Language Summary

Basic Concepts

Statements if, for, while, return, break...

Expressions arithmetic, comparison, assignment...

The most important types are int, char, bool, double, and the containers string, vector, and map. Summary of common types:

Built-in	Description
int x;	Fastest integer type (16-32 bits), also short, long, unsigned
char x;	8-bit character, '\0' to '\xFF' or -128 to 127
double x;	64 bit real + or - 1.8e308, 14 significant digits, also float
bool x;	true or false

Modifiers	Description
const T x;	Non-modifiable object
T& y=x;	Reference, y is an alias for x, which both have type T
T f(...) {...}	Defines f as a function returning T
T* p;	Pointer to T (*p is a T object)
T a[N];	Array of N elements of T, a[0] to a[N-1]
static T x;	Place x in data segment
register T x;	(rare) Hint to optimize for speed
volatile T x;	(rare) x may be modified externally

The following standard library types and functions require at the beginning of the program:

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

Library Type	Description	Header
istream	Standard input (cin)	iostream
ostream	Output (cout, cerr, clog)	iostream
ifstream	Input file	fstream
ofstream	Output file	fstream
string	Sequence of char	string
vector<T>	Expandable array/stack of T	vector
deque<T>	Array/double ended queue	deque
list<T>	List/stack/queue of T	list
map<T1,T2>	Associative mapping of T1 to T2	map
set<T1>	A map with keys only	set
pair<T1,T2>	Two objects of type T1 and T2	map or utility
priority_queue<T>	Sorted queue	queue
stack<T>	Stack	stack
bitset<N>	Array of N bool with logical operations	bitset
valarray<T>	Array with arithmetic operations	valarray
complex<T>	Complex number	complex
iterator	Pointer into a container	(Included with container)
const_iterator	Pointer not allowing element assignment	(Included with container)
exception	Hierarchy of exception types	stdexcept, exception

C++ Standard Library Functions	Header
min(), max(), swap(), sort(), copy(), equal()	algorithm
accumulate(), inner_product()	numeric
back_inserter()	iterator
equal_to(), less(), bind2nd()	functional
set_new_handler()	new

C Library Functions	Header
atoi(), atof(), abs(), rand(), system(), exit()	cstdlib
isalpha(), isdigit(), tolower(), toupper()	cctype
sqrt(), log(), exp(), pow(), sin(), cos(), atan()	cmath
clock(), time()	ctime
strlen(), memset(), memmove(), memcmp()	cstring
printf(), fopen(), getc(), perror()	cstdio
assert()	cassert

C++ allows you to create your own types and libraries. The most important type is a class, allowing object oriented programming. A class is an abstract data type with a hidden representation and a set of public member functions and

types. Classes can be organized into a hierarchy (inheritance), and you can write code that accepts any type in this hierarchy (polymorphism). Functions and classes can be parameterized by type (templated).

```
class T {...};  Defines T as a collection of types, objects, and member functions
template <class T> ... Defines a set of functions or classes over all T
typedef T U;    Defines type U is a synonym for T
enum T {...};   Defines T as an int, and set of int constants
struct T {...}; Like a class, except default scope of members is public
union T {...};  A struct with object members overlapping in memory
namespace N {...};  Defines a scope for a collection of types, objects, and functions
```

Basics

C++ is a compiled language, an upward compatible superset of C and an (incompatible) predecessor to Java. C++ compiles C programs but adds object oriented (OO) features (classes, inheritance, polymorphism), templates (generic functions and classes), function and operator overloading, namespaces (packages), exception handling, a library of standard data structures (string, vector, map, etc.) and formatted text I/O (istream, ostream). Unlike Java, C++ lacks a standard graphical user interface (GUI), network interface, garbage collection, and threads, and allows non-OO programming and unchecked low-level machine operations with pointers. However, C++ executes faster than Java and requires no run-time support.

A C++ program is a collection of function, object, and type declarations. Every program must have a function `int main() { ... }` where the curly braces enclose a block, a sequence of declarations and statements ending in semicolons which are executed in order. A statement is an expression, block, or control statement that alters the order of execution, such as `if, while, for, break, return`. Some types (`std::string`), objects (`std::cout`), and functions are defined in header files, requiring the line `#include <`*header*`>` before use. Items defined in the standard headers are in the namespace`std`. The `std::` prefix may be dropped after the statement `using namespace std;`. For instance,

```
// Comment: prints "Hello world!" and an OS-independent newline
#include <string>    // Defines type std::string
#include <iostream>  // Defines global object std::cout
using namespace std; // Allow std:: to be dropped
int main() {         // Execution starts here
  string s="Hello world!\n"; // Declares object s of type string
  cout << s;         // An expression as a statement, << is the output operator
  return 0;          // Execution ends here
}
```

The symbol `//` denotes a comment to the end of the line. You may also use `/* ... */` for multiline comments. Spacing and indentation is used for readability. C++ is mostly free-form, except that the end of line is significant after `#` and `//`. C++ is case sensitive.

C++ source code files should be created with a text editor and have the extension `.cpp`. If the above is called `hello.cpp`, it may be compiled and run as follows in a UNIX shell window:

```
g++ hello.cpp -o hello -Wall -O
./hello
```

The `-o` option renames the executable file, by default `a.out`. `-Wall` turns on all warnings (recommended). `-O` optimizes (compiles slower but runs faster).

In Windows, the GNU C++ compiler is called DJGPP. To compile and run from an MS-DOS box:

```
gxx hello.cpp -o hello.exe
hello
```

The output file must have a .EXE extension (default is A.EXE). There is also a .OBJ file which you can delete.

To use the network or GUI interface in UNIX, you must use the X and socket libraries, which don't work in Windows. In Windows, you must use the Windows API and a compiler that supports them, such as from Microsoft, Borland, or Symantec. GUI/network programming is nonportable and outside the scope of this document.

Links to free and commercial C++ compilers can be found at cplusplus.com.

Statements

A program consists of a collection of functions (one of which must be `int main() {...}`) and type and object declarations. A function may contain declarations and statements. Statements have the following forms, where `s` is a statement, and `t` is a true/false expression.

```
s;                              // Expression or declaration
;                               // Empty statement
{s; s;}                         // A block of 0 or more statements is a statement
if (t) s;                       // If t is true then s
if (t) s; else s;               // else is optional
while (t) s;                    // Loop 0 or more times
for (s1; t; s2) s;              // s1; while (t) {s; s2;}
break;                          // Jump from while, for, do, switch
return x;                       // Return x to calling function
try {throw x;}                  // Throw exception, abort if not caught, x has any type
  catch (T y) {s;}                // if x has type T then y=x, jump to s
  catch (...) {s;}                // else jump here (optional)
do s; while (t);                // (uncommon) s; while (t) s;
continue;                       // (uncommon) Start next loop of while, for, do
switch (i) {                    // (uncommon) Test int expression i to const C
  case C: s; break;               // if (i==C) go here
  default: s;                     // optional, else go here
}
label: goto label;              // (rare) Jump to label within a function
```

A statement may be a declaration or an expression. Objects and types declared in a block are local to that block. (Functions cannot be defined locally). It is normal (but not required) to show statements on separate lines and to indent statements enclosed in a block. If braces are optional, we indent anyway. For instance,

```
{                       // start of block
  int a[10], i=0, j;    // declaration
  a[i+2]=3;             // expression
}                       // end of block, a, i, and j are destroyed
```

declares the array of int `a` with elements `a[0]` through `a[9]` (whose values are initially undefined), `i` with initial value 0, and `j` with an undefined initial value. These names can only be used in scope, which is from the declaration to the closing brace.

The `for` loop is normally used for iteration. For instance, the following both exit the loop with `i` set to the index of the first element of `a` such that `a[i]` is 0, or to 10 if not found.

```
  for (i=0; i<10; i=i+1) {    i=0;
    if (a[i]==0) {            while (i<10) {
      break;                    if (a[i]==0)
    }                             break;
  }                             i=i+1;
                              }
```

The braces in the `for` loop are optional because they each enclose a single statement. In the `while` loop, the outer braces are required because they enclose 2 statements. All statements are optional:`for (;;)` loops forever. The first statement in a `for` loop may declare a variable local to the loop.

```
  for (int i=0; i<10; i=i+1)
```

It is only possible to `break` from the innermost loop of a nested loop. `continue` in a `for` loop skips the rest of the block but executes the iteration (s2) and test before starting the next loop.

`return x;` causes the current function to return to the caller, evaluating to x. It is required except in functions returning void, in which case `return;` returns without a value. The value returned by`main()` has no effect on program behavior and is normally discarded. However it is available as the $status in a UNIX csh script or ERRORLEVEL in a Windows .BAT file.

```
int sum(int x, int y) {  // Function definition
  return x+y;
}
int main() {
  int a=sum(1,2);        // a=3;
  return 0;              // By convention, nonzero indicates an error
}
```

A test of several alternatives usually has the form `if (t) s; else if (t) s; else if (t) s; ... else s;`. A `switch` statement is an optimization for the special case where an int expression is tested against a small range of constant values. The following are equivalent:

```
switch (i) {                if (i==1)
  case 1: j=1; break;         j=1;
  case 2: // fall thru      else if (i==2 || i==3) // || means "or else"
  case 3: j=23; break;        j=23;
  default: j=0;             else
}                             j=0;
```

`throw x` jumps to the first `catch` statement of the most recently executed `try` block where the parameter declaration matches the type of x, or a type that x can be converted to, or is `...`. At most one catch block is executed. If no matching `catch` block is found, the program aborts (Unexpected exception). `throw;` with no expression in a `catch` block throws the exception just caught. Exceptions are generally used when it is inconvenient to detect and handle an error in the same place.

```
void f() {
  throw 3;
}

int main() {
  try {
    f();
  }
  catch(int i) {  // Execute this block with i = 3
    throw;        // throw 3 (not caught, so program aborts)
  }
  catch(...) {    // Catch any other type
  }
}
```

Expressions

There are 18 levels of operator precedence, listed highest to lowest. Operators at the same level are evaluated left to right unless indicted, Thus, a=b+c means a=(b+c) because + is higher than =, and a-b-c means (a-b)-c. Order of evaluation is undefined, e.g. for `sin(x)+cos(x)` we cannot say whether sin() or cos() is called first.

The meaning of an expression depends on the types of the operands. `(x,y)` denotes a comma separated list of 0 or more objects, e.g. `()`, `(x)`, or `(1,2,3,4)`.

```
1
X::m           Member m of namespace or class X
::m            Global name m when otherwise hidden by a local declaration

2
p[i]           i'th element of container p (array, vector, string)
x.m            Member m of object x
p->m           Member m of object pointed to by p
f(x,y)         Call function f with 0 or more arguments
i++            Add 1 to i, result is original value of i
i--            Subtract 1 from i, result is original value of i
```

static cast<T>(x) Convert x to type T using defined conversions
const cast<T>(x) (rare) Convert x to equivalent but non-const T
reinterpret cast<T>(x) (rare, dangerous) Pretend x has type T
dynamic cast<T>(x) (rare) Convert base pointer or reference to derived if possible
typeid(x) (rare) If x is type T, then typeid(x)==typeid(T) (in <typeinfo>)

3 (right to left)
p Contents of pointer p, or p[0]. If p is type T, *p is T
&x Address of (pointer to) x. If x is type T, &x is T*
-a Negative of numeric a
!i Not i, true if i is false or 0
~i Bitwise compliment of i, -1 - i
(T)x Convert (cast) object x to type T (by static, const, or reinterpret)
T(x,y) Convert, initializing with 0 or more arguments
new T Create a T object on heap, return its address as T*
new T(x,y) Create, initializing with 0 or more arguments
new(p) T (rare) Initialize T at address p without allocating from heap
new(p) T(x,y) (rare) Initialize T with 0 or more arguments at p
new T[i] Create array of i objects of type T, return T* pointing to first element
delete p Destroy object pointed to by p obtained with new T or new T()
delete[] p Destroy array obtained with new T[]
++i Add 1 to i, result is the new i
--i Subtract 1 from i, result is the new i
sizeof x Size of object x in bytes
sizeof(T) Size of objects of type T in bytes

4
x.*p (rare) Object in x pointed to by pointer to member p
q->*p (rare) Object in *q pointed to by pointer to member p

5
a*b Multiply numeric a and b
a/b Divide numeric a and b, round toward 0 if both are integer
i%j Integer remainder i-(i/j)*j

6
a+b Addition, string concatenation
a-b Subtraction

7
x<<y Integer x shifted y bits to left, or output y to ostream x
x>>y Integer x shifted y bits to right, or input y from istream x

8
x<y Less than
x>y Greater than
x<=y Less than or equal to
x>=y Greater than or equal to

9
x==y Equals
x!=y Not equals

10
i&j Bitwise AND of integers i and j

11
i^j Bitwise XOR of integers i and j

12
i|j Bitwise OR of integers i and j

13
i&&j i and then j (evaluate j only if i is true/nonzero)

14
i||j i or else j (evaluate j only if i is false/zero)

15 (right to left)
x=y Assign y to x, result is new value of x
x+=y x=x+y, also -= *= /= %= &= |= ^= <<= >>=

16
i?x:y If i is true/nonzero then x else y

17
throw x Throw exception x (any type)

18
x,y Evaluate x and y (any types), result is y

Expressions that don't require creating a new object, such as `a=b, ++a, p[i], p->m, x.m, a?b:c, a,b` etc. are *lvalues*, meaning they may appear on the left side of an assignment. Other expressions and conversions create temporary objects to hold the result, which are `const` (constant). An expression used as a statement discards the final result.

```
int a, b, c;
a+b;         // Legal, add a and b, discard the sum
a=b=c;       // Legal, assign c to b, then assign the new b to a
(a+=b)+=c;   // Legal, add b to a, then add c to a
a+b=c;       // Error, a+b is const
double(a)=b; // Error, double(a) is const
```

`static_cast<T>(x)` converts x to type T if a conversion is defined. Usually the value of x is preserved if possible. Conversions are defined between all numeric types (including char and bool), from 0 to pointer, pointer to bool or void*, istream to bool, ostream to bool, char* to string, from a derived class to base class (including pointers or references), and from type T to type U if class U has a constructor taking T or class T has a member `operator U()`. A conversion will be implicit (automatically applied) whenever an otherwise invalid expression, assignment, or function argument can be made legal by applying one, except for T to U where U's constructor taking T is declared `explicit`, for example, the constructor for `vector` taking `int`.

```
double d; d=static_cast<double>(3);  // Explicit 3 to 3.0
d=3;                                  // Implicit conversion
d=sqrt(3);                            // Implicit 3.0, sqrt() expects double
vector<int> v(5);                     // This constructor is explicit
v=5;                                  // Error, no implicit conversion
v=static_cast<vector<int> >(5);       // OK
```

`const_cast<T>(x)` allows an object to be modified through a const pointer or reference. It must always be explicit.

```
int x=3;
const int& r=x; r=4;     // Error, r is const
const_cast<int&>(r)=4;   // OK, x=4
const int* p=&x; *p=5;   // Error, *p is const
*const_cast<int*>(p)=5;  // OK, x=5
```

If x were const, then this code would still be allowed but it is undefined whether x actually changes.

`reinterpret_cast<T>(x)` turns off normal type checking between int and different pointer types, which are normally incompatible. The only safe conversion is to convert a pointer back to its original type. Conversion is always explicit.

```
int x=3, *p=&x; *p=5;              // OK, x=5
*reinterpret_cast<double*>(p)=5;  // Crash, writing 8 bytes into 4
```

The expression `(T)x` applies whatever combination of static, const, and reinterpret casts are needed to convert x to type T. `T(x)` is a static_cast.

```
const char* s="hello";
int(*s);                  // static_cast
(char*)s;                 // const_cast
(const int*)s;            // reinterpret_cast
(int*)s;                  // reinterpret_cast and const_cast
```

Declarations

A declaration creates a type, object, or function and gives it a name. The syntax is a type name followed by a list of objects with possible modifiers and initializers applying to each object. A name consists of upper or lowercase letters, digits, and underscores (_) with a leading letter. (Leading underscores are allowed but may be reserved). An initializer appends the form `=x` where x is an expression, or (x,y) for a list of *one* or more expressions. For instance,

```
string s1, s2="xxx", s3("xxx"), s4(3,'x'), *p, a[5], next_Word();
```

declares s1 to be a string with initial value "", s2, s3, and s4 to be strings with initial value "xxx", p to be a pointer to string, a to be an array of 5 strings (a[0] to a[4] with initial values ""), and`next_Word` to be a function that takes no

parameters and returns a string.

Built-in Types

All built-in types are numeric. They are not automatically initialized to 0 unless global or static.

```
int a, b=0;       // a's value is undefined
static double x;  // 0.0
```

Types and their usual ranges are listed below. Actual ranges could be different. The most important types are int, bool, char, and double.

Integer types	**Bits**	**Range**
bool	1	false (0) or true (1)
signed char	8	'\x80' to '\x7f' (-128 to 127)
unsigned char	8	'\x00' to '\XFF' (0 to 255)
char	8	Usually signed
short	16	-32768 to 32767
unsigned short	16	0u to 65535U
int	32	Usually -2147483648 to 2147483647
unsigned int	32	Usually 0 to 4294967295U
long	32-64	At least -2147483648l to 2147483647L
unsigned long	32-64	0ul to at least 4294967295LU

Floating point types	**Bits**	**Range**
float	32	-1.7e38f to 1.7E38F, 6 significant digits
double	64	-1.8e308 to 1.8E308, 14 significant digits
long double	64-80	At least double

There are implicit conversions between all types. When types are mixed in an expression, both operands are converted to the type that has the higher upper bound, but at least to int. This conversion only loses representation when mixing signed and unsigned types.

```
7/4                   // 1, int division rounds toward 0
7.0/4                 // 1.75, implicit double(4) = 4.0
'\x05'+true           // 6, implicit int('\x05') = 5, int(true) = 1
3U > -1               // false, implicit (unsigned int)(-1) = 2^32-1
```

Conversion from a floating point type to an integer type drops the decimal part and rounds toward 0. If the value is outside the range of the target, then the result is undefined.

```
int(-3.8)             // -3
```

Conversion of one integer type to another is performed modulo the range of the target. For a B-bit number (except bool), we add or subtract 2^B to bring the value within range. (In terms of a 2's complement number, we drop the most significant bits and reinterpret the sign bit without changing any bits). For bool, any nonzero value is true.

```
(unsigned char)(-1)     // '\xff' (255)
bool(3)                 // true
short a=x12345678;      // x5678 hex
```

Integer Types

int is the most common integer type, normally the underlying word size of the computer or 32 bits, representing numbers from -2^{31} to $2^{31}-1$ (-2147483648 to 2147483647). On some older systems such as real mode DOS, it may be 16 bits (-32768 to 32767). You should use int unless you need the range of some other type.

An int value may be written in decimal (e.g. 255), hexadecimal with a leading X (e.g. xff or XFF) or octal (base 8) with a leading 0 (e.g. 0377). A trailing L denotes long (e.g. 255L or 255l), and U denotes unsigned. These may be combined (e.g. 255lu or 255UL is unsigned long). Most integer operations translate to a single machine instruction and are very fast.

```
+ - * / % -i      Add, subtract, multiply, divide, mod, unary negation
=                 Assignment
```

```
== != < <= > >=   Comparison, returns true or false
++i i++ --i i--   Pre/post increment and decrement
& | ^ ~i << >>    Bitwise and, or, xor, not, left shift, right shift
+= -= *= /= %= &= |= ^= <<= >>=    Operate and assign, e.g. x+=y means x=x+y
&& || !i          Logical and then, or else, not
```

Division rounds toward 0, e.g. 7/4 is 1, -7/4 is -1. x%y is the remainder with the sign of x, e.g. -7%4 is -3. Division or mod by 0 is a run time error and should be avoided. Operations that yield results outside the range of an int are converted modulo 2^{32}, or more generally, 2^{B} for a B bit number. For instance, 65535*65537 is -1, not 2^{32}-1.

Assignment returns the value assigned, e.g. x=y=0 assigns 0 to y and the new y to x. The result is an lvalue, e.g. (x=y)=0 is also legal (but useless). It assigns y to x, then 0 to x.

++i and i++ both add 1 to i. However, ++i returns the new value, and i++ returns the old value. Likewise for decrement, --i and i--, which subtracts 1. The *pre* forms, ++i, --i, are lvalues.

Bitwise operators treat an int as a 2's compliment B-bit binary number (B=32) with weights -2^{B-1}, 2^{B-2}, 2^{B-3}, ... 8, 4, 2, 1. The leftmost bit is negative, and serves as the sign bit. Thus, 0 is all zero bits and -1 is all 1 bits. Bitwise operators x&y x|y x^y and ~x perform B simultaneous logical operations on the bits of x and y. For instance, if y is a power of 2, then x&(y-1) has the effect x%y, but is usually faster, and the result is always positive in the range 0 to y-1.

x<<y returns x shifted left by y places, shifting in zeros. The result is $x*2^{y}$. x>>y returns x shifted right by y places, shifting in copies of the sign bit (or zeros if unsigned). The result is $x/2^{y}$ but rounding negative instead of toward 0. For instance, -100>>3 is -13. y must be in the range 0 to B-1 (0 to 31). Shifting is usually faster than * and /.

Any binary arithmetic or bitwise operator may be combined with assignment. The result is an lvalue.
e.g. `(x+=2)*=3;` has the effect x=x+2; x=x*3;

Logical operators treat 0 as false and any other value as true. They return true (1) or false (0), as do comparisons. The && and || operators do not evaluate the right operand if the result is known from the left operand.

```
if (i>=0 && i<n && a[i]==x)  // Do bounds check on i before indexing array a
if (x=3)                     // Legal but probably wrong, assign 3 to x and test true
```

`char`

A `char` is a one byte value. Unlike other numeric types, it prints as a character, although it can be used in arithmetic expressions. Character constants are enclosed in single quotes, as `'a'`. A backslash has special meaning. '\n' is a newline, '\\' is a single backslash, '\'' is a single quote, '\"' is a double quote. A backslash may be followed by 3 octal digits ('\377') or an X and 2 hex digits ('\xFF') (but not decimal). Most computers use ASCII conversion as follows:

```
8-13:   \b\t\n\v\f\r  (bell, tab, newline, vertical tab, formfeed, return)
32-47:   !\"#$%&\'()*+,-./                (32=space, \' and \" are one char)
48-63:  0123456789:;<=>\?                  (\? is one char)
64-95:  @ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_  (\\ is one char)
96-126: `abcdefghijklmnopqrstuvwxyz{|}~
```

Floating Point Types

A number with a decimal point is double (e.g. 3.7) unless a trailing F is appended (e.g. 3.7f or 3.7F), in which case it is float. Double is preferred. A double may be written in the form xey meaning $x*10^{y}$, e.g. 3.7E-2 (0.037) or 1e4 (10000.0).

A double is usually represented as a 64 bit number with a sign bit, an 11 bit exponent, and 52 bit mantissa. Therefore it can only represent numbers of the form $M*2^{E}$ exactly, where $-2^{52} < M < 2^{52}$and $2^{-10} < E < 2^{10}$. This is about + or - 1.797e308 with about 15 decimal digits of precision. Therefore, numbers like 1e14 and 0.5 have exact representations, but 1e20 and 0.1 do not.

```
0.1 * 10 == 1    // false, they differ by about 10^-15
```

There are no bitwise or logical operators, %, ++, or --

```
+ - * / -x        Add, subtract, multiply, divide, unary negation (no %)
= += -= *= /=     Assignment, may be combined with operators
== != < <= > >=   Comparison, however only < and > are meaningful
```

Operations may produce values outside the range of a double resulting in infinity, -infinity or NaN (not a number). These values cannot be written in C++.

Additional mathematical functions (sqrt(), log(), pow(), etc.) can be found in <cmath>.

Modifiers

In a declaration, modifiers before the type name apply to all objects in the list. Otherwise they apply to single objects.

```
int* p, q;           // p is a pointer, q is an int
const int a=0, b=0;  // a and b are both const
```

const

`const` objects cannot be modified once created. They must be initialized in the declaration. By convention, const objects are UPPERCASE when used globally or as parameters.

```
const double PI=3.14159265359;  // Assignment to PI not allowed
```

References

A reference creates an alias for an object that already exists. It must be initialized. A reference to a const object must also be const.

```
int i=3;
int& r=i;          // r is an alias for i
r=4;               // i=4;
double& pi=PI;     // Error, would allow PI to be modified
const double& pi=PI;  // OK
```

Functions

A function has a list of parameter declarations, a return type, and a block of statements. Execution must end with a return statement returning an expression that can be converted to the return type, unless void, in which case there is an implied `return;` at the end. Arguments passed to a function must match the parameters or allow implicit conversion (such as int to double). Functions must be defined before use, or have a matching declaration that replaces the block with a semicolon and may optionally omit parameter names. Functions are always global (not defined in other functions).

```
void f(double x, double); // Declaration
double g() {              // Definition
  return 3;               // Implied conversion to double (3.0)
}
int main() {              // Execution starts with function main
  f(g(), 5);              // Calls g, then f with implicit 5.0
  return 0;               // Return UNIX $status or Windows ERRORLEVEL
}
void f(double x, double y) { // Definition must match declaration
  cout << x+y;
  return;                 // Optional
}
```

Command line arguments may be passed to `main(int argc, char** argv)` where `argv` is an array of `argc` elements

of type char* ('\0' terminated array of char), one element for each word (separated by white spaces). In UNIX, the command line is expanded before being passed (* becomes a directory listing, etc). The following program prints the command line.

```
// echo.cpp
#include <iostream>
using namespace std;
int main(int argc, char** argv) {
  for (int i=0; i<argc; ++i)
    cout << argv[i] << endl;
  return 0;
}
```

```
g++ echo.cpp
./a.out hello world
./a.out
hello
world
```

Function parameters have local scope. They are initialized by copying the argument, which may be an expression. Reference parameters are not copied; they become references to the arguments passed, which must be objects that the function may modify. If the reference is const, then the argument may be an expression. Const reference is the most common for passing large objects because it avoids the run time overhead of copying.

```
void assign_if(bool cond, string& to,  const string& from) {
            // value      reference    const reference
  if (cond)
    to=from;
}
int main() {
  string s;
  assign_if(true, s, "a");   // OK, s="a"
  assign_if(false, "b", s);  // Error: to refers to a const
```

Functions returning a reference must return an object which can be assigned to, and that object must exist after returning (global or static, but not local). The function may be called on the left side of an assignment. Functions returning by value make a temporary copy which is const.

```
int  a=1;                             // Global
int  f() {return a;}                  // OK, returns copy of a
int& g() {return a;}                  // OK, g() is alias for a
int& h() {return a+1;}                // Error, reference to const
int& i() {int b; return b;}           // Error, b destroyed after return
int& j() {static int b; return b;}  // OK, static has global lifespan
int main() {
  f()=2;    // Error, assignment to const
  g()=f();  // OK, a=1;
  return 0;
}
```

Functions with the same name may be overloaded by matching the arguments to the parameters.

```
int abs(int);
double abs(double);
int main() {
  abs(3);    // int
  abs(3.0);  // double
  abs("3");  // Error, no match
  abs('a');  // Error, ambiguous, could convert char to int or double
  return 0;
}
```

Most operators X can be overloaded by defining a function named `operator X()` taking the operands as arguments. At least one argument has to be a class type.

```
string operator - (const string& s, int i);  // Defines s-i
string operator - (const string& s);         // Defines -s
```

Operators `. :: ?:` and `sizeof` cannot be overloaded. Operators `= [] ->` cannot be overloaded except as class members. Postfix `++ --` are overloaded as binary operators with a second dummy `int` parameter to distinguish from

the prefix form.

```
string& operator++(const string& s);      // defines ++s
string  operator++(const string& s, int); // defines s++
```

Functions may have default arguments by initializing the parameters. Defaults should be specified only once. Defaulted parameters must appear after all non-default parameters.

```
void f(int i, int j=0, int k=0);  // OK
void g(int i=0, int j);           // Error
int main() {
  f(1, 2);  // f(1, 2, 0);
  f(1);     // f(1, 0, 0);
  f();      // Error
  return 0;
}
void f(int i, int j, int k) {}    // Defaults not specified again
```

A **template** overloads a function for all types. The declaration `template <class T, class U>` before a function definition allows T and U to be used in the code as types. The compiler will figure out appropriate substitutions from the arguments. A non-templated overloaded function takes precedence over a template.

```
template <class T>
void swap(T& a, T& b) {
  T tmp=a;
  a=b;
  b=tmp;
}
void swap(string& a, string& b);  // Overrides the case T=string
int main() {
  int i=1, j=2;
  string a, b;
  swap(i, j);        // OK, T is int
  swap(a, b);        // OK, calls non-templated swap
  swap(i, a);        // Error, cannot resolve T
  swap(cout, cerr);  // Error, ostream does not allow =
```

`inline` is a hint to the compiler to optimize for speed by expanding the code where it is called, saving a call and return instruction. Unlike a macro, semantics are preserved. Only short functions should be inlined.

```
inline int min1(int a, int b) {return a<b?a:b;}
#define min2(a,b) ((a)<(b)?(a):(b))
int main() {
  min1(f(), 0);  // calls f() once
  min2(f(), 0);  // calls f() twice, expands to ((f())<(0)?(f()):(0))
```

Pointers

A pointer stores the address of another object, and unlike a reference, may be moved to point elsewhere. The expression `&x` means "address of x" and has type "pointer to x". If `x` has type `T`, then `&x`has type `T*`. If `p` has type `T*`, then `*p` is the object to which it points, which has type `T`. The * and & operators are inverses, e.g. `*&x == x`.

Two pointers are equal if they point to the same object. All pointer types are distinct, and can only be assigned pointers of the same type or 0 (NULL). There are no run time checks against reading or writing the contents of a pointer to invalid memory. This usually causes a segmentation fault or general protection fault.

```
int i=3, *p=&i;    // p points to i, *p == 3
*p=5;              // i=5
p=new int(6);      // OK, p points to an int with value 6
p=new char('a');   // Error, even though char converts to int
p=6;               // Error, no conversion from int to pointer
p=0;               // OK
p=i-5;             // Error, compiler can't know this is 0
*p=7;              // Segmentation fault: writing to address 0
int *q; *q;        // Segmentation fault: q is not initialized, reading random memory
```

A pointer to a const object of type T must also be const, of type `const T*`, meaning that the pointer may be assigned to but its contents may not.

```
const double PI=3.1415926535898;
double* p=&PI;              // Error, would allow *p=4 to change PI
const double* p=&PI;        // OK, can't assign to *p (but may assign to p)
double* const p=&PI;        // Error, may assign to *p (but not to p)
const double* const p=&PI;  // OK, both *p and p are const
```

A function name used without parenthesis is a pointer to a function. Function pointers can be assigned values and called.

```
int f(double);     // functions f and g take double and return int
int g(double);
int *h(double);    // function h takes double and returns pointer to int
int (*p)(double);  // p is a pointer to a function that takes double and returns int
int main() {
  p=f; p(3.0);     // calls f(3.0)
  p=g; p(3.0);     // calls g(3.0)
  p=h;             // Error, type mismatch
```

Explicit pointer conversions are allowed but usually unsafe.

```
int i, *p=&i;
i=int(3.0);        // OK, rounds 3.0
*(double*)p = 3.0; // Crash, writes beyond end of i
*(double*)&PI = 4; // Overwrites a const
```

These may also be written (with the same results):

```
i=static_cast<int>(3.0);            // Apply standard conversions
*reinterpret_cast<double*>p = 3.0;  // Pretend p has type double*
*const_cast<double*>&PI = 4;        // Same type except for const
```

Arrays

The size of an array must be specified by a constant, and may be left blank if the array is initialized from a list. Array bounds start at 0. There are no run time checks on array bounds. Multi-dimensional arrays use a separate bracket for each dimension. An array name used without brackets is a pointer to the first element.

```
int a[]={0,1,2,3,4};    // Array with elements a[0] to a[4]
int b[5]={6,7};         // Implied ={6,7,0,0,0};
int c[5];               // Not initialized, c[0] to c[4] could have any values
int d[2][3]={{1,2,3},{4,5,6}};  // Initialized 2-D array
int i=d[1][2];          // 6
d[-1][7]=0;             // Not checked, program may crash
```

The bare name of an array is a const pointer to the first element. If p is a pointer to an array element, then p+i points i elements ahead, to p[i]. By definition, p[i] is *(p+i).

```
int a[5];               // a[0] through a[4]
int* p=a+2;             // *p is a[2]
p[1];                   // a[3]
p-a;                    // 2
p>a;                    // true because p-a > 0
p-1 == a+1              // true, both are &a[1]
*a;                     // a[0] or p[-2]
a=p;                    // Error, a is const (but not *a)
```

A literal string enclosed in double quotes is an unnamed static array of const char with an implied '\0' as the last element. It may be used either to initialize an array of char, or in an expression as a pointer to the first char. Special chars in literals may be escaped with a backslash as before. Literal strings are concatenated without a + operator (convenient to span lines).

```
char s[]="abc";                       // char s[4]={'a','b','c','\0'};
const char* p="a" "b\n";              // Points to the 'a' in the 4 element array "ab\n\0"
const char* answers[2]={"no","yes"};  // Array of pointers to char
cout << answers[1];                   // prints yes (type const char*)
cout << answers[1][0];                // prints y (type const char)
"abc"[1]                              // 'b'
```

Arrays do not support copying, assignment, or comparison.

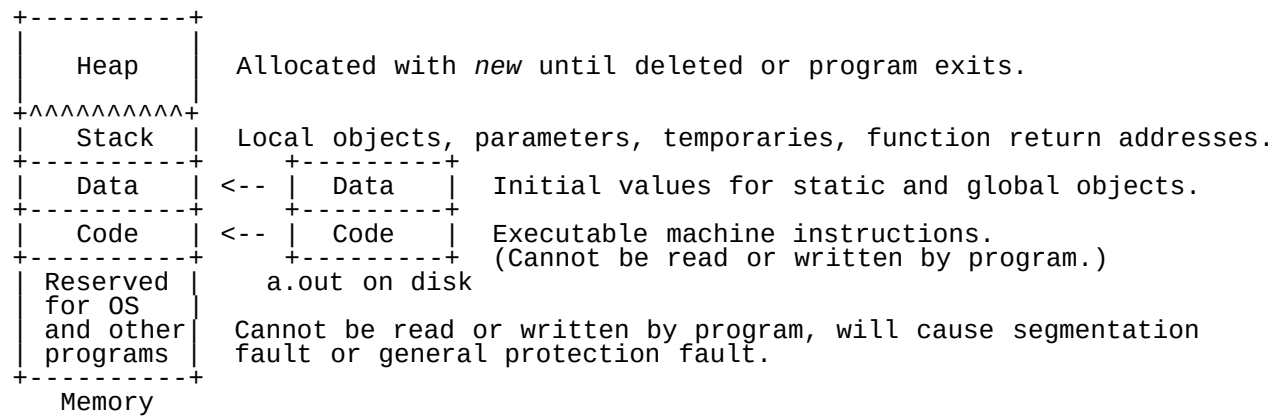
```
int a[5], b[5]=a;       // Error: can't initialize b this way
b=a;                    // Error: can't assign arrays
b==a;                   // false, comparing pointers, not contents
"abc"=="abc"            // false, comparing pointers to 2 different locations
```

The size of an array created with `new[]` may be an expression. The elements cannot be initialized with a list. There is no run time check against accessing deleted elements.

```
int n, *p;
cin >> n;
p=new int[n];  // Elements p[0] to p[n-1] with values initially undefined
delete[] p;    // Use delete with new or new(), delete[] with new[]
p[0] = 1;      // May crash
```

static

Normally, objects are placed on the stack. Memory is allocated by growing the stack at the top; thus objects are destroyed in the reverse order in which they are created. An object's life span is the same as its scope. If an object comes into scope more than once, then it is reinitialized each time, and destroyed when leaving its scope.



`static` objects are placed in the data segment. They are initialized from values stored in the executable file, and therefore these values must be known at compile time. Initialization occurs only once. Values are maintained when the object is out of scope (e.g. between function calls), and it is safe to return a pointer or reference to them. Numeric values not explicitly initialized are set to 0.

```
int& f() {        // Return by reference, f() is an alias for s, not a temporary copy
  static int s=1; // Initialized only once
  ++s;
  return s;       // Safe to return by reference
}
int main() {
  cout << f();    // 2
  cout << f();    // 3
  f()=5;          // OK, s=5;
  s=6;            // Error, s is not in scope
```

register

(Rare) A hint to the compiler to optimize an int or pointer for speed. It is no longer used because most optimizers can do a better job.

```
register int x;
```

volatile

(Rare) Indicates that an object might be modified from outside the program (e.g. a hardware input port) and that the optimizer should not make copies of it. Its use is machine dependent.

```
const volatile unsigned short& port=*(const short*)0xfffe; // 16 bit port at address xfffe
```

Standard Library Types

Standard library types (string, vector, map...) and objects (cin, cout...) require a `#include <header>` and must be extracted from namespace `std`, either with a `using namespace std;` statement or by using the fully qualified names preceded with `std::`, as in `std::cout`.

```
#include <iostream>
int main() {
  std::cout << "Hello\n";
  return 0;
}
```

```
#include <iostream>
using namespace std;
int main() {
  cout << "Hello\n";
  return 0;
}
```

<iostream>

The header <iostream> defines global object `cin` of type `istream`, and global objects `cout, cerr, clog` of type `ostream`. `cin` represents standard input, normally the keyboard, unless redirected to a file or piped on the command line. `cout` represents standard output, which is normally the screen unless redirected or piped. Writing to `cerr` or `clog` both write to the screen even if output is redirected. The difference is that writing a newline ('\n') flushes any buffered output to `cerr` but not to `cout` or `clog`.

In the following, `in` is an istream (cin), `out` is an ostream (cout, cerr, clog), i is int, c is char, and cp is char*.

```
in >> x;               // Read 1 word to numeric, string, or char* x, return in
in.get();              // Read 1 char (0-255) or EOF (-1) as an int
in.get(c);             // Read 1 char into c, return in
in.unget();            // Put back last char read, return in
in.getline(cp, i);     // Read up to i chars into char cp[i] or until '\n', return in
in.getline(cp, i, c);  // Read to c instead of '\n', return in
getline(in, s);        // Read up to '\n' into string s, return in
in.good();             // true if no error or EOF
bool(in);              // in.good();
in.bad();              // true if unexpected char in formatted input
in.clear();            // Allow more input after bad, or throw an ios::failure
in.eof();              // true if end of file
in.fail();             // true if system input error

out << x;                // Formatted output, redirected with >
out << endl;             // Print '\n' and flush
```

Input with >> reads a contiguous sequence of non-whitespace characters. If x is numeric and the next word contains invalid characters (such as "1.5" or "foo" for an int), then the first offending character remains unread, in.bad() is set, and no further input can occur until in.clear() is called. Input into a char* array is not bounds checked. Input returns the istream to allow chaining, and has a conversion to bool to test for success. Output also returns the ostream to allow chaining.

```
// Read and print pairs of strings and ints until something goes wrong
// Input:  hi 3 there 5 this is 1 test
// Output: hi 3
           there 5

string s; int i;
while (cin >> s >> i)
  cout << s << " " << i << endl;
cin.clear();
```

The get() methods read one character including whitespace. The various getline() functions read up through the next newline character and discard the newline. The methods `good(), bad(), eof(), fail(), clear()`, and implicit conversion to bool are available in `ostream`, just as in `istream`, but are seldom used.

<iomanip>

Defines manipulators for formatted output of numeric types. They have no effect on strings. `setw()` applies only to the

next object printed, but the others remain in effect until changed.

```
out << setw(i);          // Pad next output to i chars, then back to 0
out << setfill(c);       // Pad with c (default ' ')
out << setprecision(i);  // Use i significant digits for all float, double

cout << setw(6) << setprecision(3) << setfill('0') << 3.1; // print "003.10"
```

<fstream>

Defines types `ifstream` and `ofstream` representing input and output files respectively. `ifstream` is derived from `istream`, inheriting all its operations (such as >>). In addition,

```
ifstream in(cp);          // Open file named cp for reading
ifstream in(cp, ios::in | ios::binary);  // Open in binary mode
bool(in);                 // true if open successful
```

`cp` is the file name. It must be a char*, not string (use `s.c_str()` to convert string `s`). Input is normally in text mode. In Windows, carriage returns ('\r') are discarded, and an ASCII 26 ('\032') signals end of file. In binary mode and in UNIX, no such translation occurs. The file is closed when the ifstream is destroyed.

```
{
  ifstream f("input.dat", ios::in | ios::binary);
  if (!f)
    cerr << "File not found\n";
  else {
    int i=f.get();  // First byte or EOF if empty
  }
} // f closed here
```

`ofstream` is derived from `ostream`, inheriting all its operations (such as <<). In addition,

```
ofstream os(cp);          // Open file named cp for writing
ofstream os(cp, ios::out | ios::binary);  // Open in binary mode
```

In text mode in Windows, writing '\n' actually writes "\r\n". The file named cp is overwritten if it exists, or created otherwise. The file is flushed and closed when the ofstream is destroyed.

<string>

A `string` is like an array of char, but it also supports copying, assignment, and comparison, and its size may be set or changed at run time. '\0' has no special meaning. There is implicit conversion from char* to string in mixed type expressions.

```
string()           // Empty string
string(cp)         // Convert char* cp to string
string(n, c)       // string of n copies of char c
s=s2               // Assign char* or string s2 to string s
s1<s2              // Also ==, !=, >, <=, >=, either s1 or s2 may be char*
s.size()           // Length of string s
string::size_type  // Type of s.size(), usually unsigned int
s.empty()          // True if s.size() == 0
s[i]               // i'th char, 0 <= i < s.size() (unchecked), may be assigned to
s.at(i)            // s[i] with bounds check, throws out_of_range
s1+s2              // Concatenate strings, either s1 or s2 may be char or char*
s+=s2              // Append string, char, or char* s2 to string s
s.c_str()          // string s as a const char* with trailing '\0'
s.substr(i, j)     // Substring of string s of length j starting at s[i]
s.substr(i)        // Substring from s[i] to the end
s.find(s2)         // Index of char, char*, or string s2 in s, or string::npos if not found
s.rfind(s2)        // Index of last occurrence of s2 in s
s.find_first_of(s2)     // Index of first char in s that occurs in s2
s.find_last_of(s2)      // Index of last char in s that occurs in s2
s.find_first_not_of(s2) // Index of first char in s not found in s2
s.find_last_not_of(s2)  // Index of last char in s not found in s2
s.replace(i, j, s2)     // Replace s.substr(i, j) with s2
```

`s.size()` should be converted to `int` to avoid unsigned comparison.

```
string s(3,'a');   // "aaa"
s += "b"+s;        // "aaabaaa"
for (int i=0; i!=int(s.size()); ++i) {  // print s one char at a time
  cout << s[i];
s.size() > -1;     // false!  -1 is converted to unsigned
```

`string` supports standard container operations with regard to iterators. `string` iterators are random, supporting all the pointer operators of `char*`. The notation `[b,e)` means the sequence such that pointer or iterator b points to the first element and e points one past the last element.

```
s.begin()          // Iterator pointing to s[0]
s.end()            // Iterator pointing 1 past last char
string::iterator   // Iterator type, like char*
string::const_iterator  // Type if s is const, like const char*
string(b, e)       // string initialized from sequence [b,e)
s.erase(b)         // Remove char in s pointed to by b
s.erase(b, e)      // Remove substring [b,e) from s
s.replace(b, e, s2)  // Replace substring [b,e) with string s2
```

Conversion from `iterator` to `const_iterator` is allowed, but not the other way. `const_iterator` should be used if the string is not going to be modified.

```
char* cp="ABCDE";
string s(cp, cp+5); // "ABCDE"
string s2(s.begin()+1, s.end()-1);  // "BCD"
for (string::const_iterator p=s.begin(); p!=s.end(); ++p)  // Print s one char at a time
  cout << *p;       // or p[0]
```

As with arrays and pointers, indexing and iterator dereferencing are not checked at run time. Creating a string with a negative or very large size is also trouble.

```
string s(-1, 'x');               // Crash, negative size
string s2(s.end(), s.begin());  // Crash, negative size
s[-1]='x';                       // Crash, out of bounds
*s.end()='x';                    // Crash, out of bounds
string::iterator p; *p='x';     // Crash, dereferencing uninitialized iterator
```

<vector>

A `vector<T>` is like an array of T, but supports copying, assignment, and comparison. Its size can be set and changed at run time, and it can efficiently implement a stack (O(1) time to push or pop). It has random iterators like `string`, which behave like type T* (or const T* if the vector is const). If T is numeric, elements are initialized to 0. It is not possible to have an initialization list such as {1,2,3}.

```
vector<T>()            // Empty vector, elements of type T
vector<T>(n)           // n elements, default initialized
vector<T>(n, x)        // n elements each initialized to x
vector<T> v2=v;        // Copy v to v2
v2=v;                  // Assignment
v2<v                   // Also >, ==, !=, <=, >= if defined for T
vector<T>(b, e)        // Initialize to sequence [b, e)
v.size()               // n
vector<T>::size_type   // Type of v.size(), usually unsigned int
v.empty()              // true if v.size() == 0
v[i]                   // i'th element, 0 <= i < v.size() (unchecked), may be assigned to
v.at(i)                // v[i] with bounds check, throws out_of_range
v.begin(), v.end()     // Iterators [b, e)
vector<T>::iterator    // Iterator type, also const_iterator
v.back()               // v[v.size()-1] (unchecked if empty)
v.push_back(x)         // Increase size by 1, copy x to last element
v.pop_back()           // Decrease size by 1 (unchecked if empty)
v.front()              // v[0] (unchecked)
v.resize(n)            // Change size to n >= 0 (unchecked)
v.insert(d, x)         // Insert x in front of iterator d, shift, increase size by 1
v.insert(d, n, x)      // Insert n copies of x in front of d
v.insert(d, b, e)      // Insert copy of [b, e) in front of d
v.erase(d)             // Remove *d, shift, decrease size by 1
v.erase(d, e)          // Remove subsequence [d, e)
v.clear()              // v.erase(v.begin(), v.end())
v.reserve(n)           // Anticipate that v will grow to size n >= v.size()
v.capacity()           // Reserved size
```

For insert and erase, d and e must point into v (and d <= e) or the program may crash. Elements from *d to the end are shifted and the size is changed as needed. Saved copies of iterators may become invalid after any change of size or capacity (not checked).

To implement `push_back()` efficiently, a vector typically doubles the reserved space when it runs out in order to minimize memory reallocation and copying. `reserve()` allows this strategy to be optimized.

```
// Read words from input into a stack, print in reverse order
string s;
vector<string> v;
while (cin >> s)
  v.push_back(s);
while (!v.empty()) {
  cout << v.back() << endl;
  v.pop_back();
}
```

<deque>

A `deque` (double ended queue) is just like a vector, but optimized for adding and removing elements at either end in O(1) time. It lacks `reserve()` and `capacity()` and adds

```
v.push_front(x)        // v.insert(v.begin(), x)
v.pop_front()          // v.erase(v.begin())
```

<list>

A `list` is like a deque but optimized for insert and erase at any point at the cost of random access. It lacks [] (indexing), and its iterators are *bidirectional*, not supporting [], +, -, <, >, <=, or >=. `list`adds

```
v.splice(d, v2, b);  // Move *b from list v2 to in front of d in v
v.splice(d, v2);     // Move all elements of list v2 to in front of d in v
v.splice(d, v2, b, e); // Move [b,e) in v2 to in front of d at v
v.remove(x);         // Remove all elements equal to x
v.remove_if(f);      // Remove elements x where f(x) is true
v.sort();            // Sort list
v.sort(f);           // Sort list using function bool f(x,y) instead of x < y
v.merge(v2);         // Merge sorted list v2 into sorted list v
v.merge(v2, f);      // Merge using f(x,y) instead of x < y to sort v
v.unique();          // Remove duplicates from sorted list
v.unique(f);         // Use f(x,y) instead of x == y
v.reverse();         // Reverse order of elements
```

Iterators can only be moved one element at a time using ++ or --, and compared using == or !=.

```
char* cp="ABCDE";
list<char> v(cp, cp+5);  // v.size() is 5
for (list<char>::const_iterator p=v.begin(); p!=v.end(); ++p)  // Print ABCDE
  cout << *p;
```

<map>

A `map<K,V> m` is a set of key-value pairs with unique, sorted keys of type K and values of type V. m[k] efficiently (O(log n) time) returns the value associated with k (as an lvalue), or creates a default value (0 if V is numeric) if k is used for the first time. A map iterator points to a `pair<const K, V>`, which has members `first` of type const K and `second` of type V.

```
pair<K,V> x(k,v);    // Create a pair x containing copies of k and v
x.first              // k
x.second             // v
x=make_pair(k,v)     // x.first=k; x.second=v;

map<K,V> m;          // map sorted by < on K
map<K,V,f>()         // map sorted by f(x,y) instead of x<y on K
m[k]=v;              // Associate v (type V) with unique key k of type K
m[k]                 // Retrieve v, or associate V() with k if new
m.size()             // Number of unique keys
```

```
m.empty()            // true if m.size() == 0
map<K,V>::iterator   // bidirectional, points to a pair<const K, V>
map<K,V>::const_iterator   // points to a pair<const K, const V>
m.begin()            // Points to first pair (lowest k)
m.end()              // Points 1 past last pair
m.find(k)            // Points to pair containing k or m.end() if not found
m.erase(k)           // Remove key K and its associated value
m.erase(b)           // Remove pair pointed to by iterator b
m.erase(b, e)        // Remove sequence [b, e)
m.clear()            // Make empty: m.erase(m.begin(), m.end())
m==m2                // Compare maps, also !=, <, <=, >, >=
```

We use `m.find(k)` rather than m[k] when we wish to look up k without increasing the size of m if k is not found.

```
// Read words, print an alphabetical index of words with their counts
string s;
map<string, int> m;
while (cin >> s)
  ++m[s];
for (map<string, int>::const_iterator p=m.begin(); p!=m.end(); ++p)
  cout << p->first << " " << p->second << endl;
```

A `multimap` is a `map` that allows duplicate keys. It support all `map` operations except []. Elements are added by inserting a pair<K,V> and retrieved by `m.equal_range(k)` which returns a pair of iterators defining the sequence of pairs matching k.

```
multimap<K,V,f> m;   // f defaults to < on K
m.insert(make_pair(k,v))  // Insert a pair
pair<multimap<K,V,f>::iterator, multimap<K,V,f>::iterator> p
  = m.equal_range(k) // Sequence with key k is [p->first, p->second)
```

f (when used as a template argument) is a *functoid* (or function object), a class that looks like a function by overloading (). For example:

```
template <class T> class GreaterThan {
public:
  bool operator()(const T& a, const T& b) const {return b < a;}
};

map<string, int, GreaterThan<T> > m;  // keys sorted in reverse order
```

Some function objects can be found in <functional>.

<set>

A `set<K>` and `multiset<K>` are like a map and multimap, but without values. Iterators point to a K rather than a pair. There is no [] operator.

```
set<K> m;            // Elements are sorted by < on K
m.insert(k)          // Add an element
m.erase(k)           // Remove an element
m.find(k)!=m.end()   // Test if k is in m
```

<queue>

A `queue` is a container in which elements are inserted at the back and removed from the front. This could also be done with a `deque` or `list`, so no new capabilities are provided. A `queue` does not support iterators or indexing.

```
queue<T> q;          // Queue of type T
q.size()             // Number of items in q
q.empty()            // true if q.size() == 0
q.push(x)            // Put x in the back
x=q.back()           // The item last pushed, may be assigned to
x=q.front()          // The next item to pop, may be assigned to
q.pop()              // Remove the front item
```

A `priority_queue` is more useful. It sorts the items as they are pushed so that the largest is on top and removed first.

```
priority_queue<T> q; // Element type is T
```

```
priority_queue<T, vector<T>, f> q;  // Use functoid f(x,y) instead of x < y to sort
q.size(), q.empty()  // As before
q.push(x)            // Insert x
x=q.top()            // Largest item in q, cannot be assigned to
q.pop()              // Remove top item
```

<stack>

Items are popped from the top of a `stack` in the reverse order in which they were pushed. It does not provide any new functionality beyond a vector, deque, or list, and does not support iterators or indexing.

```
stack<T> s;          // Stack with elements of type T
s.size(), s.empty()  // As with queue
s.push(x);           // Put x on top
x=s.top();           // Last item pushed, may be assigned to
s.pop();             // Remove the top item
```

<bitset>

A `bitset<N>` is like a `vector<bool>` with fixed size N, but without iterators, and supporting logical operators like an N-bit `int`. Its elements have the values 0 or 1. It is implemented efficiently, with 8 elements per byte.

```
bitset<N> b;         // N-bit bitset, N must be a compile time constant
bitset<N> b=x;       // Initialize b[0]..b[31] from bits of long x
b[i]                 // i'th bit, 0 <= i < N or throw out_of_range()
b.size()             // N, cannot be changed
b.set(i)             // b[i] = 1
b.reset(i)           // b[i] = 0
b.flip(i)            // b[i] = 1 - b[i]
b.test(i)            // true if b[i] == 1
b.set()              // Set all bits, also b.reset(), b.flip()
b & b2               // Bitwise AND, also | ^ ~ << >> &= |= ^= <<= >>= == !=
b.count()            // Number of bits set to 1
b.any()              // true if b.count() > 0
b.none()             // true if b.count() == 0
cin >> b             // Read bits as '0' and '1' e.g. "10101"
cout << b            // Write bits as '0' and '1'
bitset<N> b(s);      // Initialize from string s of '0' and '1' or throw invalid_argument()
s=b.template to_string<char>()  // Convert to string
x=b.to_ulong()       // Convert to unsigned long, throw overflow_error() if bits > 31 set
```

<valarray>

A `valarray` is like a fixed sized array or vector that supports arithmetic operations on all the elements at once. For instance, if x and y are valarrays of the same size, then x+y is a valarray containing the sums of the corresponding elements. Likewise, y=sqrt(x) assigns y[i]=sqrt(x[i]) to each element of y. In mixed type expressions, a scalar (element of type T) is promoted to a valarray of the same size by duplicating it, e.g. x+1 adds 1 to all elements of x.

```
valarray<T> v(n);    // n elements of type T, initially T() or 0
valarray<T> v(x, n); // n copies of x (note arguments are backwards)
valarray<T> v(a, n); // Initialize from array a[0]..a[n-1]
valarray<T> v;       // size is 0
v.size()             // Number of elements, n
v[i]                 // i'th element, 0 <= i < n, not checked
v+=x, v+=v           // Add x or v[i] to all v[i], also = -= *= /= %= ^= &= |= <<= >>=
v+v, v+x, x+v        // Also - * / % ^ & | << >> and unary + - ~ !
sqrt(v)              // Also all functions in cmath
x=v.sum()            // Sum of all elements
v.shift(n)           // Move all v[i] to v[i+n], shift in 0
v.cshift(n)          // Move v[i] to v[(i+n) % v.size()]
v.resize(n)          // Change size to n, but reset all elements to 0
v.resize(n, x)       // Change size to n, set all elements to x
```

<complex>

A `complex` supports complex arithmetic. It has real and imaginary parts of type T. Mixed type expressions promote real to complex (e.g. double to complex<double> and lower precision to higher precision (e.g. complex<int> to

complex<double>).

```
complex<T> x;         // (0,0), T is a numeric type
complex<T> x=r;       // (r,0), convert real r to complex
complex<T> x(r, i);   // (r,i)
x=polar<T>(rho, theta); // Polar notation: radius, angle in radians
x.real()              // r
x.imag()              // i
abs(x)                // rho = sqrt(r*r+i*i)
arg(x)                // tan(theta) = i/r
norm(x)               // abs(x)*abs(x)
conj(x)               // (r,-i)
x+y                   // Also - * / == != = += -= *= /= and unary + -
sin(x)                // Also sinh, sqrt, tan, tanh, cos, cosh, exp, log, log10, pow(x,y)
cout << x             // Prints in format "(r,i)"
cin >> x              // Expects "r", "(r)", or "(r,i)"
```

<stdexcept>, <exception>

The standard library provides a hierarchy of exception types. Not all of them are used by the library, but any may be thrown.

```
  Type                  Header                    Thrown by
  exception             stdexcept, exception
    logic_error         stdexcept
      length_error      stdexcept
      domain_error      stdexcept
      out_of_range      stdexcept                 .at(i) (vector/string/deque index out of
bounds)
      invalid_argument  stdexcept, bitset         bitset("xxx") (not '0' or '1')
    runtime_error       stdexcept
      range_error       stdexcept
      overflow_error    stdexcept
      underflow_error   stdexcept
    bad_alloc           new                       new, new[] (out of memory)
    bad_cast            typeinfo                  dynamic_cast<T&> (can't convert to derived)
    bad_typeid          typeinfo                  typeid(*p) when p==0
    bad_exception       exception
    ios_base::failure   ios, iostream, fstream    istream::clear(), ostream::clear()
```

Catching a base class catches all derived classes, thus `catch(exception e)` catches all of the above types. However, C++ allows throwing exceptions not derived from `exception`, so this may not catch everything. All exceptions provide the following interface:

```
throw exception(msg)   // Throw exception with char* or string msg
throw exception();     // Default msg
catch(exception e) {e.what();}  // msg as a char*
```

New exceptions may be derived from existing types to maintain this interface (see inheritance).

```
class MyError: public exception {
public:
  MyError(const string& msg=""): exception(msg) {}
}
```

C++ Standard Library Functions

Many C++ standard library functions operate on sequences denoted by iterators or pointers. **Iterators** are a family of types that include pointers. They are classified by the operators they support.

- **input:** `++p, p++, p=q, p==q, p!=q, *p` (read-only)
- **output:** `p=q, p==q, p!=q, *p++ = x` (alternating write/increment)
- **forward:** input and output and `p->m, *p` (multiple read-write)
- **bidirectional:** forward and `--p, p--`, implemented by list, map, multimap, set, multiset.
- **random:** bidirectional and `p<q, p>q, p<=q, p>=q, p+i, i+p, p-i, p-q, p[i]`, implemented by arrays (as pointers), string, vector, deque.

Some algorithms require certain iterator types, but will accept more powerful types. For example, `copy(b, e, d)` require b and e to be at least input iterators and d to be at least an output iterator. But it will accept forward, bidirectional, or random iterators because these all support input and output operations. `sort()` requires random iterators and will accept no other type.

The notation [b,e) denotes the sequence of e-b objects from b[0] to e[-1], i.e. b points to the beginning of the sequence and e points one past the end. For most containers, v, the sequence is [v.begin(), v.end()). For an array of n elements, the sequence is [a, a+n).

<algorithm>

In the following, b and e are input iterators, and d is an output iterator, unless otherwise specified. Parameters eq and lt are optional, and default to functions that take 2 arguments x and y and return x==y and x<y respectively, e.g. `bool eq(x,y) {return x==y;}`. x and y are objects of the type pointed to by the iterators. p is a pair of iterators. f is a function or function object as noted.

```
// Operations on ordinary objects
swap(x1, x2);              // Swap values of 2 objects of the same type
min(x1, x2);               // Smaller of x1 or x2, must be same type
max(x1, x2);               // Larger of x1 or x2, must be same type

// Properties of sequences (input iterators)
equal(b, e, b2, eq);       // true if [b,e)==[b2,...)
lexicographical_compare(b, e, b2, e2, lt);  // true if [b,e)<[b2,e2)
i=min_element(b, e);       // Points to smallest in [b,e)
i=max_element(b, e);       // Points to largest
n=count(b, e, x);          // Number of occurrences of x in [b,e)
n=count_if(b, e, f);       // Number of f(x) true in [b,e)

// Searching, i points to found item or end (e) if not found
i=find(b, e, x);           // Find first x in [b,e)
i=find_if(b, e, f);        // Find first x where f(x) is true
i=search(b, e, b2, e2, eq);// Find first [b2,e2) in [b,e) (forward)
i=find_end(b, e, b2, e2, eq); // Find last [b2,e2) in [b,e) (forward)
i=search_n(b, e, n, x, eq);// Find n copies of x in [b,e) (forward)
p=mismatch(b, e, b2, eq);  // Find first *p.first in [b,e) != *p.second in [b2,.) (forward)
i=adjacent_find(b, e, eq); // Find first of 2 equal elements (forward)

// Modifying elements
i=copy(b, e, d);           // Copy [b,e) to [d,i)
fill(d, i, x);             // Set all in [d,i) to x (forward)
i=fill_n(d, n, x);         // Set n elements in [d,i) to x
generate(d, i, f);         // Set [d,i) to f() (e.g. rand) (forward)
i=generate_n(d, n, f);     // Set n elements in [d,i) to f()
f=for_each(b, e, f);       // Call f(x) for each x in [b,e)
i=transform(b, e, d, f);   // For x in [b,e), put f(x) in [d,i)
i=transform(b, e, b2, d, f);  // For x in [b,e), y in [b2,.), put f(x,y) in [d,i)
replace(b, e, x, y)        // Replace x with y in [b,e)
replace_if(b, e, f, y);    // Replace with y in [b,e) where f(x) is true
i=replace_copy(b, e, d, x, y);    // Copy [b,e) to [d,i) replacing x with y
i=replace_copy_if(b, e, d, f, y); // Copy replacing with y where f(x) is true

// Rearranging sequence elements
sort(b, e, lt);            // Sort [b,e) by < (random)
stable_sort(b, e, lt);     // Sort slower, maintaining order of equal elements (random)
partial_sort(b, m, e, lt); // Sort faster but leave [m,e) unsorted (random)
nth_element(b, m, e, lt);  // Sort fastest but only *m in proper place (random)
iter_swap(b, e);           // swap(*b, *e) (forward)
i=swap_ranges(b, e, b2);   // swap [b,e) with [b2,i) (forward)
i=partition(b, e, f);      // Moves f(x) true to front, [i,e) is f(x) false (bidirectional)
i=stable_partition(b, e, f);  // Maintains order within each partition
i=remove(b, e, x);         // Move all x to end in [i,e) (forward)
i=remove_if(b, e, f);      // Move f(x) true to front in [b,i) (forward)
i=remove_copy(b, e, d, x); // Copy elements matching x to [d,i)
i=remove_copy_if(b, e, d, f);  // Copy elements x if f(x) is false to [d,i)
replace(b, e, x1, x2);     // Replace x1 with x2 in [b,e)
i=replace_copy(b, e, d, x1, x2);  // Copy [b,e) to [d,i) replacing x1 with x2
reverse(b, e);             // Reverse element order in [b,e) (bidirectional)
i=reverse_copy(b, e, d);   // Copy [b,e) to [d,i) reversing the order (b,e bidirectional)
rotate(b, m, e);           // Move [b,m) behind [m,e) (forward)
i=rotate_copy(b, m, e, d); // Rotate into [d,i)
random_shuffle(b, e, f);   // Random permutation, f() defaults to rand()
next permutation(b, e, lt);// Next greater sequence, true if successful (bidirectional)
```

```
prev_permutation(b, e, lt);// Previous permutation, true if successful (bidirectional)

// Operations on sorted sequences
i=unique(b, e, eq);            // Move unique list to [b,i), extras at end
i=unique_copy(b, e, d, eq);    // Copy one of each in [b,d) to [d,i)
i=binary_search(b, e, x, lt);  // Find i in [b,e) (forward)
i=lower_bound(b, e, x, lt);    // Find first x in [b,e) or where to insert it (forward)
i=upper_bound(b, e, x, lt);    // Find 1 past last x in [b,e) or where to insert it (forward)
p=equal_range(b, e, x, lt);    // p.first = lower bound, p.second = upper bound (forward)
includes(b, e, b2, e2, lt);    // true if [b,e) is a subset of [b2,e2)
i=merge(b, e, b2, e2, d, lt);  // Merge [b,e) and [b2,e2) to [d,i)
inplace_merge(b, m, e, lt);    // Merge [b,m) and [m,e) to [b,e) (bidirectional)
i=set_union(b, e, b2, e2, d, lt);  // [d,i) = unique elements in either [b,e) or [b2,e2)
i=set_intersection(b, e, b2, e2, d, lt);  // [d,i) = unique elements in both
i=set_difference(b, e, b2, e2, d, lt);    // [d,i) = unique elements in [b,e) but not [b2,e2)
i=set_symmetric_difference(b, e, b2, e2, d, lt);  // [d,i) = elements in one but not both
```

Algorithms never change the size of a container. When copying, the destination must be large enough to hold the result.

```
int a[5]={3,1,4,1,6};
vector b(5);
copy(a, a+5, v.begin());   // Copy a to v
remove(a, a+5, 1);         // {3,4,6,1,1}, returns a+3
sort(a, a+4);              // {1,3,4,6,1}
```

<numeric>

In the following, `plus`, `minus`, and `times` are optional functions taking 2 arguments x and y that return x+y, x-y, and x*y respectively, e.g. `int plus(int x, int y) {return x+y;}`

```
x=accumulate(b, e, x, plus);                // x + sum over [b,e)
x=inner_product(b, e, b2, x, plus, times);  // x + sum [b,e)*[b2,e2)
adjacent_difference(b, e, minus);           // for i in (b,e) *i -= i[-1]
partial_sum(b, e, plus);                    // for i in [b,e) *i += sum [b,i)
```

<iterator>

An inserter is an output iterator that expands the container it points to by calling push_back(), push_front(), or insert(). The container must support this operation. A stream iterator can be used to do formatted input or output using >> or <<

```
back_inserter(c);             // An iterator that appends to container c
front_inserter(c);            // Inserts at front of c
inserter(c, p);               // Inserts in front of p
ostream_iterator<T>(out, cp); // Writes to ostream separated by char* cp (default " ")
istream_iterator<T>(in);      // An input iterator that reads T objects from istream
```

The most common use is to copy to an empty vector, deque, or list.

```
vector<int> from(10), to;
copy(from.begin(), from.end(), back_inserter(to));
```

This header also defines tag types to be used for creating iterator types that work with algorithms. See defining iterators.

<functional>

Functions in <functional> create *function objects*, which are objects that behave like functions by overloading `operator()`. These can be passed to algorithms that take function arguments, e.g.

```
vector<int> v(10);
sort(v.begin(), v.end(), greater<int>());  // Sort v in reverse order
int x=accumulate(v.begin(), v.end(), 1, multiplies<T>);  // Product of elements
```

The following create function objects that take one or two parameters x and y of type T and return the indicated

```
equal to<int>()(3,4)
```

expression, i.e., returns false.

```
// Predicates (return bool)
equal_to<T>()                 // x==y
not_equal_to<T>()             // x!=y
greater<T>()                  // x>y
less<T>()                     // x<y
greater_equal<T>()            // x>=y
less_equal<T>()               // x<=y
logical_and<bool>()           // x&&y
logical_or<bool>()            // x||y
logical_not<bool>()           // !x (unary)

// Arithmetic operations (return T)
plus<T>()                     // x+y
minus<T>()                    // x-y
multiplies<T>()               // x*y
divides<T>()                  // x/y
modulus<T>()                  // x%y
negate<T>()                   // -x (unary)
```

A *binder* converts a 2-argument function object into a 1-argument object by binding a fixed value c to the other argument, e.g. `bind2nd(less<int>(), 10)` returns a function object that takes one argument `x` and returns true if `x<10`.

```
bind1st(f, c)                 // An object computing f(c,y)
bind2nd(f, c)                 // An object computing f(x,c)

i=find_if(v.begin(), v.end(), bind2nd(equal_to<int>(), 0));  // Find first 0
```

The following convert ordinary functions and member functions into function objects. All functions must be converted to be passed to bind1st and bind2nd. Member functions must also be converted to be passed to algorithms.

```
ptr_fun(f)                    // Convert ordinary function f to object
mem_fun(&T::f)                // Convert member function of class T
mem_fun_ref(T::f)             // Same

i=find_if(v.begin(), v.end(), mem_fun(&string::empty));  // Find ""
transform(v.begin(), v.end(), v.begin(), bind2nd(ptr_fun(pow), 2.0));  // Square elements
```

`not1()` and `not2()` negate a unary or binary function object.

```
not1(f)                       // Object computing !f(x)
not2(f)                       // Object computing !f(x,y)

i=find_if(v.begin(), v.end(), not1(bind2nd(equal_to<int>(), 0)));  // Find nonzero
```

<new>

The default behavior of `new` is to throw an exception of type `bad_alloc` if out of memory. This can be changed by writing a function (taking no parameters and returning void) and passing it to`set_new_handler()`.

```
void handler() {throw bad_alloc();} // The default
set_new_handler(handler);
```

`new(nothrow)` may be used in place of `new`. If out of memory, it returns 0 rather than throw bad_alloc.

```
int* p = new(nothrow) int[1000000000];  // p may be 0
```

C Library Functions

The C library is provided for backwards compatibility with the C language. Because C lacked namespaces, all types and functions were defined globally. For each C header, C++ provides an additional header by prefixing "c" and dropping the ".h" suffix, which places everything in namespace `std`. For instance, `<stdio.h>` becomes `<cstdio>`.

<cstdlib>

Miscellaneous functions. s is type char*, n is int

```
atoi(s); atol(s); atof(s);// Convert char* s to int, long, double e.g. atof("3.5")
abs(x); labs(x);          // Absolute value of numeric x as int, long
rand();                   // Pseudo-random int from 0 to RAND_MAX (at least 32767)
srand(n);                 // Initialize rand(), e.g. srand(time(0));
system(s);                // Execute OS command s, e.g. system("ls");
getenv(s);                // Environment variable or 0 as char*, e.g. getenv("PATH");
exit(n);                  // Kill program, return status n, e.g. exit(0);
void* p = malloc(n);      // Allocate n bytes or 0 if out of memory.  Obsolete, use new.
p = calloc(n, 1);         // Allocate and set to 0 or return NULL.  Obsolete.
p = realloc(p, n);        // Enlarge to n bytes or return NULL.  Obsolete.
free(p);                  // Free memory.  Obsolete: use delete
```

<cctype>

Character tests take a char c and return bool.

```
isalnum(c);               // Is c a letter or digit?
isalpha(c); isdigit(c);   // Is c a letter?  Digit?
islower(c); isupper(c);   // Is c lower case?  Upper case?
isgraph(c); isprint(c);   // Printing character except/including space?
isspace(c); iscntrl(c);   // Is whitespace?  Is a control character?
ispunct(c);               // Is printing except space, letter, or digit?
isxdigit(c);              // Is hexadecimal digit?
c=tolower(c); c=toupper(c);   // Convert c to lower/upper case
```

<cmath>

Functions take double and return double.

```
sin(x); cos(x); tan(x);   // Trig functions, x in radians
asin(x); acos(x); atan(x);// Inverses
atan2(y, x);              // atan(y/x)
sinh(x); cosh(x); tanh(x);// Hyperbolic
exp(x); log(x); log10(x); // e to the x, log base e, log base 10
pow(x, y); sqrt(x);       // x to the y, square root
ceil(x); floor(x);        // Round up or down (as a double)
fabs(x); fmod(x, y);      // Absolute value, x mod y
```

<ctime>

Functions for reading the system clock. `time_t` is an integer type (usually long). `tm` is a struct.

```
clock()/CLOCKS_PER_SEC;   // Time in seconds since program started
time_t t=time(0);         // Absolute time in seconds or -1 if unknown
tm* p=gmtime(&t);         // 0 if UCT unavailable, else p->tm_X where X is:
  sec, min, hour, mday, mon (0-11), year (-1900), wday, yday, isdst
asctime(p);               // "Day Mon dd hh:mm:ss yyyy\n"
asctime(localtime(&t));   // Same format, local time
```

<cstring>

Functions for performing string-like operations on arrays of char marked with a terminating '\0' (such as `"quoted literals"` or as returned by `string::c_str()`. Mostly obsoleted by type `string`.

```
strcpy(dst, src);         // Copy src to dst. Not bounds checked
strcat(dst, src);         // Concatenate to dst. Not bounds checked
strcmp(s1, s2);           // Compare, <0 if s1<s2, 0 if s1==s2, >0 if s1>s2
strncpy(dst, src, n);     // Copy up to n chars, also strncat(), strncmp()
strlen(s);                // Length of s not counting \0
strchr(s,c); strrchr(s,c);// Address of first/last char c in s or 0
strstr(s, sub);           // Address of first substring in s or 0
  // mem... functions are for any pointer types (void*), length n bytes
memcpy(dst, src, n);      // Copy n bytes from src to dst
memmove(dst, src, n);     // Same, but works correctly if dst overlaps src
memcmp(s1, s2, n);        // Compare n bytes as in strcmp
memchr(s, c, n);          // Find first byte c in s, return address or 0
```

```
  memset(s, c, n);          // Set n bytes of s to c
```

<cstdio>

The `stdio` library is made mostly obsolete by the newer `iostream` library, but many programs still use it. There are facilities for random access files and greater control over output format, error handling, and temporary files. Mixing both I/O libraries is not recommended. There are no facilities for string I/O.

Global objects `stdin, stdout, stderr` of type `FILE*` correspond to `cin, cout, cerr`. s is type char*, c is char, n is int, f is FILE*.

```
  FILE* f=fopen("filename", "r");  // Open for reading, NULL (0) if error
    // Mode may also be "w" (write) "a" append, "a+" random access read/append,
    // "rb", "wb", "ab", "a+b" are binary
  fclose(f);                  // Close file f
  fprintf(f, "x=%d", 3);      // Print "x=3"  Other conversions:
    "%5d %u %-8ld"              // int width 5, unsigned int, long left justified
    "%o %x %X %lx"              // octal, hex, HEX, long hex
    "%f %5.1f"                  // double: 123.000000, 123.0
    "%e %g"                     // 1.23e2, use either f or g
    "%c %s"                     // char, char*
    "%%"                        // %
  sprintf(s, "x=%d", 3);      // Print to array of char s
  printf("x=%d", 3);          // Print to stdout (screen unless redirected)
  fprintf(stderr, ...         // Print to standard error (not redirected)
  getc(f);                    // Read one char (as an int, 0-255) or EOF (-1) from f
  ungetc(c, f);               // Put back one c to f
  getchar();                  // getc(stdin);
  putc(c, f)                  // fprintf(f, "%c", c);
  putchar(c);                 // putc(c, stdout);
  fgets(s, n, f);             // Read line including '\n' into char s[n] from f.  NULL if EOF
  gets(s)                     // fgets(s, INT_MAX, f); no '\n' or bounds check
  fread(s, n, 1, f);          // Read n bytes from f to s, return number read
  fwrite(s, n, 1, f);         // Write n bytes of s to f, return number written
  fflush(f);                  // Force buffered writes to f
  fseek(f, n, SEEK_SET);      // Position binary file f at n
    // or SEEK_CUR from current position, or SEEK_END from end
  ftell(f);                   // Position in f, -1L if error
  rewind(f);                  // fseek(f, 0L, SEEK_SET); clearerr(f);
  feof(f);                    // Is f at end of file?
  ferror(f);                  // Error in f?
  perror(s);                  // Print char* s and last I/O error message to stderr
  clearerr(f);                // Clear error code for f
  remove("filename");         // Delete file, return 0 if OK
  rename("old", "new");       // Rename file, return 0 if OK
  f = tmpfile();              // Create temporary file in mode "wb+"
  tmpnam(s);                  // Put a unique file name in char s[L_tmpnam]
```

Example: input file name and print its size

```
  char filename[100];               // Cannot be a string
  printf("Enter filename\n");       // Prompt
  gets(filename, 100, stdin);       // Read line ending in "\n\0"
  filename[strlen(filename)-1]=0;   // Chop off '\n';
  FILE* f=fopen(filename, "rb");    // Open for reading in binary mode
  if (f) {                           // Open OK?
    fseek(f, 0, SEEK_END);          // Position at end
    long n=ftell(f);                // Get position
    printf("%s has %ld bytes\n", filename, n);
    fclose(f);                      // Or would close when program ends
  }
  else
    perror(filename);               // fprintf(stderr, "%s: not found\n", filename);
                                    // or permission denied, etc.
```

`printf(), fprintf()`, and `sprintf()` accept a variable number of arguments, one for each "%" in the format string, which must be the appropriate type. The compiler does not check for this.

```
  printf("%d %f %s", 2, 2.0, "2");  // OK
  printf("%s", 5);  // Crash: expected a char* arg, read from address 5
  printf("%s");     // Crash
  printf("%s", string("hi"));  // Crash: use "hi" or string("hi").c_str()
```

<cassert>

Provides a debugging function for testing conditions where all instances can be turned on or off at once. `assert(false);` prints the asserted expression, source code file name, and line number, then aborts. Compiling with **`g++ -DNDEBUG`** effectively removes these statements.

```
assert(e);                // If e is false, print message and abort
#define NDEBUG            // (before #include <assert.h>), turn off assert
```

Classes

Classes provide data abstraction, the ability to create new types and hide their implementation in order to improve maintainability. A `class` is a data structure and an associated set of *member functions* (methods) and related type declarations which can be associated with the class or instances (objects) of the class. A class is divided into a `public` interface, visible wherever the class or its instances are visible, and a `private` implementation visible only to member functions of the class.

```
class T {          // Create a new type T
private:           // Members are visible only to member functions of T (default)
public:            // Members are visible wherever T is visible
  // Type, object, and function declarations
};
T::m;              // Member m of type T
T x;               // Create object x of type T
x.m;               // Member m of object x
T* p=&x; p->m;     // Member m of object pointed to by p
```

Typically the data structure is private, and functionality is provided by member functions. Member function definitions should be separated from the declaration and written outside the class definition, or else they are assumed to be inline (which is appropriate for short functions). A member function should be declared const (before the opening brace) if it does not modify any data members. Only const member functions may be called on const objects.

```
class Complex {    // Represents imaginary numbers
private:
  double re, im;   // Data members, represents re + im * sqrt(-1)
public:
  void set(double r, double i) {re=r; im=i;}  // Inlined member function definition
  double real() const {return re;}            // const - does not modify data members
  double imag() const;                        // Declaration for non-inlined function
};
double Complex::imag() const {return im;}     // Definition for imag()
int main() {
  Complex a, b=a;                 // Objects of type Complex
  a.set(3, 4);                    // Call a member function
  b=a;                            // Assign b.re=a.re; b.im=a.im
  b==a;                           // Error, == is not defined
  cout << a.re;                   // Error, re is private
  cout << a.real();               // OK, 3
  cout << Complex().real();       // OK, prints an undefined value
  Complex().set(5, 6);            // Error, non-const member called on const object
```

A class has two special member functions, a *constructor*, which is called when the object is created, and a *destructor*, called when destroyed. The constructor is named *`class`*`::`*`class`*, has no return type or value, may be overloaded and have default arguments, and is never const. It is followed by an optional initialization list listing each data member and its initial value. Initialization takes place before the constructor code is executed. Initialization might not be in the order listed. Members not listed are default-initialized by calling their constructors with default arguments. If no constructor is written, the compiler provides one which default-initializes all members. The syntax is:

```
class::class(parameter list): member(value), member(value) { code...}
```

The destructor is named *`class`*`::~`*`class`*, has no return type or value, no parameters, and is never const. It is usually not needed except to return shared resources by closing files or deleting memory. After the code executes, the data members are destroyed using their respective destructors in the reverse order in which they were constructed.

```
class Complex {
public:
  Complex(double r=0, double i=0): re(r), im(i) {}  // Constructor
  ~Complex() {}                                     // Destructor
  // Other members...
};
Complex a(1,2), b(3), c=4, d;  // (1,2) (3,0) (4,0) (0,0)
```

A constructor defines a conversion function for creating temporary objects. A constructor that allows 1 argument allows implicit conversion wherever needed, such as in expressions, parameter passing, assignment, and initialization.

```
Complex(3, 4).real();  // 3
a = 5;                 // Implicit a = Complex(5) or a = Complex(5, 0)

void assign_if(bool, Complex&, const Complex&);
assign_if(true, a, 6); // Implicit Complex(6) passed to third parameter
assign_if(true, 6, a); // Error, non-const reference to Complex(6), which is const
```

Operators may be overloaded as members. The expression `aXb` for operator X can match either `operator X(a, b)` (global) or `a.operator X(b)` (member function), but not both. Unary operators omit b. Operators =, [], and -> can only be overloaded as member functions.

```
class Complex {
public:
  Complex operator + (const Complex& b) const { // const because a+b doesn't change a
    return Complex(re+b.re, im+b.im);
  }
  // ...
};

Complex operator - (const Complex& a, const Complex& b) {
  return Complex(a.real()-b.real(), a.imag()-b.imag());
}

Complex a(1, 2), b(3, 4);
a+b;                    // OK, a.operator+(b) == Complex(4, 6)
a-b;                    // OK, operator-(a, b) == Complex(-2, -2)
a+10;                   // OK, Complex(1, 12), implicit a+Complex(10, 0)
10+a;                   // Error, 10 has no member operator+(Complex)
a-10;                   // OK, Complex(1, -8)
10-a;                   // OK, Complex(7, -4)
```

The member function (+) has the advantage of private access (including to other objects of the same class), but can only do implicit conversions on the right side. The global function (-) is symmetrical, but lacks private access.
A `friend` declaration (in either the private or public section) allows private access to a global function.

```
class Complex {
  friend Complex operator-(const Complex&, const Complex&);
  friend class T;       // All member functions of class T are friends
  // ...
};
```

A conversion operator allows implicit conversion to another type. It has the form of a member function named `operator T() const` with implied return type T. It is generally a good idea to allow implicit conversions in only one direction, preferably with constructors, so this member function is usually used to convert to pre-existing types.

```
class Complex {
public:
  operator double() const {return re;}
  // ...
}

Complex a(1, 2);
a-10;                  // Error, double(a)-10 or a-Complex(10) ?
a-Complex(10);         // Complex(-9, 2);
double(a)-10;          // -9
```

An `explicit` constructor does not allow implicit conversions.

```
class Complex {
```

```
  explicit Complex(double r=0, double i=0);
  // ...
};

Complex a=1;           // Error
Complex a(1);          // OK
a-10;                  // OK, double(a)-10 = -9
a-Complex(10);         // OK, Complex(-9, 0)
```

A class or member function may be **templated**. The type parameter must be passed in the declaration for objects of the class.

```
template <class T>
class Complex {
  T re, im;
public:
  T real() const {return re;}
  T imag() const {return im;}
  Complex(T r=0, T i=0);
  friend Complex<T> operator - (const Complex<T>&, const Complex<T>&);
};

template <class T>
Complex<T>::Complex(T r, T i): re(r), im(i) {}

Complex<int> a(1, 2);                // Complex of int
Complex<double> b(1.0, 2.0);         // Complex of double
a=a-Complex<int>(3, 4);              // Complex<int>(-2, -2)
Complex<Complex<double> > c(b, b);   // Note space, not >>
c.real().imag();                     // 2.0
```

Templates can have default arguments and int parameters. The argument to an int parameter must be a value known at compile time.

```
template <class T, class U=T, int n=0> class V {};
V<double, string, 3> v1;
V<char> v2;  // V<char, char, 0>
```

Classes define default behavior for copying and assignment, which is to copy/assign each data member. This behavior can be overridden by writing a *copy constructor* and `operator=` as members, both taking arguments of the same type, passed by const reference. They are usually required in classes that have destructors, such as the `vector<T>`-like class below. If we did not overload these, the default behavior would be to copy the data pointer, resulting in two Vectors pointing into the same array. The assignment operator normally returns itself (`*this`) by reference to allow expressions of the form `a=b=c;`, but is not required to do so. `this` means the address of the current object; thus any member `m` may also be called `this->m` within a member function.

```
template <class T>
class Vector {
private:
  T* data;  // Array of n elements
  int n;    // size()
public:
  typedef T* iterator;                       // Vector::iterator means T*
  typedef const T* const_iterator;           // Iterators for const Vector
  int size() const {return n;}               // Number of elements
  T& operator[](int i) {return data[i];}     // i'th element
  const T& operator[](int i) const {return data[i];}  // i'th element of const Vector
  iterator begin() {return data;}            // First, last+1 elements
  iterator end() {return data+size();}
  const_iterator begin() const {return data;}  // Const versions
  const_iterator end() const {return data+size();}
  Vector(int i=0): data(new T[i]), n(i) {}  // Create with size i
  ~Vector() {delete[] data;}                 // Return memory
  Vector(const Vector<T>& v): data(new T[v.n]), n(v.n) {  // Copy constructor
    copy(v.begin(), v.end(), begin());
  }
  Vector& operator=(const Vector& v) {       // Assignment
    if (&v != this) {                        // Assignment to self?
      delete[] data;                         // If not, resize and copy
      data=new T[n=v.n];
      copy(v.begin(), v.end(), begin());
    }
    return *this;                            // Allow a=b=c;
  }
```

```
  template <class P> Vector(P b, P e): data(new T[e-b]), n(e-b) {  // Templated member
    copy(b, e, data);                        // Initialize from sequence [b, e)
  }
};
```

A type defined in a class is accessed through *class*`::type`

```
Vector<int>::iterator p;        // Type is int*
Vector<int>::const_iterator cp; // Type is const int*
```

Member functions may be overloaded on `const`. Overloaded member functions need not have the same return types. `const` member functions should not return non-const references or pointers to data members.

```
Vector<int> v(10);              // Uses non-const [], begin(), end()
const Vector<int> cv(10);       // Uses const [], begin(), end()
cv=v;                           // Error, non-const operator= called on cv
v[5]=cv[5];                     // OK. assigns to int&
cv[5]=v[5];                     // Error, assigns to const int&
p=cv.begin();                   // Error, would allow *p=x to write into cv
cp=cv.begin();                  // OK because can't assign to *cp
```

Defining Iterators. Sometimes a container's iterator types must be defined as nested classes overloading the usual pointer operations rather than typedef'ed to pointers. In order to work properly with functions defined in <algorithm>, iterators should define the following 5 public typedefs:

- `iterator_category`: one of the following (defined in `<iterator>`):
  - `output_iterator_tag` (if sequential writing is supported)
  - `input_iterator_tag` (if sequential reading is supported)
  - `forward_iterator_tag` (if both are supported)
  - `bidirectional_iterator_tag` (if the iterator can be decremented)
  - `random_access_iterator_tag` (if all pointer operations are supported)
- `value_type`: the type of the elements, for example, `T`
- `difference_type`: the result of iterator subtraction, usually `ptrdiff_t` (a signed int type)
- `pointer`: the type returned by `operator->()`, usually `T*` or `const T*`
- `reference`: the type returned by `operator*()`, usually `T&` or `const T&`

Operator -> should be overloaded as a unary function returning a pointer to a class to which -> will be applied, i.e. `x->m` is interpreted as `x.operator->()->m`. Nested class members are named Outer::Inner::member. Outer and inner classes cannot access each other's private members. Templated members defined outside the class need their own template declarations.

```
template <class T> class Vector {
public:

  // Reverse iterator for Vector, i.e. ++p goes to the previous element.
  class reverse_iterator {
  private:
    T* p;                                // Points to current element
  public:

    // typedefs needed to work with <algorithm> functions
    typedef std::random_access_iterator_tag iterator_category;  // Defined in <iterator>
    typedef T value_type;                // Type of element
    typedef ptrdiff_t difference_type;   // Result of iterator subtraction, usually int
    typedef T* pointer;                  // Type returned by operator ->
    typedef T& reference;                // Type returned by operator *

    reverse_iterator(T* a=0): p(a) {}    // Implicit conversion from T* and iterator
    iterator base() const {return p;}    // Convert to normal iterator

    // Forward operators
    reverse_iterator& operator++() {--p; return *this;} // prefix
    reverse_iterator  operator++(int);                  // postfix, we pretend it's binary
    reference operator*() const {return *p;}
    pointer operator->() const {return p;}              // We pretend it's unary
    bool operator==(Vector<T>::reverse_iterator b) const {return p==b.p;}
    bool operator!=(Vector<T>::reverse_iterator b) const {return p!=b.p;}
    // Also, bidirectional and random operators
```

```
  };
  reverse_iterator rbegin() {return end()-1;}
  reverse_iterator rend() {return begin()-1;}
  // Other members...
};

// Code for postfix ++
template <class T>
inline Vector<T>::reverse_iterator Vector::reverse_iterator::operator++(int dummy) {
  Vector<T>::reverse_iterator result = *this;
  ++*this;
  return result;
};

// Print a Vector in reverse order
int main() {
  Vector<int> a(10);
  for (Vector<int>::reverse_iterator p=a.rbegin(); p!=a.rend(); ++p)
    cout << *p << endl;
```

`vector<T>` supplies random `reverse_iterator` and `const_reverse_iterator` as above. Const iterators would typedef `pointer` as `const T*` and `reference` as `const T&`.

A `static` data member is shared by all instances of a class. It must be initialized in a separate declaration, not in the class definition or in the constructor initialization list. A `static` member function cannot refer to `this` or any non-static members (and therefore it makes no sense to make them `const`). Static members may be referenced either as *`object.member`* or *`class`*`::`*`member`*.

```
class Counter {
  static int count;  // Number of Counters that currently exist (private)
public:
  static int get() {return count;}
  Counter() {++count;}
  ~Counter() {--count;}
  Counter(const Counter& c) {++count;}  // Default would be wrong
  Counter& operator=(const Counter& c) {return *this;}  // Default would be OK
};
int Counter::count = 0;  // Initialize here, OK if private
main() {
  Counter a, b, c;
  cout << b.get();          // 3
  cout << Counter::get();  // 3
}
```

Inheritance

Inheritance is used to write a specialized or enhanced version of another class. For example, an `ofstream` is a type of `ostream`. `class` *`D`*`: public` *`B`* defines class D as *derived* from (subclass of) *base*class (superclass) B, meaning that D *inherits* all of B's members, except the constructors, destructor, and assignment operator. The default behavior of these special member functions is to treat the base class as a data member.

```
class String: public Vector<char> {
public:
  String(const char* s=""): Vector<char>(strlen(s)) {
    copy(s, s+strlen(s), begin());  // Inherits Vector<char>::begin()
  }
};
String a="hello"; // Calls Vector<char>::Vector(5);
a.size();         // 5, inherits Vector<char>::size()
a[0]='j';         // "jello", inherits Vector<char>::operator[]
String b=a;       // Default copy constructor uses Vector's copy constructor on base part
b=a;              // Default = calls Vector's assignment operator on base part
```

The default destructor `String::~String() {}` is correct, since in the process of destroying a String, the base is also destroyed, calling `Vector<char>::~Vector() {delete data[];}`. Since there is no need to write a destructor, there is no need to redefine copying or assignment either.

Although `String` inherits `Vector<char>::data`, it is private and inaccessible. A `protected` member is accessible to derived classes but private elsewhere.

```
class B {
protected:
  int x;
} b;                            // Declare class B and object b
b.x=1;                          // Error, x is protected

class D: public B {
  void f() {x=1;}               // OK
};
```

By default, a base class is private, making all inherited members private. Private base classes are rare and typically used as implementations rather than specializations (A string is a vector, but a stack is not).

```
class Stack: Vector<int> {  // or class Stack: private Vector<int>
public:
  bool empty() const {return size()==0;}  // OK
} s;
s.size();   // Error, private
s.empty();  // OK, public
```

A class may have more than one base class (called *multiple inheritance*). If both bases are in turn derived from a third base, then we derive from this root class using `virtual` to avoid inheriting its members twice further on. Any indirectly derived class treats the virtual root as a direct base class in the constructor initialization list.

```
class ios {...};                                        // good(), binary, ...
class fstreambase: public virtual ios {...};            // open(), close(), ...
class istream: public virtual ios {...};                // get(), operator>>(), ...
class ifstream: public fstreambase, public istream {  // Only 1 copy of ios
  ifstream(): fstreambase(), istream(), ios() {...}   // Normally ios() would be omitted
};
```

Polymorphism

Polymorphism is the technique of defining a common interface for a hierarchy of classes. To support this, a derived object is allowed wherever a base object is expected. For example,

```
String s="Hello";
Vector<char> v=s;    // Discards derived part of s to convert
Vector<char>* p=&s;  // p points to base part of s
try {throw s;} catch(Vector<char> x) {}  // Caught with x set to base part of s
s=Vector<char>(5);   // Error, can't convert base to derived

// Allow output of Vector<char> using normal notation
ostream& operator << (ostream& out, const Vector<char>& v) {
  copy(v.begin(), v.end(), ostream_iterator<char>(out, ""));  // Print v to out
  return out;        // To allow (cout << a) << b;
}
cout << s;           // OK, v refers to base part of s
ofstream f("file.txt");
f << s;              // OK, ofstream is derived from ostream
```

A derived class may redefine inherited member functions, overriding any function with the same name, parameters, return type, and const-ness (and hiding other functions with the same name, thus the overriding function should not be overloaded). The function call is resolved at compile time. This is incorrect in case of a base pointer or reference to a derived object. To allow run time resolution, the base member function should be declared `virtual`. Since the default destructor is not virtual, a virtual destructor should be added to the base class. If empty, no copy constructor or assignment operator is required. Constructors and = are never virtual.

```
class Shape {
public:
  virtual void draw() const;
  virtual ~Shape() {}
};
class Circle: public Shape {
public:
  void draw() const;      // Must use same parameters, return type, and const
};

Shape s; s.draw();        // Shape::draw()
Circle c; c.draw();       // Circle::draw()
Shape& r=c; r.draw();     // Circle::draw() if virtual
```

```
Shape* p=&c; p->draw();   // Circle::draw() if virtual
p=new Circle; p->draw();  // Circle::draw() if virtual
delete p;                 // Circle::~Circle() if virtual
```

An *abstract* base class defines an interface for one or more derived classes, which are allowed to instantiate objects. Abstractness can be enforced by using protected (not private) constructors or using *pure virtual* member functions, which must be overridden in the derived class or else that class is abstract too. A pure virtual member function is declared `=0;` and has no code defined.

```
class Shape {
protected:
  Shape();                  // Optional, but default would be public
public:
  virtual void draw() const = 0; // Pure virtual, no definition
  virtual ~Shape() {}
};
// Circle as before

Shape s;                    // Error, protected constructor, no Shape::draw()
Circle c;                   // OK
Shape& r=c; r.draw();       // OK, Circle::draw()
Shape* p=new Circle();      // OK
```

Run time type identification

C++ provides for run time type identification, although this usually indicates a poor design. `dynamic_cast<T>`(x) checks at run time whether a base pointer or reference is to a derived object, and if so, does a conversion. The base class must have at least one virtual function to use run time type checking.

```
#include <typeinfo>    // For typeid()
typeid(*p)==typeid(T)  // true if p points to a T
dynamic_cast<T*>(p)    // Convert base pointer to derived T* or 0.
dynamic_cast<T&>(r)    // Convert base reference to derived T& or throw bad_cast()
```

For example,

```
class B {public: virtual void f(){}};
class D: public B {public: int x;} d;  // Bad design, public member in D but not B
B* p=&d; p->x;                         // Error, no B::x
D* q=p; q->x;                          // Error, can't convert B* to D*
q=(D*)p;  q->x;                        // OK, but reinterpret_cast, no run time check
q=dynamic_cast<D*>(p); if (q) q->x;    // OK
```

Other Types

`typedef` defines a synonym for a type.

```
typedef char* Str;  // Str is a synonym for char*
Str a, b[5], *c;    // char* a; char* b[5]; char** c;
char* d=a;          // OK, really the same type
```

`enum` defines a type and a set of symbolic values for it. There is an implicit conversion to int and explicit conversion from int to enum. You can specify the int equivalents of the symbolic names, or they default to successive values beginning with 0. Enums may be anonymous, specifying the set of symbols and possibly objects without giving the type a name.

```
enum Weekday {MON,TUE=1,WED,THU,FRI};  // Type declaration
enum Weekday today=WED;                // Object declaration, has value 2
today==2                               // true, implicit int(today)
today=Weekday(3);                      // THU, conversion must be explicit
enum {N=10};                           // Anonymous enum, only defines N
int a[N];                              // OK, N is known at compile time
enum {SAT,SUN} weekend=SAT;            // Object of anonymous type
```

A **`struct`** is a class where the default protection is public instead of private. A `struct` can be initialized like an array.

```
struct Complex {double re, im;};     // Declare type
```

```
Complex a, b={1,2}, *p=&b;           // Declare objects
a.re = p->im;                         // Access members
```

A **`union`** is a struct whose fields overlap in memory. Unions can also be anonymous. They may be used to implement variant records.

```
union U {int i; double d;};  // sizeof(U) is larger of int or double
U u; u.i=3;                  // overwrites u.d

// A variant record
class Token {
  enum {INT, DOUBLE} type;   // which field is in use?
  union {int i; double d;} value;  // An anonymous union
public:
  void print() const {
    if (type==INT) cout << value.i;
    else cout << value.d;
  }
};
```

An enum, struct, class, or union type and a list of objects may be declared together in a single statement.

```
class Complex {public: double re, im;} a, b={1,2}, *p=&b;
```

Program Organization

For C++ programs that only use one source code file and the standard library, the only rule is to declare things before using them: type declarations before object declarations, and function declarations or definitions before calling them. However, implicitly inlined member functions may use members not yet declared, and templates may use names as long as they are declared before instantiation.

```
class Complex {
  double real() const {return re;}  // OK
  double re, im;
};
```

Global and member functions (unless inlined or templated) and global or class static objects are separately compilable units, and may appear in separate source code (.cpp) files. If they are defined and used in different files, then a declaration is needed. To insure that the declaration and definition are consistent, the declaration should be in a shared header file. A shared header conventionally has a `.h` extension, and is inserted with a `#include "`*`filename`*`.h"`, using double quotes to indicate that the file is in the current directory. Global variables are declared with `extern` without initialization.

```
// prog.h          // prog1.cpp         // prog2.cpp
extern int x;      #include "prog.h"   #include "prog.h"
int f();           int x=0;            int f() {
                   int main() {          return x;
                     f();              }
                     return 0;
                   }
```

To compile,

```
g++ prog1.cpp prog2.cpp -o prog
```

This produces two object files (`prog1.o, prog2.o`), and then links them to produce the executable `prog`. g++ also accepts `.o` files, which are linked only, saving time if the `.cpp` file was not changed. To compile without linking, use `-c`. To optimize (compile slower but run faster), use `-O`.

The UNIX `make` command updates the executable as needed based on the timestamps of source and `.o` files. It requires a file named `Makefile` containing a set of dependencies of the form:

```
file: files which should be older than file
(tab) commands to update file
```

Dependencies may be in any order. The Makefile is executed repeatedly until all dependencies are satisfied.

```
# Makefile comment
prog: prog1.o prog2.o
      g++ prog1.o prog2.o -o prog

prog1.o: prog1.cpp prog.h
      g++ -c prog1.cpp

prog2.o: prog2.cpp prog.h
      g++ -c prog2.cpp
```

Compiler options for g++. Other compilers may vary.

```
g++ file1.cpp              Compile, produce executable a.out in UNIX
g++ file1.cpp file2.o      Compile .cpp and link .o to executable a.out
g++ -Wall                  Turn on all warnings
g++ -c file1.cpp           Compile to file1.o, do not link
g++ -o file1               Rename a.out to file1
g++ -O                     Optimize executable for speed
g++ -v                     Verbose mode
g++ -DX=Y                  Equivalent to #define X Y
g++ --help                 Show all g++ options
gxx file1.cpp              Compile in Windows MS-DOS box (DJGPP) to A.EXE
```

Anything which is not a separately compilable unit may appear in a header file, such as class definitions (but not function code unless inlined), templated classes (including function code), templated functions, and other `#include` statements.

Creating Libraries (namespaces)

Libraries usually come in the form of a header and an object (`.o`) file. To use them, `#include "header.h"` and link the `.o` file using g++. If the `.o` was compiled in C rather than C++, then indicate this with `extern "C" {}` to turn off name mangling. C++ encodes or "mangles" overloaded function names to allow them to be linked, but C does not since it doesn't allow overloading.

```
extern "C" {          // Turn off name mangling
#include "header.h"   // Written in C
}
```

When writing your own library, use a unique namespace name to prevent conflicts with other libraries. A namespace may span multiple files. Types, objects, and functions declared in a namespace N must be prefixed with N:: when used outside the namespace, or there must be a `using namespace N;` in the current scope.

Also, to guard against possible multiple inclusions of the header file, `#define` some symbol and test for it with #ifndef ... #endif on the first and last lines. Don't have a `using namespace std;`, since the user may not want `std` visible.

```
#ifndef MYLIB_H       // mylib.h, or use #if !defined(MYLIB_H)
#define MYLIB_H
#include <string>
// No using statement
namespace mylib {
  class B {
  public:
    std::string f();  // No code
  }
}
#endif

// mylib.cpp, becomes mylib.o
#include <string>
#include "mylib.h"
using namespace std;  // OK
namespace mylib {
  string B::f() {return "hi";}
}
```

`#define` could be used to create constants through text substitution, but it is better to use `const` to allow type checking. `#define X Y` has the effect of replacing symbol X with arbitrary text Y before compiling, equivalent to the

g++ option `-DX=Y`. Each compiler usually defines a different set of symbols, which can be tested with `#if, #ifdef, #ifndef, #elsif, #else`, and `#endif`.

```
#ifdef unix  // Defined by most UNIX compilers
// ...
#else
// ...
#endif
```

Preprocessor statements are one line (no semicolon). They perform text substitutions in the source code prior to compiling.

```
#include <header>          // Standard header
#include "header.h"        // Include header file from current directory
#define X Y                // Replace X with Y in source code
#define f(a,b) a##b        // Replace f(1,2) with 12
#define X \                // Continue a # statement on next line
#ifdef X                   // True if X is #defined
#ifndef X                  // False if X is #defined
#if !defined(X)            // Same
#else                      // Optional after #if...
#endif                     // Required
```

History of C++

C++ evolved from C, which in turn evolved from B, written by Ken Thompson in 1970 as a variant of BCPL. C was developed in the 1970's by Brian Kernighan and Dennis Ritchie as a "portable assembly language" to develop UNIX. C became widely available when they published "The C Programming Language" in 1983. C lacked standard containers (string, vector, map), iostreams, bool, const, references, classes, exceptions, namespaces, new/delete, function and operator overloading, and object-oriented capabilities. I/O was done using `<stdio.h>`. Strings were implemented as fixed sized char[] arrays requiring functions to assign or compare them (strcpy(), strcmp()). Structs could not be assigned, and had to be copied using memcpy(). Function arguments were not type checked. Functions could only modify arguments by passing their addresses. Memory allocation was done using malloc(), which requires the number of bytes to allocate and returns an untyped pointer or NULL if it fails. The language allowed unsafe implicit conversions such as int to pointers. Variables had to be declared before the first statement. There was no `inline`, so macros were often used in place of small functions. Hardware was slow and optimizers were not very good, so it was common to declare `register` variables. There were no // style comments. For instance,

```
/* Copy argv[1] to buf and print it */
#include <stdio.h>         /* No cout, use printf()                              */
#include <string.h>        /* No string type, use char*                          */
#include <stdlib.h>        /* No new/delete, use malloc/free                    */
main(argc, argv)           /* Return type defaults to int */
int argc;                  /* Old style parameter declaration, no type checking */
char** argv;
{                          /* No namespace std                                   */
  char* buf;               /* All declarations before the first statement        */
  if (argc>1) {
    buf=(char*)malloc((strlen(argv[1])+1)*sizeof(char));  /* Cast optional      */
    strcpy(buf, argv[1]);  /* Can't assign, no range check                       */
    printf("%s\n", buf);   /* Arguments not type checked                         */
    free(buf);             /* No delete */
  }
}                          /* Return value is undefined (unchecked)             */
```

The ANSI C standard was finished in 1988. It added `const`, new style function declaration with type checking, `struct` assignment, strict type checking of pointer assignments, and specified the standard C library, which until now was widely used but with minor, annoying variations. However, many compilers did not become ANSI compliant until the early 1990's.

In the 1980's Bjarne Stroustrup at AT&T developed "C with Classes", later C++. Early implementations were available for UNIX as `cfront` (cc), a C++ to C translator around 1990. It added object oriented programming with classes, inheritance, and polymorphism, also references, the iostream library, and minor enhancements such as // style comments and the ability to declare variables anywhere. Because there were no namespaces, the iostream header was

`using`

named <iostream.h> and no statement was required. Unlike C programs which always have a .c extension, C++ didn't say, so .cpp, .cc and .C were all common, and .hpp for headers.

GNU gcc and g++, which compiled C and C++ directly to machine code, were developed in the early 1990's. Templates were added in 1993. Exceptions were added in 1994. The standard container library (originally called the standard template library or STL) was developed by researchers at Hewlett-Packard and made available free as a separate download in the mid 1990's and ported to several compilers.

ANSI standard C++ compilers became available in 1998. This added STL to the standard library, added multiple inheritance, namespaces, type bool, and run time type checking (dynamic_cast, typeid). The `.h` extension on headers was dropped.

C++ most likely succeeded where other early object oriented languages failed (Simula67, Actor, Eiffel, SmallTalk) because it was backwards compatible with C, allowing old code to be used, and because C programmers could use it immediately without learning the new features. However, there are a few incompatibilities.

- Old style function declarations are not allowed.
- Conversion from `void*` (returned by `malloc()`) requires a cast.
- There are many new reserved words.

There are also some incompatibilities between old (before 1998) and new versions of C++.

- `new` was changed to throw type `bad_alloc` if out of memory, instead of returning 0.
- The scope of a variable declared in a `for` loop was changed to be local to the loop and not beyond it (not yet implemented by Microsoft Visual C++)

g++ does not yet implement all ANSI C++ features. For instance,

- Type `ostringstream` allowing formatted writing to strings.
- Run time bounds checking of vector indexes using v.at(i)

The largest integer type is 32 bits in most implementations, but as 64 bit machines become common it is possible that type `long` could become a 64 bit type (as in Java) in the future. g++ supports the nonstandard 64-bit integer type `long long`, e.g.

```
unsigned long long bigzero=0LLU;
```

Most implementations of `time()` return the number of seconds since Jan. 1, 1970 as a `time_t`, normally a signed 32-bit `long`. Programs that use this implementation will fail on Jan. 19, 2038 at 3:14:08 AM as this value overflows and becomes negative.