topics:

binary_search
is_sorted
partial_sort
partial_sort_copy
sort

swap

Syntax:

```
#include <algorithm>
void swap( Assignable& a, Assignable& b );
```

The swap() function swaps the values of *a* and *b*.

swap() expects that its arguments will conform to the Assignable model; that is, they should have a copy constructor and work with the = operator. This function performs one copy and two assignments.

Related topics:

copy
copy_backward
copy_n
iter_swap
swap_ranges

swap_ranges

Syntax:

```
#include <algorithm>
iterator swap_ranges( iterator start1, iterator end1, iterator start2 );
```

The swap_ranges() function exchanges the elements in the range [*start1*,*end1*) with the range of the same size starting at *start2*.

The return value of swap_ranges() is an iterator to *start2* + (*end1*-*start1*).

Related topics:
iter_swap
swap

transform

Syntax:

```
#include <algorithm>
iterator transform( iterator start, iterator end, iterator result, UnaryFunction f );
iterator transform( iterator start1, iterator end1, iterator start2, iterator result, BinaryFunction f );
```

The transform() algorithm applies the function *f* to some range of elements, storing the result of each application of the function in *result*.

The first version of the function applies *f* to each element in [*start*,*end*) and assigns the first output of the function to *result*, the second output to (*result*+1), etc.

The second version of the transform() works in a similar manner, except that it is given two ranges of elements and calls a binary function on a pair of elements.

Related topics:
copy
fill
generate

unique

Syntax:

```
#include <algorithm>
iterator unique( iterator start, iterator end );
iterator unique( iterator start, iterator end, BinPred p );
```

The unique() algorithm removes all consecutive duplicate elements from the range [*start*,*end*). If the binary predicate *p* is given, then it is used to test to test two elements to see if they are duplicates.

The return value of unique() is an iterator to the end of the modified range.

unique() runs in linear time.

Related topics:

adjacent_find
remove
unique_copy

unique_copy

Syntax:

```
#include <algorithm>
iterator unique_copy( iterator start, iterator end, iterator result );
iterator unique_copy( iterator start, iterator end, iterator result, BinPred p );
```

The unique_copy() function copies the range [*start*,*end*) to *result*, removing all consecutive duplicate elements. If the binary predicate *p* is provided, then it is used to test two elements to see if they are duplicates.

The return value of unique_copy() is an iterator to the end of the new range.

unique_copy() runs in linear time.

Related topics:
adjacent_find
remove
unique

upper_bound

Syntax:

```
#include <algorithm>
iterator upper_bound( iterator start, iterator end, const TYPE& val );
iterator upper_bound( iterator start, iterator end, const TYPE& val, StrictWeakOrdering cmp );
```

The upper_bound() algorithm searches the ordered range [*start*,*end*) for the last location that *val* could be inserted without disrupting the order of the range.

If the strict weak ordering function object *cmp* is given, it is used to compare elements instead of the < operator.

upper_bound() runs in logarithmic time.

Related topics:
binary_search
equal_range

C++ Bitsets

C++ Bitsets give the programmer a set of bits as a data structure. Bitsets can be manipulated by various binary operators such as logical AND, OR, and so on.

Display all entries for C++ Bitsets on one page, or view entries individually:

Bitset Constructors	create new bitsets
Bitset Operators	compare and assign bitsets
any	true if any bits are set
count	returns the number of set bits
flip	reverses the bitset
none	true if no bits are set
reset	sets bits to zero
set	sets bits
size	number of bits that the bitset can hold
test	returns the value of a given bit
to_string	string representation of the bitset
to_ulong	returns an integer representation of the bitset

any

Syntax:

```
#include <bitset>
bool any();
```

The any() function returns true if any bit of the bitset is 1, otherwise, it returns false.

Related topics:

count

none

Bitset Constructors

Syntax:

```
#include <bitset>
bitset();
bitset( unsigned long val );
```

Bitsets can either be constructed with no arguments or with an unsigned long number val that will be converted into binary and inserted into the bitset. When creating bitsets, the number given in the place of the template determines how long the bitset is.

For example, the following code creates two bitsets and displays them:

```
// create a bitset that is 8 bits long
bitset<8> bs;
// display that bitset
for( int i = (int) bs.size()-1; i >= 0; i-- ) {
  cout << bs[i] << " ";
}
cout << endl;
// create a bitset out of a number
bitset<8> bs2( (long) 131 );
// display that bitset, too
for( int i = (int) bs2.size()-1; i >= 0; i-- ) {
  cout << bs2[i] << " ";
}
cout << endl;
```

Bitset Operators

Syntax:

```
#include <bitset>
!=, ==, &=, ^=, |=, ~, <<=, >>=, []
```

These operators all work with bitsets. They can be described as follows:

- != returns true if the two bitsets are not equal.
- == returns true if the two bitsets are equal.
- &= performs the AND operation on the two bitsets.
- ^= performs the XOR operation on the two bitsets.
- |= performs the OR operation on the two bitsets.
- ~ reverses the bitset (same as calling flip())
- <<= shifts the bitset to the left
- >>= shifts the bitset to the right
- [x] returns a reference to the xth bit in the bitset.

For example, the following code creates a bitset and shifts it to the left 4 places:

```
// create a bitset out of a number
bitset<8> bs2( (long) 131 );
cout << "bs2 is " << bs2 << endl;
// shift the bitset to the left by 4 digits
bs2 <<= 4;
cout << "now bs2 is " << bs2 << endl;
```

When the above code is run, it displays:

```
bs2 is 10000011
now bs2 is 00110000
```

any

Syntax:

```
#include <bitset>
bool any();
```

The any() function returns true if any bit of the bitset is 1, otherwise, it returns false.

Related topics:
count
none

count

Syntax:

```
#include <bitset>
size_type count();
```

The function count() returns the number of bits that are set to 1 in the bitset.

Related topics:
any

flip

Syntax:

```
#include <bitset>
bitset<N>& flip();
bitset<N>& flip( size_t pos );
```

The flip() function inverts all of the bits in the bitset, and returns the bitset. If *pos* is specified, only the bit at position *pos* is flipped.

none

Syntax:

```
#include <bitset>
bool none();
```

The none() function only returns true if none of the bits in the bitset are set to 1.

Related topics:

any

reset

Syntax:

```
#include <bitset>
bitset<N>& reset();
bitset<N>& reset( size_t pos );
```

The reset() function clears all of the bits in the bitset, and returns the bitset. If *pos* is specified, then only the bit at position *pos* is cleared.

set

Syntax:

```
#include <bitset>
bitset<N>& set();
bitset<N>& set( size_t pos, int val=1 );
```

The set() function sets all of the bits in the bitset, and returns the bitset. If *pos* is specified, then only the bit at position *pos* is set.

size

Syntax:

```
#include <bitset>
size_t size();
```

The size() function returns the number of bits that the bitset can hold.

test

Syntax:

```
#include <bitset>
bool test( size_t pos );
```

The function test() returns the value of the bit at position *pos*.

to_string

Syntax:

```
#include <bitset>
string to_string();
```

The to_string() function returns a string representation of the bitset.

Related topics:

to_ulong

to_ulong

Syntax:

```
#include <bitset>
unsigned long to_ulong();
```

The function to_ulong() returns the bitset, converted into an unsigned long integer.

Related topics:
to_string

C++ Double-ended Queues

Double-ended queues are like vectors, except that they allow fast insertions and deletions at the beginning (as well as the end) of the container.

Display all entries for C++ Double-ended Queues on one page, or view entries individually:

Container constructors	create dequeues and initialize them with some data
Container operators	compare, assign, and access elements of a dequeue
assign	assign elements to a dequeue
at	returns an element at a specific location
back	returns a reference to last element of a dequeue
begin	returns an iterator to the beginning of the dequeue
clear	removes all elements from the dequeue
empty	true if the dequeue has no elements
end	returns an iterator just past the last element of a dequeue
erase	removes elements from a dequeue
front	returns a reference to the first element of a dequeue
insert	inserts elements into the dequeue
max_size	returns the maximum number of elements that the dequeue can hold
pop_back	removes the last element of a dequeue
pop_front	removes the first element of the dequeue
push_back	add an element to the end of the dequeue
push_front	add an element to the front of the dequeue
rbegin	returns a reverse_iterator to the end of the dequeue
rend	returns a reverse_iterator to the beginning of the dequeue
resize	change the size of the dequeue
size	returns the number of items in the dequeue
swap	swap the contents of this dequeue with another

assign

Syntax:

```
#include <deque>
void assign( size_type num, const TYPE& val );
void assign( input_iterator start, input_iterator end );
```

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

This function will destroy the previous contents of the dequeue.

For example, the following code uses assign() to put 10 copies of the integer 42 into a vector:

```
vector<int> v;
v.assign( 10, 42 );
for( int i = 0; i < v.size(); i++ ) {
  cout << v[i] << " ";
}
cout << endl;
```

The above code displays the following output:

```
42 42 42 42 42 42 42 42 42 42
```

The next example shows how assign() can be used to copy one vector to another:

```
vector<int> v1;
for( int i = 0; i < 10; i++ ) {
  v1.push_back( i );
}

vector<int> v2;
v2.assign( v1.begin(), v1.end() );

for( int i = 0; i < v2.size(); i++ ) {
  cout << v2[i] << " ";
}
cout << endl;
```

When run, the above code displays the following output:

```
0 1 2 3 4 5 6 7 8 9
```

Related topics:

(C++ Strings) assign
insert
push_back
push_front

Container constructors

Syntax:

```
#include <deque>
container();
container( const container& c );
container( size_type num, const TYPE& val = TYPE() );
container( input_iterator start, input_iterator end );
~container();
```

The default deque constructor takes no arguments, creates a new instance of that deque.

The second constructor is a default copy constructor that can be used to create a new deque that is a copy of the given deque *c*.

The third constructor creates a deque with space for *num* objects. If *val* is specified, each of those objects will be given that value. For example, the following code creates a vector consisting of five copies of the integer 42:

```
vector<int> v1( 5, 42 );
```

The last constructor creates a deque that is initialized to contain the elements between *start* and *end*. For example:

```
// create a vector of random integers
cout << "original vector: ";
vector<int> v;
for( int i = 0; i < 10; i++ ) {
  int num = (int) rand() % 10;
  cout << num << " ";
  v.push_back( num );
}
cout << endl;

// find the first element of v that is even
vector<int>::iterator iter1 = v.begin();
while( iter1 != v.end() && *iter1 % 2 != 0 ) {
  iter1++;
}

// find the last element of v that is even
vector<int>::iterator iter2 = v.end();
do {
  iter2--;
} while( iter2 != v.begin() && *iter2 % 2 != 0 );

cout << "first even number: " << *iter1 << ", last even number: " << *iter2
<< endl;

cout << "new vector: ";
vector<int> v2( iter1, iter2 );
for( int i = 0; i < v2.size(); i++ ) {
  cout << v2[i] << " ";
```

```
}
cout << endl;
```

When run, this code displays the following output:

```
original vector: 1 9 7 9 2 7 2 1 9 8
first even number: 2, last even number: 8
new vector: 2 7 2 1 9
```

All of these constructors run in linear time except the first, which runs in constant time.

The default destructor is called when the dequeue should be destroyed.

Container operators

Syntax:

```
#include <deque>
TYPE& operator[]( size_type index );
const TYPE& operator[]( size_type index ) const;
container operator=(const container& c2);
bool operator==(const container& c1, const container& c2);
bool operator!=(const container& c1, const container& c2);
bool operator<(const container& c1, const container& c2);
bool operator>(const container& c1, const container& c2);
bool operator<=(const container& c1, const container& c2);
bool operator>=(const container& c1, const container& c2);
```

All of the C++ containers can be compared and assigned with the standard comparison operators: ==, !=, <=, >=, <, >, and =. Individual elements of a dequeue can be examined with the [] operator.

Performing a comparison or assigning one dequeue to another takes linear time. The [] operator runs in constant time.

Two `containers` are equal if:

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

Comparisons among dequeues are done lexicographically.

For example, the following code uses the [] operator to access all of the elements of a vector:

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

Related topics:

at

assign

Syntax:

```
#include <deque>
void assign( size_type num, const TYPE& val );
void assign( input_iterator start, input_iterator end );
```

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

This function will destroy the previous contents of the dequeue.

For example, the following code uses assign() to put 10 copies of the integer 42 into a vector:

```
vector<int> v;
v.assign( 10, 42 );
for( int i = 0; i < v.size(); i++ ) {
  cout << v[i] << " ";
}
cout << endl;
```

The above code displays the following output:

```
42 42 42 42 42 42 42 42 42 42
```

The next example shows how assign() can be used to copy one vector to another:

```
vector<int> v1;
for( int i = 0; i < 10; i++ ) {
  v1.push_back( i );
}

vector<int> v2;
v2.assign( v1.begin(), v1.end() );

for( int i = 0; i < v2.size(); i++ ) {
  cout << v2[i] << " ";
}
cout << endl;
```

When run, the above code displays the following output:

```
0 1 2 3 4 5 6 7 8 9
```

Related topics:

(C++ Strings) assign

insert

push_back

push_front

at

Syntax:

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

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

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 overrunns 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
Deque operators

back

Syntax:

```
#include <deque>
TYPE& back();
const TYPE& back() const;
```

The back() function returns a reference to the last element in the dequeue.

For example:

```
vector<int> v;
for( int i = 0; i < 5; i++ ) {
  v.push_back(i);
}
cout << "The first element is " << v.front()
     << " and the last element is " << v.back() << endl;
```

This code produces the following output:

```
The first element is 0 and the last element is 4
```

The back() function runs in constant time.

Related topics:

front

pop_back

begin

Syntax:

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

The function begin() returns an iterator to the first element of the dequeue. 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

clear

Syntax:

```
#include <deque>
void clear();
```

The function clear() deletes all of the elements in the dequeue. clear() runs in linear time.

Related topics:
erase

empty

Syntax:

```
#include <deque>
bool empty() const;
```

The empty() function returns true if the dequeue has no elements, false otherwise.

For example, the following code uses empty() as the stopping condition on a (C/C++ Keywords) while loop to clear a dequeue and display its contents in reverse order:

```
vector<int> v;
for( int i = 0; i < 5; i++ ) {
  v.push_back(i);
}
while( !v.empty() ) {
  cout << v.back() << endl;
  v.pop_back();
}
```

Related topics:

size

end

Syntax:

```
#include <deque>
iterator end();
const_iterator end() const;
```

The end() function returns an iterator just past the end of the dequeue.

Note that before you can access the last element of the dequeue 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 <deque>
iterator erase( iterator loc );
iterator erase( iterator start, iterator end );
```

The erase() function either deletes the element at location *loc*, or deletes the elements between *start* and *end* (including *start* but not including *end*). The return value is the element after the last element erased.

The first version of erase (the version that deletes a single element at location *loc*) runs in constant time for lists and linear time for vectors, dequeues, and strings. The multiple-element version of erase always takes linear time.

For example:

```
// Create a vector, load it with the first ten characters of the alphabet
vector<char> alphaVector;
for( int i=0; i < 10; i++ ) {
  alphaVector.push_back( i + 65 );
}
int size = alphaVector.size();
vector<char>::iterator startIterator;
vector<char>::iterator tempIterator;
for( int i=0; i < size; i++ ) {
  startIterator = alphaVector.begin();
  alphaVector.erase( startIterator );
  // Display the vector
  for( tempIterator = alphaVector.begin(); tempIterator != alphaVector.end();
tempIterator++ ) {
    cout << *tempIterator;
  }
  cout << endl;
}
```

That code would display the following output:

```
BCDEFGHIJ
CDEFGHIJ
DEFGHIJ
EFGHIJ
FGHIJ
GHIJ
HIJ
IJ
J
```

In the next example, erase() is called with two iterators to delete a range of elements from a vector:

```
// create a vector, load it with the first ten characters of the alphabet
vector<char> alphaVector;
for( int i=0; i < 10; i++ ) {
  alphaVector.push_back( i + 65 );
}
```

```
// display the complete vector
for( int i = 0; i < alphaVector.size(); i++ ) {
  cout << alphaVector[i];
}
cout << endl;

// use erase to remove all but the first two and last three elements
// of the vector
alphaVector.erase( alphaVector.begin()+2, alphaVector.end()-3 );
// display the modified vector
for( int i = 0; i < alphaVector.size(); i++ ) {
  cout << alphaVector[i];
}
cout << endl;
```

When run, the above code displays:

```

ABCDEFGHIJ
ABHIJ
```

Related topics:
clear
insert
pop_back
pop_front
(C++ Lists) remove
(C++ Lists) remove_if

front

Syntax:

```
#include <deque>
TYPE& front();
const TYPE& front() const;
```

The front() function returns a reference to the first element of the dequeue, and runs in constant time.

Related topics:

back
pop_front
push_front

insert

Syntax:

```
#include <deque>
iterator insert( iterator loc, const TYPE& val );
void insert( iterator loc, size_type num, const TYPE& val );
template<TYPE> void insert( iterator loc, input_iterator start,
input_iterator end );
```

The insert() function either:

- inserts *val* before *loc*, returning an iterator to the element inserted,
- inserts *num* copies of *val* before *loc*, or
- inserts the elements from *start* to *end* before *loc*.

For example:

```
// Create a vector, load it with the first 10 characters of the alphabet
vector<char> alphaVector;
for( int i=0; i < 10; i++ ) {
  alphaVector.push_back( i + 65 );
}

// Insert four C's into the vector
vector<char>::iterator theIterator = alphaVector.begin();
alphaVector.insert( theIterator, 4, 'C' );

// Display the vector
for( theIterator = alphaVector.begin(); theIterator != alphaVector.end();
theIterator++ )    {
  cout << *theIterator;
}
```

This code would display:

```
CCCCABCDEFGHIJ
```

Related topics:

assign
erase
(C++ Lists) merge
push_back
push_front
(C++ Lists) splice

max_size

Syntax:

```
#include <deque>
size_type max_size() const;
```

The max_size() function returns the maximum number of elements that the dequeue can hold. The max_size() function should not be confused with the size() or (C++ Strings) capacity() functions, which return the number of elements currently in the dequeue and the the number of elements that the dequeue will be able to hold before more memory will have to be allocated, respectively.

Related topics:
size

pop_back

Syntax:

```
#include <deque>
void pop_back();
```

The pop_back() function removes the last element of the dequeue.

pop_back() runs in constant time.

Related topics:

back
erase
pop_front
push_back

pop_front

Syntax:

```
#include <deque>
void pop_front();
```

The function pop_front() removes the first element of the dequeue.

The pop_front() function runs in constant time.

Related topics:
erase
front
pop_back
push_front

push_back

Syntax:

```
#include <deque>
void push_back( const TYPE& val );
```

The push_back() function appends *val* to the end of the dequeue.

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:
assign
insert
pop_back
push_front

push_front

Syntax:

```
#include <deque>
void push_front( const TYPE& val );
```

The push_front() function inserts *val* at the beginning of dequeue.

push_front() runs in constant time.

Related topics:

assign
front
insert
pop_front
push_back

rbegin

Syntax:

```
#include <deque>
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
```

The rbegin() function returns a reverse_iterator to the end of the current dequeue.

rbegin() runs in constant time.

Related topics:

begin
end
rend

rend

Syntax:

```
#include <deque>
reverse_iterator rend();
const_reverse_iterator rend() const;
```

The function rend() returns a reverse_iterator to the beginning of the current dequeue.

rend() runs in constant time.

Related topics:

begin
end
rbegin

resize

Syntax:

```
#include <deque>
void resize( size_type num, const TYPE& val = TYPE() );
```

The function resize() changes the size of the dequeue 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
(C++ Strings) capacity
size

size

Syntax:

```
#include <deque>
size_type size() const;
```

The size() function returns the number of elements in the current deque.

Related topics:

(C++ Strings) capacity
empty
(C++ Strings) length
max_size
resize

swap

Syntax:

```
#include <deque>
void swap( container& from );
```

The swap() function exchanges the elements of the current dequeue 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++ Lists

Lists are sequences of elements stored in a linked list. Compared to vectors, they allow fast insertions and deletions, but slower random access.

Display all entries for C++ Lists on one page, or view entries individually:

Container constructors	create lists and initialize them with some data
Container operators	assign and compare lists
assign	assign elements to a list
back	returns a reference to last element of a list
begin	returns an iterator to the beginning of the list
clear	removes all elements from the list
empty	true if the list has no elements
end	returns an iterator just past the last element of a list
erase	removes elements from a list
front	returns a reference to the first element of a list
insert	inserts elements into the list
max_size	returns the maximum number of elements that the list can hold
merge	merge two lists
pop_back	removes the last element of a list
pop_front	removes the first element of the list
push_back	add an element to the end of the list
push_front	add an element to the front of the list
rbegin	returns a reverse_iterator to the end of the list
remove	removes elements from a list
remove_if	removes elements conditionally
rend	returns a reverse_iterator to the beginning of the list
resize	change the size of the list
reverse	reverse the list
size	returns the number of items in the list
sort	sorts a list into ascending order
splice	merge two lists in constant time
swap	swap the contents of this list with another
unique	removes consecutive duplicate elements

assign

Syntax:

```
#include <list>
void assign( size_type num, const TYPE& val );
void assign( input_iterator start, input_iterator end );
```

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

This function will destroy the previous contents of the list.

For example, the following code uses assign() to put 10 copies of the integer 42 into a vector:

```
vector<int> v;
v.assign( 10, 42 );
for( int i = 0; i < v.size(); i++ ) {
  cout << v[i] << " ";
}
cout << endl;
```

The above code displays the following output:

```
42 42 42 42 42 42 42 42 42 42
```

The next example shows how assign() can be used to copy one vector to another:

```
vector<int> v1;
for( int i = 0; i < 10; i++ ) {
  v1.push_back( i );
}

vector<int> v2;
v2.assign( v1.begin(), v1.end() );

for( int i = 0; i < v2.size(); i++ ) {
  cout << v2[i] << " ";
}
cout << endl;
```

When run, the above code displays the following output:

```
0 1 2 3 4 5 6 7 8 9
```

Related topics:

(C++ Strings) assign
insert
push_back
push_front

Container constructors

Syntax:

```
#include <list>
container();
container( const container& c );
container( size_type num, const TYPE& val = TYPE() );
container( input_iterator start, input_iterator end );
~container();
```

The default list constructor takes no arguments, creates a new instance of that list.

The second constructor is a default copy constructor that can be used to create a new list that is a copy of the given list *c*.

The third constructor creates a list with space for *num* objects. If *val* is specified, each of those objects will be given that value. For example, the following code creates a vector consisting of five copies of the integer 42:

```
vector<int> v1( 5, 42 );
```

The last constructor creates a list that is initialized to contain the elements between *start* and *end*. For example:

```
// create a vector of random integers
cout << "original vector: ";
vector<int> v;
for( int i = 0; i < 10; i++ ) {
  int num = (int) rand() % 10;
  cout << num << " ";
  v.push_back( num );
}
cout << endl;

// find the first element of v that is even
vector<int>::iterator iter1 = v.begin();
while( iter1 != v.end() && *iter1 % 2 != 0 ) {
  iter1++;
}

// find the last element of v that is even
vector<int>::iterator iter2 = v.end();
do {
  iter2--;
} while( iter2 != v.begin() && *iter2 % 2 != 0 );

cout << "first even number: " << *iter1 << ", last even number: " << *iter2
<< endl;

cout << "new vector: ";
vector<int> v2( iter1, iter2 );
for( int i = 0; i < v2.size(); i++ ) {
  cout << v2[i] << " ";
```

```
}
cout << endl;
```

When run, this code displays the following output:

```
original vector: 1 9 7 9 2 7 2 1 9 8
first even number: 2, last even number: 8
new vector: 2 7 2 1 9
```

All of these constructors run in linear time except the first, which runs in constant time.

The default destructor is called when the list should be destroyed.

Container operators

Syntax:

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

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

Two lists are equal if:

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

Comparisons among lists are done lexicographically.

Related topics:
(C++ Strings) String operators
(C++ Strings) at
merge
unique

assign

Syntax:

```
#include <list>
void assign( size_type num, const TYPE& val );
void assign( input_iterator start, input_iterator end );
```

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

This function will destroy the previous contents of the list.

For example, the following code uses assign() to put 10 copies of the integer 42 into a vector:

```
vector<int> v;
v.assign( 10, 42 );
for( int i = 0; i < v.size(); i++ ) {
  cout << v[i] << " ";
}
cout << endl;
```

The above code displays the following output:

```
42 42 42 42 42 42 42 42 42 42
```

The next example shows how assign() can be used to copy one vector to another:

```
vector<int> v1;
for( int i = 0; i < 10; i++ ) {
  v1.push_back( i );
}

vector<int> v2;
v2.assign( v1.begin(), v1.end() );

for( int i = 0; i < v2.size(); i++ ) {
  cout << v2[i] << " ";
}
cout << endl;
```

When run, the above code displays the following output:

```
0 1 2 3 4 5 6 7 8 9
```

Related topics:
(C++ Strings) assign
insert
push_back
push_front

back

Syntax:

```
#include <list>
TYPE& back();
const TYPE& back() const;
```

The back() function returns a reference to the last element in the list.

For example:

```
vector<int> v;
for( int i = 0; i < 5; i++ ) {
  v.push_back(i);
}
cout << "The first element is " << v.front()
     << " and the last element is " << v.back() << endl;
```

This code produces the following output:

```
The first element is 0 and the last element is 4
```

The back() function runs in constant time.

Related topics:
front
pop_back

begin

Syntax:

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

The function begin() returns an iterator to the first element of the list. 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

clear

Syntax:

```
#include <list>
void clear();
```

The function clear() deletes all of the elements in the list. clear() runs in linear time.

Related topics:
erase

empty

Syntax:

```
#include <list>
bool empty() const;
```

The empty() function returns true if the list has no elements, false otherwise.

For example, the following code uses empty() as the stopping condition on a (C/C++ Keywords) while loop to clear a list and display its contents in reverse order:

```
vector<int> v;
for( int i = 0; i < 5; i++ ) {
  v.push_back(i);
}
while( !v.empty() ) {
  cout << v.back() << endl;
  v.pop_back();
}
```

Related topics:

size

end

Syntax:

```
#include <list>
iterator end();
const_iterator end() const;
```

The end() function returns an iterator just past the end of the list.

Note that before you can access the last element of the list 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 <list>
iterator erase( iterator loc );
iterator erase( iterator start, iterator end );
```

The erase() function either deletes the element at location *loc*, or deletes the elements between *start* and *end* (including *start* but not including *end*). The return value is the element after the last element erased.

The first version of erase (the version that deletes a single element at location *loc*) runs in constant time for lists and linear time for vectors, dequeues, and strings. The multiple-element version of erase always takes linear time.

For example:

```
// Create a vector, load it with the first ten characters of the alphabet
vector<char> alphaVector;
for( int i=0; i < 10; i++ ) {
  alphaVector.push_back( i + 65 );
}
int size = alphaVector.size();
vector<char>::iterator startIterator;
vector<char>::iterator tempIterator;
for( int i=0; i < size; i++ ) {
  startIterator = alphaVector.begin();
  alphaVector.erase( startIterator );
  // Display the vector
  for( tempIterator = alphaVector.begin(); tempIterator != alphaVector.end();
tempIterator++ ) {
    cout << *tempIterator;
  }
  cout << endl;
}
```

That code would display the following output:

```
BCDEFGHIJ
CDEFGHIJ
DEFGHIJ
EFGHIJ
FGHIJ
GHIJ
HIJ
IJ
J
```

In the next example, erase() is called with two iterators to delete a range of elements from a vector:

```
// create a vector, load it with the first ten characters of the alphabet
vector<char> alphaVector;
for( int i=0; i < 10; i++ ) {
  alphaVector.push_back( i + 65 );
}
```

```
// display the complete vector
for( int i = 0; i < alphaVector.size(); i++ ) {
  cout << alphaVector[i];
}
cout << endl;

// use erase to remove all but the first two and last three elements
// of the vector
alphaVector.erase( alphaVector.begin()+2, alphaVector.end()-3 );
// display the modified vector
for( int i = 0; i < alphaVector.size(); i++ ) {
  cout << alphaVector[i];
}
cout << endl;
```

When run, the above code displays:

```

ABCDEFGHIJ
ABHIJ
```

Related topics:
clear
insert
pop_back
pop_front
remove
remove_if

front

Syntax:

```
#include <list>
TYPE& front();
const TYPE& front() const;
```

The front() function returns a reference to the first element of the list, and runs in constant time.

Related topics:

back
pop_front
push_front

insert

Syntax:

```
#include <list>
iterator insert( iterator loc, const TYPE& val );
void insert( iterator loc, size_type num, const TYPE& val );
template<TYPE> void insert( iterator loc, input_iterator start,
input_iterator end );
```

The insert() function either:

- inserts *val* before *loc*, returning an iterator to the element inserted,
- inserts *num* copies of *val* before *loc*, or
- inserts the elements from *start* to *end* before *loc*.

For example:

```
// Create a vector, load it with the first 10 characters of the alphabet
vector<char> alphaVector;
for( int i=0; i < 10; i++ ) {
  alphaVector.push_back( i + 65 );
}

// Insert four C's into the vector
vector<char>::iterator theIterator = alphaVector.begin();
alphaVector.insert( theIterator, 4, 'C' );

// Display the vector
for( theIterator = alphaVector.begin(); theIterator != alphaVector.end();
theIterator++ )    {
  cout << *theIterator;
}
```

This code would display:

```
CCCCABCDEFGHIJ
```

Related topics:

assign
erase
merge
push_back
push_front
splice

max_size

Syntax:

```
#include <list>
size_type max_size() const;
```

The max_size() function returns the maximum number of elements that the list can hold. The max_size() function should not be confused with the size() or (C++ Strings) capacity() functions, which return the number of elements currently in the list and the the number of elements that the list will be able to hold before more memory will have to be allocated, respectively.

Related topics:
size

merge

Syntax:

```
#include <list>
void merge( list &lst );
void merge( list &lst, BinPred compfunction );
```

The function merge() merges the list with lst, producing a combined list that is ordered with respect to the < operator. If compfunction is specified, then it is used as the comparison function for the lists instead of <.

merge() runs in linear time.

Related topics:

Container operators
insert
splice

pop_back

Syntax:

```
#include <list>
void pop_back();
```

The pop_back() function removes the last element of the list.

pop_back() runs in constant time.

Related topics:

back
erase
pop_front
push_back

pop_front

Syntax:

```
#include <list>
void pop_front();
```

The function pop_front() removes the first element of the list.

The pop_front() function runs in constant time.

Related topics:

erase
front
pop_back
push_front

push_back

Syntax:

```
#include <list>
void push_back( const TYPE& val );
```

The push_back() function appends *val* to the end of the list.

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:
assign
insert
pop_back
push_front

push_front

Syntax:

```
#include <list>
void push_front( const TYPE& val );
```

The push_front() function inserts *val* at the beginning of list.

push_front() runs in constant time.

Related topics:

assign
front
insert
pop_front
push_back

rbegin

Syntax:

```
#include <list>
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
```

The rbegin() function returns a reverse_iterator to the end of the current list.

rbegin() runs in constant time.

Related topics:

begin
end
rend

remove

Syntax:

```
#include <list>
void remove( const TYPE &val );
```

The function remove() removes all elements that are equal to val from the list.

For example, the following code creates a list of the first 10 characters of the alphabet, then uses remove() to remove the letter 'E' from the list:

```
// Create a list that has the first 10 letters of the alphabet
list<char> charList;
for( int i=0; i < 10; i++ )
  charList.push_front( i + 65 );
// Remove all instances of 'E'
charList.remove( 'E' );
```

Remove runs in linear time.

Related topics:
erase
remove_if
unique

remove_if

Syntax:

```
#include <list>
void remove_if( UnPred pr );
```

The remove_if() function removes all elements from the list for which the unary predicate *pr* is true.

remove_if() runs in linear time.

Related topics:

erase
remove
unique

rend

Syntax:

```
#include <list>
reverse_iterator rend();
const_reverse_iterator rend() const;
```

The function rend() returns a reverse_iterator to the beginning of the current list.

rend() runs in constant time.

Related topics:

begin
end
rbegin

resize

Syntax:

```
#include <list>
void resize( size_type num, const TYPE& val = TYPE() );
```

The function resize() changes the size of the list 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
(C++ Strings) capacity
size

reverse

Syntax:

```
#include <list>
void reverse();
```

The function reverse() reverses the list, and takes linear time.

Related topics:
sort

size

Syntax:

```
#include <list>
size_type size() const;
```

The size() function returns the number of elements in the current list.

Related topics:
(C++ Strings) capacity
empty
(C++ Strings) length
max_size
resize

sort

Syntax:

```
#include <list>
void sort();
void sort( BinPred p );
```

The sort() function is used to sort lists into ascending order. Ordering is done via the < operator, unless *p* is specified, in which case it is used to determine if an element is less than another.

Sorting takes N log N time.

Related topics:
reverse

splice

Syntax:

```
#include <list>
void splice( iterator pos, list& lst );
void splice( iterator pos, list& lst, iterator del );
void splice( iterator pos, list& lst, iterator start, iterator end );
```

The splice() function inserts *lst* at location *pos*. If specified, the element(s) at *del* or from *start* to *end* are removed.

splice() simply moves elements from one list to another, and doesn't actually do any copying or deleting. Because of this, splice() runs in constant time.

Related topics:
insert
merge
swap

swap

Syntax:

```
#include <list>
void swap( container& from );
```

The swap() function exchanges the elements of the current list 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:

splice

unique

Syntax:

```
#include <list>
void unique();
void unique( BinPred pr );
```

The function unique() removes all consecutive duplicate elements from the list. Note that only consecutive duplicates are removed, which may require that you sort() the list first.

Equality is tested using the == operator, unless *pr* is specified as a replacement. The ordering of the elements in a list should not change after a call to unique().

unique() runs in linear time.

Related topics:

Container operators
remove
remove_if

C++ Maps

C++ Maps are sorted associative containers that contain unique key/value pairs. For example, you could create a map that associates a string with an integer, and then use that map to associate the number of days in each month with the name of each month.

Display all entries for C++ Maps on one page, or view entries individually:

Map constructors & destructors	default methods to allocate, copy, and deallocate maps
Map operators	assign, compare, and access elements of a map
begin	returns an iterator to the beginning of the map
clear	removes all elements from the map
count	returns the number of elements matching a certain key
empty	true if the map has no elements
end	returns an iterator just past the last element of a map
equal_range	returns iterators to the first and just past the last elements matching a specific key
erase	removes elements from a map
find	returns an iterator to specific elements
insert	insert items into a map
key_comp	returns the function that compares keys
lower_bound	returns an iterator to the first element greater than or equal to a certain value
max_size	returns the maximum number of elements that the map can hold
rbegin	returns a reverse_iterator to the end of the map
rend	returns a reverse_iterator to the beginning of the map
size	returns the number of items in the map
swap	swap the contents of this map with another
upper_bound	returns an iterator to the first element greater than a certain value
value_comp	returns the function that compares values

begin

Syntax:

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

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

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

```
map<string,int> stringCounts;
string str;

while( cin >> str ) stringCounts[str]++;

map<string,int>::iterator iter;
for( iter = stringCounts.begin(); iter != stringCounts.end(); iter++ ) {
  cout << "word: " << iter->first << ", count: " << iter->second << endl;
}
```

When given this input:

```
here are some words and here are some more words
```

...the above code generates this output:

```
word: and, count: 1
word: are, count: 2
word: here, count: 2
word: more, count: 1
word: some, count: 2
word: words, count: 2
```

Related topics:

end
rbegin
rend

Map Constructors & Destructors

Syntax:

```
#include <map>
map();
map( const map& m );
map( iterator start, iterator end );
map( iterator start, iterator end, const key_compare& cmp );
map( const key_compare& cmp );
~map();
```

The default constructor takes no arguments, creates a new instance of that map, and runs in constant time. The default copy constructor runs in linear time and can be used to create a new map that is a copy of the given map *m*.

You can also create a map that will contain a copy of the elements between *start* and *end*, or specify a comparison function *cmp*.

The default destructor is called when the map should be destroyed.

For example, the following code creates a map that associates a string with an integer:

```
struct strCmp {
  bool operator()( const char* s1, const char* s2 ) const {
    return strcmp( s1, s2 ) < 0;
  }
};

...

map<const char*, int, strCmp> ages;
ages["Homer"] = 38;
ages["Marge"] = 37;
ages["Lisa"] = 8;
ages["Maggie"] = 1;
ages["Bart"] = 11;

cout << "Bart is " << ages["Bart"] << " years old" << endl;
```

Related topics:

Map Operators

Map operators

Syntax:

```
#include <map>
TYPE& operator[]( const key_type& key );
map operator=(const map& c2);
bool operator==(const map& c1, const map& c2);
bool operator!=(const map& c1, const map& c2);
bool operator<(const map& c1, const map& c2);
bool operator>(const map& c1, const map& c2);
bool operator<=(const map& c1, const map& c2);
bool operator>=(const map& c1, const map& c2);
```

Maps can be compared and assigned with the standard comparison operators: ==, !=, <=, >=, <, >, and =. Individual elements of a map can be examined with the [] operator.

Performing a comparison or assigning one map to another takes linear time.

Two maps are equal if:

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

Comparisons among maps are done lexicographically.

For example, the following code defines a map between strings and integers and loads values into the map using the [] operator:

```
struct strCmp {
  bool operator()( const char* s1, const char* s2 ) const {
    return strcmp( s1, s2 ) < 0;
  }
};

map<const char*, int, strCmp> ages;
ages["Homer"] = 38;
ages["Marge"] = 37;
ages["Lisa"] = 8;
ages["Maggie"] = 1;
ages["Bart"] = 11;

cout << "Bart is " << ages["Bart"] << " years old" << endl;

cout << "In alphabetical order: " << endl;
for( map<const char*, int, strCmp>::iterator iter = ages.begin(); iter !=
ages.end(); iter++ ) {
  cout << (*iter).first << " is " << (*iter).second << " years old" << endl;
}
```

When run, the above code displays this output:

```
Bart is 11 years old
In alphabetical order:
Bart is 11 years old
Homer is 38 years old
Lisa is 8 years old
Maggie is 1 years old
Marge is 37 years old
```

Related topics:

insert

Map Constructors & Destructors

begin

Syntax:

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

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

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

```
map<string,int> stringCounts;
string str;

while( cin >> str ) stringCounts[str]++;

map<string,int>::iterator iter;
for( iter = stringCounts.begin(); iter != stringCounts.end(); iter++ ) {
  cout << "word: " << iter->first << ", count: " << iter->second << endl;
}
```

When given this input:

```
here are some words and here are some more words
```

...the above code generates this output:

```
word: and, count: 1
word: are, count: 2
word: here, count: 2
word: more, count: 1
word: some, count: 2
word: words, count: 2
```

Related topics:

end
rbegin
rend

clear

Syntax:

```
#include <map>
void clear();
```

The function clear() deletes all of the elements in the map. clear() runs in linear time.

Related topics:

erase

count

Syntax:

```
#include <map>
size_type count( const key_type& key );
```

The function count() returns the number of occurrences of *key* in the map.

count() should run in logarithmic time.

empty

Syntax:

```
#include <map>
bool empty() const;
```

The empty() function returns true if the map has no elements, false otherwise.

For example, the following code uses empty() as the stopping condition on a while loop to clear a map and display its contents in order:

```
struct strCmp {
  bool operator()( const char* s1, const char* s2 ) const {
    return strcmp( s1, s2 ) < 0;
  }
};

...

map<const char*, int, strCmp> ages;
ages["Homer"] = 38;
ages["Marge"] = 37;
ages["Lisa"] = 8;
ages["Maggie"] = 1;
ages["Bart"] = 11;

while( !ages.empty() ) {
  cout << "Erasing: " << (*ages.begin()).first << ", " <<
(*ages.begin()).second << endl;
  ages.erase( ages.begin() );
}
```

When run, the above code displays:

```
Erasing: Bart, 11
Erasing: Homer, 38
Erasing: Lisa, 8
Erasing: Maggie, 1
Erasing: Marge, 37
```

Related topics:

begin
erase
size

end

Syntax:

```
#include <map>
iterator end();
const_iterator end() const;
```

The end() function returns an iterator just past the end of the map.

Note that before you can access the last element of the map 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

equal_range

Syntax:

```
#include <map>
pair<iterator, iterator> equal_range( const key_type& key );
```

The function equal_range() returns two iterators - one to the first element that contains *key*, another to a point just after the last element that contains *key*.

erase

Syntax:

```
#include <map>
void erase( iterator pos );
void erase( iterator start, iterator end );
size_type erase( const key_type& key );
```

The erase function() either erases the element at *pos*, erases the elements between *start* and *end*, or erases all elements that have the value of *key*.

For example, the following code uses erase() in a while loop to incrementally clear a map and display its contents in order:

```
struct strCmp {
  bool operator()( const char* s1, const char* s2 ) const {
    return strcmp( s1, s2 ) < 0;
  }
};

...

map<const char*, int, strCmp> ages;
ages["Homer"] = 38;
ages["Marge"] = 37;
ages["Lisa"] = 8;
ages["Maggie"] = 1;
ages["Bart"] = 11;

while( !ages.empty() ) {
  cout << "Erasing: " << (*ages.begin()).first << ", " <<
(*ages.begin()).second << endl;
  ages.erase( ages.begin() );
}
```

When run, the above code displays:

```
Erasing: Bart, 11
Erasing: Homer, 38
Erasing: Lisa, 8
Erasing: Maggie, 1
Erasing: Marge, 37
```

Related topics:

begin
clear
empty
size

find

Syntax:

```
#include <map>
iterator find( const key_type& key );
```

The find() function returns an iterator to *key*, or an iterator to the end of the map if *key* is not found.

find() runs in logarithmic time.

For example, the following code uses the find() function to determine how many times a user entered a certain word:

```
  map<string,int> stringCounts;
  string str;

  while( cin >> str ) stringCounts[str]++;

  map<string,int>::iterator iter = stringCounts.find("spoon");
  if( iter != stringCounts.end() ) {
    cout << "You typed '" << iter->first << "' " << iter->second << " time(s)"
<< endl;
  }
```

When run with this input:

```
my spoon is too big.  my spoon is TOO big!  my SPOON is TOO big!  I am a
BANANA!
```

...the above code produces this output:

```
You typed 'spoon' 2 time(s)
```

insert

Syntax:

```
#include <map>
iterator insert( iterator i, const TYPE& pair );
void insert( input_iterator start, input_iterator end );
pair<iterator,bool> insert( const TYPE& pair );
```

The function insert() either:

- inserts *pair* after the element at *pos* (where *pos* is really just a suggestion as to where *pair* should go, since sets and maps are ordered), and returns an iterator to that element.
- inserts a range of elements from *start* to *end*.
- inserts *pair<key,val>*, but only if no element with key *key* already exists. The return value is an iterator to the element inserted (or an existing pair with key *key*), and a boolean which is true if an insertion took place.

For example, the following code uses the insert() function (along with the make_pair() function) to insert some data into a map and then displays that data:

```
map<string,int> theMap;
theMap.insert( make_pair( "Key 1", -1 ) );
theMap.insert( make_pair( "Another key!", 32 ) );
theMap.insert( make_pair( "Key the Three", 66667 ) );

map<string,int>::iterator iter;
for( iter = theMap.begin(); iter != theMap.end(); ++iter ) {
  cout << "Key: '" << iter->first << "', Value: " << iter->second << endl;
}
```

When run, the above code displays this output:

```
Key: 'Another key!', Value: 32
Key: 'Key 1', Value: -1
Key: 'Key the Three', Value: 66667
```

Note that because maps are sorted containers, the output is sorted by the key value. In this case, since the map key data type is string, the map is sorted alphabetically by key.

Related topics:
Map operators

key_comp

Syntax:

```
#include <map>
key_compare key_comp() const;
```

The function key_comp() returns the function that compares keys.

key_comp() runs in constant time.

Related topics:

value_comp

lower_bound

Syntax:

```
#include <map>
iterator lower_bound( const key_type& key );
```

The lower_bound() function returns an iterator to the first element which has a value greater than or equal to key.

lower_bound() runs in logarithmic time.

Related topics:
upper_bound

max_size

Syntax:

```
#include <map>
size_type max_size() const;
```

The max_size() function returns the maximum number of elements that the map can hold. The max_size() function should not be confused with the size() or (C++ Strings) capacity() functions, which return the number of elements currently in the map and the the number of elements that the map will be able to hold before more memory will have to be allocated, respectively.

Related topics:
size

rbegin

Syntax:

```
#include <map>
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
```

The rbegin() function returns a reverse_iterator to the end of the current map.

rbegin() runs in constant time.

Related topics:

begin
end
rend

rend

Syntax:

```
#include <map>
reverse_iterator rend();
const_reverse_iterator rend() const;
```

The function rend() returns a reverse_iterator to the beginning of the current map.

rend() runs in constant time.

Related topics:

begin
end
rbegin

size

Syntax:

```
#include <map>
size_type size() const;
```

The size() function returns the number of elements in the current map.

Related topics:

empty

max_size

swap

Syntax:

```
#include <map>
void swap( container& from );
```

The swap() function exchanges the elements of the current map 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**

upper_bound

Syntax:

```
#include <map>
iterator upper_bound( const key_type& key );
```

The function upper_bound() returns an iterator to the first element in the map with a key greater than *key*.

Related topics:
lower_bound

value_comp

Syntax:

```
#include <map>
value_compare value_comp() const;
```

The value_comp() function returns the function that compares values.

value_comp() runs in constant time.

Related topics:

key_comp

C++ Multimaps

C++ Multimaps are like maps, in that they are sorted associative containers, but differ from maps in that they allow duplicate keys.

Display all entries for C++ Multimaps on one page, or view entries individually:

Multimap constructors & destructors	default methods to allocate, copy, and deallocate multimaps
Multimap operators	assign and compare multimaps
begin	returns an iterator to the beginning of the multimap
clear	removes all elements from the multimap
count	returns the number of elements matching a certain key
empty	true if the multimap has no elements
end	returns an iterator just past the last element of a multimap
equal_range	returns iterators to the first and just past the last elements matching a specific key
erase	removes elements from a multimap
find	returns an iterator to specific elements
insert	inserts items into a multimap
key_comp	returns the function that compares keys
lower_bound	returns an iterator to the first element greater than or equal to a certain value
max_size	returns the maximum number of elements that the multimap can hold
rbegin	returns a reverse_iterator to the end of the multimap
rend	returns a reverse_iterator to the beginning of the multimap
size	returns the number of items in the multimap
swap	swap the contents of this multimap with another
upper_bound	returns an iterator to the first element greater than a certain value
value_comp	returns the function that compares values

begin

Syntax:

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

The function begin() returns an iterator to the first element of the multimap. 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

Multimap constructors & destructors

Syntax:

```
#include <map>
multimap();
multimap( const multimap& c );
multimap( iterator begin, iterator end,
          const key_compare& cmp = Compare(), const allocator& alloc =
Allocator() );
~multimap();
```

Multimaps have several constructors:

- The default constructor takes no arguments, creates a new instance of that multimap, and runs in constant time.
- The default copy constructor runs in linear time and can be used to create a new multimap that is a copy of the given multimap *c*.
- Multimaps can also be created from a range of elements defined by *begin* and *end*. When using this constructor, an optional comparison function *cmp* and allocator *alloc* can also be provided.

The default destructor is called when the multimap should be destroyed.

The template definition of multimaps requires that both a key type and value type be supplied. For example, you can instantiate a multimap that maps strings to integers with this statement:

```
multimap<string,int> m;
```

You can also supply a comparison function and an allocator in the template:

```
multimap<string,int,myComp,myAlloc> m;
```

For example, the following code uses a multimap to associate a series of employee names with numerical IDs:

```
multimap<string,int> m;

int employeeID = 0;
m.insert( pair<string,int>("Bob Smith",employeeID++) );
m.insert( pair<string,int>("Bob Thompson",employeeID++) );
m.insert( pair<string,int>("Bob Smithey",employeeID++) );
m.insert( pair<string,int>("Bob Smith",employeeID++) );

cout << "Number of employees named 'Bob Smith': " << m.count("Bob Smith") <<
endl;
cout << "Number of employees named 'Bob Thompson': " << m.count("Bob
Thompson") << endl;
cout << "Number of employees named 'Bob Smithey': " << m.count("Bob
Smithey") << endl;

cout << "Employee list: " << endl;
```

```
  for( multimap<string, int>::iterator iter = m.begin(); iter != m.end();
++iter ) {
    cout << " Name: " << iter->first << ", ID #" << iter->second << endl;
  }
```

When run, the above code produces the following output. Note that the employee list is displayed in alphabetical order, because multimaps are sorted associative containers:

```

  Number of employees named 'Bob Smith': 2
  Number of employees named 'Bob Thompson': 1
  Number of employees named 'Bob Smithey': 1
  Employee list:
   Name: Bob Smith, ID #0
   Name: Bob Smith, ID #3
   Name: Bob Smithey, ID #2
   Name: Bob Thompson, ID #1
```

Related topics:

count

insert

Multimap operators

Syntax:

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

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

Two multimaps are equal if:

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

Comparisons among multimaps are done lexicographically.

Related topics:
Multimap Constructors

begin

Syntax:

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

The function begin() returns an iterator to the first element of the multimap. 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

clear

Syntax:

```
#include <map>
void clear();
```

The function clear() deletes all of the elements in the multimap. clear() runs in linear time.

Related topics:
(C++ Lists) erase

count

Syntax:

```
#include <map>
size_type count( const key_type& key );
```

The function count() returns the number of occurrences of *key* in the multimap.

count() should run in logarithmic time.

empty

Syntax:

```
#include <map>
bool empty() const;
```

The empty() function returns true if the multimap has no elements, false otherwise.

For example, the following code uses empty() as the stopping condition on a (C/C++ Keywords) while loop to clear a multimap and display its contents in reverse order:

```
vector<int> v;
for( int i = 0; i < 5; i++ ) {
  v.push_back(i);
}
while( !v.empty() ) {
  cout << v.back() << endl;
  v.pop_back();
}
```

Related topics:

size

end

Syntax:

```
#include <map>
iterator end();
const_iterator end() const;
```

The end() function returns an iterator just past the end of the multimap.

Note that before you can access the last element of the multimap 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

equal_range

Syntax:

```
#include <map>
pair<iterator, iterator> equal_range( const key_type& key );
```

The function equal_range() returns two iterators - one to the first element that contains *key*, another to a point just after the last element that contains *key*.

For example, here is a hypothetical input-configuration loader using multimaps, strings and equal_range():

```
  multimap<string,pair<int,int> > input_config;

  // read configuration from file "input.conf" to input_config
  readConfigFile( input_config, "input.conf" );

  pair<multimap<string,pair<int,int> >::iterator,multimap<string,pair<int,int>
>::iterator> ii;
  multimap<string,pair<int,int> >::iterator i;

  ii = input_config.equal_range("key");         // keyboard key-bindings
  // we can iterate over a range just like with begin() and end()
  for( i = ii.first; i != ii.second; ++i ) {
    // add a key binding with this key and output
    bindkey(i->second.first, i->second.second);
  }

  ii = input_config.equal_range("joyb");        // joystick button
key-bindings
  for( i = ii.first; i != ii.second; ++i ) {
    // add a key binding with this joystick button and output
    bindjoyb(i->second.first, i->second.second);
  }
```

erase

Syntax:

```
#include <map>
void erase( iterator pos );
void erase( iterator start, iterator end );
size_type erase( const key_type& key );
```

The erase function() either erases the element at *pos*, erases the elements between *start* and *end*, or erases all elements that have the value of *key*.

find

Syntax:

```
#include <map>
iterator find( const key_type& key );
```

The find() function returns an iterator to *key*, or an iterator to the end of the multimap if *key* is not found.

find() runs in logarithmic time.

insert

Syntax:

```
#include <map>
iterator insert( iterator pos, const TYPE& val );
iterator insert( const TYPE& val );
void insert( input_iterator start, input_iterator end );
```

The function insert() either:

- inserts *val* after the element at *pos* (where *pos* is really just a suggestion as to where *val* should go, since multimaps are ordered), and returns an iterator to that element.
- inserts *val* into the multimap, returning an iterator to the element inserted.
- inserts a range of elements from *start* to *end*.

For example, the following code uses the insert() function to add several <name,ID> pairs to a employee multimap:

```
 multimap<string,int> m;

 int employeeID = 0;
 m.insert( pair<string,int>("Bob Smith",employeeID++) );
 m.insert( pair<string,int>("Bob Thompson",employeeID++) );
 m.insert( pair<string,int>("Bob Smithey",employeeID++) );
 m.insert( pair<string,int>("Bob Smith",employeeID++) );

 cout << "Number of employees named 'Bob Smith': " << m.count("Bob Smith") <<
endl;
 cout << "Number of employees named 'Bob Thompson': " << m.count("Bob
Thompson") << endl;
 cout << "Number of employees named 'Bob Smithey': " << m.count("Bob
Smithey") << endl;

 cout << "Employee list: " << endl;
 for( multimap<string, int>::iterator iter = m.begin(); iter != m.end();
++iter ) {
   cout << " Name: " << iter->first << ", ID #" << iter->second << endl;
 }
```

When run, the above code produces the following output:

```
 Number of employees named 'Bob Smith': 2
 Number of employees named 'Bob Thompson': 1
 Number of employees named 'Bob Smithey': 1
 Employee list:
  Name: Bob Smith, ID #0
  Name: Bob Smith, ID #3
  Name: Bob Smithey, ID #2
  Name: Bob Thompson, ID #1
```

key_comp

Syntax:

```
#include <map>
key_compare key_comp() const;
```

The function key_comp() returns the function that compares keys.

key_comp() runs in constant time.

Related topics:
value_comp

lower_bound

Syntax:

```
#include <map>
iterator lower_bound( const key_type& key );
```

The lower_bound() function returns an iterator to the first element which has a value greater than or equal to key.

lower_bound() runs in logarithmic time.

Related topics:
upper_bound

max_size

Syntax:

```
#include <map>
size_type max_size() const;
```

The max_size() function returns the maximum number of elements that the multimap can hold. The max_size() function should not be confused with the size() or (C++ Strings) capacity() functions, which return the number of elements currently in the multimap and the the number of elements that the multimap will be able to hold before more memory will have to be allocated, respectively.

Related topics:

size

rbegin

Syntax:

```
#include <map>
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
```

The rbegin() function returns a reverse_iterator to the end of the current multimap.

rbegin() runs in constant time.

Related topics:

begin
end
rend

rend

Syntax:

```
#include <map>
reverse_iterator rend();
const_reverse_iterator rend() const;
```

The function rend() returns a reverse_iterator to the beginning of the current multimap.

rend() runs in constant time.

Related topics:
begin
end
rbegin

size

Syntax:

```
#include <map>
size_type size() const;
```

The size() function returns the number of elements in the current multimap.

Related topics:
(C++ Strings) capacity
empty
(C++ Strings) length
max_size
(C++ Strings) resize

swap

Syntax:

```
#include <map>
void swap( container& from );
```

The swap() function exchanges the elements of the current multimap 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

upper_bound

Syntax:

```
#include <map>
iterator upper_bound( const key_type& key );
```

The function upper_bound() returns an iterator to the first element in the multimap with a key greater than *key*.

Related topics:
lower_bound

value_comp

Syntax:

```
#include <map>
value_compare value_comp() const;
```

The value_comp() function returns the function that compares values.

value_comp() runs in constant time.

Related topics:
key_comp

C++ Multisets

C++ Multisets are like sets, in that they are associative containers containing a sorted set of objects, but differ in that they allow duplicate objects.

Display all entries for C++ Multisets on one page, or view entries individually:

Container constructors & destructors	default methods to allocate, copy, and deallocate multisets
Container operators	assign and compare multisets
begin	returns an iterator to the beginning of the multiset
clear	removes all elements from the multiset
count	returns the number of elements matching a certain key
empty	true if the multiset has no elements
end	returns an iterator just past the last element of a multiset
equal_range	returns iterators to the first and just past the last elements matching a specific key
erase	removes elements from a multiset
find	returns an iterator to specific elements
insert	inserts items into a multiset
key_comp	returns the function that compares keys
lower_bound	returns an iterator to the first element greater than or equal to a certain value
max_size	returns the maximum number of elements that the multiset can hold
rbegin	returns a reverse_iterator to the end of the multiset
rend	returns a reverse_iterator to the beginning of the multiset
size	returns the number of items in the multiset
swap	swap the contents of this multiset with another
upper_bound	returns an iterator to the first element greater than a certain value
value_comp	returns the function that compares values

begin

Syntax:

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

The function begin() returns an iterator to the first element of the multiset. 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

Container constructors & destructors

Syntax:

```
#include <set>
container();
container( const container& c );
~container();
```

Every multiset has a default constructor, copy constructor, and destructor.

The default constructor takes no arguments, creates a new instance of that multiset, and runs in constant time. The default copy constructor runs in linear time and can be used to create a new multiset that is a copy of the given multiset *c*.

The default destructor is called when the multiset should be destroyed.

For example, the following code creates a pointer to a vector of integers and then uses the default multiset constructor to allocate a memory for a new vector:

```
vector<int>* v;
v = new vector<int>();
```

Related topics:
(C++ Strings) resize

Container operators

Syntax:

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

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

Two multisets are equal if:

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

Comparisons among multisets are done lexicographically.

Related topics:
(C++ Strings) String operators
(C++ Strings) at
(C++ Lists) merge
(C++ Lists) unique

begin

Syntax:

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

The function begin() returns an iterator to the first element of the multiset. 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

clear

Syntax:

```
#include <set>
void clear();
```

The function clear() deletes all of the elements in the multiset. clear() runs in linear time.

Related topics:
(C++ Lists) erase

count

Syntax:

```
#include <set>
size_type count( const key_type& key );
```

The function count() returns the number of occurrences of *key* in the multiset.

count() should run in logarithmic time.

empty

Syntax:

```
#include <set>
bool empty() const;
```

The empty() function returns true if the multiset has no elements, false otherwise.

For example, the following code uses empty() as the stopping condition on a (C/C++ Keywords) while loop to clear a multiset and display its contents in reverse order:

```
vector<int> v;
for( int i = 0; i < 5; i++ ) {
  v.push_back(i);
}
while( !v.empty() ) {
  cout << v.back() << endl;
  v.pop_back();
}
```

Related topics:

size

end

Syntax:

```
#include <set>
iterator end();
const_iterator end() const;
```

The end() function returns an iterator just past the end of the multiset.

Note that before you can access the last element of the multiset 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

equal_range

Syntax:

```
#include <set>
pair<iterator, iterator> equal_range( const key_type& key );
```

The function equal_range() returns two iterators - one to the first element that contains *key*, another to a point just after the last element that contains *key*.

erase

Syntax:

```
#include <set>
void erase( iterator pos );
void erase( iterator start, iterator end );
size_type erase( const key_type& key );
```

The erase function() either erases the element at *pos*, erases the elements between *start* and *end*, or erases all elements that have the value of *key*.

find

Syntax:

```
#include <set>
iterator find( const key_type& key );
```

The find() function returns an iterator to *key*, or an iterator to the end of the multiset if *key* is not found.

find() runs in logarithmic time.

insert

Syntax:

```
#include <set>
iterator insert( iterator pos, const TYPE& val );
iterator insert( const TYPE& val );
void insert( input_iterator start, input_iterator end );
```

The function insert() either:

- inserts *val* after the element at *pos* (where *pos* is really just a suggestion as to where *val* should go, since multisets and multimaps are ordered), and returns an iterator to that element.
- inserts *val* into the multiset, returning an iterator to the element inserted.
- inserts a range of elements from *start* to *end*.

key_comp

Syntax:

```
#include <set>
key_compare key_comp() const;
```

The function key_comp() returns the function that compares keys.

key_comp() runs in constant time.

Related topics:

value_comp

lower_bound

Syntax:

```
#include <set>
iterator lower_bound( const key_type& key );
```

The lower_bound() function returns an iterator to the first element which has a value greater than or equal to key.

lower_bound() runs in logarithmic time.

Related topics:
upper_bound

max_size

Syntax:

```
#include <set>
size_type max_size() const;
```

The max_size() function returns the maximum number of elements that the multiset can hold. The max_size() function should not be confused with the size() or (C++ Strings) capacity() functions, which return the number of elements currently in the multiset and the the number of elements that the multiset will be able to hold before more memory will have to be allocated, respectively.

Related topics:

size

rbegin

Syntax:

```
#include <set>
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
```

The rbegin() function returns a reverse_iterator to the end of the current multiset.

rbegin() runs in constant time.

Related topics:

begin
end
rend

rend

Syntax:

```
#include <set>
reverse_iterator rend();
const_reverse_iterator rend() const;
```

The function rend() returns a reverse_iterator to the beginning of the current multiset.

rend() runs in constant time.

Related topics:

begin
end
rbegin

size

Syntax:

```
#include <set>
size_type size() const;
```

The size() function returns the number of elements in the current multiset.

Related topics:

(C++ Strings) capacity
empty
(C++ Strings) length
max_size
(C++ Strings) resize

swap

Syntax:

```
#include <set>
void swap( container& from );
```

The swap() function exchanges the elements of the current multiset 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

upper_bound

Syntax:

```
#include <set>
iterator upper_bound( const key_type& key );
```

The function upper_bound() returns an iterator to the first element in the multiset with a key greater than *key*.

Related topics:
lower_bound

value_comp

Syntax:

```
#include <set>
value_compare value_comp() const;
```

The value_comp() function returns the function that compares values.

value_comp() runs in constant time.

Related topics:
key_comp

C++ Priority Queues

C++ Priority Queues are like queues, but the elements inside the queue are ordered by some predicate.

Display all entries for C++ Priority Queues on one page, or view entries individually:

Priority queue constructors	construct a new priority queue
empty	true if the priority queue has no elements
pop	removes the top element of a priority queue
push	adds an element to the end of the priority queue
size	returns the number of items in the priority queue
top	returns the top element of the priority queue

empty

Syntax:

```
#include <queue>
bool empty() const;
```

The empty() function returns true if the priority queue has no elements, false otherwise.

For example, the following code uses empty() as the stopping condition on a (C/C++ Keywords) while loop to clear a priority queue and display its contents in reverse order:

```
vector<int> v;
for( int i = 0; i < 5; i++ ) {
  v.push_back(i);
}
while( !v.empty() ) {
  cout << v.back() << endl;
  v.pop_back();
}
```

Related topics:

size

Priority queue constructors

Syntax:

```
  #include <queue>
  priority_queue( const Compare& cmp = Compare(), const Container& c =
Container() );
  priority_queue( input_iterator start, input_iterator end, const Compare&
comp = Compare(), const Container& c = Container() );
```

Priority queues can be constructed with an optional compare function *cmp* and an optional container *c*. If *start* and *end* are specified, the priority queue will be constructed with the elements between *start* and *end*.

empty

Syntax:

```
#include <queue>
bool empty() const;
```

The empty() function returns true if the priority queue has no elements, false otherwise.

For example, the following code uses empty() as the stopping condition on a (C/C++ Keywords) while loop to clear a priority queue and display its contents in reverse order:

```
vector<int> v;
for( int i = 0; i < 5; i++ ) {
  v.push_back(i);
}
while( !v.empty() ) {
  cout << v.back() << endl;
  v.pop_back();
}
```

Related topics:

size

pop

Syntax:

```
#include <queue>
void pop();
```

The function pop() removes the top element of the priority queue and discards it.

Related topics:

push

top

push

Syntax:

```
#include <queue>
void push( const TYPE& val );
```

The function push() adds *val* to the end of the current priority queue.

For example, the following code uses the push() function to add ten integers to the end of a queue:

```
queue<int> q;
for( int i=0; i < 10; i++ )
  q.push(i);
```

size

Syntax:

```
#include <queue>
size_type size() const;
```

The size() function returns the number of elements in the current priority queue.

Related topics:

(C++ Strings) capacity
empty
(C++ Strings) length
(C++ Multimaps) max_size
(C++ Strings) resize

top

Syntax:

```
#include <queue>
TYPE& top();
```

The function top() returns a reference to the top element of the priority queue.

For example, the following code removes all of the elements from a stack and uses top() to display them:

```
while( !s.empty() ) {
  cout << s.top() << " ";
  s.pop();
}
```

Related topics:

pop

C++ Queues

The C++ Queue is a container adapter that gives the programmer a FIFO (first-in, first-out) data structure.

Display all entries for C++ Queues on one page, or view entries individually:

Queue constructor	construct a new queue
back	returns a reference to last element of a queue
empty	true if the queue has no elements
front	returns a reference to the first element of a queue
pop	removes the first element of a queue
push	adds an element to the end of the queue
size	returns the number of items in the queue

back

Syntax:

```
#include <queue>
TYPE& back();
const TYPE& back() const;
```

The back() function returns a reference to the last element in the queue.

For example:

```
queue<int> q;
for( int i = 0; i < 5; i++ ) {
  q.push(i);
}
cout << "The first element is " << q.front()
     << " and the last element is " << q.back() << endl;
```

This code produces the following output:

```
The first element is 0 and the last element is 4
```

The back() function runs in constant time.

Related topics:

front

(C++ Lists) pop_back

Queue constructor

Syntax:

```
#include <queue>
queue();
queue( const Container& con );
```

Queues have a default constructor as well as a copy constructor that will create a new queue out of the container *con*.

For example, the following code creates a queue of strings, populates it with input from the user, and then displays it back to the user:

```
queue<string> waiting_line;
while( waiting_line.size() < 5 ) {
  cout << "Welcome to the line, please enter your name: ";
  string s;
  getline( cin, s );
  waiting_line.push(s);
}

while( !waiting_line.empty() ) {
  cout << "Now serving: " << waiting_line.front() << endl;
  waiting_line.pop();
}
```

When run, the above code might produce this output:

```
Welcome to the line, please enter your name: Nate
Welcome to the line, please enter your name: lizzy
Welcome to the line, please enter your name: Robert B. Parker
Welcome to the line, please enter your name: ralph
Welcome to the line, please enter your name: Matthew
Now serving: Nate
Now serving: lizzy
Now serving: Robert B. Parker
Now serving: ralph
Now serving: Matthew
```

back

Syntax:

```
#include <queue>
TYPE& back();
const TYPE& back() const;
```

The back() function returns a reference to the last element in the queue.

For example:

```
queue<int> q;
for( int i = 0; i < 5; i++ ) {
  q.push(i);
}
cout << "The first element is " << q.front()
     << " and the last element is " << q.back() << endl;
```

This code produces the following output:

```
The first element is 0 and the last element is 4
```

The back() function runs in constant time.

Related topics:
front
(C++ Lists) pop_back

empty

Syntax:

```
#include <queue>
bool empty() const;
```

The empty() function returns true if the queue has no elements, false otherwise.

For example, the following code uses empty() as the stopping condition on a while loop to clear a queue while displaying its contents:

```
queue<int> q;
for( int i = 0; i < 5; i++ ) {
  q.push(i);
}
while( !q.empty() ) {
  cout << q.front() << endl;
  q.pop();
}
```

Related topics:

size

front

Syntax:

```
#include <queue>
TYPE& front();
const TYPE& front() const;
```

The front() function returns a reference to the first element of the queue, and runs in constant time.

Related topics:

back
(C++ Lists) pop_front
(C++ Lists) push_front

pop

Syntax:

```
#include <queue>
void pop();
```

The function pop() removes the first element of the queue and discards it.

Related topics:

push

(C++ Priority Queues) top

push

Syntax:

```
#include <queue>
void push( const TYPE& val );
```

The function push() adds *val* to the end of the current queue.

For example, the following code uses the push() function to add ten integers to the end of a queue:

```
queue<int> q;
for( int i=0; i < 10; i++ ) {
  q.push(i);
}
```

Related topics:

pop

size

Syntax:

```
#include <queue>
size_type size() const;
```

The size() function returns the number of elements in the current queue.

Related topics:

empty
(C++ Strings) capacity
(C++ Strings) length
(C++ Multimaps) max_size
(C++ Strings) resize

C++ Sets

The C++ Set is an associative container that contains a sorted set of unique objects.

Display all entries for C++ Sets on one page, or view entries individually:

Set constructors & destructors	default methods to allocate, copy, and deallocate sets
Set operators	assign and compare sets
begin	returns an iterator to the beginning of the set
clear	removes all elements from the set
count	returns the number of elements matching a certain key
empty	true if the set has no elements
end	returns an iterator just past the last element of a set
equal_range	returns iterators to the first and just past the last elements matching a specific key
erase	removes elements from a set
find	returns an iterator to specific elements
insert	insert items into a set
key_comp	returns the function that compares keys
lower_bound	returns an iterator to the first element greater than or equal to a certain value
max_size	returns the maximum number of elements that the set can hold
rbegin	returns a reverse_iterator to the end of the set
rend	returns a reverse_iterator to the beginning of the set
size	returns the number of items in the set
swap	swap the contents of this set with another
upper_bound	returns an iterator to the first element greater than a certain value
value_comp	returns the function that compares values

begin

Syntax:

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

The function begin() returns an iterator to the first element of the set. 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

Set constructors & destructors

Syntax:

```
#include <set>
set();
set( const set& c );
~set();
```

Every set has a default constructor, copy constructor, and destructor.

The default constructor takes no arguments, creates a new instance of that set, and runs in constant time. The default copy constructor runs in linear time and can be used to create a new set that is a copy of the given set *c*.

The default destructor is called when the set should be destroyed.

For example, the following code creates a pointer to a vector of integers and then uses the default set constructor to allocate a memory for a new vector:

```
vector<int>* v;
v = new vector<int>();
```

Related topics:

(C++ Strings) resize

Set operators

Syntax:

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

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

Two sets are equal if:

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

Comparisons among sets are done lexicographically.

Related topics:
(C++ Strings) String operators
(C++ Strings) at
(C++ Lists) merge
(C++ Lists) unique

begin

Syntax:

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

The function begin() returns an iterator to the first element of the set. 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

clear

Syntax:

```
#include <set>
void clear();
```

The function clear() deletes all of the elements in the set. clear() runs in linear time.

Related topics:
(C++ Lists) erase

count

Syntax:

```
#include <set>
size_type count( const key_type& key );
```

The function count() returns the number of occurrences of *key* in the set.

count() should run in logarithmic time.

empty

Syntax:

```
#include <set>
bool empty() const;
```

The empty() function returns true if the set has no elements, false otherwise.

For example, the following code uses empty() as the stopping condition on a (C/C++ Keywords) while loop to clear a set and display its contents in reverse order:

```
vector<int> v;
for( int i = 0; i < 5; i++ ) {
  v.push_back(i);
}
while( !v.empty() ) {
  cout << v.back() << endl;
  v.pop_back();
}
```

Related topics:

size

end

Syntax:

```
#include <set>
iterator end();
const_iterator end() const;
```

The end() function returns an iterator just past the end of the set.

Note that before you can access the last element of the set 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

equal_range

Syntax:

```
#include <set>
pair<iterator, iterator> equal_range( const key_type& key );
```

The function equal_range() returns two iterators - one to the first element that contains *key*, another to a point just after the last element that contains *key*.

erase

Syntax:

```
#include <set>
void erase( iterator pos );
void erase( iterator start, iterator end );
size_type erase( const key_type& key );
```

The erase function() either erases the element at *pos*, erases the elements between *start* and *end*, or erases all elements that have the value of *key*.

find

Syntax:

```
#include <set>
iterator find( const key_type& key );
```

The find() function returns an iterator to *key*, or an iterator to the end of the set if *key* is not found.

find() runs in logarithmic time.

insert

Syntax:

```
#include <set>
iterator insert( iterator i, const TYPE& val );
void insert( input_iterator start, input_iterator end );
pair<iterator,bool> insert( const TYPE& val );
```

The function insert() either:

- inserts *val* after the element at *pos* (where *pos* is really just a suggestion as to where *val* should go, since sets and maps are ordered), and returns an iterator to that element.
- inserts a range of elements from *start* to *end*.
- inserts *val*, but only if *val* doesn't already exist. The return value is an iterator to the element inserted, and a boolean describing whether an insertion took place.

Related topics:
(C++ Maps) Map operators

key_comp

Syntax:

```
#include <set>
key_compare key_comp() const;
```

The function key_comp() returns the function that compares keys.

key_comp() runs in constant time.

Related topics:

value_comp

lower_bound

Syntax:

```
#include <set>
iterator lower_bound( const key_type& key );
```

The lower_bound() function returns an iterator to the first element which has a value greater than or equal to key.

lower_bound() runs in logarithmic time.

Related topics:
upper_bound

max_size

Syntax:

```
#include <set>
size_type max_size() const;
```

The max_size() function returns the maximum number of elements that the set can hold. The max_size() function should not be confused with the size() or (C++ Strings) capacity() functions, which return the number of elements currently in the set and the the number of elements that the set will be able to hold before more memory will have to be allocated, respectively.

Related topics:
size

rbegin

Syntax:

```
#include <set>
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
```

The rbegin() function returns a reverse_iterator to the end of the current set.

rbegin() runs in constant time.

Related topics:

begin
end
rend

rend

Syntax:

```
#include <set>
reverse_iterator rend();
const_reverse_iterator rend() const;
```

The function rend() returns a reverse_iterator to the beginning of the current set.

rend() runs in constant time.

Related topics:

begin
end
rbegin

size

Syntax:

```
#include <set>
size_type size() const;
```

The size() function returns the number of elements in the current set.

Related topics:

(C++ Strings) capacity
empty
(C++ Strings) length
max_size
(C++ Strings) resize

swap

Syntax:

```
#include <set>
void swap( container& from );
```

The swap() function exchanges the elements of the current set 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**

upper_bound

Syntax:

```
#include <set>
iterator upper_bound( const key_type& key );
```

The function upper_bound() returns an iterator to the first element in the set with a key greater than *key*.

Related topics:
lower_bound

C++ Stacks

The C++ Stack is a container adapter that gives the programmer the functionality of a stack -- specifically, a FILO (first-in, last-out) data structure.

Display all entries for C++ Stacks on one page, or view entries individually:

Stack constructors	construct a new stack
empty	true if the stack has no elements
pop	removes the top element of a stack
push	adds an element to the top of the stack
size	returns the number of items in the stack
top	returns the top element of the stack

empty

Syntax:

```
#include <stack>
bool empty() const;
```

The empty() function returns true if the stack has no elements, false otherwise.

For example, the following code uses empty() as the stopping condition on a (C/C++ Keywords) while loop to clear a stack and display its contents in reverse order:

```
vector<int> v;
for( int i = 0; i < 5; i++ ) {
  v.push_back(i);
}
while( !v.empty() ) {
  cout << v.back() << endl;
  v.pop_back();
}
```

Related topics:

size

Stack constructors

Syntax:

```
#include <stack>
stack();
stack( const Container& con );
```

Stacks have an empty constructor and a constructor that can be used to specify a container type.

empty

Syntax:

```
#include <stack>
bool empty() const;
```

The empty() function returns true if the stack has no elements, false otherwise.

For example, the following code uses empty() as the stopping condition on a (C/C++ Keywords) while loop to clear a stack and display its contents in reverse order:

```
vector<int> v;
for( int i = 0; i < 5; i++ ) {
  v.push_back(i);
}
while( !v.empty() ) {
  cout << v.back() << endl;
  v.pop_back();
}
```

Related topics:

size

pop

Syntax:

```
#include <stack>
void pop();
```

The function pop() removes the top element of the stack and discards it.

Related topics:
(C++ Priority Queues) push
top

push

Syntax:

```
#include <stack>
void push( const TYPE& val );
```

The function push() adds *val* to the top of the current stack.

For example, the following code uses the push() function to add ten integers to the top of a stack:

```
stack<int> s;
for( int i=0; i < 10; i++ )
  s.push(i);
```

Related topics:

pop

size

Syntax:

```
#include <stack>
size_type size() const;
```

The size() function returns the number of elements in the current stack.

Related topics:

(C++ Strings) capacity
empty
(C++ Strings) length
(C++ Multimaps) max_size
(C++ Strings) resize

top

Syntax:

```
#include <stack>
TYPE& top();
```

The function top() returns a reference to the top element of the stack.

For example, the following code removes all of the elements from a stack and uses top() to display them:

```
while( !s.empty() ) {
  cout << s.top() << " ";
  s.pop();
}
```

Related topics:

pop

C++ Vectors

Vectors contain contiguous elements stored as an array. Accessing members of a vector or appending elements can be done in constant time, whereas locating a specific value or inserting elements into the vector takes linear time.

Display all entries for C++ Vectors on one page, or view entries individually:

Vector constructors	create vectors and initialize them with some data
Vector operators	compare, assign, and access elements of a vector
assign	assign elements to a vector
at	returns an element at a specific location
back	returns a reference to last element of a vector
begin	returns an iterator to the beginning of the vector
capacity	returns the number of elements that the vector can hold
clear	removes all elements from the vector
empty	true if the vector has no elements
end	returns an iterator just past the last element of a vector
erase	removes elements from a vector
front	returns a reference to the first element of a vector
insert	inserts elements into the vector
max_size	returns the maximum number of elements that the vector can hold
pop_back	removes the last element of a vector
push_back	add an element to the end of the vector
rbegin	returns a reverse_iterator to the end of the vector
rend	returns a reverse_iterator to the beginning of the vector
reserve	sets the minimum capacity of the vector
resize	change the size of the vector
size	returns the number of items in the vector
swap	swap the contents of this vector with another

assign

Syntax:

```
#include <vector>
void assign( size_type num, const TYPE& val );
void assign( input_iterator start, input_iterator end );
```

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

This function will destroy the previous contents of the vector.

For example, the following code uses assign() to put 10 copies of the integer 42 into a vector:

```
vector<int> v;
v.assign( 10, 42 );
for( int i = 0; i < v.size(); i++ ) {
  cout << v[i] << " ";
}
cout << endl;
```

The above code displays the following output:

```
42 42 42 42 42 42 42 42 42 42
```

The next example shows how assign() can be used to copy one vector to another:

```
vector<int> v1;
for( int i = 0; i < 10; i++ ) {
  v1.push_back( i );
}

vector<int> v2;
v2.assign( v1.begin(), v1.end() );

for( int i = 0; i < v2.size(); i++ ) {
  cout << v2[i] << " ";
}
cout << endl;
```

When run, the above code displays the following output:

```
0 1 2 3 4 5 6 7 8 9
```

Related topics:
(C++ Strings) assign
insert
push_back
(C++ Lists) push_front

Vector constructors

Syntax:

```
#include <vector>
vector();
vector( const vector& c );
vector( size_type num, const TYPE& val = TYPE() );
vector( input_iterator start, input_iterator end );
~vector();
```

The default vector constructor takes no arguments, creates a new instance of that vector.

The second constructor is a default copy constructor that can be used to create a new vector that is a copy of the given vector *c*.

The third constructor creates a vector with space for *num* objects. If *val* is specified, each of those objects will be given that value. For example, the following code creates a vector consisting of five copies of the integer 42:

```
vector<int> v1( 5, 42 );
```

The last constructor creates a vector that is initialized to contain the elements between *start* and *end* . For example:

```
// create a vector of random integers
cout << "original vector: ";
vector<int> v;
for( int i = 0; i < 10; i++ ) {
  int num = (int) rand() % 10;
  cout << num << " ";
  v.push_back( num );
}
cout << endl;

// find the first element of v that is even
vector<int>::iterator iter1 = v.begin();
while( iter1 != v.end() && *iter1 % 2 != 0 ) {
  iter1++;
}

// find the last element of v that is even
vector<int>::iterator iter2 = v.end();
do {
  iter2--;
} while( iter2 != v.begin() && *iter2 % 2 != 0 );

// only proceed if we find both numbers
if( iter1 != v.end() && iter2 != v.begin() ) {
  cout << "first even number: " << *iter1 << ", last even number: " << *iter2
<< endl;

  cout << "new vector: ";
  vector<int> v2( iter1, iter2 );
```

```
  for( int i = 0; i < v2.size(); i++ ) {
    cout << v2[i] << " ";
  }
  cout << endl;
}
```

When run, this code displays the following output:

```
original vector: 1 9 7 9 2 7 2 1 9 8
first even number: 2, last even number: 8
new vector: 2 7 2 1 9
```

All of these constructors run in linear time except the first, which runs in constant time.

The default destructor is called when the vector should be destroyed.

Vector operators

Syntax:

```
#include <vector>
TYPE& operator[]( size_type index );
const TYPE& operator[]( size_type index ) const;
vector operator=(const vector& c2);
bool operator==(const vector& c1, const vector& c2);
bool operator!=(const vector& c1, const vector& c2);
bool operator<(const vector& c1, const vector& c2);
bool operator>(const vector& c1, const vector& c2);
bool operator<=(const vector& c1, const vector& c2);
bool operator>=(const vector& c1, const vector& c2);
```

All of the C++ containers can be compared and assigned with the standard comparison operators: ==, !=, <=, >=, <, >, and =. Individual elements of a vector can be examined with the [] operator.

Performing a comparison or assigning one vector to another takes linear time. The [] operator runs in constant time.

Two vectors are equal if:

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

Comparisons among vectors are done lexicographically.

For example, the following code uses the [] operator to access all of the elements of a vector:

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

Related topics:
at

assign

Syntax:

```
#include <vector>
void assign( size_type num, const TYPE& val );
void assign( input_iterator start, input_iterator end );
```

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

This function will destroy the previous contents of the vector.

For example, the following code uses assign() to put 10 copies of the integer 42 into a vector:

```
vector<int> v;
v.assign( 10, 42 );
for( int i = 0; i < v.size(); i++ ) {
  cout << v[i] << " ";
}
cout << endl;
```

The above code displays the following output:

```
42 42 42 42 42 42 42 42 42 42
```

The next example shows how assign() can be used to copy one vector to another:

```
vector<int> v1;
for( int i = 0; i < 10; i++ ) {
  v1.push_back( i );
}

vector<int> v2;
v2.assign( v1.begin(), v1.end() );

for( int i = 0; i < v2.size(); i++ ) {
  cout << v2[i] << " ";
}
cout << endl;
```

When run, the above code displays the following output:

```
0 1 2 3 4 5 6 7 8 9
```

Related topics:
(C++ Strings) assign
insert
push_back
(C++ Lists) push_front

at

Syntax:

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

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

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 overrunns 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:

Vector operators

back

Syntax:

```
#include <vector>
TYPE& back();
const TYPE& back() const;
```

The back() function returns a reference to the last element in the vector.

For example:

```
vector<int> v;
for( int i = 0; i < 5; i++ ) {
  v.push_back(i);
}
cout << "The first element is " << v.front()
     << " and the last element is " << v.back() << endl;
```

This code produces the following output:

```
The first element is 0 and the last element is 4
```

The back() function runs in constant time.

Related topics:

front

pop_back

begin

Syntax:

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

The function begin() returns an iterator to the first element of the vector, and runs in constant time.

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

```
vector<string> words;
string str;

while( cin >> str ) words.push_back(str);

vector<string>::iterator iter;
for( iter = words.begin(); iter != words.end(); iter++ ) {
  cout << *iter << endl;
}
```

When given this input:

```
hey mickey you're so fine
```

...the above code produces the following output:

```
hey
mickey
you're
so
fine
```

Related topics:

[] operator
at
end
rbegin
rend

capacity

Syntax:

```
#include <vector>
size_type capacity() const;
```

The capacity() function returns the number of elements that the vector 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 <vector>
void clear();
```

The function clear() deletes all of the elements in the vector.

clear() runs in linear time.

Related topics:
erase

empty

Syntax:

```
#include <vector>
bool empty() const;
```

The empty() function returns true if the vector has no elements, false otherwise.

For example, the following code uses empty() as the stopping condition on a while loop to clear a vector and display its contents in reverse order:

```
vector<int> v;
for( int i = 0; i < 5; i++ ) {
  v.push_back(i);
}
while( !v.empty() ) {
  cout << v.back() << endl;
  v.pop_back();
}
```

Related topics:
size

end

Syntax:

```
#include <vector>
iterator end();
const_iterator end() const;
```

The end() function returns an iterator just past the end of the vector.

Note that before you can access the last element of the vector using an iterator that you get from a call to end(), you'll have to decrement the iterator first. This is because end() doesn't point to the end of the vector; it points **just past the end of the vector**.

For example, in the following code, the first "cout" statement will display garbage, whereas the second statement will actually display the last element of the vector:

```
vector<int> v1;
v1.push_back( 0 );
v1.push_back( 1 );
v1.push_back( 2 );
v1.push_back( 3 );

int bad_val = *(v1.end());
cout << "bad_val is " << bad_val << endl;

int good_val = *(v1.end() - 1);
cout << "good_val is " << good_val << endl;
```

The next example shows how begin() and end() can be used 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 <vector>
iterator erase( iterator loc );
iterator erase( iterator start, iterator end );
```

The erase() function either deletes the element at location *loc*, or deletes the elements between *start* and *end* (including *start* but not including *end*). The return value is the element after the last element erased.

The first version of erase (the version that deletes a single element at location *loc*) runs in constant time for lists and linear time for vectors, dequeues, and strings. The multiple-element version of erase always takes linear time.

For example:

```
// Create a vector, load it with the first ten characters of the alphabet
vector<char> alphaVector;
for( int i=0; i < 10; i++ ) {
  alphaVector.push_back( i + 65 );
}
int size = alphaVector.size();
vector<char>::iterator startIterator;
vector<char>::iterator tempIterator;
for( int i=0; i < size; i++ ) {
  startIterator = alphaVector.begin();
  alphaVector.erase( startIterator );
  // Display the vector
  for( tempIterator = alphaVector.begin(); tempIterator !=
alphaVector.end(); tempIterator++ ) {
    cout << *tempIterator;
  }
  cout << endl;
}
```

That code would display the following output:

```
BCDEFGHIJ
CDEFGHIJ
DEFGHIJ
EFGHIJ
FGHIJ
GHIJ
HIJ
IJ
J
```

In the next example, erase() is called with two iterators to delete a range of elements from a vector:

```
// create a vector, load it with the first ten characters of the alphabet
vector<char> alphaVector;
for( int i=0; i < 10; i++ ) {
  alphaVector.push_back( i + 65 );
}
```

```
// display the complete vector
for( int i = 0; i < alphaVector.size(); i++ ) {
  cout << alphaVector[i];
}
cout << endl;

// use erase to remove all but the first two and last three elements
// of the vector
alphaVector.erase( alphaVector.begin()+2, alphaVector.end()-3 );
// display the modified vector
for( int i = 0; i < alphaVector.size(); i++ ) {
  cout << alphaVector[i];
}
cout << endl;
```

When run, the above code displays:

```

ABCDEFGHIJ
ABHIJ
```

Related topics:
clear
insert
pop_back
(C++ Lists) pop_front
(C++ Lists) remove
(C++ Lists) remove_if

front

Syntax:

```
#include <vector>
TYPE& front();
const TYPE& front() const;
```

The front() function returns a reference to the first element of the vector, and runs in constant time.

For example, the following code uses a vector and the sort() algorithm to display the first word (in alphabetical order) entered by a user:

```
vector words;
string str;

while( cin >> str ) words.push_back(str);

sort( words.begin(), words.end() );

cout << "In alphabetical order, the first word is '" << words.front() <<
"'." << endl;
```

When provided with this input:

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

...the above code displays:

```
In alphabetical order, the first word is 'aid'.
```

Related topics:

back

(C++ Lists) pop_front

(C++ Lists) push_front

insert

Syntax:

```
#include <vector>
iterator insert( iterator loc, const TYPE& val );
void insert( iterator loc, size_type num, const TYPE& val );
template<TYPE> void insert( iterator loc, input_iterator start,
input_iterator end );
```

The insert() function either:

- inserts *val* before *loc*, returning an iterator to the element inserted,
- inserts *num* copies of *val* before *loc*, or
- inserts the elements from *start* to *end* before *loc*.

Note that inserting elements into a vector can be relatively time-intensive, since the underlying data structure for a vector is an array. In order to insert data into an array, you might need to displace a lot of the elements of that array, and this can take linear time. If you are planning on doing a lot of insertions into your vector and you care about speed, you might be better off using a container that has a linked list as its underlying data structure (such as a List or a Deque).

For example, the following code uses the insert() function to splice four copies of the character 'C' into a vector of characters:

```
// Create a vector, load it with the first 10 characters of the alphabet
vector<char> alphaVector;
for( int i=0; i < 10; i++ ) {
  alphaVector.push_back( i + 65 );
}

// Insert four C's into the vector
vector<char>::iterator theIterator = alphaVector.begin();
alphaVector.insert( theIterator, 4, 'C' );

// Display the vector
for( theIterator = alphaVector.begin(); theIterator != alphaVector.end();
theIterator++ )    {
  cout << *theIterator;
}
```

This code would display:

```
CCCCABCDEFGHIJ
```

Here is another example of the insert() function. In this code, insert() is used to append the contents of one vector onto the end of another:

```
vector<int> v1;
v1.push_back( 0 );
v1.push_back( 1 );
v1.push_back( 2 );
v1.push_back( 3 );
```

```
vector<int> v2;
v2.push_back( 5 );
v2.push_back( 6 );
v2.push_back( 7 );
v2.push_back( 8 );

cout << "Before, v2 is: ";
for( int i = 0; i < v2.size(); i++ ) {
  cout << v2[i] << " ";
}
cout << endl;

v2.insert( v2.end(), v1.begin(), v1.end() );

cout << "After, v2 is: ";
for( int i = 0; i < v2.size(); i++ ) {
  cout << v2[i] << " ";
}
cout << endl;
```

When run, this code displays:

```
Before, v2 is: 5 6 7 8
After, v2 is: 5 6 7 8 0 1 2 3
```

Related topics:

assign
erase
push_back
(C++ Lists) merge
(C++ Lists) push_front
(C++ Lists) splice

max_size

Syntax:

```
#include <vector>
size_type max_size() const;
```

The max_size() function returns the maximum number of elements that the vector 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 vector and the the number of elements that the vector will be able to hold before more memory will have to be allocated, respectively.

Related topics:
size

pop_back

Syntax:

```
#include <vector>
void pop_back();
```

The pop_back() function removes the last element of the vector.

pop_back() runs in constant time.

Related topics:

back
erase
(C++ Lists) pop_front
push_back

push_back

Syntax:

```
#include <vector>
void push_back( const TYPE& val );
```

The push_back() function appends *val* to the end of the vector.

For example, the following code puts 10 integers into a vector:

```
vector<int> the_vector;
for( int i = 0; i < 10; i++ ) {
  the_vector.push_back( i );
}
```

When displayed, the resulting vector would look like this:

```
0 1 2 3 4 5 6 7 8 9
```

push_back() runs in constant time.

Related topics:
assign
insert
pop_back
(C++ Lists) push_front

rbegin

Syntax:

```
#include <vector>
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
```

The rbegin() function returns a reverse_iterator to the end of the current vector.

rbegin() runs in constant time.

Related topics:

begin
end
rend

rend

Syntax:

```
#include <vector>
reverse_iterator rend();
const_reverse_iterator rend() const;
```

The function rend() returns a reverse_iterator to the beginning of the current vector.

rend() runs in constant time.

Related topics:

begin
end
rbegin

reserve

Syntax:

```
#include <vector>
void reserve( size_type size );
```

The reserve() function sets the capacity of the vector to at least *size*.

reserve() runs in linear time.

Related topics:
capacity

resize

Syntax:

```
#include <vector>
void resize( size_type num, const TYPE& val = TYPE() );
```

The function resize() changes the size of the vector 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:
Vector constructors & destructors
capacity
size

size

Syntax:

```
#include <vector>
size_type size() const;
```

The size() function returns the number of elements in the current vector.

Related topics:

capacity
empty
(C++ Strings) length
max_size
resize

swap

Syntax:

```
#include <vector>
void swap( container& from );
```

The swap() function exchanges the elements of the current vector with those of *from*. This function operates in constant time.

For example, the following code uses the swap() function to exchange the contents of two vectors:

```
vector v1;
v1.push_back("I'm in v1!");

vector v2;
v2.push_back("And I'm in v2!");

v1.swap(v2);

cout << "The first element in v1 is " << v1.front() << endl;
cout << "The first element in v2 is " << v2.front() << endl;
```

The above code displays:

```
The first element in v1 is And I'm in v2!
The first element in v2 is I'm in v1!
```

Related topics:

= operator

(C++ Lists) splice

C/C++ Pre-processor Commands

Display all entries for C/C++ Pre-processor Commands on one page, or view entries individually:

#, ##	manipulate strings
#define	define variables
#error	display an error message
#if, #ifdef, #ifndef, #else, #elif, #endif	conditional operators
#include	insert the contents of another file
#line	set line and file information
#pragma	implementation specific command
#undef	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.

Example code:

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 x = -1;
while( absolute_value( x ) ) {
   ...
}
```

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 it's 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

#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.

Example code:

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 x = -1;
while( absolute_value( x ) ) {
   ...
}
```

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 it's 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 c_plusplus.
- 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

Display all entries for C/C++ Keywords on one page, or view entries individually:

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 );
```

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:
char
double
false
float
int
long
short
signed
true
unsigned
wchar_t

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
private
protected
public
struct
this
typename
union
virtual

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&lt;type&gt; (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 aditional 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:
bool
char
double
int
long
short
signed
unsigned
void
wchar_t

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:

break
continue
do
if
while

friend

The friend keyword allows classes or functions not normally associated with a given class to have access to the private data of that class.

Related topics:
class

goto

Syntax:

```

goto labelA;
...
labelA:
```

The goto statement causes the current thread of execution to jump to the specified label. While the use of the goto statement is generally considered harmful, it can occasionally be useful. For example, it may be cleaner to use a goto to break out of a deeply-nested for loop, compared to the space and time that extra break logic would consume.

Related topics:
break

if

Syntax:

```
if( conditionA ) {
  statement-listA;
}
else if( conditionB ) {
  statement-listB;
}
...
else {
  statement-listN;
}
```

The if construct is a branching mechanism that allows different code to execute under different conditions. The conditions are evaluated in order, and the statement-list of the first condition to evaluate to true is executed. If no conditions evaluate to true and an else statement is present, then the statement list within the else block will be executed. All of the else blocks are optional.

Related topics:
else
for
switch
while

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. Functions that contain static data, loops, switch statements, or recursive calls cannot be inlined. When a function declaration is included in a class declaration, the compiler should try to automatically inline that function.

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, in bytes, of the argument passed to it. 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;
```

When run, 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 preceeding "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.

Example code:

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