The Evolution of C++: 1985 to 1989

Bjarne Stroustrup AT&T Bell Laboratories

ABSTRACT: *The C++ Programming Language* [Stroustrup 1986] describes C++ as defined and implemented in August 1985. This paper describes the growth of the language since then and clarifies a few points in the definition. It is emphasized that these language modifications are extensions; C++ has been and will remain a stable language suitable for long term software development. The main new features of C++ are: multiple inheritance, type-safe linkage, better resolution of overloaded functions, recursive definition of assignment and initialization, better facilities for user-defined memory management, abstract classes, `static` member functions, `const` member functions, `protected` members, overloading of operator ->, and pointers to members. These features are provided in the 2.0 release of C++.

This paper is a revised and expanded version of a paper with a similar title presented at the USENIX C++ Workshop in Santa Fe, New Mexico, November 1987.

0. *Introduction*

As promised in *The C++ Programming Language*, C++ has been evolving to meet the needs of its users. This evolution has been guided by the experience of users of widely varying backgrounds working in a great range of application areas. The primary aim of the extensions has been to enhance C++ as a language for data abstraction and object-oriented programming [Stroustrup 1987a] in general and to enhance it as a tool for writing high-quality libraries of user-defined types in particular. By a high-quality library I mean a library that provides a concept to a user in the form of one or more classes that are convenient, safe, and efficient to use. In this context, *safe* means that a class provides a specific type-secure interface between the users of the library and its providers; *efficient* means that use of the class does not impose large overhead in run-time or space on the user compared with hand written C code.

Portability of at least some C++ implementations is a key design goal. Consequently, extensions that would add significantly to the porting time or to the demands on resources for a C++ compiler have been avoided. This ideal of language evolution can be contrasted with plausible alternative directions such as making programming convenient

- at the expense of efficiency or structure;
- for novices at the expense of generality;
- in a specific application area by adding special purpose features to the language;
- by adding language features to increase integration into a specific C++ environment.

For some ideas of where these ideas of language evolution might lead C++ see Stroustrup 1987a, b; 1989.

A programming language is only one part of a programmer's world. Naturally, work is being done in many other fields (such as tools, environments, libraries, education and design methods) to make C++ programming more pleasant and effective. This paper, however, deals strictly with language and language implementation issues. Furthermore, this paper discusses only features that are generally available. Speculative or experimental features, such as exception handling and parameterized types, are not presented here.

1. Overview

This paper is a brief overview of new language features; it is not a manual[1] or a tutorial. The reader is assumed to be familiar with the language as described in *The C++ Programming Language* and to have sufficient experience with C++ to recognize many of the problems that the features described here are designed to solve or alleviate. Most of the extensions take the form of removing restrictions on what can be expressed in C++.

First some extensions to C++'s mechanisms for controlling access to class members are presented. Like all extensions described here, they reflect experience with the mechanisms they extend and the increased demands posed by the use of C++ in relatively large and complicated projects:

Access Control (§2)

C++ software is increasingly constructed by combining semi-independent components (modules, classes, libraries, etc.) and much of the effort involved in writing C++ goes into the design and implementation of such components. To help these activities, the rules for overloading function names and the rules for linking separately compiled code have been refined:

Overloading Resolution (§3)
Type-Safe Linkage (§4)

1. A revised C++ manual is under review and will appear later this year.

Classes are designed to represent general or application-specific concepts. Originally, C++ provided only single inheritance, that is, a class could have at most one direct base class, so that the directly representable relations between classes had to be a tree structure. This is sufficient in a large majority of cases. However, there are important concepts for which relations cannot be naturally expressed as a tree, but where a directed acyclic graph is suitable. As a consequence, C++ has been extended to support multiple inheritance, that is, a class can have several immediate base classes, directly. The rules for ambiguity resolution and for initialization of base classes and members have been refined to cope with this extension:

Multiple Inheritance (§5)
Base and Member Initialization (§6)
Abstract Classes (§7)

The concept of a class member has been generalized. Most important, the introduction of `const` member functions allows the rules for `const` class objects to be enforced:

`static` Member Functions (§8)
`const` Member Functions (§9)
Initialization of `static` Members (§10)
Pointers to Members (§11)

The mechanisms for user-defined memory management have been refined and extended to the point where the old and inelegant "assignment to `this`" mechanism has become redundant:

User-Defined Free Store Management (§12)

The rules for assignment and initialization of class objects have been made more general and uniform to require less work from the programmer:

Assignment and Initialization (§13)

Some minor extensions are presented:

Operator `->` (§14)
Operator `,` (§15)
Initialization of `static` objects (§16)

The last section does not describe language extensions but presents the resolution of some details of the C++ language definition:

Resolutions (§17)

In addition to the extensions mentioned here, many details of the definition of C++ have been modified for greater compatibility with the proposed ANSI C standard [ANSI 1988].

2. *Access Control*

The rules and syntax for controlling access to class members have been made more flexible.

2.1 protected Members

The simple private/public model of data hiding served C++ well where C++ was used essentially as a data abstraction language and for a large class of problems where inheritance was used for object-oriented programming. However, when derived classes are used there are two kinds of users of a class: derived classes and "the general public." The members and friends that implement the operations on the class operate on the class objects on behalf of these users. The private/public mechanism allows the programmer to distinguish clearly between the implementers and the general public, but does not provide a way of catering specifically to derived classes.[2] This often caused the data hiding mechanisms to be ignored:

```
class X {         // One bad way:
    // ...
public:
    int a;  // ''a'' should have been private
            // don't use it unless you are
            // a member of a derived class
    // ...
};
```

Another symptom of this problem was overuse of friend declarations:

2. An interesting discussion of access and encapsulation problems in languages with inheritance mechanisms can be found in Snyder [1986].

```
class X {         // Another bad way:
friend class D1; // make derived classes friends to
friend class D2; // give access to private member 'a'
// ...
friend class Dn;
    // ...
    int a;
public:
    ...
};
```

The solution adopted was `protected` members. A `protected` member is accessible to members and friends of a derived class as if it were `public`, but inaccessible to "the general public" just like `private` members. For example:

```
class X {
// private by default:
    int priv;
protected:
    int prot;
public:
    int publ;
};

class Y : public X {
    void mf();
};

Y::mf()
{
    priv = 1;     // error: priv is private
    prot = 2;     // OK: prot is protected
        //    and mf() is a member of Y
    publ = 3;     // OK: publ is public
}

void f(Y* p)
{
    p->priv = 1;  // error: priv is private
    p->prot = 2;  // error: prot is protected
        //       and f() is not a friend
                  //         or a member of X or Y
    p->publ = 3;  // OK: publ is public
}
```

A more realistic example of the use of `protected` can be found in section 5.

A `friend` function has the same access to `protected` members as a member function.

A subtle point is that accessibility of protected members depends on the static type of the pointer used in the access. A member or a friend of a derived class has access only to protected members of objects that are known to be of its derived type. For example:

```
class Z : public Y {
    // ...
};

void Y::mf()
{
    prot = 2;      // OK: prot is protected and
        //     mf() is a member

    X a;
    a.prot = 3;   // error: prot is protected and
        //        a is not a Y

    X* p = this;
    p->prot = 3;  // error: prot is protected and
                  //        p is not a pointer to Y

    Z b;
    b.prot = 4;   // OK: prot is protected and
                  //     mf() is a member and
        //     a Z is a Y
}
```

A protected member of a class `base` is a protected member of a class derived from `base` if the derivation is public and private otherwise.

2.2 Access Control Syntax

The following example confuses most beginners and even experts get bitten sometimes:

```
class X {
    // ...
public:
    int f();
};

class Y : X { /* ... */ };
```

```
int g(Y* p)
{
    // ...
    return p->f();  // error!
};
```

Here `X` is by default declared to be a `private` base class of `Y`. This means that `Y` is not a subtype of `X` so the call `p->f()` is illegal because `Y` does not have a public function `f()`. Private base classes are quite an important concept, but to avoid confusion it is recommended that they be declared `private` explicitly:

```
class Y : private X { /* ... */ };
```

Several `public`, `private`, and `protected` sections are allowed in a class declaration:

```
class X {
public:
    int i1;
private:
    int i2;
public:
    int i3;
};
```

These sections can appear in any order. This implies that the public interface of a class may appear textually before the private "implementation details":

```
class S {
public:
    f();
    int i1;
    // ...
private:
    g();
    int i2;
    // ...
};
```

2.3 Adjusting Access

When a class `base` is used as a private base class all of its members are considered private members of the derived class.

The syntax *base-class-name* :: *member-name* can be used to restore access of a member to what it was in the base:

```
class base {
public:
    int publ;
protected:
    int prot;
private:
    int priv;
};

class derived : private base {
protected:
    base::prot;     // protected in derived
public:
    base::publ;     // public in derived
};
```

This mechanism cannot be used to grant access that was not already granted by the base class:

```
class derived2 : public base {
public:
    base::priv;     // error: base::priv is private
};
```

This mechanism can be used only to restore access to what it was in the base class:

```
class derived3 : private base {
protected:
    base::publ;     // error: base::publ was public
};
```

This mechanism cannot be used to remove access already granted:

```
class derived4 : public base {
private:
    base::publ;     // error: base::publ is public
};
```

We considered allowing the last two forms and experimented with them, but found that they caused total confusion among users about the access control rules and the rules for private and public derivation. Similar considerations led to the decision not to introduce the (otherwise perfectly reasonable) concept of protected base classes.

2.4 Details

A friend function has the same access to base class members as a member function. For example:

```
class base {
protected:
    int prot;
public:
    int pub;
};

class derived : private base {
public:
    friend int fr(derived *p) { return p->prot; }
    int mem() { return prot; }
};
```

In particular, a friend function can perform the conversion of a pointer to a derived class to its private base class:

```
class derived2 : private base {
public:
    friend base* fr2(derived2 *p) { return p; }
    base* mem() { return this; }
};

base* f(derived2* p)
{
    return p;   // error: cannot convert; base is a
                // private base class of derived
}
```

However, friendship is *not* transitive. For example:

```
class X {
friend class Y;
private:
    int a;
};

class Y {
    friend int fr3(Y *p)
        { return p->a; }  // error: fr3() is not
           //        a friend of X
    int mem(Y* p)
        { return p->a; }  // OK: mem() is a friend of X
};
```

3. *Overloading Resolution*

The C++ overloading mechanism was revised to allow resolution of types that used to be "too similar" and to gain independence of declaration order. The resulting scheme is more expressive and catches more ambiguity errors. Consider:

```
double abs(double);
float abs(float);
```

To cope with single precision floating point arithmetic it must be possible to declare both of these functions; now it is. The effect of any call of `abs()` given the declarations above is the same if the order of declarations was reversed:

```
float abs(float);
double abs(double);
```

Here is a slightly simplified explanation of the new rules. Note that with the exception of a few cases where the the older rules allowed order dependence the new rules are compatible and old programs produce identical results under the new rules. For the last two years or so C++ implementations have issued warnings for the now "outlawed" order dependent resolutions.

C++ distinguishes 5 kinds of "matches":

1. Match using no or only unavoidable conversions (for example, array name to pointer, function name to pointer to function, and `T` to `const T`).
2. Match using integral promotions (as defined in the proposed ANSI C standard; that is, `char` to `int`, `short` to `int` and their `unsigned` counterparts) and `float` to `double`.
3. Match using standard conversions (for example, `int` to `double`, `derived*` to `base*`, `unsigned int` to `int`).
4. Match using user defined conversions (both constructors and conversion operators).
5. Match using the ellipsis `...` in a function declaration.

Consider first functions of a single argument. The idea is always to choose the "best" match, that is the one highest on the list above. If there are two best matches the call is ambiguous and thus a compile time error. For example,

```
float abs(float);
double abs(double);
int abs(int);
unsigned abs(unsigned);
char abs(char);

abs(1);     // abs(int);
abs(1U);    // abs(unsigned);
abs(1.0);   // abs(double);
abs(1.0F);  // abs(float);
abs('a');   // abs(char);
abs(1L);    // error: ambiguous, abs(int) or abs(double)
```

Here, the calls take advantage of the ANSI C notation for `unsigned` and `float` literals and of the C++ rule that a character constant is of type `char`.[3] The call with the `long` argument `1L` is ambiguous since `abs(int)` and `abs(double)` would be equally good matches (match with standard conversion).

Hierarchies established by public class derivations are taken into account in function matching, and where a standard conversion is needed the conversion to the "most derived" class is chosen. A `void*` argument is chosen only if no other pointer argument matches. For example:

```
class B { /* ... */ };
class BB : public B { /* ... */ };
class BBB : public BB { /* ... */ };

f(B*);
f(BB*);
f(void*);

void g(BBB* pbbb, int* pi)
{
    f(pbbb);  // f(BB*);
    f(pi);    // f(void*);
}
```

3. Surprisingly, giving character constants type `char` does not cause incompatibilities with C where they have type `int`. Except for the pathological example `sizeof('a')`, every construct that can be expressed in both C and C++ gives the same result. The reason for the surprising compatibility is that even though character constants have type `int` in C, the rules for determining the values of such constants involves the standard conversion from `char` to `int`.

This ambiguity resolution rule matches the rule for virtual function calls where the member from the most derived class is chosen.

If two otherwise equally good matches differ in terms of `const`, the `const` specifier is taken into account in function matching for pointer and reference arguments. For example:

```
char* strtok(char*, const char*);
const char* strtok(const char*, const char*);

void g(char* vc, const char* vcc)
{
            // strtok(char*, char*);
    char* p1 = strtok(vc,"a");
            // strtok(const char*, char*);
    const char* p2 = strtok(vcc,"a");
    char* p3 = strtok(vcc,"a");         // error
}
```

In the third case, `strtok(const char*, const char*)` is chosen because `vcc` is a `const char*`. This leads to an attempt to initialize the `char* p3` with the `const char*` result.

For calls involving more than one argument a function is chosen provided it has a better match than every other function for at least one argument and at least as good a match as every other function for every argument. For example:

```
class complex { /* ... */ complex(double); };

f(int,double);
f(double,int);
f(complex,int);
f(int ...);
f(complex ...);

complex z = 1;

f(1, 2.0);    // f(int,double);
f(1.0, 2);    // f(double,int);
f(z, 1.2);    // f(complex,int);
f(z, 1, 3);   // f(complex ...);
f(2.0, z);    // f(int ...);
f(1, 1);      // error: ambiguous, f(int,double) and
        //        f(double,int)
```

The unfortunate narrowing from `double` to `int` in the third and the second to last cases causes warnings. Such narrowings are allowed to preserve compatibility with C. In this particular case the narrowing is harmless, but in many cases `double` to `int` conversions are value destroying and they should never be used thoughtlessly.

As ever, at most one user-defined and one built-in conversion may be applied to a single argument.

4. Type-Safe Linkage

Originally, C++ allowed a name to be used for more than one name ("to be overloaded") only after an explicit `overload` declaration. For example:

```
overload max;              // 'overload' now obsolete
int max(int,int);
double max(double,double);
```

It used to be considered too dangerous simply to use a name for two functions without previous declaration of intent. For example:

```
int abs(int);
double abs(double);     // used to be an error
```

This fear of overloading had two sources:

1. concern that undetected ambiguities could occur
2. concern that a program could not be properly linked unless the programmer explicitly declared where overloading was to take place.

The former fear proved largely groundless and the few problems found in actual use have been taken care of by the new order-independent overloading resolution rules. The latter fear proved to have a basis in a general problem with C separate compilation rules that had nothing to do with overloading.

On the other hand, the redundant `overload` declarations themselves became an increasingly serious problem. Since they had to precede (or be part of) the declarations they were to enable, it was not possible to merge pieces of software using the same function name for different functions unless both pieces had declared the function overloaded. This is not usually the case. In particular, the name one wants to overload is often the name of a C standard library function declared in a C header. For example, one might have standard headers like this:

```
/* Header for C standard math library, math.h: */
    double sqrt(double);
    /* ... */

// header for C++ standard complex arithmetic library,
// complex.h:
    overload sqrt;
    complex sqrt(complex);
    // ...
```

and try to use them like this:

```
#include <math.h>
#include <complex.h>
```

This causes a compile time error when the `overload` for `sqrt()` is seen after the first declaration of `sqrt()`. Rearranging declarations, putting constraints on the use of header files, and sprinkling `overload` declarations everywhere "just in case" can alleviate this kind of problem, but we found the use of such tricks unmanageable in all but the simplest cases. Abolishing `overload` declarations and getting rid of the `overload` keyword in the process is a much better idea.

Doing things this way does pose an implementation problem, though. When a single name is used for several functions, one must be able to tell the linker which calls are to be linked to which function definitions. Ordinary linkers are not equipped to handle several functions with the same name. However, they can be tricked into handling overloaded names by encoding type information into the names seen by the linker. For example, the names for these two functions

```
double sqrt(double);
complex sqrt(complex);
```

become

```
sqrt__Fd
sqrt__F7complex
```

in the compiler output to the linker. The user and the compiler see the C++ source text where the type information serves to disambiguate and the linker sees the names that have been disambiguated by adding a textual representation of the type information. Naturally, one might have a linker that understood about type information, but it is not necessary and such linkers are certainly not common.

Using this encoding or any equivalent scheme solves a long-standing problem with C linkage. Inconsistent function declarations in separately compiled code fragments are now caught. For example,

```
// file1.c:

extern name* lookup(table* tbl, const char* name);

// ...

void some_fct(char* s)
{
    name* n = lookup(gtbl,s);
}
```

looks plausible and the compiler can find no fault with it. However, if the definition of `lookup()` turns out to be

```
// file2.c:

int lookup(const char* name)
{
    // ...
}
```

the linker now has enough information to catch the error.

Finally, we have to face the problem of linking to code fragments written in other languages that do not know the C++ type system or use the C++ type encoding scheme. One could imagine all compilers for all languages on a system agreeing on a type system and a linkage scheme such that linkage of code fragments written in different languages would be safe. However, since this will not typically be the case we need a way of calling functions written in a language that does not use a type-safe linkage scheme and a way to write C++ functions that obey the different (and typically unsafe) linkage rules for other languages. This is done by explicitly specifying the name of the desired linkage convention in a declaration:

```
extern "C" double sqrt(double);
```

or by enclosing whole groups of declarations in a linkage directive:

```
extern "C" {
#include <math.h>
}
```

By applying the second form of linkage directive to standard header files one can avoid littering the user code with linkage directives. This type-safe linkage mechanism is discussed in detail in Stroustrup [1988].

5. *Multiple Inheritance*

Consider writing a simulation of a network of computers. Each node in the network is represented by an object of class `Switch`, each user or computer by an object of class `Terminal`, and each communication line by an object of class `Line`. One way to monitor the simulation (or a real network of the same structure) would be to display the state of objects of various classes on a screen. Each object to be displayed is represented as an object of class `Displayed`. Objects of class `Displayed` are under control of a display manager that ensures regular update of a screen and/or data base. The classes `Terminal` and `Switch` are derived from a class `Task` that provides the basic facilities for co-routine style behavior. Objects of class `Task` are under control of a task manager (scheduler) that manages the real processor(s).

Ideally, `Task` and `Displayed` are classes from a standard library. If you want to display a terminal, class `Terminal` must be derived from class `Displayed`. Class `Terminal`, however, is already derived from class `Task`. In a single inheritance language, such as Simula67, we have only two ways of solving this problem: deriving `Task` from `Displayed` or deriving `Displayed` from `Task`. Neither is ideal because they each create a dependency between the library versions of two fundamental and independent concepts. Ideally, one would want to be able to say that a `Terminal` is a `Task` *and* a `Displayed`; that a `Line` is a `Displayed` *but not* a `Task`; and that a `Switch` is a `Task` *but not* a `Displayed`.

The ability to express this class hierarchy, that is, to derive a class from more than one base class, is usually referred to as *multiple inheritance*. Other examples involve the representation of various kinds of windows in a window system [Weinreb & Moon 1981] and the representation of various kinds of processors and compilers for a multi-machine, multi-environment debugger [Cargill 1986].

In general, multiple inheritance allows a user to combine concepts represented as classes into a composite concept represented as a derived class. C++ allows this to be done in a general, type-safe, compact, and efficient manner. The basic scheme allows independent concepts to be combined and ambiguities to be detected at compile time. An extension of the base class concept, called *virtual base classes*, allows dependencies between classes in an inheritance DAG (Directed Acyclic Graph) to be expressed.

5.1 Ambiguity Detection

Ambiguous uses are detected at compile time:

```
class A { public: f(); /* ... */ };
class B { public: f(); /* ... */ };
class C : public A, public B { };

void g() {
    C* p;
    p->f(); // error: ambiguous
}
```

Note that it is not an error to combine classes containing the same member names in an inheritance DAG. The error occurs only when a name is used in an ambiguous way – and only then does the compiler have to reject the program. This is important since most potential ambiguities in a program never appear as actual ambiguities. Considering a potential ambiguity an error would be far too restrictive.[4]

Typically one would resolve the ambiguity by adding a function:

4. The strategy for dealing with ambiguities in inheritance DAGs is essentially the same as the strategy for dealing with ambiguities in expression evaluation involving overloaded operators and user-defined coercions. Note that the access control mechanism does not affect the ambiguity control mechanism. Had `B::f()` been `private` the call `p->f()` would still be ambiguous.

```
class C : public A, public B {
public:
    f()
    {
        // C's own stuff
        A::f();
        B::f();
    }
    // ...
};
```

This example shows the usefulness of naming members of a base class explicitly with the name of the base class. In the restricted case of single inheritance, this way is marginally less elegant than the approach taken by Smalltalk and other languages (simply referring to "my super class" instead of using an explicit name). However, the C++ approach extends cleanly to multiple inheritance.

In this context it might be worth noting that `Y::f` means "the `f` from class `Y` or one of `Y`'s base classes" and not simply "the `f` declared in class `Y`." For example:

```
class X { public: int f(); };
class Y : public X { };
class Z : public Y { public: f(); };

int Z::f() { return Y::f(); }   // calls the X::f()
```

5.2 Multiple Inclusion

A class can appear more than once in an inheritance DAG:

```
class A : public L { /* ... */ };
class B : public L { /* ... */ };
class C : public A, public B { /* ... */ };
```

In this case, an object of class `C` has two sub-objects of class `L`, namely `A::L` and `B::L`. This is often useful, as in the case of an implementation of lists requiring each element on a list to contain a link element. If, in the example above, `L` is a link class, then a `C` can be on both the list of `A`s and the list of `B`s at the same time.

Virtual functions work as expected; that is, the version from the most derived class is used:

```
class A { public: virtual f(); /* ... */ };
class B { public: virtual g(); /* ... */ };
class C : public A, public B
     { public: f(); g(); /* ... */ };

void ff()
{
    C obj;
    A* pa = &obj;
    B* pb = &obj;

    pa->f();        // calls C::f
    pb->g();        // calls C::g
}
```

This way of combining classes is ideal for representing the union of independent or nearly independent concepts. However, in some interesting cases, such as the window example, a more explicit way of expressing sharing and dependency is needed.

5.3 Virtual Base Classes

Virtual base classes provide a mechanism for sharing between sub-objects in an inheritance DAG and for expressing dependencies among such sub-objects:

```
class A : public virtual W { /* ... */ };
class B : public virtual W { /* ... */ };
class C : public A, public B,
     public virtual W { /* ... */ };
```

In this case there is only one object of class `W` in class `C`.

A virtual base class is considered an immediate virtual base class of every class directly or indirectly derived from it. Therefore, the explicit specification of `W` as a base of `C` is redundant. Class `C` could equivalently be declared like this:

```
class C : public A, public B { /* ... */ };
```

I prefer to mention the virtual base explicitly, though, since the presence of a virtual base typically affects the way member functions are programmed (see below).

Constructing the tables for virtual function calls can get quite complicated when virtual base classes are used. However, virtual functions work as usual by choosing the version from the most derived class in a call:

```
class W {
    // ...
public:
    virtual void f();
    virtual void g();
    virtual void h();
    virtual void k();
    // ...
};

class AW : public virtual W
      { /* ... */ public: void g(); /* ... */ };
class BW : public virtual W
      { /* ... */ public: void f(); /* ... */ };

class CW : public AW, public BW, public virtual W {
    // ...
public:
    void h();
    // ...
};

CW* pcw = new CW;

pcw->f();                // invokes BW::f()
pcw->g();                // invokes AW::g()
pcw->h();                // invokes CW::h()
((AW*)pcw)->f();         // invokes BW::f() !!!
```

The reason that `BW::f()` is invoked in the last example is that the only `f()` in an object of class `CW` is the one found in the (shared) sub-object `W`, and that one has been overridden by `BW::f()`.

Ambiguities are easily detected at the point where `CW`'s table of virtual functions is constructed. The rule for detecting ambiguities in a class DAG is that all re-definitions of a virtual function from a virtual base class must occur on a single path through the DAG. The example above can be drawn like this:

$$\begin{tikzcd}
 & W\{\ f\,g\,h\,k\ \} & \\
AW\{\ g\ \} \arrow[ur] & & BW\{\ f\ \} \arrow[ul] \\
 & CW\{\ h\ \} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Note that a call "up" through one path of the DAG to a virtual function may result in the call of a function (re-defined) in another path (as happened in the call `((AW*)pcw)->f()` in the example above). In this example, an ambiguity would occur if a function `f()` was added to `AW`. This ambiguity might be resolved by adding a function `f()` to `CW`.

Programming with virtual bases is trickier than programming with non-virtual bases. The problem is to avoid multiple calls of a function in a virtual class when that is not desired. Here is a possible style:

```
class W {
    // ...
protected:
    _f() { my stuff }
    // ...
public:
    f() { _f(); }
    // ...
};
```

Each class provides a protected function doing its "own stuff," `_f()`, for use by derived classes and a public function `f()` as the interface for use by the "general public."

```
class A : public virtual W {
    // ...
protected:
    _f() { my stuff }
    // ...
public:
    f() { _f(); W::_f(); }
    // ...
};
```

A derived class `f()` does its "own stuff" by calling `_f()` and its base classes' "own stuff" by calling their `_f()`s.

```
class B : public virtual W  {
    // ...
protected:
    _f() { my stuff }
    // ...
public:
    f() { _f(); W::_f(); }
    // ...
};
```

In particular, this style enables a class that is (indirectly) derived twice from a class `W` to call `W::f()` once only:

```
class C : public A, public B, public virtual W {
    // ...
protected:
    _f() { my stuff }
    // ...
public:
    f() { _f(); A::_f(); B::_f(); W::_f(); }
    // ...
};
```

Method combination schemes, such as the ones found in Lisp systems with multiple inheritance, were considered as a way of reducing the amount of code a programmer needed to write in cases like the one above. However, none of these schemes appeared to be sufficiently simple, general, and efficient enough to warrant the complexity it would add to C++.

5.4 Time and Space Efficiency

As described in Stroustrup [1987], a virtual function call is about as efficient as a normal function call – even in the case of multiple inheritance. The added cost is 5 to 6 memory references per call. This compares with the 3 to 4 extra memory references incurred by a virtual function call in a C++ compiler providing only single inheritance. The multiple inheritance scheme currently used causes an increase of about 50% in the size of the tables used to implement the virtual functions compared with the older single inheritance implementation. To offset that, the multiple inheritance implementation optimizes away quite a few spurious tables generated by the older single-inheritance implementations so that the memory requirement of a program using virtual functions actually decreases in most cases.

It would have been nice if there had been absolutely no added cost for the multiple inheritance scheme when only single inheritance is used. Such schemes exist, but involve the use of tricks that cannot be done by a C++ compiler generating C.

6. *Base and Member Initialization*

The syntax for initializing base classes and members has been extended to cope with multiple inheritance and the order of initialization has been more precisely defined. Leaving the initialization order unspecified in the original definition of C++ gave an unnecessary degree of freedom to language implementers at the expense of the users. In most cases, the order of initialization of members doesn't matter and in most cases where it does matter, the order dependency is an indication of bad design. In a few cases, however, the programmer absolutely needs control of the order of initialization. For example, consider transmitting objects between machines. An object must be reconstructed by a receiver in exactly the reverse order in which it was decomposed for transmission by a sender. This cannot be guaranteed for objects communicated between programs compiled by compilers from different suppliers unless the language specifies the order of construction.

Consider:

```
class A { public: A(int); A(); /* ... */ };
class B { public: B(int); B(); /* ... */ };

class C : public A, public B {
    const a;
    int& b;
public:
    C(int&);
};
```

In a constructor the sub-objects representing base classes can be referred to by their class names:

```
C::C(int& rr) : A(1), B(2), a(3), b(rr) { /* ... */ }
```

The initialization takes place in the order of declaration in the class with base classes initialized before members,[5] so the initialization order for class `C` is `A`, `B`, `a`, `b`. This order is independent of the order of explicit initializers, so

5. Virtual base classes force a modification to this rule; see below.

```
C::C(int& rr) : b(rr), B(2), a(3), A(1) { /* ... */ }
```

also initializes in the declaration order `A`, `B`, `a`, `b`.

The reason for ignoring the order of initializers is to preserve the usual FIFO ordering of constructor and destructor calls. Allowing two constructors to use different orders of initialization of bases and members would constrain implementations to use more dynamic and more expensive strategies.

Using the base class name explicitly clarifies even the case of single inheritance without member initialization:

```
class vector {
    // ...
public:
    vector(int);
    // ...
};

class vec : public vector {
    // ...
public:
    vec(int,int);
    // ...
};
```

It is reasonably clear even to novices what is going on here:

```
vec::vec(int low, int high) :
    vector(high-low-1) { /* ... */ }
```

On the other hand, this version:

```
vec::vec(int low, int high) :
    (high-low-1) { /* ... */ }
```

has caused much confusion over the years. The old-style base class initializer is of course still accepted. It can be used only in the single inheritance case since it is ambiguous otherwise.

A virtual base is constructed before any of its derived classes. Virtual bases are constructed before any non-virtual bases and in the order they appear on a depth-first left-to-right traversal of the inheritance DAG. This rule applies recursively for virtual bases of virtual bases.

A virtual base is initialized by the "most derived" class of which it is a base. For example:

```
class V { public: V(); V(int); /* ... */ };
class A : public virtual V
     { public: A(); A(int); /* ... */ };
class B : public virtual V
     { public: B(); B(int); /* ... */ };
class C : public A, public B
     { public: C(); C(int); /* ... */ };

A::A(int i) : V(i) { /* ... */ }
B::B(int i) { /* ... */ }
C::C(int i) { /* ... */ }

    V v(1); // use V(int)
    A a(2); // use V(int)
    B b(3); // use V()
    C c(4); // use V()
```

The order of destructor calls is defined to be the reverse order of appearance in the class declaration (members before bases). There is no way for the programmer to control this order – except by the declaration order. A virtual base is destroyed after all of its derived classes.

It might be worth mentioning that virtual destructors are (and always have been) allowed:

```
struct B { /* ... */ virtual ~B(); };

struct D : B { ~D(); };

void g() {
    B* p = new D;
    delete p;        // D::~D() is called
}
```

The word `virtual` was chosen for virtual base classes because of some rather vague conceptual similarities to virtual functions and to avoid introducing yet another keyword.

7. *Abstract Classes*

One of the purposes of static type checking is to detect mistakes and inconsistencies before a program is run. It was noted that a significant class of detectable errors was escaping C++'s checking. To add insult to injury, the language actually forced programmers to write extra code and generate larger programs to make this happen.

Consider the classic "shape" example. Here, we must first declare a class `shape` to represent the general concept of a shape. This class needs two virtual functions `rotate()` and `draw()`. Naturally, there can be no objects of class `shape`, only objects of specific shapes. Unfortunately C++ did not provide a way of expressing this simple notion.

The C++ rules specify that virtual functions, such as `rotate()` and `draw()`, must be defined in the class in which they are first declared. The reason for this requirement is to ensure that traditional linkers can be used to link C++ programs and to ensure that it is not possible to call a virtual function that has not been defined. So the programmer writes something like this:

```
class shape {
    point center;
    color col;
    // ...
public:
    where() { return center; }
    move(point p) { center=p; draw(); }
    virtual void rotate(int)
   { error("cannot rotate"); abort(); }
    virtual void draw()
   { error("cannot draw"); abort(); }
    // ...
};
```

This ensures that innocent errors such as forgetting to define a `draw()` function for a class derived from `shape` and silly errors such as creating a "plain" `shape` and attempting to use it cause run time errors. Even when such errors are not made, memory can easily get cluttered with unnecessary virtual tables for classes such as `shape` and with functions that are never called, such as `draw()` and `rotate()`. The overhead for this can be noticeable.

The solution is simply to allow the user to say that a virtual function does not have a definition; that is, it is a "pure virtual function." This is done by an initializer `=0`:

```
class shape {
    point center;
    color col;
    // ...
public:
    where() { return center; }
    move(point p) { center=point; draw(); }
                                // pure virtual function
    virtual void rotate(int) = 0;
    virtual void draw() = 0;  // pure virtual function
    // ...
};
```

A class with one or more pure virtual functions is an abstract class. An abstract class can only be used as a base for another class. In particular, it is not possible to create objects of an abstract class. A class derived from an abstract class must either define the pure virtual functions from its base or again declare them to be pure virtual functions.

The notion of pure virtual functions was chosen over the idea of explicitly declaring a class to be abstract because the selective definition of functions is much more flexible.

8. Static Member Functions

A static data member of a class is a member for which there is only one copy rather than one per object and which can be accessed without referring to any particular object of the class it is a member of. The reason for using static members is to reduce the number of global names, to make obvious which static objects logically belong to which class, and to be able to apply access control to their names. This is a boon for library providers since it avoids polluting the global name space and thereby allows easier writing of library code and safer use of multiple libraries.

These reasons apply for functions as well as for objects. In fact, *most* of the names a library provider wants to be local are function names. It was also observed that nonportable code, such as

```
((X*)0)->f();
```

was used to simulate static member functions. This trick is a time bomb because sooner or later someone will make an `f()` that is used this way `virtual` and the call will fail horribly because there is no `X` object at address zero. Even where `f()` is not virtual such calls will fail under some implementations of dynamic linking.

A static member function is a member so that its name is in the class scope and the usual access control rules apply. A static member function is not associated with any particular object and need not be called using the special member function syntax. For example:

```
class X {
    int mem;
    static int smem;
public:
    static void f(int,X*);
};

void g()
{
    X obj;
    f(1,&obj);        // error (unless there really is
                      // a global function f())
    X::f(1,&obj);     // fine
    obj.f(1,&obj);    // also fine
}
```

Since a static member function isn't called for a particular object it has no `this` pointer and cannot access non-static members without explicitly specifying an object. For example:

```
void X::f(int i, X* p)
{
    mem = i;        // error: which mem?
    p->mem = i;     // fine
    smem++;         // fine: only one smem
}
```

9. const Member Functions

Consider this example:

```
class s {
    int aa;
public:
    void mutate() { aa++; }
    int value() { return aa; }
};
```

```
void g()
{
    s o1;
    const s o2;
    o1.mutate();
    o2.mutate();
    int i = o1.value() + o2.value();
}
```

It seems clear that the call `o2.mutate()` ought to fail since `o2` is a `const`.

The reason this rule has not been enforced until now is simply that there was no way of distinguishing a member function that may be invoked on a `const` object from one that may not. In general, the compiler cannot deduce which functions will change the value of an object. For example, had `mutate()` been defined in a separately compiled source file the compiler would not have been able to detect the problem at compile time.

The solution to this has two parts. First, `const` is enforced so that "ordinary" member functions cannot be called for a `const` object. Then we introduce the notion of a `const` member function, that is, a member function that may be called for all objects including `const` objects. For example:

```
class X {
    int aa;
public:
    void mutate() { aa++; }
    int value() const { return aa; }
};
```

Now `X::value()` is guaranteed not to change the value of an object and can be used on a `const` object whereas `X::mutate()` can only be called for non-`const` objects:

```
int g()
{
    X o1;
    const X o2;
    o1.mutate();       // fine
    o2.mutate();       // error
    return o1.value() + o2.value(); // fine
}
```

In a `const` member function of `X` the `this` pointer points to a `const X`. This ensures that non-devious attempts to modify the value of an object through a `const` member will be caught:

```
class X {
    int a;
    void cheat() const { a++; } // error
};
```

Note that the use of `const` as a suffix to `()` is consistent with the use of `const` as a suffix to `*`.

It is occasionally useful to have objects that appear as constants to users but do in fact change their state. Such classes can be written using explicit casts:

```
class XX {
    int a;
    int calls_of_f;
    int f() const { ((XX*)this)->calls_of_f++;
   return a; }
    // ...
};
```

Since this can be quite deceptive and is error-prone in some contexts it is often better to represent the variable part of such an object as a separate object:

```
class XX {
    int a;
    int& calls_of_f;
    int f() const { calls_of_f++; return a; }
    // ...
    XX() : calls_of_f(*new int) { /* ... */ }
    ~XX() { delete &calls_of_f; /* ... */ }
    // ...
};
```

10. Initialization of static Members

A `static` data member of a class must be defined somewhere. The `static` declaration in the class declaration is only a declaration and does not set aside storage or provide an initializer.

This is a change from the original C++ definition of `static` members, which relied on implicit definition of `static` members and on implicit initialization of such members to `0`. Unfortunately, this style of initialization cannot be used for objects of all types. In particular, objects of classes with constructors cannot be initialized this way. Furthermore, this style of initialization relied on linker features that are not universally available. Fortunately, in the implementations where this used to work it will continue to work for some time, but conversion to the stricter style described here is strongly recommended.

Here is an example:

```
class X {
    static int i;
    int j;
    X(int);
    int read();
};

class Y {
    static X a;
    int b;
    Y(int);
    int read();
};
```

Now `X::i` and `Y::a` have been declared and can be referred to, but somewhere definitions must be provided. The natural place for such definitions is with the definitions of the class member functions. For example:

```
// file X.c:
X::X(int jj) { j = jj; }
int X::read() { return j; }
int X::i = 3;

// file Y.c:
Y::Y(int bb) { b = bb; }
int Y::read() { return b; }
X Y::a = 7;
```

11. Pointers to Members

As mentioned in Stroustrup 1986, it was an obvious deficiency in C++ that there was no way of expressing the concept of a pointer to a member of a class. This led to the need to "cheat" the type system in cases, such as error handling, where pointers to functions are traditionally used. Consider this example:

```
struct S {
    int mf(char*);
};
```

The structure `S` is declared to be a (trivial) type for which the member function `mf` is declared. Given a variable of type `S` the function `mf` can be called:

```
S a;
int i = a.mf("hello");
```

The question is "*What is the type of* `mf`*?*"

The equivalent type of a non-member function

```
int f(char*);
```

is

```
int (char*)
```

and a pointer to such a function is of type

```
int (*)(char*)
```

Such pointers to "normal" functions are declared and used like this:

```
int f(char*);          // declare function
          // declare and initialize pointer to function
int (*pf)(char*) = &f;
int i = (*pf)("hello");// call function through pointer
```

A similar syntax is introduced for pointers to members of a specific class. In a definition `mf` appears as:

```
int S::mf(char*)
```

The type of `S::mf` is:

```
int S:: (char*)
```

That is, "member of `S` that is a function taking a `char*` argument and returning an `int`." A pointer to such a function is of type

```
int (S::*)(char*)
```

That is, the notation for pointer to member of class `S` is `S::*`. We can now write:

```
          // declare and initialize pointer to member function
        int (S::*pmf)(char*) = &S::mf;

        S a;
          // call function through pointer for the object 'a'
        int i = (a.*pmf)("hello");
```

The syntax isn't exactly pretty, but neither is the C syntax it is modeled on.

A pointer to member function can also be called given a pointer to an object:

```
        S* p;
          // call function through pointer for the object '*p':
        int i = (p->*pmf)("hello");
```

In this case, we might have to handle virtual functions:

```
        struct B {
            virtual f();
        };

        struct D : B {
            f();
        };

        int ff(B* pb, int (B::*pbf)())
        {
            return (pb->*pbf)();
        };

        void gg()
        {
            D dd;
            int i = ff(&dd, &B::f);
        }
```

This causes a call of `D::f()`. Naturally, the implementation involves a lookup in `dd`'s table of virtual functions exactly as a call to a virtual function that is identified by name rather than by a pointer. The overhead compared to a "normal function call" is the usual (about 5 memory references [dependent on the machine architecture]).

It is also possible to declare and use pointers to members that are not functions:

```
struct S {
    int mem;
};

int S::* psm = &S::mem;

void f(S* ps)
{
    ps->*psm = 2;
}

void g()
{
    S a;
    f(&a);
}
```

This is a complicated way of assigning `2` to `a.mem`.

Pointers to members are described in greater detail in Lippman & Stroustrup 1988.

12. *User-Defined Free Store Management*

C++ provides the operators `new` and `delete` to allocate memory on the free store and to release store allocated this way for reuse. Occasionally a user needs a finer-grained control of allocation and deallocation. The first section below shows "the bad old way" of doing this and the following sections show how the usual scope and overloaded function resolution mechanisms can be exploited to achieve similar effects more elegantly. This means that assignment to `this` is an anachronism and will be removed from the implementations of C++ after a decent interval. This will allow the type of `this` in a member function of class `X` to be changed to `X *const`.

12.1 Assignment to this

Formerly, if a user wanted to take over allocation of objects of a class `X` the only way was to assign to `this` on each path through every constructor for `X`. Similarly, the user could take control of deallocation by assigning to `this` in a destructor. This is a very powerful and general mechanism. It is also non-obvious, error prone, repetitive, too subtle when derived classes are used, and essentially unmanageable when multiple inheritance is used. For example:

```
class X {
    int on_free_store;
    // ...
public:
    X();
    X(int i);
    ~X();
    // ...
}
```

Every constructor needs code to determine when to use the user-defined allocation strategy:

```
X::X() {
    if (this == 0) {        // 'new' used
        this = myalloc(sizeof(X));
        on_free_store = 1;
    }
    else { // static, automatic, or member of aggregate
      // forget this assignment at your peril
        this = this;
        on_free_store = 0;
    }
    // initialize
}
```

The assignments to `this` are "magic" in that they suppress the usual compiler generated allocation code.

Similarly, the destructor needs code to determine when to use the user-defined deallocation strategy and must use an assignment to `this` to indicate that it has taken control over deallocation:

```
X::~X() {
    // cleanup
    if (on_free_store) {
        myfree(this);
          // forget this assignment at your peril
        this = 0;
    }
}
```

This user-defined allocation and deallocation strategy isn't inherited by derived classes in the usual way.

The fundamental problem with the "assign to `this`" approach to user-controlled memory management is that initialization and memory management code are intertwined in an ad hoc manner. In particular, this implies that the language cannot provide any help with these critical activities.

12.2 Class-Specific Free Store Management

The alternative is to overload the allocation function `operator new()` and the deallocation function `operator delete()` for a class `X`:

```
class X {
    // ...
public:
    void* operator new(size_t sz)
   { return myalloc(sz); }
    void operator delete(X* p) { myfree(p); }

    X() { /* initialize */ }
    X(int i) { /* initialize */ }

    ~X() { /* cleanup */ }

    // ...
};
```

The type `size_t` is an implementation-defined integral type used to hold object sizes.[6] It is the type of the result of `sizeof`.

Now `X::operator new()` will be used instead of the global `operator new()` for objects of class `X`. Note that this does not affect other uses of operator `new` within the scope of `X`:

```
void* X::operator new(size_t s)
{
                        // global operator new as usual
    void* p = new char[s];
    //...
    return p;
}
```

6. `operator new()` used to require a `long`; `size_t` was adopted to bring C++ allocation mechanisms into line with ANSI C.

```
void X::operator delete(X* p)
{
    //...
    delete (void*) p;// global operator delete as usual
}
```

When the new operator is used to create an object of class X, operator new() is found by a lookup starting in X's scope so that X::operator new() is preferred over a global ::operator new().

12.3 *Inheritance of operator new()*

The usual rules for inheritance apply:

```
                // objects of class Y are also allocated
class Y : public X
{                          // using X::operator new
    // ...
};
```

This is the reason X::operator new() needs an argument specifying the amount of store to be allocated; sizeof(Y) is typically different from sizeof(X). Naturally, a class that is never a base class need not use the size argument:

```
void* Z::operator new(size_t) { return next_free_Z(); }
```

This optimization should not be used unless the programmer is perfectly sure that Z is never used as a base class, because if it is, disaster will happen.

An operator new(), be it local or global, is used only for free store allocation, so

```
X a1;               // allocated statically

void f()
{
    X a;        // allocated on the stack
    X v[10];    // allocated on the stack
}
```

does not involve any operator new(). Instead, store is allocated statically and on the stack.

X::operator new() is only used for individual objects of class X (and objects of classes derived from class X that do not have their own operator new()), so

```
X* p = new X[10];
```

does not involve `X::operator new()` because `X[10]` is an array.

Like the global `operator new()`, `X::operator new()` returns a `void*`. This indicates that it returns uninitialized memory. It is the job of the compiler to ensure that the memory returned by this function is converted to the proper type and – if necessary – initialized using the appropriate constructor. This is exactly what happens for the global `operator new()`.

`X::operator new()` and `X::operator delete()` are `static` member functions. In particular, they have no `this` pointer. This reflects the fact that `X::operator new()` is called before constructors so that initialization has not yet happened and `X::operator delete()` is called after the destructor so that the memory no longer holds a valid object of class `X`.

12.4 Overloading operator new()

Like other functions, `operator new()` can be overloaded. Every `operator new()` must return a `void*` and take a `size_t` as its first argument. For example:

```
void* operator new(size_t sz);  // the usual allocator

void* operator new(size_t sz, heap* h)
{                               // allocate from heap 'h'
    return h->allocate(sz);
}

void* operator new(size_t, void* p)
{                               // place object at 'p'
    return p;
}
```

The size argument is implicitly provided when operator `new` is used. Subsequent arguments must be explicitly provided by the user. The notation used to supply these additional arguments is an argument list placed immediately after the `new` operator itself.

```
static char buf [sizeof(X)];    // static buffer

class heap {
    // ...
};

heap h1;
```

```
f() {
    X* p1 = new X; // use the default allocator
                   // operator new(size_t sz):
                   // operator new(sizeof(X))

    X* p3 = new(&h1) X;  // use h1's allocator
                   // operator new(size_t sz, heap* h):
                   // operator new(sizeof(X),&h1)

    X* p2 = new(buf) X; // explicit allocation in 'buf'
                   // operator new(size_t, void* p):
                   // operator new(sizeof(X),buf)
}
```

Note that the explicit arguments go after the `new` operator but before the type. Arguments after the type go to the constructor as ever. For example:

```
class Y {
    void* operator new(size_t, const char*);
    Y(const char*);
};

Y* p = new("string for the allocator")
       Y("string for the constructor");
```

12.5 Controlling Deallocation

Where many different `operator new()` functions are used one might imagine that one would need many different and matching `operator delete()` functions. This would, however, be quite inconvenient and often unmanageable. The fundamental difference between creation and deletion of objects is that at the point of creation the programmer knows just about everything worth knowing about the object whereas at the point of deletion the programmer holds only a pointer to the object. This pointer may not even give the exact type of the object, but only a base class type. It will therefore typically be unreasonable to require the programmer writing a `delete` to choose among several variants.[7]

7. The requirement that a programmer must distinguish between `delete p` for an individual object and `delete[n] p` for an array is an unfortunate hack and is mitigated only by the fact that there is nothing that forces a programmer to use such arrays.

Consider a class with two allocation functions and a single deallocation function that chooses the proper way of deallocating based on information left in the object by the allocators:

```
class X {
    enum { somehow, other_way } which_allocator;

    void* operator new(size_t sz)
    {   void* p = allocate_somehow();
        ((X*)p)->which_allocator = somehow;
        return p;
    }

    void* operator new(size_t sz , int i)
    {   void* p = allocate_some_other_way();
        ((X*)p)->which_allocator = other_way;
        return p;
    }

    void operator delete(void*);
    // ...
};
```

Here `operator delete()` can look at the information left behind in the object by the `operator new()` and deallocate appropriately:

```
void X::operator delete(void* p)
{
    switch (((X*)p)->which_allocator) {
    case somehow:
        deallocate_somehow();
        break;
    case other_way:
        deallocate_some_other_way();
        break;
    default:
            /* something is funny */
    }
}
```

Since `operator new()` and `operator delete()` are static member functions they need to cast their "object pointers" to use member names. Furthermore, these functions will be invoked only by explicit use of operators `new` and `delete`. This implies that `X::which_allocator` is not initialized for automatic objects so in that case it may have an arbitrary value. In particular, the default case in `X::operator delete()` might occur if someone tried to `delete` an automatic (on the stack) object.

Where (as will often be the case) the rest of the member functions of `X` have no need for examining the information stored by allocators for use by the deallocator, this information can be placed in storage outside the object proper ("in the container itself"), thus decreasing the memory requirement for automatic and static objects of class `X`. This is exactly the kind of game played by "ordinary" allocators such as the C `malloc()` for managing free store.

The example of the use of assignment to `this` above contains code that depends on knowing whether the object was allocated by `new` or not. Given local allocators and deallocators, it is usually neither wise nor necessary to do so. However, in a hurry or under serious compatibility constraints, one might use a technique like this:

```
class X {
    static X* last_X;
    int on_free_store;
    // ...

    X();

    void* operator new(long s)
    {
        return last_X = allocate_somehow();
    }

    // ...
};

X::X()
{
    if (this == last_X) { // on free store
        on_free_store = 1;
    }
    else {// static or automatic or member of aggregate
        on_free_store = 0;
    }
    // ...
}
```

Note that there is no simple and implementation-independent way of determining that an object is allocated on the stack. There never was.

12.6 Placement of Objects

For ordinary functions it is possible to specifically call a non-member version of the function by prefixing a call with the scope resolution operator `::`. For example,

```
::open(filename,"rw");
```

calls the global `open()`. Prefixing a use of the `new` operator with `::` has the same effect for `operator new()`; that is,

```
X* p = ::new X;
```

uses a global `operator new()` even if a local `X::operator new()` has been defined. This is useful for placing objects at specific addresses (to cope with memory mapped I/O, etc.) and for implementing container classes that manage storage for the objects they maintain. Using `::` ensures that local allocation functions are not used and the argument(s) specified for `new` allows selection among several global `operator new()` functions. For example:

```
// place object at address p:
void* operator new(size_t, void* p) { return p; }

char buf [sizeof(X)];    // static buffer

f()
{
    X* p = ::new(buf) X;// explicit allocation in 'buf'
                        // place an X at address 0777
    p = ::new((void*)0777) X;
}
```

Naturally, for most classes the `::` will be redundant since most classes do not define their own allocators. The notation `::delete` can be used similarly to ensure use of a global deallocator.

12.7 Memory Exhaustion

Occasionally, an allocator fails to find memory that it can return to its caller. If the allocator must return in this case, it should return the value `0`. A constructor will return immediately upon finding itself called with `this==0` and the complete `new` expression will yield the value `0`. In the absence of more elegant error handling schemes, this enables critical software to defend itself against allocation problems. For example:

```
void f()
{
    X* p = new X;
    if (p == 0) { /* handle allocation error */ }
    // use p
}
```

The use of a `new_handler` [Stroustrup 1986] can make most such checks unnecessary.

12.8 Explicit Calls of Destructors

Where an object is explicitly "placed" at a specific address or in some other way allocated so that no standard deallocator can be used, there might still be a need to destroy the object. This can be done by an explicit call of the destructor:

```
p->X::~X();
```

The fully qualified form of the destructor's name must be used to avoid potential parsing ambiguities. This requirement also alerts the user that something unusual is going on. After the call of the destructor, `p` no longer points to a valid object of class `X`.

12.9 Size Argument to operator delete()

Like `X::operator new()`, `X::operator delete()` can be overloaded, but since there is no mechanism for the user to supply arguments to a deallocation function this overloading simply presents the programmer with a way of using the information available in the compiler. `X::operator delete()` can have two forms (only):

```
class X {
    void operator delete(void* p);
    void operator delete(void* p, size_t sz);
    // ...
};
```

If the second form is present it will be preferred by the compiler and the second argument will be the size of the object to the best of the compiler's knowledge. This allows a base class to provide memory management services for derived classes:

```
class X {
    void* operator new(size_t sz);
    void operator delete(void* p, size_t sz);

    virtual ~X();
    // ...
};
```

The use of a virtual destructor is crucial for getting the size right in cases where a user deletes an object of a derived class through a pointer to the base class:

```
class Y : public X {
    // ...
    ~Y();
};

X* p = new Y;
delete p;
```

13. Assignment and Initialization

C++ originally had assignment and initialization default defined as bitwise copy of an object. This caused problems when an object of a class with assignment was used as a member of a class that did not have assignment defined:

```
class X {
    // ...
public:
    X& operator=(const X&);
    // ...
};

class Y {
    X a;
    // ...
};

void f()
{
    Y y1, y2;
    // ...
```

```
        y1 = y2;
    }
```

Assuming that assignment was not defined for `Y`, `y2.a` is copied into `y1.a` with a bitwise copy. This invariably turns out to be an error and the programmer has to add an assignment operator to class `Y`:

```
    class Y {
        X a;
        // ...
        const Y& operator=(const Y& arg)
        {
            a = arg.a;
            // ...
        }
    };
```

To cope with this problem in general, assignment in C++ is now defined as memberwise assignment of non-static members and base class objects.[8] Naturally, this rule applies recursively until a member of a built-in type is found. This implies that for a class `X`, `X(const X&)` and `const X& X::operator=(const X&)` will be supplied where necessary by the compiler, as has always been the case for `X::X()` and `X::~X()`. In principle every class `X` has `X::X()`, `X::X(const X&)`, and `X::operator=(const X&)` defined. In particular, defining a constructor `X::X(T)` where `T` isn't a variant of `X&` does not affect the fact that `X::X(const X&)` is defined. Similarly, defining `X::operator=(T)` where `T` isn't a variant of `X&` does not affect the fact that `X::operator=(const X&)` is defined.

To avoid nasty inconsistencies between the predefined `operator=()` functions and user defined `operator=()` functions, `operator=()` must be a member function. Global assignment functions such as `::operator=(X&, const X&)` are anachronisms and will be disallowed after a decent interval.

Note that since access controls are correctly applied to both implicit and explicit copy operations we actually have a way of prohibiting assignment of objects of a given class `X`:

8. One could argue that the original definition of C++ was inconsistent in requiring bitwise copy of objects of class `Y`, yet guaranteeing that `X::operator=()` would be applied for copying objects of a class `X`. In this case both guarantees cannot be fulfilled.

```
class X {
    // Objects of class X cannot be copied
    // except by members of X
    void operator=(X&);
    X(X&);
    // ...
public:
    X(int);
    // ...
};

void f() {
    X a(1);
    X b = a;        // error: X::X(X&) private
    b = a;          // error: X::operator=(X&) private
}
```

The automatic creation of `X::X(const X&)` and `X::operator=(const X&)` has interesting implications on the legality of some assignment operations. Note that if `X` is a public base class of `Y` then a `Y` object is a legal argument for a function that requires an `X&`. For example:

```
class X { public: int aa; };
class Y : public X { public: int bb; };

void f() {
    X xx;
    Y yy;
    xx = yy;  // ok: a Y is an X
              //     xx=yy; means xx.operator=((X&)yy);
              //     and is optimized to xx.aa = yy.aa
}
```

Defining assignment as memberwise assignment implies that `operator=()` isn't inherited in the ordinary manner. Instead, the appropriate assignment operator is – if necessary – generated for each class. This implies that the "opposite" assignment of an object of a base class to a variable of a derived class is illegal as ever:

```
void f() {
    X xx;
    Y yy;
    yy = xx;    // error: an X is not a Y
}
```

The extension of the assignment semantics to allow assignment of an object of a derived class to a variable of a public base class had been repeatedly requested by users. The direct connection to the recursive memberwise assignment semantics became clear only through work on the two apparently independent problems.

14. Operator ->

Until now -> has been one of the few operators a programmer couldn't define. This made it hard to create classes of objects intended to behave like "smart pointers." When overloading, -> is considered a unary operator (of its left hand operand) and -> is reapplied to the result of executing `operator->()`. Hence the return type of an `operator->()` function must be a pointer to a class or an object of a class for which `operator->()` is defined. For example:

```
struct Y { int m; };

class X {
    Y* p;
    // ...
    Y* operator->() {
        if (p == 0) {
                // initialize p
        }
        else {
                // check p
        }
        return p;
    }
    // ...
};
```

Here, class `X` is defined so that objects of type `X` act as pointers to objects of class `Y`, except that some suitable computation is performed on each access.

```
void f(X x, X& xr, X* xp)
{
    x->m;   // x.p->m
    xr->m;  // xr.p->m
```

```
        xp->m;  // error: X does not have a member m
    }
```

Like `operator=()`, `operator[]()`, and `operator()()`, `operator->()` must be a member function (unlike `operator+()`, `operator-()`, `operator<()`, etc., that are often most useful as `friend` functions).

The dot operator still cannot be overloaded.

For ordinary pointers, use of `->` is synonymous with some uses of unary `*` and `[]`. For example, for

```
    Y* p;
```

it holds that:

```
    p->m == (*p).m == p[0].m
```

As usual, no such guarantee is provided for user-defined operators. The equivalence can be provided where desired:

```
    class X {
        Y* p;
    public:
        Y* operator->() { return p; }
        Y& operator*() { return *p; }
        Y& operator[](int i) { return p[i]; }
    };
```

If you provide more than one of these operators it might be wise to provide the equivalence, exactly as it is wise to ensure that `x+=1` has the same effect as `x=x+1` for a simple variable `x` of some class `X` if `+=`, `=`, and `+` are provided.

The overloading of `->` is important to a class of interesting programs, just like overloading `[]`, and not just a minor curiosity. The reason is that *indirection* is a key concept and that overloading `->` provides a clean, direct, and efficient way of representing it in a program. Another way of looking at operator `->` is to consider it a way of providing C++ with a limited, but very useful, form of *delegation* [Gul 1986].

15. Operator ,

Until now the comma operator `,` has been one of the few operators a programmer couldn't define. This restriction did not appear to have any purpose so it has been removed. The most obvious use of an overloaded comma operator is list building:

```
class X { /* ... */ };

class Xlist {
    // ...
public:
    Xlist();
    Xlist(const X&);
    friend Xlist operator,(const X&, const X&);
};

void f()
{
    X a,b,c;
    Xlist xl = (a,b,c);
                // meaning operator,(operator,(a,b),c)
}
```

If you have a bit of trouble deciding which commas mean what in this example you have found the reason overloading of comma was originally left out.

16. Initialization of static Objects

In C, a static object can only be initialized using a slightly extended form of constant expressions. In C++, it has always been possible to use completely general expressions for the initialization of static class objects. This feature has now been extended to static objects of all types. For example:

```
#include <math.h>

double sqrt2 = sqrt(2);

main()
{
    if (sqrt(2)!=sqrt2) abort();
}
```

Such dynamic initialization is done in declaration order within a file and before the first use of any object or function defined in the file. No order is defined for initialization of objects in different source files except that all static initialization takes place before any dynamic initialization.

17. Resolutions

This section does not describe additions to C++ but gives answers to questions that have been asked often and do not appear to have clear enough answers in the reference manual [Stroustrup 1986]. These resolutions involve slight changes compared to earlier rules. This was done to bring C++ closer to the ANSI C draft.

17.1 Function Argument Syntax

Like the C syntax, the C++ syntax for specifying types allows the type `int` to be implicit in some cases. This opens the possibility of ambiguities. In argument declarations, C++ chooses the longest type possible when there appears to be a choice:

```
typedef long I;
      // f1() takes an unnamed 'const long' argument
f1(const I);
      // f2() takes a 'const int' argument (called 'i')
f2(const i);
```

This rule applies to the `const` and `volatile` specifiers, but not to `unsigned`, `short`, `long`, or `signed`:

```
f3(unsigned int I);  // ok
f4(unsigned I);// ok: equivalent to f4(unsigned int I);
```

A type cannot contain two basic type specifiers so

```
f5(char I) { I++; }
f6(I I) { I++; }
```

are legal.

17.2 Declaration and Expression Syntax

There is an ambiguity in the C++ grammar involving *expression-statement*s and *declaration*s: an *expression-statement* with a "function style" explicit type conversion as its leftmost sub-expression can be indistinguishable from a *declaration* where the first *declarator* starts with a `(`. For example:

```
T(a);         // declaration or type conversion of 'a'
```

In those cases the *statement* is a *declaration.*

To disambiguate, the whole *statement* may have to be examined to determine if it is an *expression-statement* or a *declaration*. This disambiguates many examples. For example, assume `T` is the name of some type:

```
T(a)->m = 7;         // expression-statement
T(a)++;              // expression-statement
T(a,5)<<c;           // expression-statement
T(*d)(double(3));    // expression-statement

T(*e)(int);          // declaration
T(f)[];              // declaration
T(g) = { 1, 2 };     // declaration
```

The remaining cases are *declaration*s. For example:

```
T(a);          // declaration
T(*b)();       // declaration
T(c)=7;        // declaration
T(d),e,f=3;    // declaration
T(g)(h,2);     // declaration
```

The disambiguation is purely syntactic; that is, the meaning of the names, beyond whether they are names of types or not, is not used in the disambiguation.

This resolution has two virtues compared to alternatives: it is simple to explain and completely compatible with C. The main snag is that it is not well adapted to simple minded parsers, such as YACC, because the lookahead required to decide what is an *expression-statement* and what is a *declaration* in a statement context is not limited.

However, note that a simple lexical lookahead can help a parser disambiguate most cases. Consider analysing a *statement*; the troublesome cases look like this

```
T ( d-or-e )  tail
```

Here, *d-or-e* must be a *declarator*, an *expression*, or both for the statement to be legal. This implies that *tail* must be a semicolon, something that can follow a parenthesized *declarator* or something that can follow a parenthesized *expression*, that is, an *initializer*, `const`, `volatile`, `(` or `[` or a postfix or infix operator.

A user can explicitly disambiguate cases that appear obscure. For example:

```
void f()
{
    auto int(*p)();  // explicitly declaration
    (void) int(*p)();// explicitly expression-statement
    0,int(*p)();     // explicitly expression-statement
    (int(*p)());     // explicitly expression-statement
    int(*p)();       // resolved to declaration
}
```

17.3 Enumerators

An enumeration is a type. Each enumeration is distinct from all other types. The set of possible values for an enumeration is its set of enumerators. The type of an enumerator is its enumeration. For example:

```
enum wine { red, white, rose, bubbly };
enum beer { ale, bitter, lager, stout };
```

defines two types, each with a distinct set of 4 values.

```
wine w = red;
beer b = bitter;

w = b;    // error, type mismatch: beer assigned to wine
w = stout;// error, type mismatch: beer assigned to wine
w = 2;    // error, type mismatch: int assigned to wine
```

Each enumerator has an integer value and can be used wherever an integer is required; in such cases the integer value is used:

```
int i = rose;// the value of 'rose' (that is 2) is used
i = b;       // the value of 'b' is assigned to 'i'
```

This interpretation is stricter than what has been used in C++ until now and stricter than most C dialects. The reason for choosing it was ANSI C's requirement that enumerations be distinct types. Given that, the details follow from C++'s emphasis on type checking and the requirements of consistency to allow overloading, etc. For example:

```
int f(int);
int f(wine);

void g()
{
    f(i);       // f(int)
    f(w);       // f(wine)

    f(1);       // f(int)
    f(white);   // f(wine)

    f(b);       // f(int), standard conversion
                //         from beer to int used
}
```

C++'s checking of enumerations is stricter than ANSI C's, in that assignments of integers to enumerations are disallowed. As ever, explicit type conversion can be used:

```
w = wine(257);  /* caveat emptor */
```

An enumerator is entered in the scope in which the enumeration is defined. In this context, a class is considered a scope and the usual access control rules apply. For example:

```
class X {
    enum { x, y, z };
    // ...
public:
    enum { a, b, c };

    f(int i = a) { g(i+x); ... }
    // ...
}

void h() {
    int i = a;  // error: 'X::a' is not in scope
    i = X::a;   // ok
    i = X::x;   // error: 'X::x' is private
}
```

17.4 The const Specifier

Use of the `const` specifier on a non-local object implies that linkage is *internal* by default (`static`); that is, the object declared is local to the file in which it occurs. To give it external linkage it must be explicitly declared `extern`.

Similarly, `inline` implies that linkage is *internal* by default. External linkage can be obtained by explicit declaration:

```
extern const double g;
const double g = 9.81;

extern inline f(int);
inline f(int i) { return i+c; }
```

17.5 Function Types

It is possible to define function types that can be used exactly like other types, except that variables of function types cannot be defined – only variables of pointer to function types:

```
typedef int F(char*);  // function taking a char*
                       // argument and returning an int
F* pf;                 // pointer to such function
F f;    // error: no variables of function type allowed
```

Function types can be useful in friend declarations. Here is an example from the C++ task system:

```
class task : public scheduler {
friend SIG_FUNC_TYP sig_func;
          // the type of a function must be specified
          // in a friend function declaration
    // ...
}
```

The reason to use a `typedef` in the friend declaration `sig_func` and not simply to write the type directly is that the type of `signal()` is system dependent:

```
// BSD signal.h:
typedef void SIG_FUNC_TYP(int, int, sigcontext*);

// 9th edition signal.h:
typedef void SIG_FUNC_TYP(int);
```

Using the `typedef` allows the system dependencies to be localized where they belong: in the header files defining the system interface.

17.6 Lvalues

Note that the default definition of assignment of an `X` as a call of

```
X& operator=(const X&)
```

makes assignment of `X`s produce an lvalue. For uniformity, this rule has been extended to assignments of built-in types. By implication, `+=`, `-=`, `*=`, etc., now also produce lvalues. So – again by implication – do prefix `++` and `--` (but not the postfix versions of these operators).

In addition, the comma and `?:` can also produce lvalues. The result of a comma operation is an lvalue if its second operand is. The result of a `?:` operator is an lvalue provided both its second and third operands are and provided they have exactly the same type.

17.7 Multiple Name Spaces

C provides a separate name space for structure tags whereas C++ places type names in the same name space as other names. This gives important notational conveniences to the C++ programmer but severe headaches to people managing header files in mixed C/C++ environments. For example:

```
struct stat {
    // ...
};

extern "C" int stat(const char*, struct stat *);
```

was not legal C++ though early implementations accepted it as a compatibility hack. The experience has been that trying to impose the "'pure C++" single name space solution (thus outlawing examples such as the one above) has caused too much confusion and too much inconvenience to too many users. Consequently, a slightly cleaned up version of the C/C++ compatibility hack has now become part of C++. This follows the overall principle that where there is a choice between inconveniencing compiler writers and annoying users, the compiler writers should be inconvenienced.[9] It appears that the compromise provided by the rules presented below enables all accepted uses of multiple name spaces in C while preserving the notational convenience of C++ in all cases where C compatibility isn't an essential issue. In particular, every legal C++ program remains legal. The restrictions on the use of constructors and typedef names in connection with the use of multiple name spaces are imposed to prevent some nasty cases of hard to detect ambiguities that would cause trouble for the composition of C++ header files.

A typedef can declare a name to refer to the same type more than once. For example:

```
typedef struct s { /* ... */ } s;
typedef s s;
```

A name `s` can be declared as a type (struct, class, union, enum) *and* as a non-type (function, object, value, etc.) in a single scope. In this case, the name `s` refers to the non-type and `struct s` (or whatever) can be used to refer to the type. The order of declaration does not matter. This rule takes effect only after both declarations of `s` have been seen. For example:

```
struct stat { /* ... */ };
stat a;
void stat(stat* p);
        // struct is needed to avoid the function name
struct stat b;
stat(0);        // function call

int f(int);
f(1);
struct f { /* ... */ };
        // struct is needed to avoid the function name
struct f a;
```

A name cannot simultaneously refer to two types:

```
struct s { /* ... */ };
typedef int s;  // error
```

9. Sorry Jens, Mike, Mike, Mike, Phil, Walter, et al.

The name of a class with a constructor cannot also simultaneously refer to something else:

```
struct s { s(); /* ... */ };
int s();        // error

struct t* p;
int t();        // ok
int i = t();
struct t { t(); /* ... */ }     // error
i = t();
```

If a non-type name `s` hides a type name `s`, `struct s` can be used to refer to the type name. For example:

```
struct s { /* ... */ };
f(int s) { struct s a; s++; }
```

Note: if a type name hides a non-type name the usual scope rules apply:

```
int s;
f()
{
    struct s { /* ... */ };   // new 's' refers to the
                // type and the global int is hidden
    s a;
}
```

Use of the `::` scope resolution operator implies that its argument is a non-type name. For example:

```
int s;
f()
{
    struct s { /* ... */ };
    s a;
    ::s = 1;
}
```

17.8 Function Declaration Syntax

To ease the use of common C++ and ANSI C header files, `void` may be used to indicate that a function takes no arguments:

```
extern int f(void);       // same as 'extern int f();'
```

18. Conclusions

C++ is holding up nicely under the strain of large scale use in a diverse range of application areas. The extensions added so far have been have all been relatively easy to integrate into the C++ type system. The C syntax, especially the C declarator syntax, has consistently caused much greater problems than the C semantics; it remains barely manageable. The stringent requirements of compatibility and maintenance of the usual run-time and space efficiencies did not constrain the design of the new features noticeably. Except for the introduction of the keywords `catch`, `private`, `protected`, `signed`, `template`, and `volatile` the extensions described here are upward compatible. Users will find, however, that type-safe linkage, improved enforcement of `const`, and improved handling of ambiguities will force modification of some programs by detecting previously uncaught errors.

Acknowledgements

Most of the credit for these extensions goes to the literally hundreds of C++ users who provided me with bugs, mistakes, suggestions, and most importantly with sample problems.

Phil Brown, Tom Cargill, Jim Coplien, Steve Dewhurst, Keith Gorlen, Laura Eaves, Bob Kelley, Brian Kernighan, Andy Koenig, Archie Lachner, Stan Lippman, Larry Mayka, Doug McIlroy, Pat Philip, Dave Prosser, Peggy Quinn, Roger Scott, Jerry Schwarz, Jonathan Shopiro, and Kathy Stark supplied many valuable suggestions and questions.

The C++ multiple inheritance mechanism was partially inspired by the work of Stein Krogdahl from the University of Oslo [1984].

References

[ANSI 1988] Draft Proposed American National Standard X3J11/88/090 dated December 7, 1988.

Tom Cargill, PI: A Case Study in Object-Oriented Programming, OOPSLA'86 Proceedings, pages 350-360, September 1986.

Agha Gul, An Overview of Actor languages, SIGPLAN Notices, pages 58-67, October 1986.

Stein Krogdahl, An Efficient Implementation of Simula Classes with Multiple Prefixing, Research Report No. 83, June 1984, University of Oslo, Institute of Informatics.

Stan Lippman and Bjarne Stroustrup, Pointers to Members in C++, Proc. USENIX C++ Conference, Denver, October 1988.

Alan Snyder, Encapsulation and Inheritance in Object-Oriented Programming Languages, SIGPLAN Notices, November 1986, pages 38-45.

Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986.

Bjarne Stroustrup, Multiple Inheritance for C++, Proc. EUUG Spring'87 Conference, Helsinki. [Expanded version to appear in *Computing Systems* 2:4.]

Bjarne Stroustrup, What is "Object-Oriented Programming"?, Proc. 1st European Conference on Object-Oriented Programming, Paris, 1987, Springer Verlag Lecture Notes in Computer Science, Vol. 276, pages 51-70. Also in *IEEE Software*, May 1988. (1987a)

Bjarne Stroustrup, Possible Directions for C++, Proc. USENIX C++ Workshop, Santa Fe, November 1987. (1987b)

Bjarne Stroustrup, Type-safe Linkage for C++, Proc. USENIX C++ Conference, Denver, October 1988. Also in *Computing Systems* Vol. 1 No. 4 Fall 1988, pages 371-403.

Bjarne Stroustrup, Parameterized Types for C++, Proc. USENIX C++ Conference, Denver, October 1988. Also in *Computing Systems* Vol. 2 No. 1 Spring 1989, pages 55-85, and *Journal of Object-Oriented Programming*, January 1989. (1989)

Daniel Weinreb and David Moon, *Lisp Machine Manual*, Symbolics, Inc., 1981.

[*submitted May 7, 1989; revised July 27, 1989; accepted August 4, 1989*]