## C++

C++ is a language developed by Bjarne Stroustrup and many others at Bell Labs from the language C.

The major concepts in C++ are [encapsulation](http://www.cs.nmsu.edu/~rth/cs/cs177/map/encap.html), [inheritance](http://www.cs.nmsu.edu/~rth/cs/cs177/map/inherit.html), and [polymorphism](http://www.cs.nmsu.edu/~rth/cs/cs177/map/poly.html).

Programming style can follow [abstract data types](http://www.cs.nmsu.edu/~rth/cs/cs177/map/empty.html), [structured programming](http://www.cs.nmsu.edu/~rth/cs/cs177/map/structprog.html), and [object-oriented design](http://www.cs.nmsu.edu/~rth/cs/cs177/map/empty.html), although C++ can also support familiar programming styles since it contains (almost) the whole of [ANSI C](http://www.cs.nmsu.edu/~rth/cs/cs177/map/ansic.html).

## Default arguments

The parameter list of a [function signature](http://www.cs.nmsu.edu/~rth/cs/cs177/map/signat.html) can specify a default value associated with a parameter. Only the last *n* consecutive parameters can have a default value, the first *n* consecutive parameters can have no default. In both cases, *n* can be zero. Defaults must be specified at the time of declaration of the function, or the function can be declared and defined simultaneously, as it is with [inline](http://www.cs.nmsu.edu/~rth/cs/cs177/map/empty.html)functions. An example is:

void df(int, float, int = 12, char = 'A');

This can be called in one of three ways:

df(10, 2.2); // defaults used for parameters three and four

df(10, 2.2, 32); // default only used for last one

df(10, 2.2, 32, 'B'); // both defaults are over-ridden

Default arguments are a kind of function [overloading](http://www.cs.nmsu.edu/~rth/cs/cs177/map/overload.html).

Defaults can only be specified once. If a function is declared and then defined later, the definition must *not* mention the defaults:

void f(int = 10);

...

void f(int x) {

...

}

**Access control**

Whereas scope is about the names declared and used in regions of program text between braces, access control is about the use of objects, and the functions that use them. Objects can control access to their internal stateby use of the keywords private, public and protected. By default, all class members are private; they cannot be accessed externally (see the use of structures, and the use of the friend declaration for variations on this). Using the section keyword *public* makes any names declared in that section accessible from outside. e.g. in

class Access {

int x1;

void f1();

public:

int x2;

void f2();

};

x1 and f1 are private, but x2 and f2 are public. There can be any number of sections, private or public, in a class declaration. It is conventional to make explicit use of the keyword private, even though default access is private. Thus:

class Access {

private:

int x1;

void f1();

public:

int x2;

void f2();

};

is preferred over the previous declaration.

The keyword protected is used only when inheritance between classes needs to be controlled, and allows derived classes to access private members of a base class directly

**ANSI C**

C++ is based, both historically and logically upon C. While it was being developed C was undergoing the grueling business of standardization, both by the American National Standards Institute and the International Standards Organization (ISO).

Although it was intended that all of C be included in C++ as subset, in fact the ANSI version of C took over some aspects of the C++ development, changing them slightly, and the end result is that ANSI C is not a subset of C++, but is close enough for most purposes.

Only programmers intent on maintaining C programs to compile as C++ programs as well need be concerned. The best way to approach the problem is to learn C++ as a separate langauge, whose look and feel takes a lot from C. This keeps the two separate, as they should be.

## Classes

The declaration of a class adds a new type to C++ type system. A class is an encapsulation of variable and function declarations, called data members and function members respectively. Variables can have any type, , but they must have unique names within the scope of the class. Functions can have the same name, even within the scope, but must have different signatures (the functions can be overloaded). All members are subject to access control; the default is private, but any member can be made public by using an access control keyword within the body of the class declaration. Function members may be simply declared as prototypes, or defined within the class as [inline](http://www.cs.nmsu.edu/~rth/cs/cs177/map/empty.html)functions. Function members may be defined [externally](http://www.cs.nmsu.edu/~rth/cs/cs177/map/funmembd.html). Special function members are [constructors](http://www.cs.nmsu.edu/~rth/cs/cs177/map/construc.html), and the [destructor](http://www.cs.nmsu.edu/~rth/cs/cs177/map/destruc.html). Here is an example class declaration showing all these points:

class Eg {

int i1; // a private data member of base type int

C \*c; // a private pointer to an object of type C

public:

Eg() { i1 = 0; c = 0; } // the (public) default

// constructor,defined inline

Eg(int ii) { i1 = i; } // an overloaded constructor

~Eg(); // the prototype for the

// destructor

void f1(int); // a prototype for a public

// function member

private:

void f1(int, int); // a prototype for a

// (private)overloaded function

int f2() { cout << "f2" << endl; } // an inline private

// function

public:

int i2; // a public data member

float f3(int, float); // a prototype for a public function

};

## Class scope

[Names](http://www.cs.nmsu.edu/~rth/cs/cs177/map/names.html)declared inside the braces of a class declaration are local to that class. They can be [data members](http://www.cs.nmsu.edu/~rth/cs/cs177/map/datamem.html) or [function members](http://www.cs.nmsu.edu/~rth/cs/cs177/map/funmemb.html). e.g. in

class ScopeExample {

int Variable;

void Function();

};

the names Variable and Function are local to the class declaration and can be re-used in another scope. The name ScopeExample, however cannot be re-used in the same scope since it is not between the braces.

## Constructor functions

Every [class](http://www.cs.nmsu.edu/~rth/cs/cs177/map/class.html)has at least one constructor function, even when none is declared. The job of a constructor functions is to allocate space for an [object](http://www.cs.nmsu.edu/~rth/cs/cs177/map/object.html), and to set its initial internal state by assigning values to some or all of its [data members](http://www.cs.nmsu.edu/~rth/cs/cs177/map/datamem.html).

There may be any number of different constructors, since [function members](http://www.cs.nmsu.edu/~rth/cs/cs177/map/funmemb.html) may be [overloaded](http://www.cs.nmsu.edu/~rth/cs/cs177/map/empty.html). However, all must have the same name as the class that contains them, and none may have a return type. The constructor which takes no arguments is the *default constructor*. If no constructor is declared at all, the compiler will create a standard default constructor that allocates space but does no initialization.

A special constructor is the copy constructor which takes one object of the same type as an argument through a reference parameter, and copies it to create a new one. Below are some examples:

class C {

private:

int x;

public:

C() { x = 0; } // the default constructor

C(int xx) { x = xx; } // an overloaded constructor

C(C &orig) { x = orig.x; } // the copy constructor

};

## Data Members

An object may contain values which are stored internally and are unique to that object. In order to do this, each value needs an appropriate declaration as a data member in the [class](http://www.cs.nmsu.edu/~rth/cs/cs177/map/class.html). A data member may be of any type, including classes already defined, pointers to objects of any type, or even [references](http://www.cs.nmsu.edu/~rth/cs/cs177/map/pointer.html)to objects of any type.

Data members may be [private](http://www.cs.nmsu.edu/~rth/cs/cs177/map/access.html)or [public](http://www.cs.nmsu.edu/~rth/cs/cs177/map/access.html), but are usually held private so that values may only be changed at the discretion of the class [function members](http://www.cs.nmsu.edu/~rth/cs/cs177/map/funmemb.html). In the example below, the class C contains two a private data member of type int, and a public data member of type pointer to char.

class C {

private:

int x;

public:

float f;

};

**Destructor functions**

The destructor function is a special function that a class uses to recover the space allocated to an [instance](http://www.cs.nmsu.edu/~rth/cs/cs177/map/object.html)of that class. The destructor is called whenever an object goes out of [scope](http://www.cs.nmsu.edu/~rth/cs/cs177/map/scope.html). Every class has a destructor function even if none is defined. In this case the equivalent definition is

~X() {}

Where X is the name of the class. Note there is no return type (just like the [constructor](http://www.cs.nmsu.edu/~rth/cs/cs177/map/construc.html)functions), there are no parameters and the tilde (~) must be present. Destructors should be [public](http://www.cs.nmsu.edu/~rth/cs/cs177/map/access.html)to be of any use.

There can only be one destructor for a class, but its body can be defined in any chosen fashion. Typically the destructor can use [delete](http://www.cs.nmsu.edu/~rth/cs/cs177/map/dynalloc.html)on pointers initialized with [new](http://www.cs.nmsu.edu/~rth/cs/cs177/map/dynalloc.html)to ensure that [dangling pointers](http://www.cs.nmsu.edu/~rth/cs/cs177/map/pointer.html) are not left.

For example, in the class below, the constructor creates a data member with new, and the destructor destroys it.

class Y { ... };

class X {

private:

Y\* mem1;

public:

X() { mem1 = new Y; }

~X() { delete mem1; }

};

Where a data member is of a (non-pointer) class type, or when a class is derived, its destructor will be called when the containing or deriving class destructor is called. So, in class A below, there is a member of type B whose destructor will be called when an object of type A is destroyed. Since B is derived from C, the destructor for C is called when B's destructor is called.

class C { ... };

class B : public C {

...

public:

~B() {} // when this is called, ~C() is called first

};

class A {

...

B b;

public:

~A() {} // when this is called, ~B() is called first for member b;

...

};

**Dynamic Allocation**

C++ allows the program to create and destroy objects at run-time, under program control. Two global operators, new and delete, make this happen.

The operator new takes a class name as argument, together with optional arguments that are passed to the appropriate class [constructor](http://www.cs.nmsu.edu/~rth/cs/cs177/map/construc.html), if available. It returns a [pointer](http://www.cs.nmsu.edu/~rth/cs/cs177/map/pointer.html)to the new [object](http://www.cs.nmsu.edu/~rth/cs/cs177/map/object.html), which is created on the heap. A variable of type pointer-to-class can hold this returned value. For example:

class X { ... };

X\* obj = new X;

In this case, the default constructor is called, since there are no arguments mentioned. However in:

class X {

...

public:

X(int x);

...

};

X\* obj = new X(32);

the constructor with a single integer parameter is used to initialize the new object.

Pointer variables follow the usual rules of scope, so it is possible for the variable to be deleted, leaving the object to which it points stranded (the "dangling pointer" problem). To avoid this, objects created with new should be deleted before this happens. For example:

class X { ... };

void f1() {

X\* x = new X;

...

delete x;

}

Without the delete operation, the variable x would disappear, leaving the object to which it points without a pointer. Of course, pointers can be returned from functions, so they can be used with delete at a later time.

The operator delete calls the [destructor](http://www.cs.nmsu.edu/~rth/cs/cs177/map/destruc.html)function for the class of the argument.

## Encapsulation

Variables in C++ are subject to two constraints. One is the rules of [scope](http://www.cs.nmsu.edu/~rth/cs/cs177/map/scope.html), and the other is [access control](http://www.cs.nmsu.edu/~rth/cs/cs177/map/access.html). Encapsulation allows the programmer to control both the scope of names, and access to functions and/or values stored inside an [object](http://www.cs.nmsu.edu/~rth/cs/cs177/map/object.html). The main construct is the [class](http://www.cs.nmsu.edu/~rth/cs/cs177/map/class.html), which allows both variables and functions to be declared within it. These names are subject to [visibility](http://www.cs.nmsu.edu/~rth/cs/cs177/map/visib.html) rules, and can be declared as having [private](http://www.cs.nmsu.edu/~rth/cs/cs177/map/access.html), or [public](http://www.cs.nmsu.edu/~rth/cs/cs177/map/access.html) access.

## File scope

Names declared outside any function or class have file scope, since their only boundary is the file in which the name occurs. The compiler can link the same name with file scope across files, or this can be prevented by using the [keyword](http://www.cs.nmsu.edu/~rth/cs/cs177/map/empty.html) [static](http://www.cs.nmsu.edu/~rth/cs/cs177/map/empty.html). These names are typically called global.

**Friend**

Classes can also control access to their private members by using the *friend*declaration. There are two variations on this:

1. Access can be granted to individual member functions of another class by using the following form:
2. friend *return type* X::f(*param types*...);

any number of functions can be granted access in this manner.

1. Access can be granted to *all*the functions of a class by:
2. friend class X;

The best way to think about friend functions is that although they are actually declared in another scope, the friend declaration "imports" them into the scope of the declaration, thus allowing direct access to private members.

Of course, since friend functions are *not* member functions access still has to be through an object of the class declaring the friend.

For example, the function FR in class B, below is granted access to the private member x of class A, but still has to use an object of type A to effect the access:

class A {

private:

int x;

...

friend void B::FR();

};

class B {

...

void FR() {

A a;

cout << a.x; // direct access allowed by friend declaration

}

...

};

## Function members

An object may contain functions which are stored internally and are unique to that object. Each such function needs an appropriate declaration, as a [prototype](http://www.cs.nmsu.edu/~rth/cs/cs177/map/signat.html), in the [class](http://www.cs.nmsu.edu/~rth/cs/cs177/map/class.html).

Function members can be ordinary functions, but they can be [overloaded](http://www.cs.nmsu.edu/~rth/cs/cs177/map/overload.html), have [default parameter values](http://www.cs.nmsu.edu/~rth/cs/cs177/map/empty.html), or be [virtual](http://www.cs.nmsu.edu/~rth/cs/cs177/map/empty.html).

Special function members are [constructors](http://www.cs.nmsu.edu/~rth/cs/cs177/map/construc.html) and the [destructor](http://www.cs.nmsu.edu/~rth/cs/cs177/map/destruc.html). In the example below, there are two (overloaded) functions called f1, one private and one public; two constructors, one private and one public, and one destructor;

class C {

private:

void f1(int);

C();

public:

void f1(int, int);

C(int);

~C();

};

[http://www.cs.nmsu.edu/~rth/cs/cs177/map/blob.gif](mailto:rth@cs.nmsu.edu)

Function members can be called using the same [member access operator](http://www.cs.nmsu.edu/~rth/cs/cs177/map/memaccess.html) as [data members](http://www.cs.nmsu.edu/~rth/cs/cs177/map/DATAMEM.html).

## Function member definition

If a function member is not defined as inline, then it must be defined outside the class. This done by prefixing the function name with the class name and the *scope-resolution operator*, :: (colon-colon). This qualifies the name to avoid ambiguity when function names are reused in different classes. This appears to break the rules of [scope](http://www.cs.nmsu.edu/~rth/cs/cs177/map/scope.html), because references to variables declared within the [scope of the class](http://www.cs.nmsu.edu/~rth/cs/cs177/map/clsscope.html) are allowed in function member definitions. The rules are merely being extended, however, if it is accepted that the definition of the function is "really" contained in the class, but is just placed externally for readability purposes. It also provides a declaration versus implementation view that [abstract data typing](http://www.cs.nmsu.edu/~rth/cs/cs177/map/adt.html) demands. An exmple of external definition is:

class Ext {

private:

int privateVariable;

public:

void extFun();

...

};

void Ext::extFun() {

privateVariable = 10; // reference to name in the class scope

}

## Function scope

Braces also define the scope of a function's body. Variable names declared locally can be contained anywhere between the braces. One minor break in the rules is that any parameters declared in the [function header](http://www.cs.nmsu.edu/~rth/cs/cs177/map/signat.html) are also considered to be inside the function's scope (whereas the function name itsef is not). e.g. in

int Function(int a, int b) {

int c = a + b;

}

all the names a, b and c are considered local to the function. The name Function is non-local.

## Inheritance

A [class](http://www.cs.nmsu.edu/~rth/cs/cs177/map/class.html) defines a new type that extends the type system of the language. In languages like C++, a type is a skeleton for an area of memory containing values that follow a particular format. For instance, an integer may be 32 bits of binary number in the 2s complement format; a float may be 32 bits of binary number in the IEEE floating-point format. However, a type may also be seen as a *model* for a type of [object](http://www.cs.nmsu.edu/~rth/cs/cs177/map/object.html) found in the real world.

Things in the real world are classified into groupings called natural types. Each natural type has a number of attributes, called properties that are possessed by all the objects of that type. For instance, the class of all cats is a type with many properties, such as warm-blooded, four-legged, hairy and whiskered. All cats have these properties, and the presence of these properties distinguishes cats from, for instance, humans, which are warm-blooded and hairy, but only have two legs, and no whiskers. When we represent such classes using C++, we could duplicate the common properties like this:

class human {

<warm-blooded>

<hairy>

<two-legged>

};

class cat {

<warm-blooded>

<hairy>

<four-legged>

<whiskered>

};

However, we can leave the common properties in the human class, and allow the cat class to inherit these properties, avoiding the duplication:

class cat : public human {

<four-legged>

<whiskered>

};

Of course, we should also stop the inheritance of the property four-legged, and we can do that by over-riding a property with another of the same name

## C++ Inheritance

Inheritance is a way of relating two classes so that one class may use another class's members without redefining them (another way is using the [friend](http://www.cs.nmsu.edu/~rth/cs/cs177/map/empty.htm) declaration). A class may be derived from a base class by using the inheritance syntax:

class base { ... };

class derived : base { ... };

In fact if this done as above, all the members of the class base become [private](http://www.cs.nmsu.edu/~rth/cs/cs177/map/access.html) in the class derived, so it is better to use [public](http://www.cs.nmsu.edu/~rth/cs/cs177/map/access.html) inheritance:

class base { ... };

class derived : public base { ... };

In this way, all the members, whether [data](http://www.cs.nmsu.edu/~rth/cs/cs177/map/datamem.html) or [functions](http://www.cs.nmsu.edu/~rth/cs/cs177/map/funmemb.html), of base retain their [access control](http://www.cs.nmsu.edu/~rth/cs/cs177/map/access.html) category: public members become [public](http://www.cs.nmsu.edu/~rth/cs/cs177/map/access.html), private members remain [private](http://www.cs.nmsu.edu/~rth/cs/cs177/map/access.html). The rules of scope still apply, however, so that function members of the derived class cannot access the inherited members directly, unless the base class declares them to *protected* ( or public). In the example below, the class derived contains a member function accessP that accesses a data member dm1 of the class base, from which it inherits dm1. If the access control of dm1 was private then the function accessP would not compile. It would work for public access, but rather than open up access to users of the class, the protected keyword laves the member private, execept for derived classes: a useful convenience for programmers.

class base {

protected:

int dm1;

...

};

class derived: public base {

...

public:

void accessP() { ... dm1 ... }

...

};

Inheritance makes two new types which can be used separately, if necessary. However, more common is to use them as [sub- and super-type](http://www.cs.nmsu.edu/~rth/cs/cs177/map/subtype.html).

**Internal state**

An [object](http://www.cs.nmsu.edu/~rth/cs/cs177/map/object.html) has a lifetime from its time of creation to the time of its destruction. While it is alive, the object maintains the values of its internal variables unless they are altered by assignment. The conjunction of all the values of all the internal variables ([private or public](http://www.cs.nmsu.edu/~rth/cs/cs177/map/access.html)) is the state of the object. Since [class variables](http://www.cs.nmsu.edu/~rth/cs/cs177/map/class.html) can be of any type, some objects contain other objects, these being the values of those variables.

Some objects have indefinite lifetime because they maintain their state for the whole lifetime of the program - these are global variables. Objects created during function call have a definite lifetime - these are local variables.[Dynamic allocation](http://www.cs.nmsu.edu/~rth/cs/cs177/map/dynalloc.html) allows the lifetime of an object to be controlled by program execution.

**Names**

Names, or identifiers, are used to label C++ language items. They are used for:

* types, including the structured types, class, struct, enum and union;
* variables and constants (const variables)
* functions, both member functions and global C-like functions
* macros (#defines)

The meaning of names follow the rules of [scope](http://www.cs.nmsu.edu/~rth/cs/cs177/map/scope.html), the most important of which is that a name can only have one meaning within a single scope.

Names can also be placed into logically separate collections called [namespaces](http://www.cs.nmsu.edu/~rth/cs/cs177/map/namespace.html), that allow much more freedom over their choice to avoid name clashes

## Namespaces

The idea of a namespace is to allow re-use of names across collections of files so that programmers are free to choose names which might already be in use, perhaps in an imported library file. This is especially true of standard libraries, where there are many commonly-used names. It is another form of [encapsulation](http://www.cs.nmsu.edu/~rth/cs/cs177/map/encap.html), but this time (unlike the [class](http://www.cs.nmsu.edu/~rth/cs/cs177/map/class.html) encapsulation) it is merely considered to be a collection of names. To use it, preface a collection of declarations and/or definitions with the namespace declaration:

namespace myNamespace {

class C { ... };

class D { ... };

}

where myNamespace is any identifier. All of the names declared are then considered to be part of that namespace. Additional files can have the same (or a different) namespace by using a similar namespace declaration, and there could be more than namespace in a file. All the names (classes, functions, templates, variables) must now be prefixed with the namespace when used outside the namespace:

namespace ns1 {

int var;

}

namespace ns2 {

void f() {

cout << ns1::var << endl;

}

}

This is the way that the standard template library can be used:

#include <iostream>

int main () {

std::cout << "Hello world in ANSI-C++\n";

return 0;

}

More common is to avoid all these prefixes with the 'using' command:

#include <iostream>

using namespace std;

int main () {

cout << "Hello world in ANSI-C++\n";

return 0;

}

## Objects

An object is an instance of a [class](http://www.cs.nmsu.edu/~rth/cs/cs177/map/class.html), which is the OO (object-oriented) way of saying it. In more common terms, an object is a variable of a given type. In C++, a class declaration also declares a new type.

Objects are declared in C form, by preceding the name, or names of the object or objects by the type (class) name. Thus:

NewType1 obj1, obj2;

declares two objects obj1 and obj2, both of type NewType1, where NewType1 is the name of a previously declared class.

Objects can also be initialized, i.e. its [internal state](http://www.cs.nmsu.edu/~rth/cs/cs177/map/intstate.html) can be set, by passing arguments to a constructor function when the object is declared. Thus:

NewType2 obj3(32);

declares an object of type NewType2, and passes 32 to its [constructor](http://www.cs.nmsu.edu/~rth/cs/cs177/map/construc.html). There can be any number of constructors, as long as each has a different [signature](http://www.cs.nmsu.edu/~rth/cs/cs177/map/signat.html), since [member functions](http://www.cs.nmsu.edu/~rth/cs/cs177/map/funmemb.html) can be [overloaded](http://www.cs.nmsu.edu/~rth/cs/cs177/map/overload.html).

A class can also have a [default constructor](http://www.cs.nmsu.edu/~rth/cs/cs177/map/construc.html) that takes no arguments. The parentheses may then be omitted, leaving a "regular" C declaration, like the first example above. The default constructor can also initialize the object since it can use constants to do so.

Objects have a [lifetime](http://www.cs.nmsu.edu/~rth/cs/cs177/map/intstate.html), either governed by a local [scope](http://www.cs.nmsu.edu/~rth/cs/cs177/map/scope.html), or they can be [created or destroyed dynamically](http://www.cs.nmsu.edu/~rth/cs/cs177/map/dynalloc.html).

## Operator functions

Any of the normal unary or binary operators may be overloaded within a class. This allows the redefinition of an operator for use by the new type created by a class declaration. Although the body of the function can contain any code appropriate to the redefined operator, both the number of parameters (unary or binary) and the order of precedence must remain the same.

To overload an operator a function member is declared with the name operator followed by the operator symbol (operator is a reserved word in C++). e.g. in this class the operator + is overloaded to produce the effect of adding two pairs of data members:

class OPOL {

private:

int x, y;

public:

OPOL(int xx, int yy) : x(xx), y(yy) {}

OPOL operator + (OPOL arg2) {

arg2.x += x;

arg2.y += y;

return arg2;

}

};

This now allows the following code to compile correctly:

OPOL a1(1, 2), a2(3, 5), a3(0, 0);

a3 = a1 + a2;

The result is that a3 contains the values x = 4 and y = 7;

The [input/output](http://www.cs.nmsu.edu/~rth/cs/cs177/map/empty.html)classes overload the left and right shift operators (<< and >>) in order to provide insertion and extraction for [streams](http://www.cs.nmsu.edu/~rth/cs/cs177/map/empty.html).

## Overloading

The usual rules of [scope](http://www.cs.nmsu.edu/~rth/cs/cs177/map/scope.html) forbid the re-use of a name in the same scope. In C++ this relaxed for function members which can have the same name as long as their [signatures](http://www.cs.nmsu.edu/~rth/cs/cs177/map/signat.html) are different. Thus a class may have two functions called overload, one of which has no parameters, and one of which has a single integer parameter. Each function overloading must have its own unique definition, thus there really are multiple meanings for the same name, even within the same scope.

class OL {

...

public:

void overload();

void overload(int);

...

};

In general, a function may be overloaded by altering its signature, however, because of the implicit conversion of types in C, some overloadings are ambiguous. This is true of any function which only differs by its return type. e.g.

void F(int);

and

int F(int);

cannot be distinguished by the compiler given a call

F(10);

An example where there is also ambiguity is:

void F(int);

and

void F(long);

given the same call

F(10);

Again these two cannot be distinguished by the compiler due to implicit coercion among the numeric types.

[Operators](http://www.cs.nmsu.edu/~rth/cs/cs177/map/opfuncs.html) may also be overloaded, as can [constructors](http://www.cs.nmsu.edu/~rth/cs/cs177/map/construc.html).

## Pointers and references

C++ can have three kinds of variable, leading to three kinds of assignment, passing of arguments to functions, and returning values from functions. From C it inherits valueand pointervariables.

### Value variables

Value variables are declared with a plain syntax, preceding the name of the variable with the type (or class) name. For example:

int i1;

float sum\_of\_money;

Graphic g1, g2;

Telescope t1;

where int and float are examples of [base types](http://www.cs.nmsu.edu/~rth/cs/cs177/map/types.html), and Graphic and Telescope are [names](http://www.cs.nmsu.edu/~rth/cs/cs177/map/names.html)of classes already declared.

### Pointer variables

Similarly:

int\* p1;

float\* pointer\_to\_sum\_of\_money;

Graphic\* pG1, pG2;

Telescope\* pt1;

declare pointer variables of the same types.

In order to be useful, a pointer variable must be made to point to an existing object. This can be done by initialization:

int\* p1 = &i1;

Graphic\* pG1 = &g1;

or by assignment:

pG2 = &g2;

pt1 = &t1;

The ampersand (&) is the "address-of" operator that comes from C.

Pointers may also be initialized or assigned through the [new operator](http://www.cs.nmsu.edu/~rth/cs/cs177/map/dynalloc.html). They can be used in [parameter lists](http://www.cs.nmsu.edu/~rth/cs/cs177/map/arguments.html) as well as as [return types](http://www.cs.nmsu.edu/~rth/cs/cs177/map/signat.html).

### Reference variables

C++ has a third kind of variable which operates like a pointer but with a value-like syntax. To declare a reference variable, we use &, not \*:

int& ri1;

Graphic& rG1;

Telescope& rt1;

The best way to think of reference variables is as aliases, i.e. a reference variable adds a name to an existing object. For this reason, "free" reference variables, such as the ones above are not allowed unless initialized:

int& ri1 = i1;

Graphic& rG1 = g1;

Telescope& rt1 = t1;

Now ri1 is another name for the object i1, rG1 for object g1 and rt1 for object t1. References are however allowed as class members (as long as they are initialized by some [constructor](http://www.cs.nmsu.edu/~rth/cs/cs177/map/construc.html)), as parameters, and as return types.

One advantage of references over pointers is a simpler syntax. Compare the two "swap" functions below:

void pswap(int\* pi1, int\* pi2) {

int\* ptemp = pi1;

\*ptemp = \*pi1;

\*pi1 = \*pi2;

\*pi2 = \*ptemp;

}

called with:

int i1 = 12, i2 = 20;

pswap(&i1, &i2);

void rswap(int& ri1, int& ri2) {

int temp;

temp = ri1;

ri1 = ri2;

ri2 = temp;

}

called with:

int i1 = 12, i2 = 20;

rswap(i1, i2);

Both pswap and rswap produce the same result - the swapping of values in i1 and i2.

In theory the reference variable can replace most uses of pointers, but in practice, pointers are still used, especially with [dynamic allocation](http://www.cs.nmsu.edu/~rth/cs/cs177/map/dynalloc.html).

**Polymorphism**

Polymorphism literally means "many shapes". It refers to the properties of a language which alter the normal meaning of a language construct.

C++ exhibits four kinds of polymorphism:

1. [overloaded functions](http://www.cs.nmsu.edu/~rth/cs/cs177/map/overload.html)  
   the definition attached to a function name is altered according to the number of type of its parameters (its [signature](http://www.cs.nmsu.edu/~rth/cs/cs177/map/signat.html))
2. [sub-type expressions](http://www.cs.nmsu.edu/~rth/cs/cs177/map/subtype.html)  
   the evaluation of an expression is altered according to the [inheritance](http://www.cs.nmsu.edu/~rth/cs/cs177/map/inherit.html) relationship between classes
3. [generic classes](http://www.cs.nmsu.edu/~rth/cs/cs177/map/template.html) (and functions)  
   the declaration of a [class](http://www.cs.nmsu.edu/~rth/cs/cs177/map/class.html) is altered according to the value of compile-time parameters
4. [dynamic binding](http://www.cs.nmsu.edu/~rth/cs/cs177/map/virtual.html) of functions  
   the evaluation of an [function call](http://www.cs.nmsu.edu/~rth/cs/cs177/map/memaccess.html) expression is altered according to the actual type of the caller object

C++ allows the programmer to use the same name in different contexts. This re-use of names is governed by a set of rules of [visibility](http://www.cs.nmsu.edu/~rth/cs/cs177/map/visib.html), a term that gives the rules a metaphorical nature.

There are four main kinds of scope: [class](http://www.cs.nmsu.edu/~rth/cs/cs177/map/clsscope.html) scope, [function](http://www.cs.nmsu.edu/~rth/cs/cs177/map/funscope.html) scope, [block](http://www.cs.nmsu.edu/~rth/cs/cs177/map/block.html) scope, and [file](http://www.cs.nmsu.edu/~rth/cs/cs177/map/filscope.html) scope. A name has local scope if it declared between a pair of matched braces { }.

## Signatures

Function declarations (also called *prototypes*) consist of a name and a signature, the "type" of the function. The number and types of any [parameters](http://www.cs.nmsu.edu/~rth/cs/cs177/map/empty.html), together with the [return type](http://www.cs.nmsu.edu/~rth/cs/cs177/map/empty.html) make up the signature. E.g. if we combine a name, Fred, with a signature, two integer parameters and a floating point return, we get the prototype float Fred(int, int).

A special type used in signatures is *void*, which signifies no parameters, as in float Fred(void), or no return value, as in void Fred(int). A void [pointer](http://www.cs.nmsu.edu/~rth/cs/cs177/map/pointer.html) is capable of accepting a pointer to any type as its [argument](http://www.cs.nmsu.edu/~rth/cs/cs177/map/empty.html).

## Structures

In order to be compatible with [ANSI C](http://www.cs.nmsu.edu/~rth/cs/cs177/map/ansi.html), C++ allows the keyword *struct* in exactly the way that C allows - to define compound types. However, the internals of a *struct* are extended to make it identical to the [class](http://www.cs.nmsu.edu/~rth/cs/cs177/map/class.html) except in one respect. In a *class*, the default [access control](http://www.cs.nmsu.edu/~rth/cs/cs177/map/access.html)is private, whereas in a *struct*, the default access control is public. Everything else: [data members](http://www.cs.nmsu.edu/~rth/cs/cs177/map/datamem.html), [function members](http://www.cs.nmsu.edu/~rth/cs/cs177/map/funmemb.html), [constructors](http://www.cs.nmsu.edu/~rth/cs/cs177/map/construc.html), [destructors](http://www.cs.nmsu.edu/~rth/cs/cs177/map/destruct.html), [overloading](http://www.cs.nmsu.edu/~rth/cs/cs177/map/overload.html), [operators](http://www.cs.nmsu.edu/~rth/cs/cs177/map/opfuncs.html) etc. is the same.

**Sub- and super-types**

When two classes have a sub/super-type relationship, objects of the derived class may be used where objects of the base class type are expected. This says that objects are in a is-a relationship. Thus, if a class ferrari inherits from a class sports\_car, then all objects of type ferrari are also of type sports\_car. Thus the following is valid:

class sports\_car { ... };

class ferrari : public sports\_car { ...};

void gg(sports\_car);

ferrari f1;

gg(f1);

The same occurs with assignment:

sports\_car s1;

ferrari f1;

s1 = f1;

is allowed because f1 is truncated to a sports\_car, and then assignment can occur. However:

f1 = s1;

is not allowed because s1 cannot be converted to an object of type ferrari. An explicit cast:

s1 = (ferrai)s1;

fails in the same way.

Of course, whether this makes sense depends on the implementation of the classes, but the sub-typing principle must allow this to happen. In terms of argument passing, the (larger) object of type ferrari is truncated down to sports\_car size in order to be copied into the parameter. This is safe, since the truncated part could not be used by the function anyway. The strong type-checking of C++ ensures this.

This implicit type conversion (object truncation) is called *up-casting*. The inverse, *Down-casting*, is dangerous, although possible, *but only using pointers*:

ferrai\* pf1 = &f1;

sports\_car\* ps1 = &s1;

pf1 = (ferrari\*)ps1;

Here an explicit cast changes the type of the pointer to s1 before the assignment, However, the extra part of the object that corresponds to the derived class is not initialized, and accessing it may be disastrous. e.g.

pf1->part\_only\_ferraris\_have

may give an unpredictable result, since all sports car do not have such a part.

**Virtual functions**

Ordinarily functions are selected at compile time according to the static type of the caller object. e.g.

class X {

public:

void f();

};

class Y : public X {

public:

void f(); // identical [signature](http://www.cs.nmsu.edu/~rth/cs/cs177/map/signat.html) to X's function f

};

class Z : public X {

public:

...

};

For these [objects](http://www.cs.nmsu.edu/~rth/cs/cs177/map/object.html):

X x; Y y; Z z;

the following calls are valid:  
x.f() calls X's function f, y.f() calls Y's function f, and z.f() calls X's function f, which Z inherits

When a pointer to type X points to an object of type Y, or of type Z, the static type is still used, according to the [sub-type](http://www.cs.nmsu.edu/~rth/cs/cs177/map/subtype.html) principle, i.e. with

X\* px1 = &x; // px1 points to an X

X\* px2 = &y; // px2 points to a Y

X\* px3 = &z; // px3 points to a Z

the three calls: px1->f(), px2->f(), and px3->f() still call X's function f, even though, in the second and third cases, px2 and px3 actually point to objects of type Y and Z respectively.

This can be altered if the keyword *virtual* is used on the function declaration in the base class, X:

class X {

public:

virtual void f(); // needs to be repeated in [derived classes](http://www.cs.nmsu.edu/~rth/cs/cs177/map/inherit.html)

};

Now, whereas the calls for objects x, y and z remain unchanged, the situation with pointers is different:

px1->f();

still calls X's function (the function is not really virtual-- it still has a body!), but

px2->f();

now calls Y's function, the function of the object actually pointed to by px2. Also

px3->f();

calls X's function, but this time because Z (the type of the object pointed to by px3) inherits it from X.

This selection is done at run-time, and is called *dynamic*, or *late*binding.

## Visibility

A simple metaphor that helps understand the principle of scope is to imagine that a wall exists around each region of [scope](http://www.cs.nmsu.edu/~rth/cs/cs177/map/scope.html), and to consider what [names](http://www.cs.nmsu.edu/~rth/cs/cs177/map/names.html) one can see through the walls when standing inside the region - in other words which names are *visible*.

Scope regions may be nested inside each other, and rules govern what is visible and what is not. In general, a name may be seen if it is outside the walls, but it is not possible to see inside another set of walls. In the diagram below, the four kinds of scope are shown by example.

