C++ Language System Overview

Introduction

About This Guide

This manual, the *C++ Language System Overview*, is an overview of new language features of C++. Most of the extensions take the form of removing restrictions on what can be expressed in C++.

The main new features provided in the 2.1 release of C++ are: multiple inheritance, type−safe linkage, recursive definition of assignment and initialization, better facilities for user−defined memory management, abstract and derived classes, *static* member functions, *const* member functions, *protected* members, pointers to members, type checking, operator and function name overloading, free store management, constant types, references, inline functions, and virtual functions.

This manual also shows how C++ corrects most of the deficiencies of C by offering improved compile−time type checking and support for encapsulation.

Audience

This manual assumes that you are familiar with the language as described in *The C++ Programming Language* (see **"Recommended Reading on C++"**) and to have sufficient experience with C++ to recognize many of the problems that the features described in this manual are designed to solve or alleviate.

Document Overview

This guide contains the following chapters:

Chapter 1, "The Evolution of C++," is an overview of new language features of C++: 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.

Chapter 2, "An Introduction to C++," explains how C++ corrects most of the deficiencies of C by offering improved compile−time type checking and support for encapsulation.

Chapter 3, "An Overview of C++," describes the features added to C to be a better C, support data abstraction, and support object−oriented programming.

Chapter 4, "Object−Oriented Programming," discusses features necessary to support these programming styles in a general−purpose programming language. It presents one view of what "object−oriented" ought to mean in the context of a general−purpose programming language.

Chapter 5, "Multiple Inheritance for C++," describes an implementation of a multiple inheritance mechanism for C++. It provides only the most rudimentary explanation of what multiple inheritance is in general and what it can be used for.

Chapter 6, "Parameterized Types for C++," explains Type parameterization, the ability to define a type in terms of another, unspecified, type.

Chapter 7, "Automatic Template Instantiation," presents detailed information on use of the template instantiation environment with C++ Release 3.0.1.

Chapter 8, "Template Instantiation Environment," describes the problems involved in generating names for overloaded functions in C++ and in linking to C programs. It also discusses how these problems relate to library building.

Chapter 9, "Type–Safe Linkage for C++," describes the problems involved in generating names for overloaded functions in C++ and in linking to C programs. It also discusses how these problems relate to library building.

Chapter 10, "Access Rules for C++," defines the C++ access rules, as they relate to the various protection methods, and explains some of the reasoning for these rules.

Chapter 11, "Inline Functions in C++," describes the proper use of C++ inline functions and points out some of the pitfalls in their use. It discusses the implementation of inline functions in the AT&T C++ Language System Release 2.1.

Chapter 12, "As Close as Possible to C—But No Closer," summarizes the remaining differences between the ANSI C standard and C++, explains their motivation.

"Glossary", contains glossary definitions relevant to this manual.

Index, is a detail index of this manual.

Related Documentation

Refer to the *C++ Translator Release Notes*, that accompanied your C++ installation package, for information specific to your release of the C++ translator. In addition, you should have copies of the following:

- *C++ Programming Guide*, Silicon Graphics, Inc., document number 007–0704–050
- *C++ Language System Product Reference Manual*, Silicon Graphics, Inc., document number 007–1618–020
- *C++ Language System Library*, Silicon Graphics, Inc., document number 007–1619–020

Recommended Reading on C++

For a precise description of the C++ language, the following is indispensable:

- Ellis, Margaret A., and Stroustrup, Bjarne. *The Annotated C++ Reference Manual* (Reading, Mass.: Addison–Wesley, 1990)

The following books are recommended reading on programming in C++:

- Lippman, Stanley B. *C++ Primer* Second Edition (Reading, Mass.: Addison–Wesley, 1991)
- Stroustrup, Bjarne. *The C++ Programming Language* Second Edition (Addison–Wesley, 1991))

Notation and Syntax Conventions

This guide uses the following notation and syntax conventions:

italic	Indicates arguments in a command line that you must replace with a valid value. In text, it is used to indicate commands, document titles, file names, glossary items, new terms, and variables.
`courier`	Indicates computer output and program listings.
`courier bold`	Indicates user input to the computer and nonprinting keys.
[ ]	Brackets enclose optional command arguments. (Do not enter the brackets.)
. . .	An ellipsis indicates that the preceding optional items can appear more than once in succession.
|	The vertical bar separates items from which you can choose one.

Chapter 1

The Evolution of C++

This chapter is a brief overview of new language features of C++. It is assumed that you are familiar with the language as described in *The C++ Programming Language* and 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++.

The C++ Programming Language describes C++ as defined and implemented in August 1985. This chapter 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 provided in the 2.1 release 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.

This chapter contains the following sections:

- **"Introduction to C++"**
- **"Extensions to C++"**
- **"Conclusions"**

Note: This chapter is from a paper by Bjarne Stroustrup.

Introduction to C++

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 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 **Chapter 4, "Object−Oriented Programming."**

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 chapter, however, deals strictly with language and language implementation issues.

Extensions to C++

Most of the extensions to C++ take the form of removing restrictions on what can be expressed:

- Access Control

  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.
- Overloading Resolution
- Type−Safe Linkage

  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.

- Multiple Inheritance
- Base and Member Initialization
- Abstract Classes

  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.
- *static* Member Functions
- *const* Member Functions
- Initialization of *static* Members
- Pointers to Members

  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.
- User−Defined Free Store Management

  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.
- Assignment and Initialization

  The rules for assignment and initialization of class objects have been made more general and uniform to require less work from the programmer.
- Operator −>
- Operator ,
- Initialization of *static* objects
- Minor extensions
- Resolutions

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

Access Control

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

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. 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:

```
class X {              // Another bad way:
      friend class D1;  // make derived classes friends
      friend class D2;  // to 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
                     // mf2() 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 this chapter under **"Multiple Inheritance."** 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 function
      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.

Access Control Syntax

The following example confuses most beginners and sometimes even experts:

```
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 *X* is not a subtype of *Y* 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;
      // ...
};
```

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* restores 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 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.

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* fr(derived *p) { return p; }
      base* mem() { return this; }
};

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

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 older rules allowed order dependence, the new rules are compatible and old programs produce identical results under the new rules. For several years or so C++ implementations have issued warnings for the "outlawed" order dependent resolutions.

C++ distinguishes five kinds of "matches":

- Match using no or only unavoidable conversions (for example, array name to pointer, function name to pointer to function, and *T* to *constT*).
- 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*.
- Match using standard conversions (for example, *int* to *double*, *derived** to *base**, *unsigned int* to *int*).
- Match using user defined conversions (both constructors and conversion operators).
- 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*

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*.

. 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*);
}
```

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)
{
      char* p1 = strtok(vc,"a");  // strtok(char*, char*);
      const char* p2 = strtok(vcc,"a");
                                  // strtok(const char*, char*);
      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.

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:

- concern that undetected ambiguities could occur
- 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(table* tbl, const char* name, int index)
{
      // ...
}
```

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 **Chapter 6, "Parameterized Types for C++."**

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 since they both 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 and the representation of various kinds of processors and compilers for a multi−machine, multi−environment debugger.

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.

Ambiguous uses are detected at compile−time:

```
class A { f(); /* ... */ };
class B { 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

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.

. Typically one would resolve the ambiguity by adding a function:

```
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.

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.

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*.

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 *B::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 as shown in Figure 1–1:

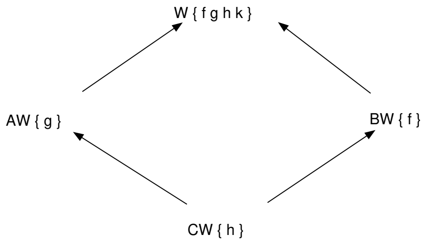

Figure 1–1 A Directed Acyclic Graph

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++.

As described in **Chapter 5, "Multiple Inheritance for C++"** a virtual function call is about as efficient as a normal function call, even in the case of multiple inheritance. The added cost is five to six memory references per call. This compares with the three to fout 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.

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 re−constructed 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 , so the initialization order for class *C* is *A*, *B*, *a*, *b*. This order is independent of the order of explicit initializers so

```
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:

Virtual base classes force a modification to this rule. *operator new*() used to require a *long; size_t* was adopted to bring C++ allocation mechanisms into line with ANSI C.

```
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.

Abstract Classes

One of the purposes of static type checking is to detect inconsistencies and mistakes before a program is run. A significant class of detectable errors has been escaping C++'s checking. In fact, the language actually forced you 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, that 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(); }
       virtual void rotate(int) = 0;  // pure virtual function
       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.

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 of which it is a member. 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 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;
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 members without explicitly specifying an object. For example:

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

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 until now has not been enforced 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 ***.

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 and initialization of *static* members. 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, 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;
```

Pointers to Members

As mentioned in *The C++ Programming Language*, it was an obvious deficiency that there was no way of expressing the concept of a pointer to a member of a class in C++. This lead 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
int (*pf)(char*) = &f;  // declare and initialize pointer to function
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 on which it is modeled.

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 five 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 to *a.mem*.

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 shows 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*.

Assignment to this

If a user wanted to take over allocation of objects of a class *X* the only way used to be 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
            this = this; // forget this assignment at your peril
            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 de-allocation strategy and use an assignment to *this* to indicate that it has taken control over deallocation:

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

This user-defined allocation and de-allocation 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.

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

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

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

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*().

Inheritance of operator new()

The usual rules for inheritance apply:

```
class Y : public X    // objects of class Y are also
{                     // allocated 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*.

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");
```

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

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*() used 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.

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'
      p = ::new((void*)0777) X;   // place an X at address 0777
}
```

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.

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. 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* can make most such checks unnecessary.

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*.

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;
```

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. 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&, 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*:

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

void f() {
```

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.

```
      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.

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., which 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 if *+=*, *=*, and *+* are provided.

The overloading of –> is important, just like overloading [], and not just a minor curiosity. The reason is that *indirection* is a key concept and 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*.

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

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

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.

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 of *The C++ Programming Language*. These resolutions involve slight changes compared to earlier rules. This was done to bring C++ closer to the ANSI C draft.

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

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

This resolution involves a slight change compared to earlier rules. This was done to bring this aspect of C++ into line with the ANSI C draft.

```
f3(unsigned int I);  // ok
f7(unsigned I);      // ok: equivalent to f7(unsigned int I);
```

A type cannot contain two basic type specifiers so the following are legal:

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

Declaration and Expression Syntax

There is an ambiguity in the C++ grammar involving *expression–statements* and *declarations*: 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 *declarations*. 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 analyzing 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
}
```

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 four 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
}
```

The const Specifier

Use of the *const* specifier on a non–local object implies that linkage is *internal* by default; that is, the object declared is local to the compilation 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; }
```

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.

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, does 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 is an *lvalue*. The result of a ?: operator is an *lvalue* provided both its second and third operands are *lvalue*s and provided they have exactly the same type.

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 struct stat(int, 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. 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 stat b;   // struct is needed to avoid the function name
stat(0);         // function call

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

A name cannot simultaneously refer to two types:

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

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++; }
```

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 = a;
}
```

Function Declaration Syntax

To ease 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();"
```

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 that 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.

Chapter 2

An Introduction to C++

In this chapter, we'll see how C++ corrects most of the deficiencies of C by offering improved compile−time type checking and support for encapsulation. We'll also introduce you to many of the new features of C++:

- classes
- type checking
- operator and function name overloading
- free store management
- constant types
- references
- inline functions
- derived classes
- virtual functions

This chapter contains the following sections:

- **"A C++ Example"**
- **"The Specification"**
- **"The Implementation"**
- **"Other Uses for Abstract Data Types"**
- **"Object−Oriented Programming in C++"**
- **"The Current Status of C++"**
- **"The Future of C++"**

Note: This chapter is derived from a paper by Keith Gorlen.

Introduction

The C++ programming language was designed and implemented by Bjarne Stroustrup of AT&T Bell Laboratories as a successor to C

The author's time and the computer facilities required to prepare this paper were provided by the Computer Systems Laboratory, Division of Computer Research and Technology, National Institutes of Health.

. It retains compatibility with existing C programs and the efficiency of C. It also adds many powerful new capabilities, making it suitable for a wide range of applications from device drivers to artificial intelligence. C++ will be of interest to UNIX users because of its intimate relation to C and its potential use for building graphical user interfaces to UNIX, for UNIX systems programming, and for supporting large−scale software development under UNIX.

C++ evolved from a dialect of C known as "C with Classes," which was invented in 1980 as a language for writing efficient event−driven simulations. Several key ideas were borrowed from the Simula67 and Algol 68 programming languages. **Figure 2−1** shows the heritage of C++.

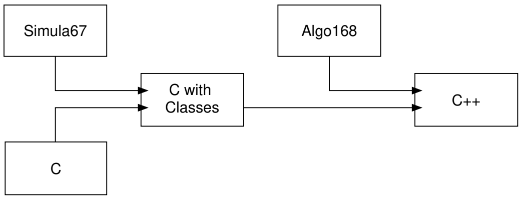

Figure 2−1 The Heritage of C++

The definitive book on C++ is Bjarne Stroustrup's *The C++ Programming Language*, which gives a detailed description of the language and contains many examples and exercises. It also includes the C++ reference manual, which is a concise, more formal definition of the language.

A C++ Example

The best way to learn about C++ is to write a program in it, and that is what we'll do in the next three sections. Let's start in familiar territory by taking a look at a simple program written in plain old C:

```
main()
{
	int a = 193;
	int b = 456;
	int c = a + b + 47;
	printf("%d\n",c);
}
```

This program declares three integer variables named *a*, *b*, and *c*, initializing *a* and *b* to the values 193 and 456, respectively. The integer *c* is initialized to the result of adding *a* and *b* and the constant 47. Finally, the standard C library function *printf*() is called to print out the value of *c*. The quoted string *%d* tells how to print the result: *%d* prints *c* as a decimal number, and *\n* adds a newline character. If we compile and execute this program, it prints out the number 696 and exits.

Now suppose we wish to perform a similar calculation, but this time *a* and *b* are big numbers, like the U. S. national debt expressed in dollars. Such numbers are too big to be stored as *ints* on most computers, so if we tried to write *int a = 25123654789456* the C compiler (hopefully!) would give us an error message and fail to compile the program. There are many practical applications for big integers, such as cryptography, symbolic algebra, and number theory, where it can be necessary to perform arithmetic on numbers with hundreds or even thousands of digits.

It isn't easy to write a program to deal with these big numbers in ordinary C. Coding and debugging the algorithms that perform arithmetic operations on big integers in C involves a significant amount of work, so we'd want to make them general–purpose. We wouldn't be able to predict how big the numbers might become in advance, so we would have to use a dynamic memory allocator to manage their storage at execution time. We'd need to write a C library of functions for creating, destroying, reading, printing, assigning, and performing basic arithmetic on big integers. These functions would have to have distinctive names such as *create_bigint*, *print_bigint*, and *add_bigints* to avoid confusion with other kinds of data that we might want to create, print, or add in the same program.

Worst of all, programmers wishing to use our big integers would have to know the names of these functions and the rules for calling them. They would have to remember to create and initialize big integers when they needed to use them, and to destroy them when they were finished. Even simple arithmetic expressions would be awkward to write; *c = a+b* would have to be coded as:

```
assign_bigint(&c,add_bigints(a,b))
```

and there might be problems with handling temporary results calculated during the evaluation of a complex expression. Also, programmers would have to be careful when combining big integers with other data types such as *int*. They would need to call a function to convert *int*s to big integers before adding them, for example. Any C program using big integers would be both difficult to write and difficult to read.

In C++, we still must write the code to manage the storage of big integers and functions to perform the same operations on them. The difference is that C++ lets us "package" this code so that using our big integers is as convenient as using the *int* data type that is built into C. We can, in effect, extend the C++ language by adding our own custom data type, which we'll call *BigInt*. Notice how similar the example C program is to this C++ program which performs a similar calculation using *BigInt*s instead of *int*s:

```
#include "BigInt.h"
main()
{
```

```
      BigInt a =  "25123654789456";
      BigInt b = "456023398798362";
      BigInt c = a + b + 47;
      c.print();    /* print the result, c */
      printf("\n");
}
```

Data Abstraction

This technique of defining new data types that are well−suited to the application to be programmed is known as *data abstraction*, and a data type such as *BigInt* is called an *abstract data type*. Data abstraction is a powerful, general−purpose technique which, when properly used, can result in shorter, more readable, more flexible programs.

Data abstraction is supported by several other modern programming languages such as Ada.

In these languages, and in C++ as well, a programmer can define a new abstract data type by specifying a data structure together with the operations permissible on that data structure, as shown in **Figure 2−2** .

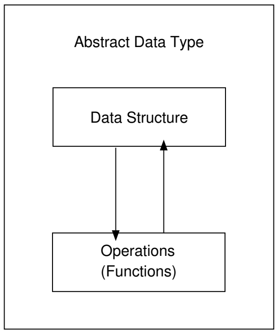

Figure 2–2 An Abstract Data Type

It is difficult or impossible to practice data abstraction in most other programming languages currently in widespread use, such as BASIC, C, COBOL, FORTRAN, PASCAL, or Modula–2. This is because data abstraction requires special language features not available in these languages. To get an idea of what these features do, let's analyze the example C++ program.

The first three statements in the body of the *main*() program declare three type *BigInt* variables, *a*, *b*, and *c*. The C++ compiler must know how to create them, how much space to allocate for them, and how to initialize them.

The first and second statements are similar; they initialize the *BigInt* variables *a* and *b* with big integer constants written as character strings containing only digits. To do this the C++ compiler must be able to convert character strings into *BigInt*s.

The third statement is the most complicated. It adds *a*, *b*, and the integer constant 47 and stores the result in *c*. The C++ compiler needs to be able to create a temporary *BigInt* variable to hold the sum of *a* and *b*. Then it must convert the *int* constant *47* into a *BigInt* and add this to the temporary variable.

Finally, it must initialize *c* from this temporary *BigInt* variable.

The fourth statement prints *c* on the standard output. The last statement calls the C library function *printf* () to print a newline character. C programmers are probably familiar with *printf*(), but *c.print*() probably looks a bit strange. It is a call on a special kind of function available in C++ called a *member function*. We'll talk more about this later, but for now just think of it as a function that prints out a variable of type *BigInt*.

Even though there are no more statements in the body of *main*(), the compiler isn't finished yet. It must destroy the *BigInt* variables *a*, *b*, and *c* and any *BigInt* temporaries it may have created before leaving a function, such as *main*(). This is to assure that the storage used by these variables is freed.

Let's summarize what the C++ compiler needs to know how to do with *BigInt*s to compile the example program:

- *Create* new instances of *BigInt* variables
- *Convert* character strings and integers to *BigInt*s
- *Initialize* the value of one *BigInt* with that of another *BigInt*
- *Add* two *BigInt*s together
- *Print* *BigInt*s
- *Destroy* *BigInt*s when they are no longer needed

Specifications and Implementations

Where does the C++ compiler obtain this know−how? From the file *BigInt.h*, which is included by the first line of the example program. This file contains the *specification* of our *BigInt* abstract data type. The specification contains the information that programs that *use* an abstract data type need to have to be successfully compiled. The details of *how* the abstract data type works, known as the *implementation*, are kept in another file. In our example, this file is named *BigInt.c*. It is compiled separately, and the object code produced from it is linked with the program that uses the abstract data type, also called the *client* program. **Figure 2−3** shows how the specification (*BigInt.h*) and implementation (*BigInt.c*) of an abstract data type (*BigInt*) are combined with the source code of a client program (*client.c*) to produce an executable program (*client*).

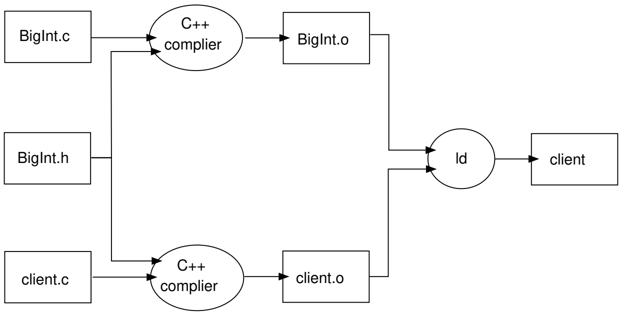

Figure 2−3 Producing an Executable Program

We separate the code for an abstract data type into a specification part and an implementation part to hide the implementation details from the client. We can then change the implementation and be confident that client programs will continue to work correctly after they are relinked with the modified object code. This is useful when a team of programmers work on a large software project. Once they agree on the specifications for the abstract data types they need, each team member can implement one or more of them independently of the rest of the team.

A well−designed abstract data type also hides its complexity in its implementation, making it as easy as possible for clients to use.

The Specification

Let's take a look at the specification for our *BigInt* data type, contained in the file *BigInt.h*. (Note that in C++, *//* begins a comment that extends to the end of the line.)

```
class BigInt {
    char* digits;     // pointer to digit array in free store
    int ndigits;      // number of digits
public:
    BigInt(const char*);    // constructor function
    BigInt(int);            // constructor function
    BigInt(const BigInt&); // initialization constructor function
    BigInt operator+(const BigInt&); //addition operator function
      void print();            // printing function
      ~BigInt();               // destructor function
};
```

Much of this code may look odd, but we'll explain it as we cover the features of C++ in the next few sections.

Classes

This is an example of one of the most important features of C++, the *class* declaration, which specifies an abstract data type. It is an extension of something C programmers are probably already familiar with, the *struct* declaration.

The *struct* declaration groups together a number of variables, which may be of different types, into a unit. For example, in C (or in C++) we can write:

```
struct BigInt {
      char* digits;
      int ndigits;
};
```

We can then declare an *instance* of this structure by writing:

```
struct BigInt a;
```

The individual *member variables* of the *struct*, *digits* and *ndigits*, can be accessed using the dot (.) operator; for example, *a.digits*, accesses the member variable *digits* of the *struct a*.

Recall that in C we can also declare a pointer to an instance of a structure:

```
struct BigInt* p;
```

in which case we can access the individual member variables by using the –> operator; for example, *p–> digits*.

C++ classes work in a similar manner, and the . and –> operators are used in the same way to access a class's member variables. In our example, class *BigInt* has two member variables named *digits* and *ndigits*. The variable *digits* points to an array of bytes (*char*s), allocated from the free storage area, that holds the digits of the big integer, one decimal digit per byte. The digits are ordered beginning with the least significant digit in the first byte of the array. The member variable *ndigits* contains the number of digits in the integer. **Figure 2–4** shows a diagram of this data structure for the number 654321.

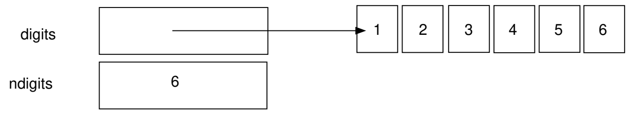

Figure 2–4 Diagram of a *BigInt* Data Structure

However, the C++ *class* can do much more than the *struct* feature of regular C. We'll now look at these extensions in detail.

Encapsulation

In C++, a client program can declare an instance of class *BigInt* by writing:

```
BigInt a;
```

But now we have a potential problem: the client program might try, for example, to use the fact that *a.ndigits* contains the number of digits in the number *a*. This would make the client program dependent on the *implementation* of class *BigInt*—after all, we might wish to change the representation of *BigInt*s to use hexadecimal instead of decimal arithmetic to save storage. We need a way to prevent unauthorized access to the member variables of the instances of a class. C++ provides this by allowing the use of the keyword *public:* within a *class* declaration to indicate which members can be accessed by anyone and which have restricted access. Members declared before the *public:* keyword are *private*, as are *digits* and *ndigits* in this example, so C++ will issue an error message if a client program attempts to use them.

Protecting the member variables of a class in this manner is known as *encapsulation*. It is a good programming practice because it enforces the separation between the specification and the implementation of abstract data types that we are trying to achieve, and it helps when debugging programs. For example, if we find that *ndigits* has the wrong value in some situation, those parts of the program that do not have access to the variable are probably not at fault.

Member Functions

But how does a client program interact with the private member variables of a class? Whereas a *struct* allows only variables to be grouped together, the C++ *class* declaration allows both variables and the functions that operate on them to be grouped. Such functions are called *member functions*, and the private member variables of the instances of a class can be accessed only by the member functions of that class. Thus, a client program can read or modify the values of the private member variables of an instance of a class indirectly, by calling the public member functions of the class, as shown in **Figure 2–5**
.

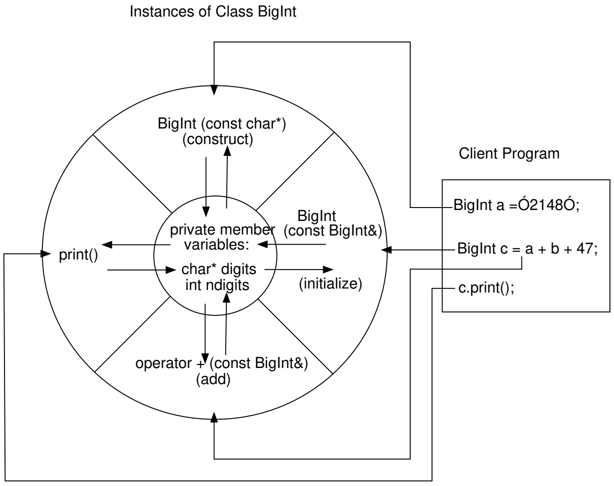

Figure 2–5 Client Program Access of Private Member Variables

Our example class *BigInt* has two private member variables, *digits* and *ndigits*, and six public member functions. The declarations of these member functions will look unusual to C programmers for several reasons: the types of the arguments of the functions are listed within parentheses in the function declarations, three of the functions declared have the same name, *BigInt*, and the function names *operator+* and *~BigInt* contain characters normally not allowed in function names.

Function Argument Type Checking

C++ strongly encourages a programmer to declare the types of the arguments of all functions. This makes it possible for C++ to check for inconsistent argument types when a function call is compiled, and can eliminate many bugs at an early stage. For example, the C statement:

```
fprintf("The answer is %d",x);
```

will compile with no problem. However, when this statement is executed the program will abort with a cryptic error message. The problem is that the standard C library function *fprintf*() expects the first argument to be a pointer to the stream to which the output is to be written, not a format string as it is here. On the other hand, in C++ we can declare the argument types of *fprintf*():

```
extern int fprintf(FILE*, const char*, ...);
```

so the compiler can give us an error message when we try to compile the incorrect function call, noting the discrepancy in the argument types. Conveniently, the argument types for most standard library functions are declared in system header files that you can include in your programs so that you don't have to write all these common declarations yourself.

Function Name Overloading

Listing the types of all of a function's arguments in its declaration has a second benefit: we can define several functions with the same name, as long as each requires a different number and/or type of argument. For example, in C++ we can declare two functions with the name *abs*:

```
int abs(int);
float abs(float);
```

We can then write:

```
x = abs(2);
y = abs(3.14);
```

The first statement will call *abs*(*int*), and the second will call *abs*(*float*)—the C++ compiler knows which *abs* to use because 2 is an *int* and 3.14 is a *float*. When more than one function has the same name like this, the name is said to be *overloaded*. One advantage of overloading is that it eliminates "funny" function names (remember *ABS*, *IABS*, *DABS*, and *CABS* from FORTRAN?). It also leads to more general programs; for example, we can write *copy*(*x,y*) to copy a *y* to an *x* without having to worry about their types; they might be arrays, or strings, or files, as long as we have written a *copy* function to handle each case.

Calling Member Functions

Getting back to our *BigInt* example and our discussion of member functions, we can now explain the next−to−last line in our first C++ program which is:

```
c.print();
```

Member functions are called in a manner analogous to the way member variables are normally accessed in C; that is, by using the . or −> operators. Since *c* is an instance of class *BigInt*, the notation *c.print*() calls the member function *print*() of class *BigInt* to print the current value of *c*. Similarly, if we declared a pointer to a *BigInt*:

```
BigInt* p;
```

then the notation *p−>print*() would call the same function. This notation prevents this particular *print*() from inadvertently being called to operate on anything other than an instance of class *BigInt*.

In C++, several different classes may all have member functions with the same name, just as in regular C several different *struct*s may all have member variables with the same name. This lets us use simple function names, like *print*, rather than distinctive names, like *print_bigint*, without worrying about naming conflicts. We could add a new class, say *BigFloat*, to a program that also used *BigInt*s, and we could also define *print*() as a member function of class *BigFloat*. Our program could contain the statements shown below and the C++ compiler would use the appropriate *print*() in both cases:.

```
BigInt a = "2934673485419";
BigFloat x = "874387430.3945798";
a.print();
x.print();
```

Constructors

One of the things the C++ compiler needs to know about our *BigInt* abstract data type is how to create new instances of *BigInt*s. We can tell C++ how we want this done by defining one or more special member functions called *constructors*. A constructor function is one which has the same name as its class. When a client program contains a declaration such as:

```
BigInt a = "123";
```

the C++ compiler reserves space for the member variables of an instance of class *BigInt* and calls the constructor function *a.BigInt~*. It is our responsibility as providers of the *BigInt* data type to write the function *BigInt*() so that it initializes the instance correctly. In our example, we'll have *BigInt~* allocate three bytes of dynamic storage, set *a.digits* to point to this storage, set the three bytes to {3,2,1}, and set *a.ndigits* to three. This will create an instance of class *BigInt* named *a* that is initialized to 123.

If a class has a constructor function, C++ *guarantees* it will be called to initialize every instance created of that class. A user of an abstract data type such as *BigInt* does not have to call an initialization function for every *BigInt* declared, which eliminates a common source of programming errors.

Constructors and Type Conversion

The second thing C++ needs to know is how to convert something that is a character string, such as *"25123654789456,"* or an integer, such as 47, to a *BigInt*. Constructors are also used for this purpose. When the C++ compiler sees a statement like:

```
BigInt c = a + b + 47;
```

it recognizes that the *int 47* must be converted to a *BigInt* before the addition can be done, and so checks to see if the constructor *BigInt*(*int*) is declared. If so, it creates a temporary instance of *BigInt* by calling *BigInt*(*int*) with the argument *47*. If an appropriate constructor is not declared, the statement is flagged as an error. We have defined *BigInt*(*char**) and *BigInt*(*int*) for class *BigInt*, so we may freely use character strings or integers wherever a *BigInt* can be used, and the C++ compiler will automatically call our constructor to do the type conversion. This is an important feature of C++ because it lets us blend our own abstract data types with others and with the fundamental types built into the language.

Constructors and Initialization

The third thing C++ must know how to do is how to initialize a *BigInt* with the value of another *BigInt*, as is required by a statement such as:

```
BigInt c = a + b + 47;
```

The *BigInt c* must be initialized with the value of a temporary *BigInt* that holds the result of the expression *a + b + 47*.

We can control how C++ initializes instances of class *BigInt* by defining the special constructor function *BigInt(const BigInt&)*. In our example, we'll make this constructor allocate storage for the new instance and make a copy of the contents of the argument instance.

Operator Overloading

The fourth thing C++ must be able to do is to add two *BigInt*s. We could just define a member function named *add* to do this, but then writing arithmetic expressions would be awkward. C++ lets us define additional meanings for most of its operators, including *+*, so we can make it mean "add" when applied to *BigInt*s. This is known as *operator overloading*, and is similar to the concept of function name overloading.

Actually, most programmers are already familiar with this idea because the operators of most programming languages, including C, are already overloaded. For example, we can write:

```
int a,b,c;
float x,y,z;
c = a+b;
z = x+y;
```

The operators *=* and *+* do quite different things in the last two statements: the first statement does *integer* addition and assignment and the second does *floating–point* addition and assignment. Operator overloading is simply an extension of this.

C++ recognizes a function name having the form *operator@* as an overloading of the C++ operator symbol *@*. We can overload the operator *+*, for example, by declaring the member function named *operator+*, as we have done in our example class *BigInt*. We can call this function using either the usual notation for calling member functions or by using just the operator:

```
BigInt a,b,c;
c = a.operator+(b);
c = a + b;
```

The last two lines are equivalent.

Binary operators such as *+* are usually not defined as member functions because automatic conversion of types is not done for the left operand. For example, the expression *a + 47* is equivalent to *a.operator+*(*47*). C++ recognizes that the function *operator+*(*const BigInt&*) is defined and that the constructor *BigInt*(*int*) can be used to convert the *int 47* to a *BigInt* before calling *operator+*. However, the expression *47 + a* is equivalent to *47.operator+*(*a*), which is an error because *47* is not an instance of a class and therefore has no member functions that can be applied to it. For this reason, binary operators are usually defined as *friend* functions, which are discussed later.

Of course, if we overload an operator, we don't change its built–in meaning, we only give it an additional meaning when used on instances of our new abstract data type. The expression *2+2* still gives *4*.

Destructors

The last thing we said was that C++ needed to know how to destroy instances of our *BigInt*s once it was finished with them. We can tell the C++ compiler how to do this by defining another special kind of member function called a *destructor*. A destructor function has the same name as its class, prefixed by the character *~*. For class *BigInt*, this is the member function *~BigInt*(). Since *~* is the C++ and C complement operator, this naming convention suggests that destructors are complementary to constructors.

We must write the function *~BigInt*() so that it properly cleans–up, or *finalizes* instances of class *BigInt* for which it is called. In our example, this means freeing the dynamic storage that was allocated by the constructor.

If a class has a destructor function, C++ *guarantees* that it will be called to finalize every instance of the class when it is no longer needed. Once again, this relieves users of an abstract data type like *BigInt* from having to remember to do this, and eliminates another source of programming errors.

Summary

We've covered a lot of territory already, so let's review where we've been.

We've seen how using the technique of data abstraction can lead to more reliable, more readable, and more flexible programs, and we've introduced many of the features of C++ that help us practice data abstraction:

- *Classes*, the basic language construct for defining new abstract data types;
- *Member variables*, which describe the data in an abstract class, and *member functions*, which define the operations on an abstract class;
- *Encapsulation*, which lets us restrict access to certain member variables and functions;
- *Function argument type checking*, which helps to ensure that functions are called with proper arguments;
- *Function name overloading*, which reduces the need for using unusual function names and helps to generalize code;
- *Constructors and destructors*, which manage the storage for an abstract data type and guarantee that instances of an abstract data type are initialized and finalized;
- *User–defined implicit type conversion*, to let us blend our abstract data types with others and with the fundamental data types of the language; and,
- *Operator overloading*, to let us give additional meaning to most of the existing operators when used with our own abstract data types, making our new data types easier to use.

We've also introduced the idea of breaking up an abstract data type into its specification, which contains

We've also introduced the idea of breaking up an abstract data type into its specification, which contains the information that the user, or client, needs to know to use the abstract data type, and its implementation, which hides the details of how the abstract data type works so that it may be programmed independently by a member of a programming team and be easily maintained.

The Implementation

We've just taken a detailed look at the specification of our *BigInt* abstract data type. Now it's time to discuss its implementation.

The implementation of an abstract data type consists of the C++ code that embodies the details of *how* the data abstraction works. For our example it is kept in a separate file named *BigInt.c*. The implementation requires the information kept in the specification, so the first line in *BigInt.c* is:

```
#include "BigInt.h"
```

Since both the implementation and client programs are compiled with the same specification, the C++ compiler ensures a consistent interface between them.

The BigInt(const char*) Constructor

Class *BigInt* has three constructors, one to create an instance of a *BigInt* from a character string of digits (a *char**), one to create an instance from an integer (an *int*), and one to initialize one *BigInt* from another. We need to be able to create a *BigInt* from a string of digits because this is the only way we can legally write very large integer constants in C++. Creating a *BigInt* from an *int* is provided as a convenience, so we can write small integers in the usual way.

Here is the implementation of the first constructor:

```
BigInt::BigInt(const char* digitString)
{
      int n = strlen(digitString);
      if (n != 0) {
          digits = new char[ndigits=n];
          char* p = digits;
          const char* q = &digitString[n];
          while (n--) *p++ = *--q - '0';
      }
      else {        // empty string
          digits = new char[ndigits=1];
          digits[0] = 0;
      }
}
```

This constructor initializes the data structure of a *BigInt* as we described previously. We determine the length of the character string argument, allocate enough memory to hold the digits of the number, then scan the character string from right to left, converting each digit character to its binary representation.

If the character string is empty we treat this as a special case and create a *BigInt* initialized to zero. C programmers will find this code quite recognizable, with a few exceptions that we'll explain in the next few sections.

The Scope Resolution Operator

The notation *BigInt::BigInt* identifies *BigInt* as a member function of class *BigInt*. Note that several C++ classes can have member functions with the same names. When it is necessary to specify exactly *which* class member we're dealing with, we can prefix the member name by the class name and the *::* operator. The *::* operator is known as the *scope resolution operator*, and it may be applied to both member functions and member variables.

Constant Types

C programmers will be familiar with use of the type *char** for arguments that are character strings, but what is a *const char** ? In C++, the keyword *const* can be used before a type to indicate that the variable being declared is constant, and therefore may not appear to the left of the assignment (=) operator. When used in an argument list as it is above, it prevents the argument from being modified by the function. This protects against another kind of common programming error.

Member Variable References

Throughout the body of the member function, you'll notice that we are able to reference the member variables of the instance for which the function is called without using the *.* or *−>* operators, as we did for example in the statement:

```
digits = new char[ndigits=n];
```

Since member functions reference the member variables of their class frequently, this provides a convenient, short notation.

The new Operator

We used the C++ *new* operator to allocate the dynamic storage needed to hold the digits of a *BigInt*. In C, we would call the standard C library function *malloc*() to do this. The *new* operator has two advantages, however. First, it returns a pointer of the appropriate data type. Thus, to allocate space for the member variables of a *struct BigInt* in C we would write:

```
(struct BigInt*)malloc(sizeof(struct BigInt))
```

whereas in C++ we can write:

```
new BigInt
```

The second advantage is that if we use *new* to allocate an instance of a class having a constructor function (such as *BigInt*), the constructor is called automatically to initialize the newly allocated instance. The result is more readable, less error−prone code.

Placement of Declarations

C programmers may have noticed that the declaration of *p* seems to be "misplaced":

```
if (n != 0) {
      digits = new char[ndigits=n];  // a statement
      char* p = digits;              // a declaration!
```

since it appears *after* the first statement in a block. In C++, declarations may be intermixed with statements as long as each variable is declared before its first use. You can frequently improve the readability of a program by placing variable declarations near the place where they are used.

The BigInt(int) Constructor

Here's the implementation of the *BigInt*(*int*) constructor, which creates a *BigInt* from an integer:

```
BigInt::BigInt(int n)
{
      char d[3*sizeof(int)+1]; // buffer for decimal digits
      char* dp = d;            // pointer to next decimal digit
      ndigits = 0;
      do {                   // convert integer to decimal digits
          *dp++ = n%10;
          n /= 10;
          ndigits++;
      } while (n > 0);
      digits = new char[ndigits];
      register int i;
      for (i=0; i<ndigits; i++) digits[i] = d[i];
```

```
      for (i=0; i<ndigits; i++) digits[i] = d[i];
}
```

This constructor works by converting the integer argument to decimal digits in the temporary array *d*. We then know how much space to allocate for the *BigInt*, so we allocate the correct amount of dynamic storage using the *new* operator, and copy the decimal digits from the temporary array into it.

The Initialization Constructor

The job of the initialization constructor is to copy the value of its *BigInt* argument into a new instance of *BigInt*:

```
void BigInt::BigInt(const BigInt& n)
{
      int i = n.ndigits;
      digits = new char[ndigits=i];
      char* p = digits;
      char* q = n.digits;
      while (i--) *p++ = *q++;
}
```

This function makes use of a *reference*, an important C++ feature we haven't seen before.

References

The argument type of the member function *BigInt*(*const BigInt&*) is an example of a C++ *reference*. References address a serious deficiency of C: the lack of a way to pass function arguments by reference. To understand what this means, suppose we wish to write a function named *inc*() that adds one to its argument. If we wrote this in C as:

```
void inc(x)
int x;
{
      x++;
}
```

and then called *inc*() with the following program:

```
int y = 1;
inc(y);
printf("%d\n",y);
```

we would discover that the program would print a 1, not a 2. This is because in C the *value* of *y* is *copied* into the argument *x*, and the statement *x++* increments this copy, leaving the value of *y* unchanged. This treatment of function arguments is known as *call by value*.

To do this correctly in C we must explicitly pass a pointer as the argument to *inc*():

```
void inc(x)
int* x;
{
      *x++;
}
int y = 1;
inc(&y);
printf("%d\n",y);
```

Notice that we had to change the program in three ways:

- the type of the function argument was changed from an *int* to an *int**;
- each occurrence of the argument in the body of the function was changed from *x* to **x*; and,
- each call of the function was changed from *inc*(*y*) to *inc*(*&y*).

The point is that passing a pointer as a function argument requires consistency in every usage of the argument within the function body and, worse yet, in every call of the function made by client programs.

This, combined with C's lack of function argument type checking, results in ample opportunity for error.

Using a C++ reference, we can write the function *inc*() as follows:

```
void inc(int& x)
{
      x++;
}
int y = 1;
inc(y);
printf("%d\n",y);
```

This requires changing only the argument type from *int* to *int&*.

In the function *inc*(), we need to pass the argument *x* using a reference because its value is modified by the function. But efficiency is another reason for passing arguments by reference. When the value of an argument requires a lot of storage, as in the case of *BigInt*s, it is less expensive to pass a pointer to the argument even though its value is not to be changed. That's why we declared the argument to *BigInt* as *const BigInt&*—the reference *BigInt&* causes just a pointer to the argument to be passed, but the *const* prevents that pointer from being used to change the argument's value from within the function.

The Addition Operator

Let's take a look at a first draft of the function *operator+*, which implements *BigInt* addition:

```
BigInt BigInt::operator+(const BigInt& n)
{
// Calculate maximum possible number of digits in sum
      int maxDigits = (ndigits>n.ndigits ? ndigits:n.ndigits)+1;
      char* sumPtr = new char[maxDigits]; // allocate storage
                                          // for sum
      BigInt sum(sumPtr,maxDigits);  // must define this
                                     // constructor
      int i = maxDigits;
      int carry = 0;
      while (i--) {
          *sumPtr = /*next digit of this*/ +
                    /*next digit of n*/ + carry;
          if (*sumPtr > 9) {
              carry = 1;
              *sumPtr -= 10;
          }
          else carry = 0;
          sumPtr++;
      }
      return sum;
}
```

We add two *BigInt*s by using the paper–and–pencil method we all learned in grammar school: we add the digits of each operand from right to left, beginning with the right–most, and also add a possible carry in from the previous column. If the sum is greater than nine, we subtract ten from the result and produce a carry.

The BigInt(char*,int) constructor

We ran into a couple of problems when writing the addition function which we indicated with comments in the code. The first problem is that we need to declare an instance of *BigInt* named *sum* in which to place the result of the addition, which will be left in the array pointed to by *sumPtr*. We must use a constructor to create this instance of *BigInt*, but none of those we have defined thus far are suitable, so we must write another.

This new constructor takes a pointer to an array containing the digits and the number of digits in the array as arguments and creates a *BigInt* from them. We don't want our client programs to use an implementation–dependent and unsafe function, so we'll declare it in the private part of class *BigInt* where it can only be used by member functions. Thus, we add the declaration:

```
BigInt(char*,int);
```

just before the keyword *public:* in the declaration of class *BigInt* in the file *BigInt.h*, and we add the implementation of this constructor to the file *BigInt.c:*

```
BigInt::BigInt(char* d, int n)
{
      digits = d;
      ndigits = n;
}
```

Class DigitStream

The second problem we encountered is that scanning the digits of the operands in the statement:

```
*sump = /*next digit of this*/ + /*next digit of n*/ + carry;
```

becomes complicated because one of the operands may contain fewer digits than the other, in which case we must pad it to the left with zeros. We would also face this problem when implementing *BigInt* subtraction, division, and multiplication, so it is worthwhile to find a clean solution. Let's use an abstract data type!

Here is the declaration for class *DigitStream* and the implementation of its member functions:

```
class DigitStream {
      char* dp;           // pointer to current digit
      int nd;             // number of digits remaining
public:
      DigitStream(const BigInt& n);   // constructor
      int operator++();   // return current digit and advance
};
DigitStream::DigitStream(BigInt& n)
{
      dp = n.digits;
      nd = n.ndigits;
}
int DigitStream::operator++()
{
      if (nd == 0) return 0;
      else {
          nd--;
          return *dp++;
      }
}
```

We can now declare an instance of a *DigitStream* for each of the operands and use the *++* operator when we need to read the next digit.

With these two problems solved, the implementation of the *BigInt* addition operator looks like:

```
BigInt BigInt::operator+(const BigInt& n)
{
      int maxDigits = (ndigits>n.ndigits ? ndigits:n.ndigits)+1;
      char* sumPtr = new char[maxDigits];
      BigInt sum(sumPtr,maxDigits);
      DigitStream a(*this);
      DigitStream b(n);
      int i = maxDigits;
```

```
        int carry = 0;
        while (i--) {
            *sumPtr = (a++) + (b++) + carry;
            if (*sumPtr > 9) {
                carry = 1;
                *sumPtr -= 10;
            }
            else carry = 0;
            sumPtr++;
        }
        return sum;
}
```

Friend Functions

Our abstract data type *DigitStream* looks quite elegant, but you may be wondering how the constructor *DigitStream*(*const BigInt&*) is able to access the member variables *digits* and *ndigits* of class *BigInt*. After all, *digits* and *ndigits* are private, and *DigitStream*(*const BigInt&*) is not a member function of class *BigInt*.

Well, it can't. We need a way to grant access to these variables to just this one function. C++ provides us with a way to do this—we can make this constructor a *friend* of class *BigInt* by adding the declaration:

```
friend DigitStream::DigitStream(const BigInt&);
```

to the declaration of class *BigInt*.

We can also make *all* of the member functions of one class friends of another by declaring the entire class as a *friend*. For example, we can make *all* of the member functions of class *DigitStream* friends of class *BigInt* by placing the declaration:

```
friend DigitStream;
```

in the declaration of class *BigInt*.

The Keyword this

Going back to the implementation of the function *operator+*(), you may be wondering where the pointer variable *this* came from in the declaration:

```
DigitStream a(*this);
```

Previously, we described how within the body of a member function we could refer to the members of the instance for which the function was called without using the *.* or *–>* operators. C++ also gives us the keyword *this* so that we may refer to the entire instance as a unit. The keyword *this* is essentially a pointer to this instance, and in our example may be thought of as a variable of type *BigInt**. Thus, the declaration *DigitStream a*(**this*) creates an instance of *DigitStream* for the left operand of *operator+*().

The Member Function BigInt::print()

The implementation of the member function *print*() is straightforward:

```
void BigInt::print()
{
        int i;
        for (i = ndigits-1; i >= 0; i--) printf("%d",digits[i]);
}
```

It loops through the *digits* array from the most–significant through the least–significant digits, calling the standard C library function *printf*() to print each digit.

The BigInt Destructor

The only thing that the *BigInt* destructor function *~BigInt*() must do is free the dynamic storage allocated by the constructors:

```
BigInt::~BigInt()
{
      delete digits;
}
```

This is done using the C++ *delete* operator, which in this case frees the dynamic storage that is pointed to by *digits*. The *delete* operator does what is usually accomplished in C by calling the standard C library function *free*, but in addition, if we use *delete* to deallocate an instance of a class having a destructor function, the destructor is called automatically to finalize the instance just before its storage is freed. The *delete* operator is thus the inverse of the *new* operator.

Inline Functions

By now you may be thinking that the overhead of calling all of these little member functions must make C++ inefficient. This would be unacceptable for a proper successor to C, which is renowned for its efficiency! So C++ allows us to declare a function to be *inline*, in which case each call of the function is replaced by a copy of the entire function, much like the substitution performed for the *#define* preprocessor command. This entirely eliminates the overhead of calling a function, and makes encapsulation practical.

To make a function such as *~BigInt*() inline, we must move its implementation from the file *BigInt.c* to the file *BigInt.h* and add the keyword *inline* to the function definition:

```
inline BigInt::~BigInt()
{
      delete digits;
}
```

The function definition must be in *BigInt.h* because it will be needed by the compiler whenever a client program uses a *BigInt*. Small functions make the best candidates for inline compilation. C++ gives us a convenient shorthand for writing *inline* functions: we can include the function body in the function declaration within the *class* declaration. Thus, we can also make *~BigInt*() *inline* by writing the following in the declaration of class *BigInt*:

```
~BigInt()   { delete digits; }
```

Below is the complete *BigInt.h* showing appropriate functions made *inline*:

```
#include <stdio.h>
class BigInt {
    char* digits;       // pointer to digit array in free store
    int ndigits;        // number of digits
    BigInt(char* d, int n) { // constructor function
        digits = d;
        ndigits = n;
    }
    friend DigitStream;
public:
    BigInt(const char*)    // constructor function
    BigInt(int);           // constructor function
    BigInt(const BigInt&); // initialization constructor function
    BigInt operator+(const BigInt&); //addition operator function
    void print();              // printing function
    ~BigInt()    { delete digits; }   // destructor function
};
class DigitStream {
      char* dp;                  // pointer to current digit
      int nd;                    // number of digits remaining
public:
      DigitStream(const BigInt& n)      // constructor function
```

```
            dp = n.digits;
            nd = n.ndigits;
        }
        int operator++() {      // return current digit and advance
            if (nd == 0) return 0;
            else {
                nd--;
                return *dp++;
            }
        }
};
```

Summary

This completes our example abstract data type *BigInt*. Let's review the C++ features presented in this section:

- the *scope resolution operator*, which allows us to specify which class we mean when one or more classes have member variables or functions with the same name;
- *constant types*, which we can use to protect variables or function arguments from unintended modification;
- *implicit member variable references* and the keyword *this*, which are used within member functions to access the instance for which the function is called;
- the *new* and *delete* operators, which manage the free storage area and call class constructors/destructors if present;
- *references*, which we can use to conveniently pass pointers to instances instead of the instances themselves as function arguments;
- *friend functions*, which give us a way to grant access to the private member variables and functions of a class to other functions and classes; and,
- *inline functions*, which make data abstraction in C++ efficient and practical.

Other Uses for Abstract Data Types

Our *BigInt* abstract data type is an obvious application for the technique of data abstraction because it is a numeric data type, like *int*, and it is natural to extend the meanings of C++'s arithmetic operators to apply to *BigInt*s. As you become more familiar with this technique, you'll discover many opportunities for using abstract data types in your programs. Here are a few examples:

Dynamic Character Strings

We can define a dynamic (i.e., variable length) character string abstract data type that works like the string variables in languages such as BASIC. We can overload the operators *&* and *&=* to concatenate character strings, overload the relational operators *<*, *<=, ==*, and so on to compare character strings, and overload the array subscript operator [ ] to address the individual characters of a string. The function call operator:

```
operator() (int position, int length)
```

can be overloaded to perform substring extraction and replacement.

Complex Numbers

C++, like C, doesn't have a built-in complex data type, but it's easy to define one in C++. In fact, one is distributed with the C++ compiler. Class *complex* has two member variables of type *double* that hold the real and imaginary parts of a complex number, and all of the usual arithmetic operators are overloaded to perform complex arithmetic when applied to instances of class *complex*. Many of the functions in the math library, such as *cos*() and *sqrt*(), are overloaded for complex arguments.

Vectors

Vectors are another useful abstract data type. We can define classes for vectors of the fundamental data types, such as *FloatVec*, *DoubleVec*, and *IntVec*, and overload the arithmetic operators to apply element−by−element to vectors. The array subscript operator [ ] can be overloaded to check the range of vector subscripts or to handle vectors with arbitrary subscript bounds. It's also possible to overload the function call *operator*() to subscript multi−dimensional arrays.

Stream I/O

A stream I/O package is distributed with the C++ compiler that defines the class *iostream* (input/output stream) for doing formatted I/O. This class defines an instance named *cin* connected to the standard input file and overloads the operator *>>* for all the fundamental data types so we can write:

```
float x;
int i;
char* s;
cin >> x >> i >> s;
```

to read a *float*, an *int*, and a character string from the standard input file, for example. The advantage of this over using the C library function *scanf*() is that it is not possible to make the following types of errors:

```
int i;
scanf("%f",&i);         // float format for int
scanf("%d",i);         // int instead of int*
```

Similarly, class *iostream* defines an instance named *cout* connected to the standard output file and an instance named *cerr* connected to the standard error file. It overloads the operator *<<* for all the fundamental data types so we can write:

```
cout << x << i << s;
```

to write a *float*, and *int*, and a character string to the standard output file.

We can also add our own overloadings for the operators *>>* and *<<* for classes we've written so we can read or write instances of these classes using the same notation.

Object−Oriented Programming in C++

Perhaps the most interesting features of C++ are those that support the style of programming known as *object−oriented programming*. Object−oriented programming is generally useful, but is particularly suited for interactive graphics, simulation, and systems programming applications.

Derived Classes

Suppose we have written a C++ class defining an abstract data type, and we need another abstract data type that is similar to it. Perhaps it requires some additional member variables or functions, or a few of its member functions must do something differently. We'd like to reuse the code we've already written and debugged as much as possible. C++ gives us a simple way to accomplish this: we can declare the new class as a *derived class* of our existing class, called the *base class*. The derived class *inherits* all of the member variables and functions of its base class. We can then differentiate the derived class from its base class by adding member variables, adding member functions, or re−defining member functions inherited from the base class.

A base class may have more than one derived class, and a derived class may, in turn, serve as the base class for other derived classes. Thus, we can define an entire tree−structured arrangement of related classes. This gives us a coherent way to organize classes and to share common code among them.

Virtual Functions

Now suppose we're writing a graphics package, and we've written some classes for various geometric shapes, such as *Line*, *Triangle*, *Rectangle*, and *Circle*. All of these classes implement some of the same member functions, for example *draw*() and *move*(). The relevant *class* declarations for class *Line* and class *Circle* would look like this:

```
class Line {
      int x1,y1,x2,y2;    // end point coordinates
public:
      Line(int xx1,int yy1,int xx2,int yy2)       // constructor
          { x1=xx1; y1=yy1; x2=xx2; y2=yy2; }
      void draw();        // draw a line from (x1,y1) to (x2,y2)
      void move(int dx, int dy);    // move line by amount dx,dy
};
class Circle {
      int x,y;        // center of circle
      int r;          // radius of circle
public:
      Circle(int xx,int yy,int rr)  // constructor
          { x=xx; y=yy; r=rr; }
      void draw();  // draw circle with center (x,y) and radius r
      void move(int dx, int dy);   // move circle by amount dx,dy
};
```

There are a couple of things we'd like to be able to do with these related classes. First, it would be useful to have an abstract data type called *Picture* that would be a collection of *Line*s, *Triangle*s, *Rectangle*s, and *Circle*s. Second, we'd like to be able to *draw*() and *move*() our *Picture*s.

It would be most elegant if class *Picture* were general, and contained no mention of the specific shapes. That way, we could introduce a new shape, say a *Pentagon*, and not have to change class *Picture* in any way.

We can do this by defining a base class *Shape* with derived classes *Line*, *Triangle*, and so on, as shown in **Figure 2–6** .

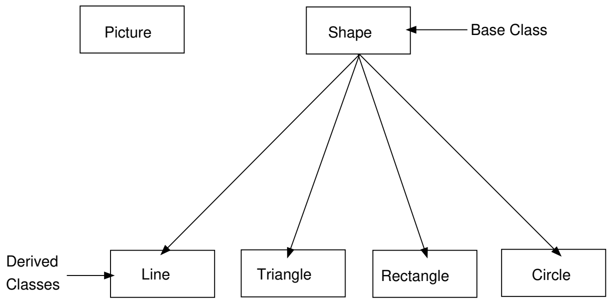

Figure 2–6 Organization of Classes for a Graphics Package

Class *Shape* declares functions applicable to any kind of shape such as *draw*() and *move*() as *virtual* functions, and implements these functions to write out an error message if called:

```
class Shape {
public:
      virtual void draw();  // Shape::draw() prints error message
      virtual void move(int dx, int dy);
                            // Shape::move() prints error message
};
```

We change the declarations of classes *Line*, *Triangle*, and so on to be derived from class *Shape* by adding the name of the base class to the declaration of the derived class; for example:

```
class Line : public Shape { ...
class Circle : public Shape { ...
```

and we also add the keyword *virtual* to the declarations of the functions *draw*() and *move*() in the derived classes. We don't have to change the implementation of these functions, however.

Now we can write class *Picture* to deal only with *Shape*s. We can represent a *Picture* by an array containing pointers to its component *Shape*s, and we can implement *Picture::draw*(), for example, simply by calling *Shape::draw*() for each shape in the picture:

```
const int PICTURE_CAPACITY = 100;
                               // max number of shapes in picture
class Picture {
      Shape* s[PICTURE_CAPACITY];// array of pointers to shapes
      int n;               // current number of shapes in picture
public:
      Picture()    { n = 0; }    // constructor
      void add(const Shape&);    // add shape to picture
      void draw();               // draw picture
      void move(int dx, int dy); // move picture
};
void Picture::add(const Shape& t) // add a shape to a picture
{
      if (n == PICTURE_CAPACITY) {
          cerr << "Picture capacity exceeded\n";
          exit(1);
      }
      s[n++] = &t;               // add pointer to shape to picture
}
void Picture::draw()           // draw a picture
{
      int i;
      for (i=0; i<n; i++) s[i]->draw();
}
```

Since *Shape::draw*() is a *virtual* function, C++ takes care of figuring out the specific class of each component *Shape* when the program is executed and calling the appropriate implementation of *draw*() for that class. This is called *dynamic binding*.

If we mistakenly forget to implement *draw*() for a derived class of *Shape*, it will inherit the implementation of *draw*() from class *Shape*. When we try to draw that shape, *Shape::draw*() will be executed, which issues an error message, as you'll recall.

Going a step further, we might want to be able to build a more complicated picture out of a number of simpler pictures. We can do this by thinking of a *Picture* as just another type of *Shape*, and making it another derived class of class *Shape*, leading to the class structure shown in **Figure 2–7** .

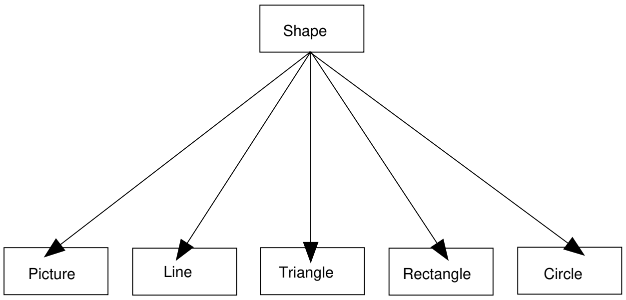

Figure 2–7 Improved Organization of Classes

Class Libraries

Taking this technique to its extreme, we can define a class named, say, *Object* and derive *every* class from it, either directly or indirectly. In class *Object* we can declare virtual functions that apply to all classes—functions for copying, printing, storing, reading, and comparing objects, for example. We then can define general data structures comprised of *Object*s and functions that operate on them that will be useful for all classes, just as class *Picture* could work with any derived class of *Shape*.

The author has written a library of about 40 general–purpose classes, modeled after the basic classes of the Smalltalk–80 programming language. The library, known as the Object–Oriented Program Support (OOPS) class library, contains classes such as *String*, Date, (hash tables), *Dictionary* (associative arrays), and *LinkedList*.

Writing C++ programs using a class library such as this is a real delight. The classes are general–purpose, and most programs of any size will have uses for some of them. They are flexible—if a particular class doesn't quite do what is needed it's usually a simple matter to derive a class that does. And the library is extensible. It provides a framework that makes it easy to add your own custom classes and make them function along with existing ones.

For example, let's see how the OOPS class library can help with the graphics package we've been discussing. The OOPS library has a class *Point* for representing x−y coordinates. We can use it in graphics classes such as *Line*:

```
class Line : public Shape {
      Point a,b;               // endpoints of the line
public:
      Line(Point p1, Point p2) { a=p1; b=p2; }  // constructor
      void draw();          // draw a line from point a to point b
      void move(Point delta);  // move line by delta
};
```

Many of the arithmetic operators are defined by class *Point*, so we can implement *move*(), for example, by writing:

```
void Line::move(Point delta)
{
      a += delta;  b += delta;
}
```

Our crude implementation of class *Picture* allocated an array of fixed size to hold the pointers to its component shapes. We can use the OOPS library class *OrderedCltn* to make this a variable−length array. An *OrderedCltn* is an array of pointers to *Object*s, so we can use it to hold pointers to instances of any class derived from *Object*, just as we used an array of pointers to *Shape*s to hold pointers to *Line*s, *Triangle*s, and so on. To make class *Shape* a derived class of *Object*, we modify its declaration:

```
class Shape : public Object { ...
```

Now we can write class *Picture as:*

```
class Picture : public Shape {
      OrderedCltn s;    // collection of pointers to shapes
public:
      Picture() {}                      // constructor
      virtual void add(const Shape&);  // add shape to picture
      virtual void draw();             // draw picture
      virtual void move(Point delta);  // move picture
};
```

Class *OrderedCltn* defines member functions such as *add*(), *remove*(), *size*(), *first*(), and *last*() to let us manipulate the pointers in the array. It also overloads the subscript operator [ ] so we can subscript *OrderedCltn*s like arrays. Using these we can write the functions *Picture::add*() and *Picture::draw* as follows:

```
void Picture::add(const Shape& t) // add a shape to a picture
{
    s.add(t);           // this calls OrderedCltn::add()
}
void Picture::draw()   // draw a picture
{
    int i;
    for (i=0; i<s.size(); i++) // s.size() returns # of objects
                               // in s
         ((Shape*)s[i])->draw(); // cast address of ith
                                 // to Shape* and call draw()
}
```

Now *Picture*s can have as many shapes in them as we need; class *OrderedCltn* manages the required storage for us.

Object I/O

Let’s write a program that uses our graphics classes to create a simple picture composed of two shapes —a line and a circle:

```
main()
{
      Picture pict;
      pict.add(*new Line(Point(0,0),Point(10,10)));
      pict.add(*new Circle(Point(10,10),2));
      pict.draw();
}
```

The first statement in the body of *main*() declares an instance of class *Picture* named *pict*, the second statement constructs an instance of *Line* with endpoints at *(0,0)* and *(10,10)* and adds it to *pict*, and the third statement constructs an instance of *Circle* with the center at (10,10) and radius 2 and also adds it to *pict*. The result is the data structure shown in **Figure 2–8** .

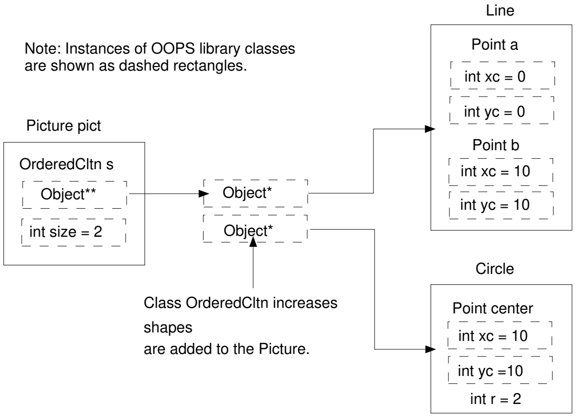

Figure 2–8 Data Structure of a Simple Picture

What if we wanted to save this data structure on a disk file so it could be read in later and used by another program? The OOPS class library makes this simple. We create an output stream (an instance of class *fstream*) named, for example, *out*, and write the picture to it with the statements:

```
#include <iostream.h>     // include header files for
#include <fstream.h>      // standard C++ stream I/O
// ...
fstream out("picturefile",output);  // create "picturefile"
pict.storeOn(out);
```

The function *storeOn*(), which is implemented in class *Object*, handles the details of finding all of the objects in the picture data structure and writing them to the output stream in a program–independent, machine–independent format. The *storeOn*() function calls the virtual function *storer*() to actually write out member variables. The *storer*() function is declared in class *Object*, and is reimplemented by each derived class to write out its own member variables. This function is already implemented for all of the

OOPS library classes, but we must write one for any classes of our own which we've derived from class *Object*. That's easy to do. For example, the *storer*() function for class *Picture* looks like:

```
void Picture::storer(iostream& strm)
{
      Shape::storer(strm); // store members of base class, if any
      s.storeOn(strm);     // store member of class Picture
}
```

To read a picture from a file, we create an input stream, *in*, (an instance of class *fstream*) connected to the file we wish to read, and read the picture from it with the statements:

```
#include <iostream.h>     // include header files for
#include <fstream.h>      // standard C++ stream I/O
// ...
fstream in("picturefile",input); // open "picturefile" read-only
readFrom(in,"Picture",pict);
```

The second argument tells *readFrom*() that we're expecting an instance of class *Picture* to be read, and to complain if the next object on the input stream is of any other class.

The function *readFrom*() works somewhat like *storeOn*(), calling a small "reader" function which we must write for each of our classes.

We can use OOPS object I/O to store and read an arbitrarily complex data structure containing instances of both OOPS library classes and our own classes. Since the data structure is converted into a program−independent, machine−independent format, we can send it through a UNIX pipe to another process running on the same machine, or over a network to another process running on a different kind of machine. This capability is particularly useful for spread sheets, forms, documents, drawings, electronic mail, and so on. The OOPS class library also gives us a framework to use when implementing object I/O for our own classes. We don't have to spend time designing a storage format, or worry about such issues as what to do with the pointers in a data structure, for example. We can use the general−purpose mechanism provided by the OOPS class library, and concentrate on our particular application.

The Current Status of C++

The C++ programming language is currently implemented as a *translator*, which accepts C++ source code as input and produces C source code as output. The C++ translator and run−time support library are written in C++, making them easily portable to most UNIX systems. AT&T first made the C++ translator available to universities and non−profit organizations in December, 1984. Release 1.0 became commercially available as an unsupported product in October, 1985.

The AT&T C++ Language System can run on any UNIX machine capable of running programs up to about 500KB in size, and having a robust C compilation system that can handle variable and external symbol names of arbitrary length. The C compiler must also allow structure assignments and the use of structures as function arguments and return values.

Training and third−party supported ports of the AT&T C++ Translator can be obtained for various UNIX systems, VAX VMS, MS−DOS, and others.

The Future of C++

The definition of the C++ programming language is not final. C++ will undoubtedly be revised to eliminate any unnecessary incompatibilities; for example, the ANSI C rules for doing floating−point arithmetic will be adopted. Historically, C++ has met the challenge of evolving while remaining compatible with C and earlier versions of C++.

Will the C++ programming language be as successful as its predecessor, or will it become just another of the countless languages that never achieve widespread use? Well, C++ has a lot going for it:

- Since C++ is, with a few minor exceptions, a superset of C, it has no fatal deficiencies. It also

possesses those attributes of C that have contributed to C's success: portability, flexibility, and efficiency.
- C++ is less error–prone than C. It thoroughly type–checks programs, as is the trend in modern programming languages, but not at the expense of flexibility or convenience. A programmer may coerce (cast) types when necessary, and define his or her own implicit type conversions for convenience.
- Support for data abstraction and object–oriented programming make C++ a much more powerful and expressive language than C. Yet the language remains one of manageable size, much smaller than PL/1 or ADA, for example.
- C++ programs are compatible with UNIX and with the large number of existing C libraries for graphics, database management, math, and statistics.
- There is a large existing community of C programmers who can begin to use C++ immediately, gradually learning and utilizing its new features.
- The AT&T C++ Language System is commercially available in source form, is inexpensive, and is highly portable. It makes the language accessible on almost all popular operating systems.
- AT&T is developing a portable C++ compiler, which will compile C++ programs more quickly than the combination of the C++ Translator and C compiler now required.
- C++ was designed at the AT&T Bell Laboratories Computer Science Research Center in Murray Hill. They have an impressive track record in producing successful software, such as the UNIX system and C language.

The main obstacle to the widespread adoption of C++ is that to realize its benefits one must master the techniques of data abstraction and/or object–oriented programming—techniques that are unfamiliar to the current generation of programmers. When this educational problem is solved, C++ should succeed C as the language of choice for a wide range of applications.

Chapter 3

An Overview of C++

This chapter describes the features added to C to be a better C, support data abstraction, and support object–oriented programming.

This chapter contains the following sections:
- **"Introduction"**
- **"What is Good about C?"**
- **"A Better C"**
- **"Support for Data Abstraction"**
- **"Support for Object–Oriented Programming"**

Note: This chapter is from a paper by Bjarne Stroustrup.

Introduction

C++ is a general–purpose programming language designed to make programming more enjoyable for the serious programmer. Except for minor details, C++ is a superset of the C language. C++ was designed to:
- be a better C
- support data abstraction
- support object–oriented programming

In addition to C, the main influences on the design of C++ were Simula67 and Algol68.

C++ has been in use for about four years and has been applied to most branches of systems programming including compiler construction, data base management, graphics, image processing, music synthesis, networking, numerical software, programming environments, robotics, simulation, and switching. It has a highly portable implementation and there are now thousands of installations including AT&T 3B, DEC VAX, Intel 80286, Motorola 68000, and Amdahl machines running UNIX and other operating systems.

What is Good about C?

C is clearly not the cleanest language ever designed nor the easiest to use; so why do so many people use it?

- C is flexible: it is possible to apply C to most every application area, and to use most every programming technique with C. The language has no inherent limitations that preclude particular kinds of programs being written.
- C is efficient: the semantics of C are "low−level"; that is, the fundamental concepts of C mirror the fundamental concepts of a traditional computer. Consequently, it is relatively easy for a compiler and/or a programmer to utilize hardware resources for a C program efficiently.
- C is available: given a computer, whether the tiniest micro or the largest super−computer, the chance is that there is an acceptable quality C compiler available and that that C compiler supports an acceptably complete and standard C language and library. There are also libraries and support tools available, so that a programmer rarely needs to design a new system from scratch.
- C is portable: a C program is not automatically portable from one machine (and operating system) to another nor is such a port necessarily easy to do. It is, however, usually possible and the level of difficulty is such that porting even major pieces of software with inherent machine dependences is typically technically and economically feasible.

Compared with these "first order" advantages, the "second order" drawbacks like the curious C declarator syntax and the lack of safety of some language constructs become less important. Designing "a better C" implies compensating for the major problems involved in writing, debugging, and maintaining C programs *without compromising the advantages of C*. C++ preserves all these advantages and compatibility with C at the cost of abandoning claims to perfection and of some compiler and language complexity. However, designing a language "from scratch" does not ensure perfection and the C++ compilers compare favorably in run−time, have better error detection and reporting, and equal the C compilers in code quality.

A Better C

The first aim of C++ is to be "a better C" by providing better support for the styles of programming for which C is most commonly used. This primarily involves providing features that make the most common errors unlikely (since C++ is a superset of C such errors cannot simply be made impossible).

Argument Type Checking and Coercion

The most common error in C programs is a mismatch between the type of a function argument and the type of the argument expected by the called function. For example:

```
double sqrt(a) double a;
{
      /* ... */
}
double sq2 = sqrt(2);
```

Since C does not check the type of the argument 2, the call *sqrt*(2) will typically cause a run−time error or give a wrong result when the square root function tries to use the integer 2 as a double−precision floating−point number. In C++, this program will cause no problem since 2 will be converted to a floating−point number at the point of the call. That is, *sqrt*(2) is equivalent to *sqrt*((*double*)2).

Where an argument type does not match the argument type specified in the function declaration and no type conversion is defined the compiler issues an error message. For example, in C++ *sqrt*() causes a compile−time error.

Naturally, the C++ syntax also allows the type of arguments to be specified in function declarations:

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

and a matching function definition syntax is also introduced:

```
double sqrt(double d)
{
      // ...
}
```

Inline Functions

Most C programs rely on macros to avoid function call overhead for small frequently−called operations. Unfortunately the semantics of macros are very different from the semantics of functions so the use of macros has many pitfalls. For example:

```
#define mul(a,b) a*b
int z = mul(x*3+2,y/4);
```

Here *z* will be wrong since the macro will expand to *x*3+2*y/4.* Furthermore, C macro definitions do not follow the syntactic rules of C declarations, nor do macro names follow the usual C scope rules. C++ circumvents such problems by allowing the programmer to declare *inline* functions:

```
inline int mul(int a, int b) { return a*b; }
```

An inline function has the same semantics as a "normal" function but the compiler can typically inline expand it so that the code−space and run−time efficiency of macros are achieved.

Scoped and Typed Constants

Since C does not have a concept of a symbolic constant macros are used. For example:

```
#define TBLMAX (TBLSIZE-1)
```

Such "constant macros" are neither scoped nor typed and can (if not properly parenthesized) cause problems similar to those of other macros. Furthermore, they must be evaluated each time they are used and their names are "lost" in the macro expansion phase of the compilation and consequently are not known to symbolic debuggers and other tools. In C++ constants of any type can be declared:

```
const int TBLMAX = TBLSIZE-1;
```

Varying Numbers of Arguments

Functions taking varying numbers of arguments and functions accepting arguments of different types are common in C. They are a notable source of both convenience and errors.

C functions where the type of arguments or the number of arguments (but not both) can vary can be handled in a simple and type−secure manner in C++. For example, a function taking one, two, or three arguments of known type can be handled by supplying default argument values which the compiler uses when the programmer leaves out arguments. For example:

```
void print(char*, char* = "-", char* = "-");
print("one", "two", "three");
print("one", "two");  // that is, print("one", "two", "-");
print("one");         // that is, print("one", "-", "-");
```

Some C functions take arguments of varying types to provide a common name for functions performing similar operations on objects of different types. This can be handled in C++ by overloading a function name. That is, the same name can be used for two functions provided the argument types are sufficiently different to enable the compiler to "pick the right one" for each call. For example:

```
void print(int);
void print(char*);
print(1);      // integer print function
print("two");  // string print function
```

The most general examples of C functions with varying arguments cannot be handled in a type−secure manner. Consider the standard output function *printf*(), which takes a format string followed by an arbitrary collection of arguments supposedly matching the format string:

A C++ I/O system that avoids the type insecurity of the *printf* approach is described in *"The C++ Programming Language."*

```
printf("a string");
printf("x = %d\m",x);
printf("name: %s\m size: %d\n", obj.name, obj.size);
```

However, in C++ one can specify the type of initial arguments and leave the number and type of the remaining arguments unspecified. For example, *printf* and its variants can be declared like this:

```
int printf(const char* ...);
int fprintf(FILE*, const char* ...);
int sprintf(char*, const char* ...);
```

These declarations allow the compiler to catch errors such as

```
printf(stderr,"x = %d\m",x);  // error: printf does not take
                                // a FILE*
fprintf("x = %d\m",x);        // error: printf needs a FILE*
```

Declarations as Statements

Uninitialized variables are another common source of errors. One cause of this class of errors is the requirement of the C syntax that declarations can occur only at the beginning of a block (before the first statement). In C++, a declaration is considered a kind of statement and can consequently be placed anywhere. It is often convenient to place the declaration where it is first needed so that it can be initialized immediately. For example:

```
void some_function(char* p)
{
      if (p==0) error("p==0 in some_function");
      int length = strlen(p);
      // ...
}
```

Support for Data Abstraction

C++ provides support for data abstraction: the programmer can define types that can be used as conveniently as built-in types and in a similar manner. Arithmetic types such as rational and complex numbers are common examples:

```
class complex {
      double re, im;
public:
      complex(double r, double i) { re=r; im=i; }
      complex(double r) { re=r; im=0; }  // float->complex
                                         // conversion
      friend complex operator+(complex, complex);
      friend complex operator-(complex, complex); // binary minus
      friend complex operator-(complex);          // unary minus
      friend complex operator*(complex, complex);
      friend complex operator/(complex, complex);
      // ...
}
```

The declaration of class (that is, user-defined type) *complex* specifies the representation of a complex number and the set of operations on a complex number. The representation is *private;* that is, *re* and *im* are accessible only to the functions defined in the declaration of class *complex.* Such functions can be defined like this:

```
complex operator+(complex a1, complex a2)
{
      return complex(a1.re+a2.re, a1.im+a2.im);
}
```

and used like this:

```
main()
{
      complex a = 2.3;
      complex b = 1/a;
      complex c = a+b+complex(1,2.3);
      // ...
}
```

Functions declared in a class declaration using the keyword *friend* are called *friend functions.* They do not differ from ordinary functions except that they may use private members of classes that name them friends. A function can be declared as a friend of more than one class. Other functions declared in a class declaration are called *member functions.* A member function is in the scope of the class and must be invoked for a specific object of that class.

Initialization and Cleanup

When the representation of a type is hidden some mechanism must be provided for a user to initialize variables of that type. A simple solution is to require a user to call some function to initialize a variable before using it. This is error prone and inelegant. A better solution is to allow the designer of a type to provide a distinguished function to do the initialization. Given such a function, allocation and initialization of a variable becomes a single operation (often called instantiation) instead of two separate operations. Such an initialization function is called a constructor. In cases where construction of objects of a type is non−trivial one often needs a complementary operation to clean up objects after their last use. In C++ such a cleanup function is called a destructor. Consider a vector type:

```
class vector {
      int  sz;          // number of elements
      int* v;           // pointer to integers
public:
      vector(int);      // constructor
      ~vector();        // destructor
      // ...
};
```

The *vector* constructor can be defined to allocate a suitable amount of space like this:

```
vector::vector(int s)
{
      if (s<=0) error("bad vector size");
      sz = s;
      v = new int[s];   // allocate an array of "s" integers
}
```

The cleanup done by the *vector* destructor consists of freeing the storage used to store the vector elements for re−use by the free store manager:

```
vector::~vector()
{
      delete v;         // deallocate the memory pointed to
                        // by v
}
```

C++ does not support garbage collection. This is, however, compensated for by enabling a type to maintain its own storage management without requiring intervention from a user. Class *vector* is an maintain its own storage management without requiring intervention from a user. Class *vector* is an example of this.

Free Store Operators

The operators *new* and *delete* were introduced to provide a standard notation for free store allocation and deallocation. A user can provide alternatives to their default implementations by defining functions called *operator new* and *operator delete.* For built-in types the *new* and *delete* operators provide only a notational convenience (compared with the standard C functions *malloc*() and *free*()). For user-defined types such as *vector* the free store operators ensure that constructors and destructors are called properly:

```
vector* fct1(int n)
{
      vector v(n);              // allocate a vector on the stack
                                // the constructor is called
      vector* p = new vector(n); // allocate a vector on the
                                 // free store
                                 // the constructor is called
      // ...
      return p;
      // the destructor is implicitly called for "v" here
}
void fct2()
{
      vector* pv = fct1(10);
      // ...
      delete pv; // call the destructor and free the store
}
```

References

C provides (only) "call by value" semantics for function argument passing; "call by reference" can be simulated by explicit use of pointers. This is sufficient, and often preferable to using "pass by value" for the built-in types of C. However, it can be inconvenient for larger objects

As indicated by an inconsistency in the C semantics, arrays are always passed by reference.

and can get seriously in the way of defining conventional notation for user-defined types in C++. Consequently, the concept of a *reference* is introduced. A reference acts as a name for an object; *T&* means reference to *T.* A reference must be initialized and becomes an alternative name for the object it is initialized with. For example:

```
int a = 1;   // "a" is an integer initialized to "1"
int& r = a;  // "r" is a reference initialized to "a"
```

The reference *r* and the integer *a* can now be used in the same way and with the same meaning. For example:

```
int b = r; // "b" is initialized to the value of "r",
             // that is, "1"
r = 2;   // the value of "r", that is, the value of "a"
           // becomes "2"
```

References enable variables of types with "large representations" to be manipulated efficiently without explicit use of pointers. Constant references are particularly useful:

```
matrix operator+(const matrix& a, const matrix& b)
{
      // code here cannot modify the value of "a" or "b"
}
matrix a = b+c;
```

In such cases the "call by value" semantics are preserved while achieving the efficiency of "call by reference."

Assignment and Initialization

Controlling construction and destruction of objects is sufficient for many, but not all, types. It can also be necessary to control all copy operations. Consider:

```
vector v1(100); // make v1 a vector of 100 elements
vector v2 = v1; // make v2 a copy of v1
v1 = v2;        // assign v1 to v2 (that is,
                  // copy the elements)
```

Declaring a function with the name *operator=* in the declaration of class *vector* specifies that vector assignment is to be implemented by that function:

```
class vector {
      int* v;
      int  sz;
public:
      // ...
      void operator=(vector&);  // assignment
};
```

Assignment might be defined like this:

```
void vector::operator=(vector& a) // check size and
                                    // copy elements
{
      if (sz != a.sz) error("bad vector size for =");
      for (int i = 0; i<sz; i++) v[i] = a.v[i];
}
```

Since the assignment operation relies on the "old value" of the vector assigned to, it cannot be used to implement initialization of one vector with another.

What is needed is a constructor that takes a vector argument:

```
class vector {
      // ...
      vector(int);       // create vector
      vector(vector&);   // create vector and copy elements
};
vector::vector(vector& a) // initialize a vector from
                            // another vector
{
      sz = a.sz;            // same size
      v = new int[sz];      // allocate element array
      for (int i = 0; i<sz; i++) v[i] = a.v[i];     // same
                                                       // values
}
```

A constructor like this (of the form *X*(*X&*)) is used to handle all initialization. This includes arguments passed "by value" and function return values:

```
vector v2 = v1;
          // use vector(vector&) constructor to initialize
void f(vector);
f(v2);  // use vector(vector&) constructor to pass a copy
          // of v2
vector g(int sz)
{
      vector v(sz);
```

```
      return v;
      // use vector(vector&) constructor to return a copy of v
}
```

Operator Overloading

As demonstrated above, standard operators like + , − , * , / can be defined for user−defined types, as can assignment and initialization in its various guises. In general, all the standard operators with the exception of

```
.  ?:
```

can be overloaded. The subscripting operator "[ ]" and the function application operator () have proven particularly useful. The C "operator assignment" operators, such as += and *= , have also found many uses.

It is not possible to redefine an operator when applied to built−in data types, to define new operators, or to redefine the precedence of operators.

Coercions

User−defined coercions, like the one from floating−point numbers to complex numbers implied by the constructor *complex*(*double*)*,* have proven unexpectedly useful in C++. Such coercions can be applied explicitly or the programmer can rely on the compiler adding them implicitly where necessary and unambiguous:

```
      complex a = complex(1);
      complex b = 1;           // implicit: 1 -> complex(1)
      a = b+complex(2);
      a = b+2;                 // implicit: 2 -> complex(2)
      a = 2+b;                 // implicit: 2 -> complex(2)
```

Coercions were introduced into C++ because mixed mode arithmetic is the norm in languages used for numerical work and because most user−defined types used for "calculation" (for example, matrices, character strings, and machine addresses) have natural mappings to and/or from other types.

Great care is taken (by the compiler) to apply user−defined conversions only where a unique conversion exists. Ambiguities caused by conversions are compile−time errors.

It is also possible to define a conversion to a type without modifying the declaration of that type. For example:

```
class point {
      float dist;
      float angle;
public:
      // ...
      operator complex()  // convert point to complex number
      {
          return polar(dist,angle);
      }
      operator double()   // convert point to real number
      {
          if (angle) error(
             "cannot convert point to real: angle!=0");
          return dist;
      }
};
```

These conversions could be used like this:

```
void some_function(point a)
{
```

```
{
      complex z = a;    // z = a.operator complex()
      double d = a;     // d = a.operator double()
      complex z3 = a+3; // z3 = a.operator complex()+complex(3);
      // ...
}
```

This is particularly useful for defining conversions to built−in types since there is no declaration for a built−in type for the programmer to modify. It is also essential for defining conversions to "standard" user−defined types where a change may have (unintentionally) wide ranging ramifications and where the average programmer has no access to the declaration.

Support for Object−Oriented Programming

C++ provides support for object−oriented programming: the programmer can define class hierarchies and a call of a member function can depend on the actual type of an object (even where the actual type is unknown at compile−time). That is, the mechanism that handles member function calls handles the case where it is known at compile−time that an object belongs to *some* class in a hierarchy, but exactly *which* class can only be determined at run−time. See examples below.

Derived Classes

C++ provides a mechanism for expressing commonality among different types by explicitly defining a class to be part of another. This allows re−use of classes without modification of existing classes and without replication of code. For example, given a class *vector:*

```
class vector {
      // ...
public:
      // ...
      vector(int);
      int& operator[](int);
          // overload the subscripting operator: []
}
```

one might define a vector for which a user can define the index bounds:

```
class vec : public vector {
      int low, high;
public:
      vec(int, int);
      int& operator[](int);
};
```

Defining *vec* as

```
: public vector
```

means that first of all a *vec* is a *vector.* That is, type *vec* has ("inherits") all the properties of type *vector* in addition to the ones declared specifically for it. Class *vector* is said to be the *base* class for *vec,* and conversely *vec* is said to be *derived* from *vector.*

Class *vec* modifies class *vector* by providing a different constructor, requiring the user to specify the two index bounds rather than the size, and by providing its own access function *operator*[](). A *vec 's operator*[]() is easily expressed in terms of *vector 's operator*[]()*:*

```
int& vec::operator[](int i)
{
      return vector::operator[](i−low);
}
```

The scope resolution operator *::* is used to avoid getting caught in an infinite recursion by calling *vec::*

operator[]() from itself. Note that *vec::operator*[]() *had* to use a function like *vector::operator*[]() to access elements. It could not just use *vector 's* members *v* and *sz* directly since they were declared *private* and therefore accessible only to *vector 's* member functions.

The constructor for *vec* can be written like this:

```
vec::vec(int lb, int hb) : vector(hb-lb+1)
{
      if (hb-lb<0) hb = lb;
      low = lb;
      high = hb;
}
```

The construct *:vector(hb−lb+1)* is used to specify the argument list needed for the base class constructor *vector*().

Class *vec* can be used like this:

```
void some_function(int l, int h)
{
      vec v1(l,h);
      const int sz = h-l+1;
      vector v2(sz);
      // ...
      for (int i=0; i<sz; i++) v2[i] = v1[l+i];
               // copy elements explicitly
      v2 = v1; // copy elements by using vector::operator=()
}
```

Virtual Functions

Class derivation (often called subclassing) is a powerful tool in its own right but a facility for run−time type resolution is needed to support object−oriented programming.

Consider defining a type *shape* for use in a graphics system. The system has to support circles, triangles, squares, and many other shapes. First specify a class that defines the general properties of all shapes:

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

The functions for which the calling interface can be defined, but where the implementation cannot be defined except for a specific shape, have been marked *virtual* (the Simula67 and C++ term for "to be defined later in a class derived from this one"). Given this definition one can write general functions manipulating shapes:

```
void rotate_all(shape* v, int size, int angle)
// rotate all members of vector "v" of size "size" "angle"
// degrees
{
      for (int i = 0; i < size; i++) v[i].rotate(angle);
}
```

For each shape *v*[*i*], the proper *rotate* function for the actual type of the object will be called. That "actual type" is not known at compile−time.

To define a particular shape we must say that it is a shape (that is, derive it from class *shape*) and specify its particular properties (including the virtual functions):

```
class circle : public shape {
      int radius;
public:
      void draw() { /* ... */ };
      void rotate(int) {}    // yes, the null function
};
```

In many contexts it is important that the C++ virtual function mechanism is very nearly as efficient as a "normal" function call. The additional run−time overhead is about 4 memory references (dependent on the machine architecture and the compiler) and the memory overhead is one word per object plus one word per virtual function per class.

Visibility Control

The basic scheme for separating the (public) user interface from the (private) implementation details has worked out very well for data abstraction uses of C++. It matches the idea that a type is a black box. It has proven to be less than ideal for object−oriented uses.

The problem is that a class defined to be part of a class hierarchy is not simply a black box. It is often primarily a building block for the design of other classes. In this case the simple binary choice *public/private* can be constraining. A third alternative is needed: a member should be private as far as functions outside the class hierarchy are concerned but accessible to member functions of a derived class in the same way that it is accessible to members of its own class. Such a member is said to be *protected.*

For example, consider a class *node* for some kind of tree:

```
class node {
      // private stuff
protected:
      node* left;
      node* right;
      // more protected stuff
public:
      virtual void print();
      // more public stuff
};
```

The pointers *left* and *right* are inaccessible to the general user but any member function of a class derived from class *node* can manipulate the tree without overhead or inconvenience.

The protection/hiding mechanism applies to names independently of whether a name refers to a function or a data member. This implies that one can have *private* and *protected* function members. Usually it is good policy to keep data *private* and present the *public* and *protected* interfaces as sets of functions. This policy minimizes the effect of changes to a class on its users and consequently maximizes its implementer’s freedom to make changes.

Another refinement of the basic inheritance scheme is that it is possible to inherit public members of a base class in such a way that they do not become public members of the derived class. This can be used to provide restricted interfaces to standard classes. For example:

```
class dequeue {
      // ...
      void insert(elem*);
      void append(elem*);
      elem* remove();
};
```

Given a dequeue a *stack* can be defined as a derived class where only the *insert*() and *remove*()

Given a dequeue a *stack* can be defined as a derived class where only the *insert*() and *remove*() operations are defined:

```
class stack : private dequeue {
          // note: just ":" not ": public" members
          // of dequeue are private members of stack
public:
      dequeue::insert;  // make "insert" a public member of stack
      dequeue::remove;  // make "remove" a public member of stack
};
```

Alternatively, inline functions can be defined to give these operations the conventional names:

```
class stack : private dequeue {
public:
      void push(elem* ee) { dequeue::insert(ee); }
      elem* pop() { return dequeue::remove(); }
};
```

What is Missing?

C++ was designed under severe constraints of compatibility, internal consistency, and efficiency: no feature was included that

- would cause a serious incompatibility with C at the source or linker levels
- would cause run−time or space overheads for a program that did not use it
- would increase run−time or space requirements for a C program
- would significantly increase the compile−time compared with C
- could only be implemented by making requirements of the programming environment (linker, loader, etc.) that could not be simply and efficiently implemented in a traditional C programming environment

Features that might have been provided but weren't because of these criteria include garbage collection, parameterized classes, exceptions, support for concurrency, and integration of the language with a programming environment. Not all of these possible extensions would actually be appropriate for C++ and unless great constraint is exercised when selecting and designing features for a language a large, unwieldy, and inefficient mess will result. The severe constraints on the design of C++ have probably been beneficial and will continue to guide the evolution of C++.

Conclusions

C++ has succeeded in providing greatly improved support for traditional C−style programming without added overhead. In addition, C++ provides sufficient language support for data abstraction and object−oriented programming in demanding (both in terms of machine utilization and application complexity) real−life applications. C++ continues to evolve to meet demands of new application areas. There still appears to be ample scope for improvement even given the (self imposed) Draconian criteria for compatibility, consistency, and efficiency. However, currently the most active areas of development are not the language itself but libraries and support tools in the programming environment.

Chapter 4

Object−Oriented Programming

This chapter discusses features necessary to support these programming styles in a general−purpose programming language. It presents one view of what "object−oriented" ought to mean in the context of a general−purpose programming language. The presentation centers around C++ but is not limited to facilities provided by that language.

Examples will be presented in C++. The reason for this is partly to introduce C++ and partly because C++ is one of the few languages that supports both data abstraction and object−oriented programming in addition to traditional programming techniques. Issues of concurrency and of hardware support for specific higher−level language constructs are ignored.

This chapter contains the following sections:

Note: This chapter is from a paper by Bjarne Stroustrup.

Introduction

"Object−Oriented Programming" and "Data Abstraction" have become very common terms. Unfortunately, few people agree on what they mean. I will offer informal definitions that appear to make sense in the context of languages like Ada, C++, Modula−2, Simula, and Smalltalk. The general idea is to equate "support for data abstraction" with the ability to define and use new types and equate "support for object−oriented programming" with the ability to express type hierarchies.

Not all programming languages can be "object−oriented." Yet claims have been made to the effect that APL, Ada, Clu, C++, CLOS, and Smalltalk are object−oriented programming languages. I have heard discussions of object−oriented design in C, Pascal, Modula−2, and CHILL, and proponents of object−oriented Fortran and Cobol programming are now appearing. "Object−oriented" has become a high−tech synonym for "good," and when you examine discussions in the trade press, you can find arguments that appear to boil down to syllogisms like:

Object−oriented is good
−−−−−−−−−−−−−−−−−−−−−−−−−−−
Ada is object−oriented

We simply *must* be more careful with our concepts and logic. In the context of a general−purpose programming language "object−oriented" should mean that it:

- Distinguishes "object−oriented programming" and "data abstraction" from each other and from other styles of programming, and presents the mechanisms that are essential for supporting the various styles of programming.
- Presents features needed to make data abstraction effective.
- Discusses facilities needed to support object−oriented programming.
- Presents some limits imposed on data abstraction and object−oriented programming by traditional hardware architectures and operating systems.

Programming Paradigms

Object−oriented programming is a technique for programming a paradigm for writing "good" programs for a set of problems. If "object−oriented programming language" means anything it must mean a programming language that provides mechanisms that support the object−oriented style of programming well. There is an important distinction here. A language is said to *support* a style of programming if it provides facilities that makes it convenient (reasonably easy, safe, and efficient) to use that style. A language does not support a technique if it takes exceptional effort or skill to write such programs; it merely *enables* the technique to be used. For example, you can write structured programs in Fortran, write type−secure programs in C, and use data abstraction in Modula−2, but it is unnecessarily hard to do because these languages do not support those techniques.

Support for a paradigm comes not only in the obvious form of language facilities that allow direct use of the paradigm, but also in the more subtle form of compile−time and/or run−time checks against unintentional deviation from the paradigm. Type checking is the most obvious example of this; ambiguity detection and run−time checks can be used to extend linguistic support for paradigms. Extra−linguistic facilities such as standard libraries and programming environments can also provide significant support for paradigms.

A language is not necessarily better than another because it possesses a feature the other does not. There are many examples to the contrary. The important issue is not so much what features a language possesses but that the features it does possess are sufficient to support the desired programming styles in the desired application areas:

1. All features must be cleanly and elegantly integrated into the language.
2. It must be possible to use features in combination to achieve solutions that would otherwise have required extra separate features.
3. There should be as few "special–purpose" features as possible.
4. A feature should be such that its implementation does not impose significant overheads on programs that do not require it.
5. A user need only know about the subset of the language explicitly used to write a program.

The last two principles can be summarized as "what you don't know won't hurt you." If there are any doubts about the usefulness of a feature it is better left out. It is *much* easier to add a feature to a language than to remove or modify one that has found its way into the compilers or the literature.

Procedural Programming

The original (and probably still the most commonly used) programming paradigm is:

Decide which procedures you want;
use the best algorithms you can find.

The focus is on the design of the processing, the algorithm needed to perform the desired computation. Languages support this paradigm by facilities for passing arguments to functions and returning values from functions. The literature related to this way of thinking is filled with discussion of ways of passing arguments, ways of distinguishing different kinds of arguments, different kinds of functions (procedures, routines, macros, ...), etc. Fortran is the original procedural language; Algol60, Algol68, C and Pascal are later inventions in the same tradition.

A typical example of "good style" is a square root function. Given an argument, it produces a result. To do this, it performs a well understood mathematical computation:

```
double sqrt(double arg)
{
      // the code for calculating a square root
}
void some_function()
{
      double root2 = sqrt(2);
      // ...
}
```

From a program organization point of view, functions are used to create order in a maze of algorithms.

Data Hiding

Over the years, the emphasis in the design of programs has shifted away from the design of procedures towards the organization of data. Among other things, this reflects an increase in program size. A set of related procedures with the data they manipulate is often called a *module.* The programming paradigm becomes:

Decide which modules you want;
partition the program so that data is hidden in modules.

This paradigm is also known as the "data hiding principle." Where there is no grouping of procedures with related data the procedural programming style suffices. In particular, the techniques for designing "good procedures" are now applied for each procedure in a module. The most common example is a definition of a stack module. The main problems that have to be solved are:

1. Provide a user interface for the stack (for example, functions *push*() and *pop*()).
2. Ensure that the representation of the stack (for example, a vector of elements) can only be

accessed through this user interface.

3. Ensure that the stack is initialized before its first use.

Here is a plausible external interface for a stack module:

```
// declaration of the interface of module stack of characters
char pop();
void push(char);
const stack_size = 100;
```

Assuming that this interface is found in a file called *stack.h,* the "internals" can be defined like this:

```
#include "stack.h"
static char v[stack_size];
               // "static" means local to this file/module
static char* p = v;        // the stack is initially empty
char pop()
{
      // check for underflow and pop
}
void push(char c)
{
      // check for overflow and push
}
```

It would be quite feasible to change the representation of this stack to a linked list. A user does not have access to the representation anyway (since *v* and *p* were declared *static,* that is, local to the file/module in which they were declared). Such a stack can be used like this:

```
#include "stack.h"
void some_function()
{
      push('c');
      char c = pop();
      if (c != 'c') error("impossible");
}
```

Pascal (as originally defined) doesn't provide any satisfactory facilities for such grouping: the only mechanism for hiding a name from "the rest of the program" is to make it local to a procedure. This leads to strange procedure nestings and over−reliance on global data.

C fares somewhat better. As shown in the example above, you can define a "module" by grouping related function and data definitions together in a single source file. The programmer can then control which names are seen by the rest of the program (a name can be seen by the rest of the program *unless* it has been declared *static*). Consequently, in C you can achieve a degree of modularity. However, there is no generally accepted paradigm for using this facility and the technique of relying on *static* declarations is rather low−level.

One of Pascal's successors, Modula−2, goes a bit further. It formalizes the concept of a module, making it a fundamental language construct with well defined module declarations, explicit control of the scopes of names (import/export), a module initialization mechanism, and a set of generally known and accepted styles of usage.

The differences between C and Modula−2 in this area can be summarized by saying that C only *enables* the decomposition of a program into modules, while Modula−2 *supports* that technique.

Data Abstraction

Programming with modules leads to the centralization of all data of a type under the control of a type manager module. If one wanted two stacks, one would define a stack manager module with an interface like this:

```
class stack_id;  // stack_id is a type
```

```
          // no details about stacks or stack_ids are known here
stack_id create_stack(int size);
          // make a stack and return its identifier
destroy_stack(stack_id);
          // call when stack is no longer needed
void push(stack_id, char);
char pop(stack_id);
```

This is an improvement over the traditional unstructured code, but "types" implemented this way are clearly very different from the built−in types in a language. Each type manager module must define a separate mechanism for creating "variables" of its type. There is no established norm for assigning object identifiers. A "variable" of such a type has no name known to the compiler or programming environment, nor do such "variables" obey the usual scope rules or argument passing rules. A type created through a module mechanism is different from a built−in type and enjoys support inferior to the support provided for built−in types. For example:

```
void f()
{
      stack_id s1;
      stack_id s2;
      s1 = create_stack(200);
      // Oops: forgot to create s2
      push(s1,'a');
      char c1 = pop(s1);
      if (c1 != 'a') error("impossible");
      push(s2,'b');
      char c2 = pop(s2);
      if (c2 != 'b') error("impossible");
      destroy_stack(s2);
      // Oops: forgot to destroy s1
}
```

In other words, the module concept that supports the data hiding paradigm enables this style of programming, but does not support it.

Languages such as Ada, Clu, and C++ attack this problem by allowing a user to define types that behave almost the same way as built−in types. Such a type is often called an *abstract data type*. I prefer the term "user−defined type." A way of defining types that are somewhat more abstract is demonstrated in the **"Multiple Implementations"** below.

I prefer the term "user defined type": "Those *types are not "abstract"; they are as real as int* and *float."*—Doug McIlroy. An alternative definition of *abstract data types* would require a mathematical "abstract" specification of all types (both built−in and user−defined). What are referred to as types in this paper would, given such a specification, be concrete specifications of such truly abstract entities.

What are referred to as types in this paper would, given such a specification, be concrete specifications of such truly abstract entities. The programming paradigm becomes:

Decide which types you want;
provide a full set of operations for each type.

Where there is no need for more that one object of a type the data hiding programming style using modules suffices. Arithmetic types such as rational and complex numbers are common examples of user−defined types:

```
class complex {
      double re, im;
public:
      complex(double r, double i) { re=r; im=i; }
      complex(double r) { re=r; im=0; }   // float->complex conversion
      friend complex operator+(complex, complex);
      friend complex operator-(complex, complex);    // binary minus
      friend complex operator-(complex);             // unary minus
      friend complex operator*(complex, complex);
```

```
      friend complex operator/(complex, complex);
      // ...
};
```

Declaration of user−defined type class *complex* specifies representation of and the set of operations on a complex number. Representation is *private;* that is, *re* and *im* are accessible only to the functions specified in the declaration of class *complex.* Such functions can be defined like this:

```
complex operator+(complex a1, complex a2)
{
      return complex(a1.re+a2.re,a1.im+a2.im);
}
```

and used like this:

```
complex a = 2.3;
complex b = 1/a;
complex c = a+b*complex(1,2.3);
// ...
c = −(a/b)+2;
```

Most, but not all, modules are better expressed as user−defined types. For concepts where the "module representation" is desirable even when a proper facility for defining types is available, the programmer can declare a type and only a single object of that type. Alternatively, a language might provide a module concept in addition to and distinct from the class concept.

Problems with Data Abstraction

An abstract data type defines a sort of black box. Once it has been defined, it does not really interact with the rest of the program. There is no way of adapting it to new uses except by modifying its definition. This can lead to severe inflexibility. Consider defining a type *shape* for use in a graphics system. Assume for the moment that the system has to support circles, triangles, and squares. Assume also that you have some classes:

```
class point{ /* ... */ };
class color{ /* ... */ };
```

You might define a shape like this:

```
enum kind { circle, triangle, square };
class shape {
      point center;
      color col;
      kind k;
      // representation of shape
public:
      point where()       { return center; }
      void move(point to) { center = to; draw(); }
      void draw();
      void rotate(int);
      // more operations
};
```

The "type field" *k* is necessary to allow operations such as *draw*() and *rotate*() to determine what kind of shape they are dealing with (in a Pascal−like language, one might use a variant record with tag *k*). The function *draw*() might be defined like this:

```
void shape::draw()
{
      switch (k) {
      case circle:
          // draw a circle
          break;
```

```
                break;
            case triangle:
                // draw a triangle
                break;
            case square:
                // draw a square
            }
}
```

This is a mess. Functions such as *draw*() must "know about" all the kinds of shapes there are. Therefore the code for any such function grows each time a new shape is added to the system. If you define a new shape, every operation on a shape must be examined and (possibly) modified. You are not able to add a new shape to a system unless you have access to the source code for every operation. Since adding a new shape involves "touching" the code of every important operation on shapes, it requires great skill and potentially introduces bugs into the code handling other (older) shapes. The choice of representation of particular shapes can get severely cramped by the requirement that (at least some of) their representation must fit into the typically fixed sized framework presented by the definition of the general type *shape*.

Object−Oriented Programming

The problem is that there is no distinction between the general properties of any shape (a shape has a color, it can be drawn, etc.) and the properties of a specific shape (a circle is a shape that has a radius, is drawn by a circle−drawing function, etc.). Expressing this distinction and taking advantage of it defines object−oriented programming. A language with constructs that allows this distinction to be expressed and used supports object−oriented programming. Other languages don't.

The Simula inheritance mechanism provides a solution. First, specify a class that defines the general properties of all shapes:

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

The functions for which the calling interface can be defined, but where the implementation cannot be defined except for a specific shape, have been marked "virtual" (the Simula and C++ term for "may be re−defined later in a class derived from this one"). Given this definition, we can write general functions manipulating shapes:

```
void rotate_all(shape* v, int size, int angle)
// rotate all members of vector "v"
// of size "size" "angle" degrees
{
      for (int i = 0; i < size; i++) v[i].rotate(angle);
}
```

To define a particular shape, we must say that it is a shape and specify its particular properties (including the virtual functions):

```
class circle : public shape {
      int radius;
public:
      void draw() { /* ... */ };
      void rotate(int) {}    // yes, the null function
```

```
};
```

In C++, class *circle* is said to be *derived* from class *shape*, and class *shape* is said to be a *base* of class *circle*. An alternative terminology calls *circle* and *shape* subclass and superclass, respectively.

The programming paradigm is:

Decide which classes you want;
provide a full set of operations for each class;
make commonality explicit by using inheritance.

Where there is no such commonality data abstraction suffices. The amount of commonality between types that can be exploited by using inheritance and virtual functions is the litmus test of the applicability of object–oriented programming to an application area. In some areas, such as interactive graphics, there is clearly enormous scope for object–oriented programming. For other areas, such as classical arithmetic types and computations based on them, there appears to be hardly any scope for more than data abstraction and the facilities needed for the support of object–oriented programming seem unnecessary.

However, more advanced mathematics may benefit from the use of inheritance: fields are specializations of rings; vector spaces a special case of modules.

Finding commonality among types in a system is not a trivial process. The amount of commonality to be exploited is affected by the way the system is designed. When designing a system, commonality must be actively sought, both by designing classes specifically as building blocks for other types, and by examining classes to see if they exhibit similarities that can be exploited in a common base class.

Having examined the minimum support needed for procedural programming, data hiding, data abstraction, and object–oriented programming we will go into some detail describing features that—while not essential—can make data abstraction and object–oriented more effective.

Support for Data Abstraction

The basic support for programming with data abstraction consists of facilities for defining a set of operations (functions and operators) for a type and for restricting the access to objects of the type to that set of operations. Once that is done, however, the programmer soon finds that language refinements are needed for convenient definition and use of the new types. Operator overloading is a good example of this.

Initialization and Cleanup

When the representation of a type is hidden some mechanism must be provided for a user to initialize variables of that type. A simple solution is to require a user to call some function to initialize a variable before using it. For example:

```
class vector {
      int  sz;
      int* v;
public:
      void init(int size);   // call init to initialize sz and v
                             // before the first use of a vector
      // ...
};
vector v;
// don't use v here
v.init(10);
// use v here
```

This is error prone and inelegant. A better solution is to allow the designer of a type to provide a distinguished function to do the initialization. Given such a function, allocation and initialization of a variable becomes a single operation (often called instantiation or construction) instead of two separate operations. Such an initialization function is often called a constructor. In cases where construction of objects of a type is non−trivial, one often needs a complementary operation to clean up objects after their last use. In C++, such a cleanup function is called a destructor. Consider a vector type:

```
class vector {
      int  sz;                        // number of elements
      int* v;                         // pointer to integers
public:
      vector(int);                    // constructor
      ~vector();                      // destructor
      int& operator[](int index);     // subscript operator
};
```

The *vector* constructor can be defined to allocate space like this:

```
vector::vector(int s)
{
      if (s<=0) error("bad vector size");
      sz = s;
      v = new int[s];   // allocate an array of "s" integers
}
```

The *vector* destructor frees the storage used:

```
vector::~vector()
{
      delete v;        // deallocate the memory pointed to by v
}
```

C++ does not support garbage collection. This is compensated for, however, by enabling a type to maintain its own storage management without requiring intervention by a user. This is a common use for the constructor/destructor mechanism, but many uses of this mechanism are unrelated to storage management.

Assignment and Initialization

Controlling construction and destruction of objects is sufficient for many types, but not for all. It can also be necessary to control all copy operations. Consider class *vector*:

```
vector v1(100);
vector v2 = v1;  // make a new vector v2 initialized to v1
v1 = v2;         // assign v2 to v1
```

It must be possible to define the meaning of the initialization of *Pv2* and the assignment to *v1*. Alternatively it should be possible to prohibit such copy operations; preferably both alternatives should be available. For example:

```
class vector {
      int* v;
      int  sz;
public:
      // ...
      void operator=(const vector&);  // assignment
      vector(const vector&);          // initialization
};
```

specifies that user−defined operations should be used to interpret *vector* assignment and initialization. Assignment might be defined like this:

```
vector::operator=(const vector& a) // check size and copy elements
{
      if (sz != a.sz) error("bad vector size for =");
      for (int i = 0; i<sz; i++) v[i] = a.v[i];
```

```
}
```

Since the assignment operation relies on the "old value" of the vector being assigned to, the initialization operation *must* be different. For example:

```
vector::vector(const vector& a)
                          // initialize a vector from another vector
{
      sz = a.sz;          // same size
      v = new int[sz];    // allocate element array
      for (int i = 0; i<sz; i++) v[i] = a.v[i];   // copy elements
}
```

In C++, a copy constructor, for example *X*(*const X&*) defines all initialization of objects of type *X* with another object of type *X*. In addition to explicit initialization copy constructors are used to handle arguments passed "by value" and function return values.

In C++ assignment of an object of class *X* can be prohibited by declaring assignment private:

```
class X {
      void operator=(const X&);   // only members of X can
      X(const X&);                // copy an X
      // ...
public:
      // ...
};
```

Ada does not support constructors, destructors, overloading of assignment, or user–defined control of argument passing and function return. This severely limits the class of types that can be defined and forces the programmer back to "data hiding techniques"; that is, the user must design and use type manager modules rather than proper types.

Parameterized Types

Why would you want to define a vector of integers anyway? A user typically needs a vector of elements of some type unknown to the writer of the *vector* type. Consequently the vector type ought to be expressed in such a way that it takes the element type as an argument:

```
template<class T> class vector {  // vector of elements of type T
      T* v;
      int sz;
public:
      vector(int s)
      {
          if (s <= 0) error("bad vector size");
          v = new T[sz = s];    // allocate an array of "s" "T"s
      }
      T& operator[](int i);
      int size() { return sz; }
      // ...
};
```

A *template* specifies a family of types generated by specifying the template argument(s).

Vectors of specific types can now be defined and used:

```
vector<int> v1(100);      // v1 is a vector of 100 integers
vector<complex> v2(200);  // v2 is a vector of 200 complex numbers
v2[i] = complex(v1[x],v1[y]);
```

Ada, Clu, ML, and C++ support parameterized types.

The ANSI C++ committee, X3J16, only accepted templates into C++ in July 1990, so few C++ implementations support templates at the time of writing.

There need not be any run−time overheads compared with a class where all types involved are specified directly.

A problem with parameterized types is that each instantiation creates an independent type. For example, the type *vector<char>* is unrelated to the type *vector<complex>*. Ideally one would like to be able to express and utilize the commonality of types generated from the same parameterized type. For example, both *vector<char>* and *vector<complex>* have a *size*() function that is independent of the parameter type. It is possible, but not trivial, to deduce this from the definition of class *vector* and then allow *size*() to be applied to any *vector*. An interpreted or dynamically compiled language (such as Smalltalk) or a language supporting both parameterized types and inheritance (such as C++) has an advantage here.

Exception Handling

As programs grow, and especially when libraries are used extensively, standards for handling errors (or more generally: "exceptional circumstances") become important. Ada, Algol68, Clu, and C++ each support a standard way of handling exceptions.

The ANSI C++ committee, X3J16, only accepted exception handling into C++ in November 1990, so C++ implementations that support exception handling are rare at the time of writing.

Consider again the *vector* example:

```
class vector {
      // ...
      class range { }; // type to be used for exceptions
};
int& vector::operator[](int i)
{
      if (i<0 || sz<=i) throw range();
      return v[i];
}
```

Instead of calling an error function, *vector::operator*[]() can invoke the exception handling code, "throw the range exception." This will cause the call stack to be unraveled until an exception handler for *vector::range* is found; this handler will than be executed.

An exception handler may be defined for a specific block:

```
void f(int i) {
      try {   // exceptions in this try block are handled by
              // the exception handler defined below
          vector v(i);
          // ...
          v[i] = 7;     //  causes vector::range exception
          // ...
          int i = g();  // might cause a vector::range exception
      }
      catch (vector::range) {
          error("f(): vector range error");
          return;
      }
}
```

There are many ways of defining exceptions and the behavior of exception handlers. The facility sketched here resembles the ones found in Clu and ML.

A poor implementation of exception handling can be a serious drain on run−time efficiency and the portability of language implementations. The C++ exception handling can be implemented so that code is not executed unless an exception is thrown or portable across C implementations by (implicitly) using the C standard library functions *setjmp*() and *longjmp*().

Type conversions

User−defined type conversions, such as the one from floating−point numbers to complex numbers implied by the constructor *complex*(*double*), have proven unexpectedly useful in C++. Such conversions can be applied explicitly or the programmer can rely on the compiler to add them implicitly where necessary and unambiguous:

```
complex a = complex(1);
complex b = 1;            // implicit: 1 -> complex(1)
a = b+complex(2);
a = b+2;                  // implicit: 2 -> complex(2)
```

User−defined type conversions were introduced into C++ because mixed mode arithmetic is the norm in languages for numerical work and because most user−defined types used for "calculation" (for example, matrices, character strings, and machine addresses) have natural mappings to and/or from other types.

One use of coercions has proven especially useful from a program organization point of view:

```
complex a = 2;
complex b = a+2;  // interpreted as operator+(a,complex(2))
b = 2+a;          // interpreted as operator+(complex(2),a)
```

Only one function is needed to interpret "+" operations and the two operands are handled identically by the type system. Furthermore, class *complex* is written without any need to modify the concept of integers to enable the smooth and natural integration of the two concepts. This is in contrast to a "pure object−oriented system" where the operations would be interpreted like this:

```
a+2;    // a.operator+(2)
2+a;    // 2.operator+(a)
```

making it necessary to modify class *integer* to make 2*+a* legal. Modifying existing code should be avoided as far as possible when adding new facilities to a system. Typically, object−oriented programming offers superior facilities for adding to a system without modifying existing code. In this case, however, data abstraction facilities provide a better solution.

Iterators

It has been claimed that a language supporting data abstraction must provide a way of defining control structures. In particular, a mechanism that allows a user to define a loop over the elements of some type containing elements is often needed. This must be achieved without forcing a user to depend on details of the implementation of the user−defined type. Given a sufficiently powerful mechanism for defining new types and the ability to overload operators, this can be handled without a separate mechanism for defining control structures.

For a vector, defining an iterator is not necessary since an ordering is available to a user through the indices. I'll define one anyway to demonstrate the technique. There are several possible styles of iterators. My favorite relies on overloading the function application operator ():

This style also relies on the existence of a distinguished value to represent "end of iteration." Often, in particular for C++ pointer types, 0 can be used.

```
class vector_iterator {
      vector& v;
      int i;
public:
      vector_iterator(vector& r) { i = 0; v = r; }
      int operator()() { return i<v.size() ? v.elem(i++) : 0; }
};
```

A *vector_iterator* can now be declared and used for a *vector* like this:

```
void f(vector& v)
{
      vector_iterator next(v);
```

```
      int i;
      while (i=next()) print(i);  // maybe too ‘cute’
}
```

More than one iterator can be active for a single object at one time, and a type may have several different iterator types defined for it so that different kinds of iteration may be performed. An iterator is a rather simple control structure. More general mechanisms can also be defined. For example, the C++ standard library provides a co−routine *clas.s*

For many "container" types, such as *vector*, one can avoid introducing a separate iterator type by defining an iteration mechanism as part of the type itself. A *vector* might be defined to have a "current element":

```
class vector {
      int* v;
      int  sz;
      int current;
public:
      // ...
      int next() { return (current < sz) ? v[current++] : 0; }
      int prev() { return (0 <= --current) ? v[current] : 0; }
};
```

Then the iteration can be performed like this:

```
vector v(sz);
int i;
while (i=v.next()) print(i);
```

This solution is not as general as the iterator solution, but avoids overhead in the important special case where only one kind of iteration is needed and where only one iteration at a time is needed for a vector. If necessary, a more general solution can be applied in addition to this simple one. Note that the "simple" solution requires more foresight from the designer of the container class than the iterator solution does. The iterator−type technique can also be used to define iterators that can be bound to several different container types thus providing a mechanism for iterating over different container types with a single iterator type.

Multiple Implementations

The basic mechanism for supporting object−oriented programming, derived classes, and virtual functions can be used to support data abstraction by allowing several different implementations for a given type. Consider again the stack example:

```
template<class T>
      class stack {
      public:
          virtual void push(T) = 0; // pure virtual function
          virtual T pop() = 0;      // pure virtual function
      };
```

The *=0* notation specifies that no definition is required for the virtual function and that the class is abstract, that is, the class can only be used as a base class. This allows stacks to be used, but not created:

```
stack<cat> s; // error: stack is abstract
void some_function(stack<cat>& s, cat kitty) // ok
{
      s.push(kitty);
      cat c2 = s.pop();
      // ...
}
```

Since no representation is specified in the stack interface, its users are totally insulated from implementation details.

We can now provide several distinct implementations of stacks. For example, we can provide a stack implemented with an array:

```
template<class T>
      class astack : public stack<T> {
          // actual representation of a stack object
          // in this case an array
          // ...
      public:
          astack(int size);
          ~astack();
          void push(T);
          T pop();
      };
```

and elsewhere a stack implemented using a linked list:

```
template<class T>
      class lstack : public stack<T> {
          // ...
      };
```

We can now create and use stacks:

```
void g()
{
      lstack<cat> s1(100);
      astack<cat> s2(100);
      cat ginger;
      cat snowball;
      some_function(s1,ginger);
      some_function(s2,snowball);
}
```

Only the creator of *stack*s, *g*(), needs to worry about different kinds of *stack*s, the user *some_function*() is totally insulated from such details. The price of this flexibility is that each operation on such a type must be a virtual function.

Implementation Issues

The support needed for data abstraction is primarily provided in the form of language features implemented by a compiler. However, parameterized types are best implemented with support from a linker with some knowledge of the language semantics, and exception handling requires support from the run−time environment. Both can be implemented to meet the strictest criteria for both compile−time speed and efficiency without compromising generality or programmer convenience.

As the power to define types increases, programs to a larger degree depend on types from libraries (and not just those described in the language manual). This naturally puts greater demands on facilities to express what is inserted into or retrieved from a library, facilities for finding out what a library contains, facilities for determining what parts of a library are actually used by a program, etc.

For a compiled language, facilities for calculating the minimal compilation necessary after a change become important. It is essential that the linker/loader with suitable help from the compiler is capable of bringing a program into memory for execution without also bringing in large amounts of related, but unused, code. In particular, a library/linker/loader system that brings the code for every operation on a type into core just because the programmer used one or two operations on the type is worse than useless.

Support for Object−Oriented programming

The basic support a programmer needs to write object−oriented programs consists of a class mechanism with inheritance and a mechanism that allows calls of member functions to depend on the actual type of an object (in cases where the actual type is unknown at compile−time). The design of the member function calling mechanism is critical. In addition, facilities supporting data abstraction techniques (as described above) are important because the arguments for data abstraction and for its refinements to support elegant use of types are equally valid where support for object−oriented programming is available. The success of both techniques hinges on the design of types and on the ease, flexibility, and efficiency of such types. Object−oriented programming allows user−defined types to be far more flexible and general than the ones designed using only data abstraction techniques.

Calling Mechanisms

The key language facility supporting object−oriented programming is the mechanism by which a member function is invoked for a given object. For example, given a pointer *p*, how is a call *p−>f*(*arg*) handled? There is a range of choices.

In languages such as C++ and Simula, where static type checking is extensively used, the type system can be employed to select between different calling mechanisms. In C++, two alternatives are available:

1. A normal function call: the member function to be called is determined at compile−time (through a lookup in the compiler's symbol tables) and called using the standard function call mechanism with an argument added to identify the object for which the function is called. Where the "standard function call" is not considered efficient enough, the programmer can declare a function *inline* and the compiler will attempt to expand its body inline. In this way, one can achieve the efficiency of a macro expansion without compromising the standard function semantics. This optimization is equally valuable as a support for data abstraction.
2. A virtual function call: The function to be called depends on the type of the object for which it is called. This type cannot, in general, be determined until run−time. Typically, the pointer *p* will be of some base class *B* and the object will be an object of some derived class *D* (as was the case with the base class *shape* and the derived class *circle* above). The call mechanism must look into the object and find some information placed there by the compiler to determine which function *f* is to be called. Once that function is found, say *D::f*, it can be called using the mechanism described above. The name *f* is converted at compile−time into an index into a table of pointers to functions. This virtual call mechanism can be made essentially as efficient as the "normal function call" mechanism. In the standard C++ implementation, only five additional memory references are used.

In languages with weak static type checking a more elaborate mechanism must be employed. What is done in a language like Smalltalk is to store a list of the names of all member functions (methods) of a class so that they can be found at run−time:

1. A method invocation: First the appropriate table of method names is found by examining the object pointed to by *p*. In this table (or set of tables) the string *"f"* is looked up to see if the object has an *f*(). If an *f*() is found it is called; otherwise some error handling takes place. This lookup differs from the lookup done at compile−time in a statically checked language in that the method invocation uses a method table for the actual object.

A method invocation is inefficient compared with a virtual function call, but more flexible. Since static type checking of arguments typically cannot be done for a method invocation, the use of methods must be supported by dynamic type checking.

Type Checking

The shape example showed the power of virtual functions. What, in addition to this, does a method invocation mechanism do for you? You can attempt to invoke *any* method for *any* object.

The ability to invoke any method for any object enables the designer of general−purpose libraries to push the responsibility for handling types onto the user. Naturally this simplifies the design of libraries. However, it then becomes the responsibility of the user to avoid type mismatches like this:

```
// assume dynamic type checking.
// *** NOT C++ ***
```

```
Stack s;  // Stack can hold pointers to objects of any type
cs.push(new Saab900);
cs.push(new Saab37B);
cs.pop()->takeoff();   // fine: a Saab37B is a plane
cs.pop()->takeoff();   // Oops! run-time error: a Saab 900
                         // is a car
                         // a car does not have a takeoff
                         // method.
```

An attempt to use a *car* as a *plane* will be detected by the message handler and an appropriate error handler will be called. However, that is only a consolation when the user is also the programmer. The absence of static type checking makes it difficult to guarantee that errors of this class are not present in systems delivered to end-users.

Combinations of parameterized classes and the use of virtual functions can approach the flexibility, ease of design, and ease of use of libraries designed with method lookup without relaxing the static type checking or incurring significant run-time overheads (in time or space). For example:

```
stack<plane*> cs;
cs.push(new Saab900);  // compile-time error:
                         // type mismatch: car* passed, plane*
                         // expected
cs.push(new Saab37B);
cs.pop()->takeoff();   // fine: a Saab 37B is a plane
cs.pop()->takeoff();
```

The use of static type checking and virtual function calls leads to a somewhat different style of programming than does dynamic type checking and method invocation. For example, a Simula or C++ class specifies a fixed interface to a set of objects (of any derived class) whereas a Smalltalk class specifies an initial set of operations for objects (of any subclass). In other words, a Smalltalk class is a minimal specification and the user is free to try operations not specified, whereas a C++ class is an exact specification and the user is guaranteed that only operations specified in the class declaration will be accepted by the compiler.

Inheritance

Consider a language having some form of method lookup without having an inheritance mechanism. Could that language be said to support object-oriented programming? I think not. Clearly, you could do interesting things with the method table to adapt the objects' behavior to suit conditions. However, to avoid chaos, there must be some systematic way of associating methods and the data structures they assume for their object representation. To enable a user of an object to know what kind of behavior to expect, there would also have to be some standard way of expressing what is common to the different behaviors the object might adopt. This "systematic and standard way" would be an inheritance mechanism.

Consider a language having an inheritance mechanism without virtual functions or methods. Could that language be said to support object-oriented programming? I think not: the shape example does not have a good solution in such a language. However, such a language would be noticeably more powerful than a "plain" data abstraction language. This contention is supported by the observation that many Simula and C++ programs are structured using class hierarchies without virtual functions. The ability to express commonality (factoring) is an extremely powerful tool. For example, the problems associated with the need to have a common representation of all shapes could be solved. No union would be needed. However, in the absence of virtual functions, the programmer would have to resort to the use of "type fields" to determine actual types of objects, so the problems with the lack of modularity of the code would remain.

This is the problem with Simula's *inspect* statement and the reason it does not have a counterpart in C++.

This implies that class derivation (subclassing) is an important programming tool in its own right. It can be used to support object-oriented programming, but it has wider uses. This is particularly true if one identifies the use of inheritance in object-oriented programming with the idea that a base class expresses a general concept of which all derived classes are specializations. This idea captures only part of the expressive power of inheritance, but it is strongly encouraged by languages where every member function is virtual (or a method). Given suitable controls of what is inherited (see Stroustrup, B., *The C++ Programming Language*, Addison−Wesley, 1986. 2nd Edition 1991.), class derivation can be a powerful tool for creating new types. Given a class, derivation can be used to add and/or subtract features. The relation of the resulting class to its base cannot always be completely described in terms of specialization; factoring may be a better term.

Derivation is another tool in the hands of a programmer and there is no foolproof way of predicting how it is going to be used and it is too early (even after almost 25 years of Simula) to tell which uses are simply misuses.

Multiple Inheritance

When a class *A* is a base of class *B*, a *B* inherits the attributes of an *A*; that is, a *B* is an *A* in addition to whatever else it might be. Given this explanation it seems obvious that it might be useful to have a class *B* inherit from two base classes *A1* and *A2*. This is called multiple inheritance.

A fairly standard example of the use of multiple inheritance would be to provide two library classes *displayed* and *task* for representing objects under the control of a display manager and co−routines under the control of a scheduler, respectively. A programmer could then create classes such as

```
class my_displayed_task : public displayed, public task {
      // my stuff
};
class my_task : public task {  // not displayed my stuff
};
class my_displayed : public displayed {  // not a task my stuff
};
```

Using (only) single inheritance only two of these three choices would be open to the programmer. This leads to either code replication or loss of flexibility—and typically both. In C++ this example can be handled as shown above with to no significant overheads (in time or space) compared to single inheritance and without sacrificing static type checking. Ambiguities are handled at compile−time:

```
class A { public: void f(); /* ... */ };
class B { public: void f(); /* ... */ };
class C : public A, public B { /* ... */ };
void g(C* p)
{
      p->f();    // error: ambiguous
}
```

C++ differs from the object−oriented Lisp dialects that support multiple inheritance. In Lisp dialects ambiguities are resolved by considering the order of declarations significant, by considering objects of the same name in different base classes identical, or by combining methods of the same name in base classes into a more complex method of the highest class.

In C++, one would typically resolve the ambiguity by adding a function:

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

In addition to this straightforward concept of independent multiple inheritance there appears to be a need for a more general mechanism for expressing dependencies between classes in a multiple inheritance lattice. In C++, the requirement that a sub−object should be shared by all other sub−objects in a class object is expressed through the mechanism of a virtual base class:

```
class W { /* ... */ }; // window
class Bwindow: public virtual W { // window with border ...
};
class Mwindow : public virtual W { // window with menu ...
};
class BMW : public Bwindow, public Mwindow {
      // window with border and menu ...
};
```

Here the (single) *window* sub−object is shared by the *Bwindow* and *Bwindow* sub−objects of a *BMW*. The Lisp dialects provide concepts of method combination to ease programming using such complicated class hierarchies. C++ does not.

Encapsulation

Consider a class member (either a data member or a function member) that needs to be protected from "unauthorized access." What choices can be reasonable for delimiting the set of functions that may access that member? The "obvious" answer for a language supporting object−oriented programming is "all operations defined for this object," that is, all member functions. A non−obvious implication of this answer is that there cannot be a complete and final list of all functions that may access the protected member since one can always add another by deriving a new class from the protected member's class and define a member function of that derived class. This approach combines a large degree of protection from accident (since you do not easily define a new derived class "by accident") with the flexibility needed for "tool building" using class hierarchies (since you can "grant yourself access" to protected members by deriving a class). For example:

```
class Window {
      // ...
protected:
      Rectangle inside;
      // ...
public:
      // ...
};
class Dumb_terminal : Window {
      // ...
public:
      void prompt();
      // ...
};
```

Here *Window* specifies *inside* as protected so that derived classes such as *Dumb_terminal* can read it and figure out what part of the *Window*'s area it may manipulate.

Unfortunately, the "obvious" answer for a language oriented towards data abstraction is different: "list the functions that need access in the class declaration." There is nothing special about these functions. In particular, they need not be member functions. A non−member function with access to private class members is called a *friend* in C++. Class *complex* above was defined using *friend* functions. It is sometimes important that a function may be specified as a *friend* in more than one class. Having the full list of members and friends available is a great advantage when you are trying to understand the behavior of a type and especially when you want to modify it.

Encapsulation issues increase dramatically in importance with the size of the program and with the number and geographical dispersion of its users. (See *The C++ Programming Language for more detailed discussions of language support for encapsulation.)*

Implementation Issues

The support needed for object−oriented programming is primarily provided by the run−time system and by the programming environment. Part of the reason is that object−oriented programming builds on the language improvements already pushed to their limit to support data abstraction so that relatively few additions are needed.

This assumes that an object−oriented language does indeed support data abstraction. However, the support for data abstraction is often deficient in such languages. Conversely, languages that support data abstraction are typically deficient in their support of object−oriented programming.

The use of object−oriented programming blurs the distinction between a programming language and its environment further. Since more powerful special− and general−purpose user−defined types can be defined, their use pervades user programs. This requires further development of both the run−time system, library facilities, debuggers, performance measuring, monitoring tools, etc. Ideally these are integrated into a unified programming environment. Smalltalk is the best example of this.

Limits to Perfection

A major problem with a language defined to exploit the techniques of data hiding, data abstraction, and object−oriented programming is that to claim to be a general−purpose programming language it must

1. Run on traditional machines.
2. Coexist with traditional operating systems.
3. Compete with traditional programming languages in terms of run−time efficiency.
4. Cope with every major application area.

This implies that facilities must be available for effective numerical work (floating−point arithmetic without overheads that would make Fortran appear attractive), and that facilities must be available for access to memory in a way that allows device drivers to be written. It must also be possible to write calls that conform to the often rather strange standards required for traditional operating system interfaces. In addition, it should be possible to call functions written in other languages from an object−oriented programming language and for functions written in the object−oriented programming language to be called from a program written in another language.

Another implication is that an object−oriented programming language cannot completely rely on mechanisms that cannot be efficiently implemented on a traditional architecture and still expect to be used as a general−purpose language. A very general implementation of method invocation can be a liability unless there are alternative ways of requesting a service.

Similarly, garbage collection can become a performance and portability bottleneck. Most object−oriented programming languages employ garbage collection to simplify the task of the programmer and to reduce the complexity of the language and its compiler. However, it ought to be possible to use garbage collection in non−critical areas while retaining control of storage use in areas where it matters. As an alternative, it is feasible to have a language without garbage collection and then provide sufficient expressive power to enable the design of types that maintain their own storage. C++ is an example of this.

Exception handling and concurrency features are other potential problem areas. Any feature that is best implemented with help from a linker can become a portability problem.

The alternative to having "low−level" features in a language is to handle major application areas using separate "low−level" languages.

Conclusions

Object−oriented programming is programming using inheritance. Data abstraction is programming using user−defined types. With few exceptions, object−oriented programming can and ought to be a superset of data abstraction. These techniques need proper support to be effective. Data abstraction primarily needs support in the form of language features and object−oriented programming needs further support from a programming environment. To be general−purpose, a language supporting data abstraction or object−oriented programming must enable effective use of traditional hardware.

Chapter 5

Multiple Inheritance for C++

This chapter describes an implementation of a multiple inheritance mechanism for C++ (described in *The C++ Programming Language*). It provides only the most rudimentary explanation of what multiple inheritance is in general and what it can be used for. The particular variation of the general concept implemented here is primarily explained in terms of this implementation.

For more information on multiple inheritance see **Chapter 1, "The Evolution of C++"** and *"What is Object−Oriented Programming?"*

This chapter contains the following sections:
- **"Introduction"**
- **"Multiple Inheritance"**
- **"C++ Implementation Strategy"**
- **"Multiple Base Classes"**
- **"Virtual Functions"**
- **"Multiple Inclusions"**
- **"Virtual Base Classes"**
- **"Constructors and Destructors"**
- **"Visibility"**
- **"Overheads"**
- **"But is it Simple to Use?"**
- **"Conclusions"**

Note: This chapter is from a paper by Bjarne Stroustrup.

Introduction

Multiple Inheritance is the ability of a class to have more than one base class (super class). In a language where multiple inheritance is supported a program can be structured as a set of inheritance lattices instead of (just) as a set of inheritance trees. This is widely believed to be an important structuring tool. It is also widely believed that multiple inheritance complicates a programming language significantly, is hard to implement, and is expensive to run. I will demonstrate that none of these last three conjectures are true.

First a bit of background on multiple inheritance and C++ implementation technique is presented, then the multiple inheritance scheme implemented for C++ is introduced in two stages:
- the basic scheme for multiple inheritance, the basic strategy for ambiguity resolution, and the way to implement virtual functions
- handling of classes included more than once in an inheritance lattice; the programmer has the choice whether a multiply included base class will result in one or more sub−objects being created

Finally, some the complexities and overheads introduced by this multiple inheritance scheme are summarized.

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 processors.

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 C++ or Simula67, we have only two ways of solving this problem: deriving *Task* from *Displayed* or deriving *Displayed* from *Task.* Neither is ideal since they both create a dependency between the library versions of two fundamental and independent concepts. Ideally one would want to be able to choose between saying 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 using a 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 and the representation of various kinds of processors and compilers for a multi−machine, multi−environment debugger.

In general, multiple inheritance allows a user to combine independent (and not so independent) concepts represented as classes into a composite concept represented as a derived class. A common way of using multiple inheritance is for a designer to provide sets of base classes with the intention that a user creates new classes by choosing base classes from each of the relevant sets. Thus a programmer creates new concepts using a recipe like "pick an A and/or a B." In the window example, a user might specify a new kind of window by selecting a style of window interaction (from the set of interaction base classes) and a style of appearance (from the set of base classes defining display options). In the debugger example, a programmer would specify a debugger by choosing a processor and a compiler.

Given multiple inheritance and N concepts each of which might somehow be combined with one of M other concepts, we need N+M classes to represent all the combined concepts. Given only single inheritance, we need to replicate information and provide N+M+N*M classes. Single inheritance handles cases where N==1 or M==1. The usefulness of multiple inheritance for avoiding replication hinges on the importance of examples where the values of N and M are both larger than 1. It appears that examples with N>=2 and M>=2 are not uncommon; the window and debugger examples described above will typically have both N and M larger than 2.

C++ Implementation Strategy

Before discussing multiple inheritance and its implementation in C++ I will first describe the main points in the traditional implementation of the C++ single inheritance class concept.

An object of a C++ class is represented by a contiguous region of memory. A pointer to an object of a class points to the first byte of that region of memory. The compiler turns a call of a member function into an "ordinary" function call with an "extra" argument; that "extra" argument is a pointer to the object for which the member function is called.

Consider a simple class *A:*

In most of this paper data hiding issues are ignored to simplify the discussion and shorten the examples. This makes some examples illegal. Changing the word *class* to *struct* would make the examples legal, as would adding *public* specifiers in the appropriate places.

```
class A {
      int a;
      void f(int i);
};
```

An object of class A will look like Figure 5−1:

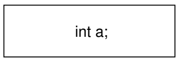

Figure 5–1 Class A Object

No information is placed in an *A* except the integer *a* specified by the user. No information relating to (non–virtual) member functions is placed in the object.

A call of the member function *"A::f" :*

```
A* pa;
pa–>f(2);
```

is transformed by the compiler into an "ordinary function call":

```
f__F1A(pa,2);
```

Objects of derived classes are composed by concatenating the members of the classes involved:

```
class A { int a; void f(int); };
class B : A { int b; void g(int); };
class C : B { int c; void h(int); };
```

Again, no "housekeeping" information is added, so an object of class *C* looks like Figure 5–2:

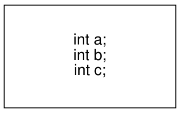

Figure 5–2 Class C Object - 1

The compiler "knows" the position of all members in an object of a derived class exactly as it does for an object of a simple class and generates the same (optimal) code in both cases.

Implementing virtual functions involves a table of functions. Consider:

```
class A {
      int a;
      virtual void f(int);
      virtual void g(int);
      virtual void h(int);
};
class B : A { int b; void g(int); };
```

```
class C : B { int c; void h(int); };
```

In this case, a table of virtual functions, the *vtbl,* contains the appropriate functions for a given class and a pointer to it is placed in every object. A class *C* object looks like Figure 5–3:

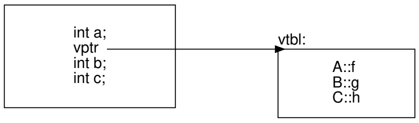

Figure 5–3 Class C Object - 2

A call to a virtual function is transformed into an indirect call by the compiler. For example,

```
C* pc;
pc–>g(2);
```

becomes something like:

```
(*(pc–>vptr[1]))(pc,2);
```

A multiple inheritance mechanism for C++ must preserve the efficiency and the key features of this implementation scheme.

Multiple Base Classes

Given two classes

```
class A { ... };
class B { ... };
```

one can design a third using both as base classes:

```
class C : A , B { ... };
```

This means that a *C* is an *A* and a *B*. One might equivalently define *C* like this:

```
class C : B , A { ... };
```

This definition is equivalent except for possible side effects in constructors and destructors (access to global variables, input operations, output operations, etc.).

Object Layout

An object of class *C* can be laid out as a contiguous object like Figure 5-4:

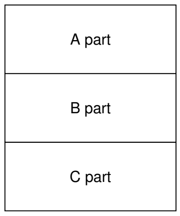

Figure 5–4 Contiguous Class C Object

Accessing a member of classes *A*, *B*, or *C* is handled exactly as before: the compiler knows the location in the object of each member and generates the appropriate code (without spurious indirections or other overhead).

Member Function Call

Calling a member function of *A* or *C* is identical to what was done in the single inheritance case. Calling a member function of *B* given a *C** is slightly more involved:

```
C* pc;
pc->bf(2);  // assume that bf is a member of B
              // and that C has no member named bf
              // except the one inherited from B
```

Naturally, *B::bf*() expects a *B** (to become its *this* pointer). To provide it, a constant must be added to pc.

This constant, *delta*(*B*), is the relative position of the *B* part of *C.* This delta is known to the compiler that transforms the call into:

```
bf__F1B((B*)((char*)pc+delta(B)),2);
```

The overhead is one addition of a constant per call of this kind. During the execution of a member function of *B* the function's *this* pointer points to the *B* part of *C* as shown in Figure 5–5:

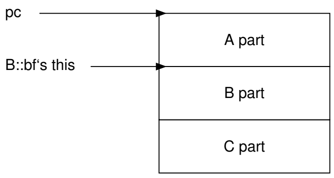

Figure 5–5 Pointer to the *B* Part of *C*

Note that there is no space penalty involved in using a second base class and that the minimal time penalty is incurred only once per call.

Ambiguities

Consider potential ambiguities if both *A* and *B* have a public member *ii*:

```
class A { int ii; };
```

```
class B { char* ii; };
class C : A, B { };
```

In this case *C* will have two members called *ii*, *A::ii* and *B::ii*. Then

```
C* pc;
pc->ii;         // error: A::ii or B::ii ?
```

is illegal since it is ambiguous. Such ambiguities can be resolved by explicit qualification:

```
pc->A::ii;    // C’s A’s ii
pc->B::ii;    // C’s B’s ii
```

A similar ambiguity arises if both *A* and *B* have a function *f*():

```
class A { void f(); };
class B { int f(); };
class C : A, B { };
C* pc;
pc->f();    // error: A::f or B::f ?
pc->A::f();    // C’s A’s f
pc->B::f();    // C’s B’s f
```

As an alternative to specifying which base class in each call of an *f*(), one might define an *f*() for *C*. *C::f*() might call the base class functions. For example:

```
class C : A, B {
      int f() { A::f(); return B::f(); }
};
C* pc;
pc->f();        // C::f is called
```

Casting

Explicit and implicit casting may also involve modifying a pointer value with a delta:

```
C* pc;
B* pb;
pb = (B*)pc;    // pb = (B*)((char*)pc+delta(B))
pb = pc;        // pb = (B*)((char*)pc+delta(B))
pc = pb;        // error: cast needed
pc = (C*)pb;    // pc = (C*)((char*)pb-delta(B))
```

Casting yields the pointer referring to the appropriate part of the same object as shown in Figure 5–6:

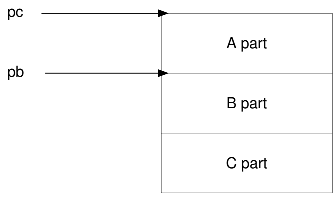

Figure 5–6 Modified Pointer Value

Comparisons are interpreted in the same way:

```
pc == pb;    // that is, pc == (C*)pb
               // or equivalently (B*)pc == pb
               // that is, (B*)((char*)pc+delta(B)) == pb
               // or equivalently pc == (C*)((char*)pb-delta(B))
```

Note that in both C and C++ casting has always been an operator that produced one value given another value rather than an operator that simply reinterpreted a bit pattern. For example, on almost all machines (*int*).2 causes code to be executed; (*float*)(*int*).2 is not equal to .2. Introducing multiple inheritance as described here will introduce cases where (char*)(B*)v!=(char*)v for some pointer type B*. Note, however, that when *B* is a base class of *C*, (B*)v==(C*)v==v.

Zero Valued Pointers

Pointers with the value zero cause a separate problem in the context of multiple base classes. Consider applying the rules presented above to a zero−valued pointer:

```
C* pc = 0;
B* pb = 0;
if (pb == 0) ...
pb = pc;    // pb = (B*)((char*)pc+delta(B))
if (pb == 0) ...
```

The second test would fail since *pb* would have the value (*B**)((*char**)*0+delta*(*B*)).

The solution is to elaborate the conversion (casting) operation to test for the pointer−value 0:

```
C* pc = 0;
B* pb = 0;
if (pb == 0) ...
pb = pc;        // pb = (pc==0)?0:(B*)((char*)pc+delta(B))
if (pb == 0) ...
```

The added complexity and run−time overhead are a test and an increment.

Virtual Functions

Naturally, member functions may be virtual:

```
class A { virtual void f(); };
class B { virtual void f(); virtual void g(); };
class C : A , B { void f(); };
A* pa = new C;
B* pb = new C;
C* pc = new C;
pa->f();
pb->f();
pc->f();
```

All these calls will invoke *C::f*(). This follows directly from the definition of *virtual* since class *C* is derived from class *A* and from class *B*.

Implementation

On entry to *C::f,* the *this* pointer must point to the beginning of the *C* object (and not to the *B* part). However, it is not in general known at compile−time that the *B* pointed to by *pb* is part of a *C* so the compiler cannot subtract the constant *delta(B).* Consequently *delta*(*B*) must be stored so that it can be found at run−time. Since it is only used when calling a virtual function the obvious place to store it is in the table of virtual functions (*vtbl*). For reasons that will be explained below the delta is stored with each function in the *vtbl* so that a *vtbl* entry will be of the form:

```
struct vtbl_entry {
      void    (*fct)();
      int    delta;
};
```

An object of class *C* will look like Figure 5−7:

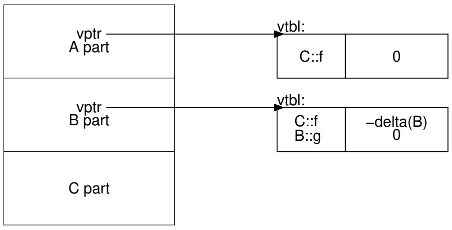

Figure 5−7 Class C Object

```
pb->f(); // call of C::f:
           // register vtbl_entry* vt = &pb->vtbl[index(f)];
           // (*vt->fct)((B*)((char*)pb+vt->delta))
```

Note that the object pointer may have to be adjusted to point to the correct sub−object before looking for the member pointing to the *vtbl.* Note also that each combination of base class and derived class has its own *vtbl.* For example, the *vtbl* for *B* in *C* is different from the *vtbl* of a separately allocated *B.* This implies that in general an object of a derived class needs a *vtbl* for each base class plus one for the derived class. However, as with single inheritance, a derived class can share a *vtbl* with its first base so that in the example above only two *vtbl s* are used for an object of type *C* (one for *A* in *C* combined with *C 's* own plus one for *B* in *C*).

Using an *int* as the type of a stored delta limits the size of a single object; that might not be a bad thing.

Ambiguities

The following demonstrates a problem:

```
class A { virtual void f(); };
class B { virtual void f(); };
class C : A , B { void f(); };
C* pc = new C;
pc->f();
pc->A::f();
pc->B::f();
```

Explicit qualification "suppresses" *virtual* so the last two calls really invoke the base class functions. Is this a problem? Usually, no. Either *C* has an *f*() and there is no need to use explicit qualification or *C* has no *f*() and the explicit qualification is necessary and correct. Trouble can occur when a function *f*() is added to *C* in a program that already contains explicitly qualified names. In the latter case one could wonder why someone would want to both declare a function virtual and also call it using explicit qualification. If *f*() is virtual, adding an *f*() to the derived class is clearly the correct way of resolving the ambiguity.

The case where no *C::f* is declared cannot be handled by resolving ambiguities at the point of call. Consider:

```
class A { virtual void f(); };
class B { virtual void f(); };
class C : A , B {  };    // error: C::f needed
C* pc = new C;
pc->f();    // ambiguous
A* pa = pc;    // implicit conversion of C* to A*
pa->f();    // not ambiguous: calls A::f();
```

The potential ambiguity in a call of *f*() is detected at the point where the virtual function tables for *A* and *B* in *C* are constructed. In other words, the declaration of *C* above is illegal because it would allow calls, such as *pa->f*(), which are unambiguous *only* because type information has been "lost" through an implicit coercion; a call of *f*() for an object of type *C* is ambiguous.

Multiple Inclusions

A class can have any number of base classes. For example,

```
class A : B1, B2, B3, B4, B5, B6 { ... };
```

It illegal to specify the same class twice in a list of base classes. For example,

```
class A : B, B { ... };    // error
```

The reason for this restriction is that every access to a *B* member would be ambiguous and therefore illegal; this restriction also simplifies the compiler.

Multiple Sub−objects

A class may be included more than once as a base class. For example:

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

In such cases multiple objects of the base class are part of an object of the derived class. For example, an object of class *C* has two *L 's:* one for *A* and one for *B* as shown in Figure 5−8:

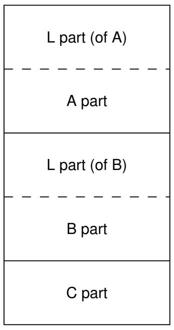

Figure 5–8 Class C Object with Two L’s

This can even be useful. Think of *L* as a link class for a Simula–style linked list. In this case a *C* can be on both the list of *A s* and the list of *B s.*

Naming

Assume that class *L* in the example above has a member *m.* How could a function *C::f* refer to *L::m?* The obvious answer is "by explicit qualification":

```
void C::f() { A::m = B::m; }
```

This will work nicely provided neither *A* nor *B* has a member *m* (except the one they inherited from *L*). If necessary, the qualification syntax of C++ could be extended to allow the more explicit:

```
void C::f() { A::L::m = B::L::m; }
```

Casting

Consider the example above again. The fact that there are two copies of *L* makes casting (both explicit and implicit) between *L** and *C** ambiguous, and consequently illegal:

```
C* pc = new C;
L* pl = pc;        // error: ambiguous
pl = (L*)pc;       // error: still ambiguous
pl = (L*)(A*)pc;   // The L in C's A
pc = pl;           // error: ambiguous
pc = (L*)pl;       // error: still ambiguous
pc = (C*)(A*)pl;   // The C containing A's L
```

I don't expect this to be a problem. The place where this will surface is in cases where *A s* (or *B s*) are handled by functions expecting an *L*; in these cases a *C* will not be acceptable despite a *C* being an *A*:

```
extern f(L*);     // some standard function
A aa;
C cc;
f(&aa);        // fine
f(&cc);        // error: ambiguous
f((A*)&cc);    // fine
```

Casting is used for explicit disambiguation.

Virtual Base Classes

When a class *C* has two base classes *A* and *B* these two base classes give rise to separate sub−objects that do not relate to each other in ways different from any other *A* and *B* objects. I call this *independent multiple inheritance.* However, many proposed uses of multiple inheritance assume a dependence among base classes (for example, the style of providing a selection of features for a window described in this chapter under "Multiple Inheritance"). Such dependencies can be expressed in terms of an object shared between the various derived classes. In other words, there must be a way of specifying that a base class must give rise to only one object in the final derived class even if it is mentioned as a base class several times. To distinguish this usage from independent multiple inheritance such base classes are specified to be virtual:

```
class AW : virtual W { ... };
class BW : virtual W { ... };
class CW : AW , BW { ... };
```

A single object of class *W* is to be shared between *AW* and *BW*; that is, only one *W* object must be included in *CW* as the result of deriving *CW* from *AW* and *BW.* Except for giving rise to a unique object in a derived class, a *virtual* base class behaves exactly like a non−virtual base class.

The "virtualness" of *W* is a property of the derivation specified by *AW* and *BW* and not a property of *W* itself. Every *virtual* base in an inheritance DAG refers to the same object. This object is constructed once using a default constructor. A class that can only be constructed given an argument cannot be a virtual base.

A class may be both a normal and a virtual base in an inheritance DAG:

```
class A : virtual L { ... };
class B : virtual L { ... };
class C : A , B { ... };
class D : L, C { ... };
```

A *D* object will have two sub−objects of class *L,* one virtual and one "normal.

Representation

The object representing a virtual base class *W* object cannot be placed in a fixed position relative to both

AW and *BW* in all objects. Consequently, a pointer to *W* must be stored in all objects directly accessing the *W* object to allow access independently of its relative position as shown in Figure 5–9.

```
AW* paw = new AW;
BW* pbw = new BW;
CW* pcw = new CW;
```

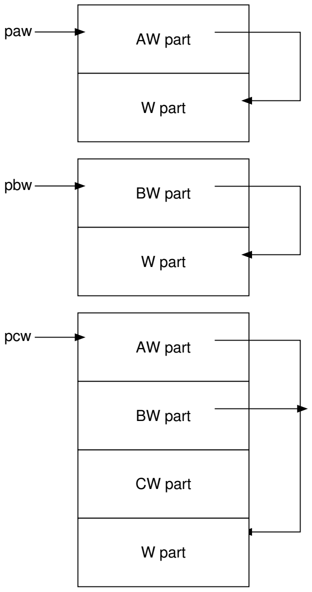

Figure 5–9 W Object Pointers

A class can have an arbitrary number of virtual base classes.

One can cast from a derived class to a virtual base class, but not from a virtual base class to a derived class. The former involves following the virtual base pointer; the latter cannot be done given the information available at run–time. Storing a "back–pointer" to the enclosing object(s) is non–trivial in general and was considered unsuitable for C++ as was the alternative strategy of dynamically keeping track of the objects "for which" a given member function invocation operates.

Virtual Functions

Consider:

```
class W {
      virtual void f();
      virtual void g();
      virtual void h();
      virtual void k();
      ...
};
class AW : virtual W { void g(); ... };
class BW : virtual W { void f(); ... };
class CW : AW , BW { void h(); ... };
i
CW* pcw = new CW;
pcw->f();        // BW::f()
pcw->g();        // AW::g()
pcw->h();        // CW::h()
((AW*)pcw)->f();    // BW::f();
```

A *CW* object might look as shown in Figure 5–10:

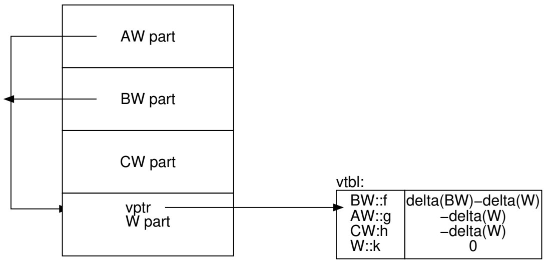

Figure 5−10 *CW* Object

In general, the delta stored with a function pointer in a *vtbl* is the delta of the class defining the function minus the delta of the class for which the *vtbl* is constructed.

If *W* has a virtual function *f* that is re−defined in both *AW* and *BW* but not in *CW* an ambiguity results. Such ambiguities are easily detected at the point where *CW 's* *vtbl* is constructed.

The rule for detecting ambiguities in a class lattice, or more precisely a directed acyclic graph (DAG) of classes, 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 as a DAG like Figure 5−11:

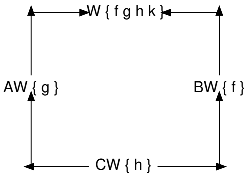

Figure 5−11 Directed Acyclic Graph (DAG)

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).

Constructors and Destructors

Constructors for base classes are called before the constructor for their derived class. Destructors for base classes are called after the destructor for their derived class. Destructors are called in the reverse order of their declaration.

Arguments to base class constructors can be specified like this:

```
class A { A(int); };
class B { B(int); };
class C : A , virtual B {
```

```
      C(int a, int b) : A(a) , B(b) { ... }
};
```

Constructors are executed in the order their objects are declared. This rule is applied to members and base classes separately and the base class constructors and applied before the member constructors. When a class has more than one base class *all* argument lists for its base class constructor *must* be qualified with the name of the base class. This rule applies even if only one of the base classes actually requires arguments.

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 (directed acyclic graph). 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.

Assignment to *this* in the constructor of a class that takes part in a multiple inheritance lattice is likely to lead to disaster. See **Chapter 1, "The Evolution of C++,"** for alternatives.

Visibility

The previous examples ignored visibility considerations. A base class may be *public* or *private.* In addition, it may be *virtual.* For example:

```
class D
      : B1          // private (by default), non−virtual (by default)
      , virtual B2  // private (by default), virtual
      , public B3   // public, non−virtual (by default)
      , public virtual B4 {
      // ...
};
```

Note that a visibility or *virtual* specifier applies to a single base class only. For example,

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

declares a public base *A* and a private base *B.*

Overheads

The overhead in using this scheme is:

1. one subtraction of a constant for each use of a member in a base class that is included as the second or subsequent base
2. one word per function in each *vtbl* (to hold the delta)
3. one memory reference and one subtraction for each call of a virtual function

4. one memory reference and one subtraction for access of a base class member of a virtual base class

Note that overheads [1] and [4] are only incurred where multiple inheritance is actually used, but overheads [2] and [3] are incurred for each class with virtual functions and for each virtual function call even when multiple inheritance is not used. Overheads [1] and [4] are only incurred when members of a second or subsequent base are accessed "from the outside"; a member function of a virtual base class does not incur special overheads when accessing members of its class.

This implies that except for [2] and [3] you pay only for what you actually use; [2] and [3] impose a minor overhead on the virtual function mechanism even where only single inheritance is used. This latter overhead could be avoided by using an alternative implementation of multiple inheritance, but I don’t know of such an implementation that is also faster in the multiple inheritance case and as portable as the scheme described here.

Fortunately, these overheads are not significant. The time, space, and complexity overheads imposed on the compiler to implement multiple inheritance are not noticeable to the user.

But is it Simple to Use?

What makes a language facility hard to use?

1. Lots of rules.
2. Subtle differences between rules.
3. Inability to automatically detect common errors.
4. Lack of generality.
5. Deficiencies.

The first two cases lead to difficulty of learning and remembering, causing bugs due to misuse and misunderstanding. The last two cases cause bugs and confusion as the programmer tries to circumvent the rules and "simulate" missing features. Case [3] causes frustration as the programmer discovers mistakes the hard way.

The multiple inheritance scheme presented here provides two ways of extending a class’s name space:

- a base class
- a virtual base class

These are two ways of creating/specifying a new class rather than ways of creating two different kinds of classes. The rules for using the resulting classes do not depend on how the name space was extended:

- ambiguities are illegal
- rules for use of members are what they were for single inheritance
- visibility rules are what they were for single inheritance
- initialization rules are what they were for single inheritance

Violations of these rules are detected by the compiler.

In other words, the multiple inheritance scheme is only more complicated to use than the existing single inheritance scheme in that

- you can extend a class’s name space more than once (with more than one base class)
- you can extend a class’s name space in two ways rather than in only one way

This appears minimal and constitutes an attempt to provide a formal and (comparatively) safe set of mechanisms for observed practices and needs. I think that the scheme described here is "as simple as possible, but no simpler."

A potential source of problems exists in the absence of "system provided back−pointers" from a virtual base class to its enclosing object.

In some contexts, it might also be a problem that pointers to sub−objects are used extensively. This will affect programs that use explicit casting to non−object−pointer types (such as *char*)* and "extra linguistic" tools (such as debuggers and garbage collectors). Otherwise, and hopefully normally, all manipulation of object pointers follows the consistent rules explained previously and is invisible to the user.

Conclusions

Multiple inheritance is reasonably simple to add to C++ in a way that makes it easy to use. Multiple inheritance is not too hard to implement, since it requires only very minor syntactic extensions, and fits naturally into the (static) type structure. The implementation is very efficient in both time and space. Compatibility with C is not affected. Portability is not affected.

Chapter 6

Parameterized Types for C++

This chapter explains Type parameterization, the ability to define a type in terms of another, unspecified, type.

This chapter contains the following sections:

- **"Introduction"**
- **"Class Templates"**
- **"Member Function Templates"**
- **"Outline of an Implementation"**
- **"Some Design Considerations"**
- **"Type Inquiries"**
- **"More about Implementation"**
- **"Function Templates"**
- **"Syntax Issues"**
- **"Templates and Typedef"**
- **"Type Equivalence"**
- **"Derivation and Templates"**
- **"Members and Friends"**
- **"Examples of Templates"**
- **"Conclusions"**
- **"Caveat"**

Note: This chapter is from a paper by Bjarne Stroustrup.

Introduction

Type parameterization is the ability to define a type in terms of another, unspecified, type. Versions of the parameterized type may then be created for several particular parameter types. A language supporting type parameterization allows specification of general container types such as *list, vector,* and *associative array* where the specific type of the elements is left as a parameter. Thus, a parameterized class specifies an unbounded set of related types; for example: list of int, list of name, list of shape, etc. Type parameterization is one way of making a language more extensible.

In the context of C++, the problems are:

1. Can type parameterization be easy to use?
2. Can objects of a parameterized type be used as efficiently as objects of a "hand−coded" type?
3. Can a general form of parameterized types be integrated into C++?
4. Can parameterized types be implemented so that the compilation and linking speed is similar to that achieved by a compilation system that does not support type parameterization?
5. Can such a compilation system be simple and portable?

A design is presented for which the answer to all of these questions is *yes.* The implementation of this scheme is a fairly simple extension of current C++ implementations.

Caution: The scheme for providing parameterized types described here is not implemented. It is not part of the C++ language, nor is there any guarantee that it ever will be. This paper was written to stimulate and focus discussion about the usefulness of a parameterized type facility for C++ and about the possible forms such a facility might take.

For many people, the largest single problem using C++ is the lack of an extensive standard library. A major problem in producing such a library is that C++ does not provide a sufficiently general facility for defining "container classes" such as lists, vectors, and associative arrays.

There are two approaches for providing such classes/types:

1. The Smalltalk approach: rely on dynamic typing and inheritance.
2. The Clu approach: rely on static typing and a facility for arguments of type *type.*

The former is very flexible, but carries a high run−time cost, and more importantly defies attempts to use static type checking to catch interface errors. The latter approach has traditionally given rise to fairly complicated language facilities and also to slow and elaborate compile/link time environments. This approach also suffered from inflexibility because languages where it was used, notably Ada, had no inheritance mechanism.

Ideally we would like a mechanism for C++ that is as structured as the Clu approach with ideal run−time and space requirements, and with low compile−time overheads. It also ought to be as flexible as Smalltalk's mechanisms. The former is possible; the latter can be approximated for many important cases.

Note that C++ appears to have sufficient expressive power to cope with the demands of library writing provided there is a single basic kind of object, such as a character (for string manipulation, pattern matching, character I/O, etc.), a double−precision floating−point number (for engineering libraries), or a bitmap (for graphics libraries). The "container class problem" is particularly serious, though, since container classes are needed to specify all but the simplest interfaces; they are the "glue" for larger systems.

Class Templates

A C++ parameterized type will be referred to as a class template. A class template specifies how individual classes can be constructed much like the way a class specifies how individual objects can be constructed. A vector class template might be declared like this:

```
template<class T> class vector {
        T* v;
        int sz;
public:
        vector(int);
        T& operator[](int);
        T& elem(int i) { return v[i]; }
        // ...
};
```

The *template <class T>* prefix specifies that a template is being declared and that an argument *T* of type *type* will be used in the declaration. After its introduction, *T* is used exactly like other type names within the scope of the template declaration. Vectors can then be used like this:

```
vector<int> v1(20);
vector<complex> v2(30);
typedef vector<complex> cvec;
                        // make cvec a synonym for vector<complex>
cvec v3(40);       // v2 and v3 are of the same type
v1[3] = 7;
v2[3] = v3.elem(4) = complex(7,8);
```

Clearly class templates are no harder to use than classes. The complete names of instances of a class template, such as *vector<int>* and *vector<complex>,* are quite readable. They might even be considered more readable than the notation for the built–in array type: *int[]* and *complex[].* When the full name is considered too long, abbreviations can be introduced using *typedef.*

It is only trivially more complicated to declare a class template than it is to declare a class. The keyword *class* is used to indicate arguments of type *type* partly because it appears to be an appropriate word, partly because it saves introducing a new keyword. In this context, *class* means "any type" and not just "some user–defined type."

The *<...>* brackets are used in preference to the parentheses *(...)* partly to emphasize the different nature of template arguments (they will be evaluated at compile–time) and partly because parentheses are already hopelessly overused in C++.

The keyword *template* is introduced to make template declarations easy to find, for humans and for tools, and to provide a common syntax for class templates and function templates.

Member Function Templates

The operations on a class template must also be defined. This implies that in addition to class templates, we need function templates. For example:

```
template<class T> T& vector<T>::operator[](int i)
{
      if (i<0 || sz<=i) error("vector: range error");
      return v[i];
}
```

A function template is a specification of a family of functions; *template<class T>* specifies that *T* is a template argument (of type *type)* that must somehow be supplied to specify a particular function.

Note that you don’t usually have to specify the template arguments to use a function template. For example, the template argument for *vector<T>::operator*[] will be determined by the vector to which the subscripting operation is applied:

```
vector<int> v1(20);
vector<complex> v2(30);
v1[3] = 7;             // vector<int>::operator[]()
v2[3] = complex(7,8);  // vector<complex>::operator[]()
```

Member functions of a class template are themselves function templates with the template arguments specified in the class templates. Function templates and member function templates will be discussed in greater detail below.

Outline of an Implementation

The basic idea for an implementation that incurs no additional costs in run–time or space compared with "hand coding" is to "macro–expand" a template for each different set of template arguments with which it is used. Naturally, template expansion is not really just macro expansion; it obeys proper scope and syntax rules. Names such as *vector<int>* can be encoded into composite class names such as *__PTvector_int.*

The example above expands into:

```
class __PTvector_int {
      int* v;
      int sz;
public:
      __PTvector_int(int);
      int& operator[](int);
      int& elem(int i) { return v[i]; }
      // ...
```

```
};
class __PTvector_complex {
      complex* v;
      int sz;
public:
      __PTvector_complex(int);
      complex& operator[](int);
      complex& elem(int i) { return v[i]; }
      // ...
};
__PTvector_int v1(20);
__PTvector_complex v2(30);
__PTvector_complex v3(40);
v1[3] = 7;
v2[3] = v3.elem(4) = complex(7,8);
```

A compiler need not have a separate template expansion pass. Since the information to do such an expansion exists in the compiler's tables, the appropriate actions can simply be taken at the proper places in the analysis and code generation process.

In addition to this expansion mechanism, a facility is needed for detecting which member functions have been used for which instances of a parameterized type. The example above used:

```
__PTvector_int::__PTvector_int();         // constructor
__PTvector_complex::__PTvector_complex();     // constructor
__PTvector_int::operator[]();             // subscripting
__PTvector_complex::operator[]();         // subscripting
__PTvector_complex::elem();
```

Note that the full list of such functions for a program can be known only after examining every source file. The linker provides a form of this list as part of its list of undefined objects and functions.

The definition of an operation on a class template might look like this:

```
template<class T> T& vector<T>::operator[](int i)
{
      if (i<0 || sz<=i) error("vector: range error");
      return v[i];
}
```

From this, the following two function definitions will have to be generated:

```
int& __PTvector_int::operator[](int i)
{
      if (i<0 || sz<=i) error("vector: range error");
      return v[i];
}
complex& __PTvector_complex::operator[](int i)
{
      if (i<0 || sz<=i) error("vector: range error");
      return v[i];
}
```

This approach ensures that no run−time efficiency is lost compared to "hand−coding." Code space might wasted by creating separate copies of functions that could have shared implementation. For example, *vector<int>* and *vector<unsigned>* need not have separate subscripting operations. Such waste can, if necessary, be reduced through suitable coding practices (see **"Type Equivalence"** below) and/or through a clever compile−time environment.

A programmer can provide a definition for a particular version of an operation on a class by specifying the template argument(s) in a function definition:

```
int& vector<int>::operator[](int i) { return v[i]; }
```

The general version of such a function as defined by its template will be used to create a function for a particular argument type only when no user–provided version is specified for that type.

Replacing the default implementation of a function as defined by a template is useful where implementations with greater precision, higher efficiency, etc. can be provided given some understanding of a particular type. It may also be useful for debugging and for supplying different versions of a function to different parts of a program (using *static* functions).

Some Design Considerations

Let us consider a few choices that were made to write the example above:

- Should all template arguments be of type *type?*
- Should a user be required to specify the set of operations that may be used for a template argument of type *type?*
- Should a user be required to explicitly declare what versions of a template can be used in a program?
- Should it be possible for a user to declare variables of type *type?*

The answer to all (in the context of C++) is *no.* Let us examine them in turn.

Template Arguments

"Should all template arguments be of type *type?"* No, there appear to be useful examples of type parameters of "normal" types. For example, a vector template might be parameterized with an error handling function:

```
typedef void (*PF)(char*);
template<class T, PF error> class vector {
      T* V;
      int sz;
public:
      T& operator[](int i) {
          if (i<= || sz<=i) error("vector: range error");
          return v[i];
      }
};

void my_error_fct() { ... }
vector<complex,&my_error_fct> v(10);
```

This example implies that default arguments might be useful:

```
template <class T, PF error=&standard_error_fct> class vector { ... }
```

Another example is a buffer type with a size argument:

```
template<class T, int sz=128> class buffer {
      T v[sz];
      // ...
};
void f()
{
      buffer<char> buf1;
      buffer<complex,20> buf2;
      // ...
}
buffer<char*,1000> glob;
```

Making *sz* an argument of the template *buffer* itself rather than of its objects implies that the size of a *buffer* is known at compile–time so that a *buffer* can be allocated without use of free store. It appears that default arguments will be at least as useful for template arguments as they are for ordinary arguments. Default arguments of type *type* might even be useful:

```
template<class T, class TEMP = double> class store {
      // ...
      T sum() { TEMP sum = 0; ... return sum; }
};
store<int,long> istore;
store<float> fstore;
```

These examples demonstrate that the range of templates with which a type can be parameterized should be restricted only if there are compelling arguments that the restriction will significantly ease the implementation of templates. I see no such argument.

Type Argument Attributes

"Should a user be required to specify the set of operations that may be used for a template argument of type *type?"* For example:

```
// The operations =, ==, <, and <=
// must be defined for an argument type T
template <
      class T {
         T& operator=(const T&);
         int operator==(const T&, const T&);
         int operator<=(const T&, const T&);
         int operator<(const T&, const T&);
      };
>
      class vector {
      // ...
};
```

No. Requiring the user to provide such information decreases the flexibility of the parameterization facility without easing the implementation or increasing the safety of the facility.

Consider *vector<T>.* To provide a sort operation one must require that type *T* has some order relation. This is not the case for all types. If the set of operations on *T* must be specified in the declaration of *vector* one would have to have two vector types: one for objects of types with an ordering relation and another for types without one. If the set of operations on *T* need not be specified in the declaration of *vector* one can have a single vector type. Naturally, one still cannot sort a vector of objects of a type *glob* that does not have an order relation. If that is tried, the generated sort function *vector<glob>::sort*() would be rejected by the compiler.

It has been argued that it is easier to read and understand parameterized types when the full set of operations on a type parameter is specified. I see two problems with this: such lists list would often be long enough to be de facto unreadable and a higher number of templates would needed for many applications.

Should experience show a need for specifying the operations on a parameterized type then such a facility can be easily and compatibly added later.

Source Code

There might be a more fundamental reason for requiring that the operations performed on a template argument of type *type* should be listed in the template declaration. The implementation technique outlined here achieves near optimal run-time characteristics by requiring the complete source code of a template to be available to the compiler when processing a use of the template. In some contexts, this is considered a deficiency and an implementation of templates that requires only the object code for functions implementing the function templates would be preferable.

At first glance it would appear that requiring the full set of operations on a template argument to be specified would make it significantly easier to produce such an implementation. In this case, a function template would be implemented by code using calls through vectors of function pointers to perform operations on template arguments of type *type.* The specification of the set of operations for a *type* argument would provide the definition for such vectors. Such an implementation would trade run−time for compile and link time, but would be semantically equivalent to the implementation scheme presented here.

Could an implementation along these lines be provided without requiring the user to list the set of operations needed for each function template argument of type *type?* I think so. Given a function template, the compiler can create a vector layout for the required set of operations without the help of a user. Given the full set of function definitions for the members of a class, the compiler can again create a vector layout for the required set of operations without the help of a user. If the compile and link environment cannot provide such a list a less optimized scheme where each member function has its own vector of operations can be used.

It thus appears that both implementation styles can be used even in the absence of template argument attribute lists so that we need not require them to preserve the implementers' freedom of action. It might be noticed that a virtual function table is in many ways similar to a vector of operations for a template so that the benefits of the vector of operations approach can often be achieved by a coding style relying on virtual functions rather than the expansion of function templates. Class *pvector* presented in **"Type Equivalence"** below on Type Equivalence is an example of this.

Type Instantiation

"Should a user be required to explicitly declare what versions of a template can be used in a program?" For example, should one require the use of an operation like Ada's *new?* No. Such a requirement would increase the size of the program text and decrease the flexibility of the template facility without yielding any benefits to the programmer or the implementer.

Type Variables

"Should it be possible for a user to declare variables of type *type?"* For example:

```
type t = int;
void f(type t)
{
      switch (t) {
      case int:
          ...
      case char*:
          ...
      case complex:
          ...
      default:
          ...
      }
}
```

Such a facility would be useful in many contexts, but does not appear suitable for C++. In particular, it is not possible to assign integer values to represent constants of type *type* such as *int, line_module*, double*(*)(*complex*,int*), and *vector<complex>* while maintaining the current style of separate compilation. Since the C++ type system is open such assignment of values in general requires an unbounded number of bits to represent a type. In practice, even simple cases require lots of bits (the current *cfront* scheme for encoding function types in character strings regularly uses dozens of characters) or some system of hashing involving a database of types. Furthermore, the introduction of such variables would require an order of magnitude greater changes to the C++ language and its implementations than the scheme (without type variables) described here.

Type Inquiries

It would be possible to enable a programmer to inquire about properties of a template argument of type

type. This would allow the programmer to write code that depends on specific properties of the actual types used.

An Inquiry Operator

Consider providing a print function for a vector type that sorts the elements before printing if and only if sorting is possible. A facility for inquiring if a certain operation, such as *<,* can be performed on objects of a given type can be provided. For example:

```
template<class T> void vector<T>::print()
{
      if (?T::operator<) sort();    // if (T has a <) sort_this_vector
      for (int i=0; i<sz; i++) { ... }
}
```

Because the *<* operation is defined for *int s,* printing of a *vector<int>* gives rise to an expansion:

```
void __PTvector_int::print()
{
      sort();        // that is, this->sort()
      for (int i=0; i<sz; i++) { ... }
}
```

On the other hand, printing a *vector<glob>* where the *<* operation is not defined for *glob s* gives rise to an expansion:

```
void __PTvector_glob::print()
{
      for (int i=0; i<sz; i++) { ... }
}
```

Tests on expressions of the form *?typ::oper* ("does type *typ* have an operation *oper?")* must be evaluated at compile–time and can be part of constant expressions.

It would probably be wise *not* to include such a type inquiry feature in the initial experimental implementation but to wait and see what properties (if any) programmers would find useful. Potentially every aspect of a type known to the compiler can be made available to the programmer; *sizeof* is a most rudimentary version of this kind of facility.

The absence of a type inquiry facility will be compensated for by the ability to define a function for a particular set of template arguments, thus overriding the generation of the "standard" version from the template. Furthermore, it can sometimes be preferable to define separate templates to represent the different concepts. For example, one might have both a *vector<T>* class and a *sorted_vector<T>* class derived from it.

The typeof Operator

Writing code where the control flow depends of the properties of a type parameter doesn't appear to be necessary, but defining variables of types dependent on type parameters does. Given a template argument of type *type,* *T,* one can express a variety of derived types using the declarator syntax; for example: *T*, T&, T*[*10*], *T*(***)(*T,T*). One can also express types obtained by template expansion such as *vector<T>.* However, this does not conveniently express all types one might like. In particular, the ways of expressing types that depend on two or more template arguments are weak. To compensate, one might introduce a *typeof* operator that yields the type of its argument. For example:

```
template<class X, class Y> void f(X x, Y y)
{
      typeof(x*y) temp = x*y;
      // ...
}
```

It would probably be wise *not* to introduce a *typeof* operator before gaining more experience. The uses of *typeof* appears to be quite limited and the scope for misuses large. In particular, *typeof* appears more suited for the writing of macros (which templates are designed to replace in many contexts) than for templates and heavy use of *typeof will reduce the compilers ability to pinpoint type errors.*

More about Implementation

So how can we generate the proper code for definitions of operations on a template for a given set of arguments? Assume that we know that versions of *vector 's* subscripting operation

```
template<class T> vector<T>::operator[](int) { ... }
```

are needed for *T==int* and *T==complex.* How can we create the proper expansions (as presented above)?

We might have a compiler option, *−X,* for creating such expansions. Assuming that the definitions for *vector 's* member functions resides in a file called *vector.c,* one might call the compiler like this:

```
CC −X "vector<int>" vector.c
CC −X "vector<complex>" vector.c
```

and have the appropriate *.o* files created. This would create not only the required subscript operator functions but also functions for any other vector operation that has its definition stored in *vector.h.* The strategy for splitting a program into separately compiled parts is in the hands of the programmer. Where a finer granularity is required of *.o* files for a library, the programmer can handle it using standard C library techniques.

Note that an expansion using the template expansion option, *−X,* may give rise to a program that uses an instance of a template that has not already been used in the program. This implies that another stage of "missing template implementation detection" is required after each expansion. Expansion is really a recursive activity. The depth of this recursion will typically be 1. It will be necessary to have a mechanism protecting against recursive expansion. For example:

```
template<class T> void f(T a) { T* p; ... f(p); }
```

Naturally, one would try to ensure that *CC −X* is used to generate *.o* files only for definitions of templates when

- a new template was used, or
- a new set of template arguments was used, or
- the declaration of a template was changed.

I imagine that after a short startup period all the necessary *.o* files for templates for a program/project will reside in a library and not interfere with the compilation process. When a program/project reaches this state the compilation overhead incurred by using templates becomes negligible.

Tools for Ensuring Consistent Linking

Consider having the tools described above:

- a C++ compiler handling the expansion of class templates into class declarations, and
- a *−X* option on this compiler to handle the expansion of function templates into function definitions.

One could then compile a C++ program using templates. A little manual intervention would be needed to get a complete program to link and load.

What additional tools would be needed to

- guarantee consistent and complete expansion and linking?
- make programming reasonably convenient?

I conjecture that [1] is perfectly feasible, but non−trivial, where portability across operating systems, compile and link time efficiency, and flexibility are all required. I also conjecture that very little is needed to achieve. Experience using such a system is clearly needed, but it might well be sufficient to modify a tool with access to the complete compiled program, such as *munch* or the linker itself, to produce

1. a list of function definitions required, or
2. a list of files for which *CC −X* needs to be run (assuming some correspondence between type names

and file names), or

3. a make script for running *CC −X* for an appropriate set of files.

It would also be important to ensure that *CC* produces readable error messages when an operation is applied to a particular template argument of type *type* for which it is not defined. For example:

```
"foo.c", line 144: error: operator<= applied to glob in vector<glob>::sort()
```

This discussion of how one might provide a minimal and portable mechanism supporting templates in C++ should not be taken as an indication that such a mechanism provides the ideal programming environment. On the contrary, it is exactly a *minimal* facility. Much better facilities can be built (think of a smart *make,* an incremental compiler, a Smalltalk−like browser, etc.), However, a minimal facility *must* exist to ensure portability of C++ programs between all implementations since there is no hope that a single maximal programming environment will ever be agreed on and implemented on every system supporting C++.

Function Templates

In addition to providing class templates, it is necessary to provide function templates. Consider providing a general *sort* function:

```
template<class T> void sort(vector<T>);
```

Given a vector *v,* one might call such a function like this:

```
sort(v);
```

The compiler can deduce the type of the sort function from the type of the vector. For example, had *v* been declared

```
vector<int> v(10);
```

the sort function *sort<int>* would have been required. On the other hand had the declaration of *v* been

```
vector<double> v(2000);
```

the sort function *sort<double>* would have been used.

Overloading

Declaring a function template is simply a way of declaring a whole bundle of overloaded functions at one time. This implies that we can use functions with arguments that can be distinguished by the overloaded function resolution mechanism only. The following function cannot be used because it takes no argument:

```
template<class T> T* create() { return (T*) malloc(sizeof(T)); }
```

The C++ syntax could be extended to cope with this by allowing the full generality of the *name<type>* notation so that template arguments could be supplied explicitly in a call:

```
int* pi = create<int>();    // create_int()
char* pc = create<char>();    // create_char()
```

Unless programmers define templates sensibly this form of resolution can become quite cryptic:

```
template<class X, class Y> f(Y,X);  // template argument order differs
                                      // from function argument order
// ...
f<char*,int>(1,"asdf");
```

I think it would be wise not to include any explicit resolution method in an initial implementation. I suspect that the implicit resolution provided by the overloaded function resolution rules are sufficient—and more elegant—in almost all cases and it is not obvious that a mechanism for explicit overloading is worth the added complexity.

Allowing explicit resolution would imply that a C++ compiler should treat function template names differently from other names and similarly to the way keywords and class names are treated. For example, without special rules for template names the last expression above would be parsed as two comparisons and a parenthesized comma expression:

```
(g<123)>(vv,10);
```

A Problem

Consider writing a function *apply*() that applies another function *g*() to all the elements of a vector. A traditional first cut would look something like this:

```
template<class T> void apply(vector<T>& v, T& (*g)(T&))
{
      for (int i = 0; i<v.size(); i++) v[i] = (*g)(v[i]);
}
```

This follows the C style of using a pointer to function. Potential problems with this are

- efficiency, because there can be no inline expansion of the applied function, and
- generality, because standard operations of built−in types such as − and ~ for *int s* cannot be applied.

Naturally, these are not problems to all people. However, an ideal template mechanism will provide solutions.

A Solution

One might consider the function to be applied by *apply*() a template argument rather than a function argument:

```
template<class T, T& (*g)(T&)> void apply(vector<T>& v)
{
      for (int i = 0; i<v.size(); i++) v[i] = (*g)(v[i]);
}
```

To call *apply*() one must specify the function to be applied. Since this version of *apply*() takes only a single *vector* argument the syntax for disambiguating an overloaded function call using *<...>* must be used:

```
class X { ... };
vector<X> v2(200);
inline void hh(X&) { ... };
void gg(X&);    // not inline
apply<X,hh>(v2);
apply<X,gg>(v2);
```

Clearly, the *X* is redundant and not elegant. Since in principle each such call of *apply*() results in writing a new function *apply*() inlining can be applied where sufficient information is available. Consequently, one would expect a C++ compiler to inline *hh*() in the first call in the example above and generate a standard function call of *gg*(). The fact that function pointers and not functions are passed in C++ is at most a minor annoyance for the compiler writer.

Operators for built−in types can be considered inline functions in this context:

```
vector<int> v(100);
apply< int, &int::operator-- >(v);
```

However, as for the explicit resolution scheme itself, it remains to be seen if this degree of sophistication and complexity is actually needed.

Syntax Issues

Consider the declarations:

```
template<class T> class vector { ... };
template<class T> T* index<class T>(vector<T>,int);
```

1. Why use the *template* keyword?
2. Why use *<...>* brackets and not parentheses?
3. Why use the *class* keyword?
4. What is the scope of a template argument?

The template keyword

If a keyword is to be used *template* seems to be a reasonable choice, but it is actually not necessary to introduce a new keyword at all. For class templates, the alternative syntax seems more elegant at first glance:

```
class vector<class T> {        // possible alternative class syntax
      ...
};
```

Here the template arguments are placed after the template name in exactly the way they are in the use of a class template:

```
vector<int> vi(200);
vector<char*> vpc(400);
```

The function syntax at first glance also looks nicer without the extra keyword:

```
T& index<class T>(vector<T> v, int i) { ... }
```

There is typically no parallel in the usage, though, since function template arguments are not usually specified explicitly:

```
int i = index(vi,10);
char* p = index(vpc,29);
```

However, there appears to be nagging problems with this "simpler" syntax. It is too clever. It is relatively hard to spot a template declaration in a program because the template arguments are deeply embedded in the syntax of functions and classes and the parsing of some function templates is a minor nightmare. It is possible to write a C++ parser that handles function template declarations where a template argument is used before it is defined, as in *index*() above. I know, because I wrote one, but it is not easy nor does the problem appear amenable to traditional parsing techniques. In retrospect, I think that not using a keyword and not requiring a template argument to be declared before it is used would result in a set of problems similar to those arising from the clever and convoluted C and C++ declarator syntax.

<...> vs. (...)

But why use brackets instead of parentheses? As mentioned before, parentheses already have many uses in C++. A syntactic clue (the *<...>* brackets) can be useful for reminding the user about the different nature of the type parameters (they are evaluated at compile–time). Furthermore, the use of parentheses could lead to pretty obscure code:

```
template(int sz = 20) class buffer {
      buffer(int i = 10);
      // ...
};
buffer b1(100)(200);
buffer b2(100);        // b2(100)(10) or b2(20)(100)?
buffer b3;        // legal?
```

These problems would become a serious practical concern if the notation for explicit disambiguation of overloaded function calls were adopted. The chosen alternative seems much cleaner:

```
template<int sz = 20> class buffer {
      buffer(sz)(int i = 10);
      // ...
};
buffer b1<100>(200);
buffer b2<100>;        // b2<100>(10)
buffer b3;        // b3<20>(10)
buffer b4(100);        // b4<20>(100)
```

The class keyword

Unfortunately, the ideal word for introducing the name of a parameter of type *type,* that is, *type* cannot be used; *type* appears as an identifier in too many programs. Why use the *class* keyword then? Why not? Classes are already types to the extent that the built-in types can be considered second class citizens in some contexts (you cannot derive a class from a built in type, you cannot take the address of an operation on a built-in type, etc.). What is done here is simply to use *class* in a slightly more general form than is done elsewhere.

Scope of Template Argument Names

The scope of a template argument name is the template declaration and the template name obeys the usual scope rules:

```
const int T;
template<class T>    // hides the const int T
class vector {
      int sz;
      T* v;
public:
      // ...
};
int T2 = T;        // here const int T is visible again
```

Template declarations may not be declaration lists:

```
template<class T> f(T*), g(T);    // error: two declarations
```

This restriction is made to avoid users making unwarranted assumptions about relations between the template arguments in the different templates.

Templates and Typedef

The template concept is easily extended to cover all types

This section has been changed since the USENIX C++ conference proceedings version of this paper based on comments by George Gonthier.

. For example:

```
template<class T, int i> typedef T array[i];
// ...
array<int,10> v;    // array of 10 ints
```

This allows great freedom in defining type names. The *typedef* keyword is necessary because

```
template<class T, int i> T array[i];
```

would define a family of arrays (all called *array)* and not a family of array type.

Consequently, only class, function, and typedef templates will be implemented.

Type Equivalence

Consider:

```
template<class T, int i> class X {
      T vec[i];
      // ...
};
array<int,10> x;
array<int,10> y;
array<int,11> z;
```

Here, *x* and *y* is of the same type, but *z* is of the different type. Since the template arguments used in the declarations of *x* and *y* are identical they refer to the same class. Naturally, only a single class declaration is generated by a C generating C++ compiler. On the other hand, the template arguments used in the declaration of *z* differs and gives rise to a different class.

Different template arguments give rise to different classes even if the argument is used in a way that does not affect the type of the generated class:

```
template<class T, int i> class Y {
public:
      foo() { int buf[i]; ... }
};
Y<int,10> xx;
Y<int,10> yy;
Y<int,11> zz;
```

Template arguments must be types, constants, or integer expression that can be evaluated at compile−time.

Derivation and Templates

Among other things, derivation (inheritance) ensures code sharing among different types (the code for a non−virtual base class function is shared among its derived classes). Different instances of a template do not share code unless some clever compilation strategy has been employed. I see no hope for having such cleverness available soon. So, can derivation be used to reduce the problem of code replicated because templates are used? This would involve deriving a template from an ordinary class. For example:

If that really is a problem: memory is cheap, etc. I think it is a problem and will remain so for the foreseeable future. People's expectations of computers have consistently outstripped even the astounding growth in hardware performance.

```
template<class T> class vector {  // general vector type
      T* v;
      int sz;
public:
      vector(int);
      T& elem(int i) { return v[i]; }
      T& operator[](int i);
      // ...
};
template<class T>
class pvector : vector<void*> {  // build all vector of pointers
                                   // based on vector<void*>
public:
      pvector(int i) : (i) {}
      T*& elem(int i) { return (T*&) vector<void*>::elem(i); }
      T*& operator[](int i) { return (T*&)
                              vector<void*>::operator[](i); }
      // ...
```

```
};
pvector<int*> pivec(100);
pvector<complex*> icmpvec(200);
pvector<char*> pcvec(300);
```

The implementations of the three vector of pointer classes will be completely shared. They are all implemented exclusively through derivation and inline expansion relying on the implementation of *vector<void*>.* The *vector<void*>* implementation is a good candidate for a standard library.

I conjecture that many class templates will in fact be derived from another template. For example:

```
template<class T> class D : B<T> {
      ...
};
```

This also ensures a degree of code sharing.

Members and Friends

Here are some more details:

Member Functions

A member function of a class template is implicitly a template with the template arguments of its class. Consider:

```
template<class T> class C {
      T p;
      T m1() { T a = p; p++; return a; }
};
C<int> c1;
C<char*> c2;
int i = c1.m1();    // int C<int>::m1() { int a = p; p++; return a; }
char* s = c2.m1();  // char* C<char*>::m1() { char* a = p; p++; return a; }
```

These two calls of *m1*() gives rise to two expansions of the definition of *m1*().

Naturally a member template may also be declared:

```
template<class T> class C {
      template<class TT> void m(TT*,T*);
};
```

This case will be discussed below. However, explicit use of class template arguments in member function names is unnecessary and illegal:

```
template<class T> class C {
      T m<T>();               // error
};
template<class T> C<T>::m<T>() { ... }      // error
template<class T> C<T>::m() { ... }         // correct
```

This also applies to constructors:

```
template<class T> class C {
      C();             // correct, a constructor
      C<T>(int);         // error constructor
};
template<class T> C<T>::C() { ... }    // correct
```

To avoid confusion it is not legal to define a template as a member with the same template argument name as was used for the class template:

```
template<class T> class C {
      template<class T> T m(T*);               // error
};
```

Friend Functions

A friend function differs from other functions only in its access to class members. In particular, a friend of a class template is not implicitly a template. Consider:

```
template<class T> class C {
      friend f1(T a);
      template<class TT> friend f2(TT a);
};
```

The definitions of *f1*() and *f2*() are legal, but clearly not equivalent.

The *friend* declaration of *f1(T)* specifies that for all types *T, f1<T>* is a friend of *C<T>.* For example, *f1<int>* is a friend of *C<int>.* However, *f1<int>* is not a friend of *C<double>.* The definition of *f1*() would probably look something like this:

```
template<class TT> f1(TT a) { ... };
```

The friend *f1*() need not be a template, but if it isn't the programmer might have a tedious time constructing the necessary set of overloaded functions "by hand."

The declaration of *f2*() specifies that for all types *T* and *TT, f2<TT>* is a friend of *C<T>.* For example *f2<int>* is a friend of *C<double>.*

Note that a friend function of a parameterized class need not itself be parameterized:

```
template<class T> class C {
      static int i;
      friend f() { i++; }
};
```

Static Members

Each version of a class template has its own copy of the static members of the class:

```
template<class T> class C { static T a; static int b; ... };
C<int> xx;
C<double> yy;
```

This implies allocation of the static variables:

```
static int C<int>::a;
static int C<int>::b;
static double C<double>::a;
static int C<double>::b;
```

Similarly, each version of a parameterized function has its own copy of static local variables:

```
template<class T> f() { static T a; static int b; ... };
```

Friend Classes

Friend classes can (as usual) be declared as a shorthand for declaring all functions friends:

```
template<class T> class C {
      friend template<class TT> class X;    // all X<TT>s
      friend class Y<T>;             // only Y<T>
      friend class Z<int>;             // only Z<int>
};
```

Examples of Templates

Here are some more examples of potentially useful templates. Versions of many of the templates used as examples in this paper have been created using macros and actually used in real programs. "Faking" templates using macros have been a major design technique for the template facilities. In this way the language facilities could be designed in parallel with the key examples and techniques they were to support.

An associative array:

```
template<class E, class I> class Map {
      // arrays of Es indexed by Is
      // ...
      E& operator[](I);
};
```

A "record" stream; the usual stream of characters is a special case:

```
template<class R> class ostream {
      // ...
      ostream<R>& operator<<(R&);    // output an R
};
```

An array for mapping information from files into primary memory:

```
template<class T, int bsz> class huge {
      T in_core_buf[bsz];
      // ...
      T& operator[](int i);
      seek(long);
      // ...
```

A linked list class:

```
template<class T> class List { ... };
```

A queue tail template for sending messages of various types:

```
template<class T> class mtail : public qtail {
      // ...
      void send(T arg)
      {
          // bundle "arg" into a new message buffer
          // and put than on the queue
      }
};
```

A counted pointer template (for user–defined automatic memory management):

```
template<class T> class CP {
      // ...
public:
      CP();
      CP(T);
      CP(CP<T>&);
      // ...
};
```

Conclusions

A general form of parameterized types can be cleanly integrated into C++. It will be easy to use and easy to document. The implementation can be efficient in both run−time and space. It can be implemented portably and efficiently (in terms of compiler and link time) provided some responsibility for generating the complete set of definitions of function templates is placed on the programmer. This implementation can be refined, but probably not without loss of either portability or efficiency. The required compiler modifications are manageable. In particular, cfront can be modified to cope with templates. Compatibility with C is maintained.

Caveat

The key thing to get right for a C++ template facility is assuring that basic parameterized classes are implemented in an easy to use and efficient way for the relatively simple key examples. The compilation system *must* be efficient and portable at least for these examples. The most reasonable approach to building a template system for C++ would be to achieve this first, make the inevitable changes in concepts based on that experience, and proceed with more advanced features *only* as far as they makes sense *then*.

Chapter 7

Automatic Template Instantiation

This chapter describes how the USL C++ compiler version 3.0.1 handles automatic template instantiation and briefly reviews alternate strategies. The chapter does not cover practical usage of the instantiation system; this topic is discussed in **Chapter 8, "Template Instantiation Environment."**

The template environment is an *implementation* of the template language feature described in *The Annotated C++ Reference Manual* (M.A. Ellis and B. Stroustrup, Addison−Wesley, 1990). Please note that, unlike the language itself, implementations are not subject to formal standardization and are therefore subject to change.

This chapter contains the following sections:

- **"Templates and Instantiation"**
- **"Definitions of Terms"**
- **"Overall Instantiation Strategies"**
- **"Executive Summary"**
- **"Detailed Description of the Instantiation Process"**
- **"Limitations of Our Approach"**
- **"Other Approaches to Instantiation"**
- **"Summary"**

Note: This chapter is from a paper by Glen McCluskey and Robert B. Murray.

Templates and Instantiation

A *template* is a skeleton for defining a set of types or functions. Each type or function of the set is created by combining the template with a set of arguments that are themselves types or values. This process is known as *instantiation*.

For example, consider a template such as the following:

```
template <class T> class List {
      T* items;
      int count;
public:
      void additem(T);
      /* ... */
};
```

This declares a template called *List*, with two data members *items* and *count*, and one member function *additem*. If one were to use this template in an application by saying

```
main()
{
	List<double> li;
	li.additem(12.34);
}
```

then a particular combination of the template *List* and the argument *double* would be instantiated, with two data members and one publicly-accessible member function *List<double>::additem(double)*.

The instantiation problem is one of creating compiled files (object files) containing the instantiations needed by an application. These files should be minimal in size; only members of the template used in the application should be instantiated. The user may have provided a special case for a template member or a complete template class, and this *specialization* should override any automatic instantiation procedure.

Definitions of Terms

We will use the following terms in this chapter (a complete glossary is given at the end).

The *template declaration file* of a template is the file that contains the template declaration (for example *List.h* might contain the declarations for *List*). These declarations must be present in any file that refers to the template, that is, *#included*.

The *template definition file* of a template is the file that contains the definitions of the template members (for class templates), or the definition of a function template (*List.c* might contain the definitions for *List*). These definitions are required to instantiate a template, but not to refer to it.

The *argument declaration file* of a named type is the file that contains the declaration of that type (*String.h* might contain the declarations for *String*). The argument declarations of all the type arguments are needed to instantiate a template.

To instantiate a template, you need the template declaration file, the template definition file, and the argument declaration files for each template type argument. For example, to instantiate *List<String>*, you might need *List.h* (template declaration file for *List*), *List.c* (template definition file for *List*), and *String.h* (argument declaration file for *String*).

While we are using file names that correspond to the class names, the instantiation scheme cannot force such a relationship between a class name and the name of the files that declare and/or define it. For example, a scheme that requires the declarations or definitions of *String* to be in *String.h* and *String.c* is unacceptable, because such a scheme makes it impossible to declare or define two templates in the same file. This scheme also obviously falls short in reflecting real-world naming conventions.

Overall Instantiation Strategies

The basic idea of template instantiation is that the linker figures out which template functions you used and for which argument types, finds the template function definitions, and generates the functions needed. This will work completely automatically provided that a class or template *C* is declared in a file *f.h* and its non-inline functions are defined in *f.c*. The following sections will describe template instantiation in detail, explain some of the design choices made, and show how the default conventions can be overridden.

The instantiation schemes that we have seen so far fall into one of three families. Two of these use a *repository*; this is conceptually an archive of template instantiations, although it need not be implemented as a UNIX™ system archive.

Instantiate as You Go

Every compile is done with reference to a repository of existing template instantiations. When a reference to a template is compiled, the repository is checked to see if a valid instantiation for that template exists. If it does not, the template is instantiated (in a separate object file) and the result put into the repository.

The advantage of these schemes is that the declarations of both the templates and the argument types are available (since the file that referenced the template must have had them available). This avoids problems with having to "go back later" and find the template and argument declaration files.

If the general version of a template does not compile, but a special case version (specialization) will be presented at link time, these schemes will still attempt to compile the general version and produce error messages. The user would have to know enough to ignore these messages, if and only if a specialization is going to be presented at link time.

In addition, these schemes place severe constraints on makefiles. If a template declaration or definition, or an argument declaration, is changed, the makefiles must be smart enough to remove the changed instantiations from the repository *before any other source files are compiled to object files*. In general this requires adding an extra dependency to every makefile rule. If the makefile has a bug, the application may appear to compile but would not link until the files that referred to the missing instantiations were recompiled (even though those files had not changed). Or worse, the application may fail at runtime because of an incorrect instantiation being used. This behavior is just too mysterious and error prone.

Manual Instantiation

With a manual instantiation scheme, the user is responsible for writing one or more files that specify exactly which templates are to be instantiated. Or the user might specify *#pragma* directives to achieve the same end. This may seem to be the easiest to implement; however, there is a snag. If a template *Stack<T>* uses *List<T>* in its implementation, how does the instantiation of *Stack<T>* cause *List<T>* to get instantiated?

One way is to force the user to ask for the instantiation of *List<T>*. This violates encapsulation: the user should not have to know that *Stack<T>* uses *List<T>*.

Alternatively, the system could be smart enough to figure out that *Stack<T>* uses *List<T>*, and to instantiate it if and only if *List<T>* is not instantiated anywhere else. However, if the system is smart enough to do this, it should also be smart enough to figure out which templates the *user's* code referred to! In this case it ought to be able to do instantiation automatically.

The point is that *a manual instantiation scheme that does not violate encapsulation must be smart enough to do automatic instantiation.* There is no "middle ground." We have therefore abandoned totally manual approaches.

Link−Directed Instantiation

Every compile is done with a repository (or path of repositories) of existing template instantiations. Objects from this repository are linked when the linker is called. If the link fails because of unresolved references, *nm* is run on the output and a list of required template members is generated. These templates are then instantiated, added to the repository, and the link phase repeated until all needed templates have been instantiated.

Why a repository? Instantiation of template functions for a large program can be a relatively slow process. It is therefore important that a function is not reinstantiated unnecessarily. The repository ensures that a template function is instantiated the first time a program using it is linked and after each change to the program that affects the definition of a template function − and *not* simply at each compilation or linking. This implies that compiling a program for the first time and after a major change can be relatively slow but that a typical compile and link step is hardly affected by template instantiation overhead.

These schemes have important advantages:

- Only the template members that are used are instantiated. If a special case instantiation is provided at link time, no attempt is made to instantiate the corresponding general version.
- Since the instantiation process takes place at link time, the repository only has to be updated by link time, not by the compile−time of any source file that uses a template. This avoids complicating the makefiles.

The disadvantages of link−directed instantiation are:

- There must be a well−defined and easily understood set of rules that the instantiator uses to find the template declaration, template definition, and argument type declaration files.

- References to templates that will not instantiate are not discovered until link time. We do not believe this is serious, especially since any apparent reference to a general template member may be a reference to a specialization provided at link time. Because of this, checking that a template is not bogus at the point of reference is not feasible.
- The number of iterations needed to initially populate the repository can be large. If we examine the call graph of the final program, give every call of a template member or function template cost 1, and every other call cost 0, then the number of iterations is potentially the diameter of the resulting graph. (This is the longest of the shortest paths between every two nodes in the graph.) We do not have a good handle on how many iterations this will be, though observations from existing programs suggest an upper limit of 3−4 iterations in practice. However, to reduce the overhead, we will emulate some of the linker functionality in a "pre−linker" to avoid the overhead of multiple links. This tool reads the symbol tables of the objects and archives, determining which unresolved references would draw in which files. It does not do any relocation. This allows us to only do one link at the end when we know that all the required instantiations are available.

The remainder of this chapter describes our instantiation scheme.

Executive Summary

This section gives a high−level overview of the instantiation process. Later sections will go into more detail.

1. For all compilations, there is a *current repository* into which instantiations are stored. The instantiation system modifies the repository but does not clean up obsolete files within it.
2. When a C++ file that references templates is compiled, the references are compiled normally into external unresolved symbols, but nothing is instantiated. (Exception: a specialization of a general−purpose template is compiled in the object file that it appears in, and is the same as a normal class declaration and implementation).
3. Every class, union, struct, or enum that is declared is logged in a *name mapping file* in the current repository. The entry includes the type name and the basename (not full pathname) of the file in which the declaration appears; the basename−only rule is used to simplify moving applications from one directory to another.
4. Every template that is declared is also logged in the name mapping file. Definitions of members of class templates are *ignored*; there is no use for individual member information.
5. At link time, a *pre−linker* determines whether a link would succeed, that is, it looks at all the files and archives, and the current repository, to determine whether there are any referenced template symbols that are unresolved. Header caches are used to make sure that any instantiations in the repository that are out of date are not used. If there are no unresolved symbols, we do a link, and are done.
6. If there are unresolved symbols, the instantiator builds a list of class templates and function templates that must be instantiated. For each template, the name mapping file is consulted to find the template declaration file, and the argument declaration files for all of the template arguments.
7. Unless explicitly specified by the user (see **"User Overriding of Name Mapping Files"** below on overriding name mapping files), the template definition file is assumed to be the file with the same name as the corresponding template declaration file, except that it has a *.c* suffix.
8. When templates are used, a *−I* path (the same one used for compiling) must be passed to the pre−linker. This −I path is used to find the template declaration file, the template definition file, and the argument declaration files. If any of these files are not found, the link fails with an appropriate diagnostic.
9. The template declaration file, the template definition file, and the argument declaration files are used to build a temporary *instantiation file*. The C++ compiler is then called to instantiate the template. The compile is done in *directed mode*; that is, the compiler is directed to only generate code for a specified list of symbols. If this is a class template, only the members that were needed are instantiated. The resulting object file is added to the repository.

10. Steps 6–8 are repeated for every template class or function that needs to be instantiated.
11. When all the instantiations have been done, the pre–linker is called again. Since some of the new instantiations may refer to new symbols, this process may iterate (back to step 5).
12. The result of the pre–linker is a set of object files containing the instantiations. These objects are passed to the actual link step.

The rest of this paper describes the process in detail, including the ways that users can exercise finer control over the instantiation process.

Detailed Description of the Instantiation Process

Repositories

A *repository* is a UNIX™ directory used to store information about instantiations and types. It should not be confused with an archive library of object files. A repository pathname can be specified by the user at compile or link time. The default repository is a subdirectory of the current directory and is automatically created (but not automatically cleaned up).

The repository contains several kinds of information:

- Name mapping files that map a template or named type to the name of the file that declares it. By default, only the basename of the file (after stripping *–I* prefixes), not its full pathname, is stored.
- Object files that contain template instantiations.
- Checksum files containing a list of needed members for each instantiation.

Compile–Time Actions

The C++ compiler maintains a name mapping file, which records the name of the file in which a class, struct, union, or enum type is declared. The extraction of type information is done by the C++ compiler after preprocessing. The compiler relies on *#line* entries in preprocessor output to determine what source file is being translated. For all class, enum, and template types, the compiler notes the basename of the file currently being translated. Typedefs, except for typedefs of anonymous structs, are ignored because they are expanded to the underlying type, which is used to encode external names. Local and unnamed types are ignored, and only the outermost type in a nested type is used.

For each type extracted, the name of the type and the basename of the file where it is declared is recorded in the name mapping files, together with the basename of the application file being compiled:

```
@dec Vector app1
"Vector.h"
@dec A app2
<A.h>
1@dec B app2
<A.h>
@dec C
"C.h"
```

The *@dec* keyword indicates a declaration. This file shows that template *Vector* is declared in *Vector.h*, class *A* is declared in *A.h*, and class *B* is also declared in *A.h*. The *#include* quote type is recorded with the file to preserve preprocessor semantics.

The header basename is computed by comparing the pathname reported by the preprocessor against all the *–I* directives and deleting any prefix that matches. Sometimes this will result in a name such as *sys/stat.h*.

Two application files *app1* and *app2* are used in this example. Application file names are recorded to support the case where two applications share one repository. If no application basename is given, then the *@dec* entry is a default one that is used as a last resort; the fourth entry is an example. The type lookup algorithm is described in **"Class Templates."**

For function templates, the type name is the function name without arguments. For example:

```
@dec f app3
"f.h"
```

for the function *f(char*)* that came from:

```
template <class T> void f(T t);
```

Because arguments are not recorded, map file entries for function templates contain the union of all files where the template was declared.

The name mapping files in the repository are not updated if the compile fails. If the resulting mapping files have multiple entries for the same type and application basenames, as in:

```
@dec A app1
<A.h>
@dec A app1
<B.h>
```

then an error is given at pre-link time, if the type is needed (lazy type lookup).

Also, if a header file previously existed in some of the entries in the default name mapping file, all the entries for that header are deleted before updating with new information. When a source file is compiled a list is made of all headers it contains. Each entry in the map file that has one of the same headers, and that contains a matching application name, is deleted before new entries are added. This tends to reduce the buildup of garbage.

User Overriding of Name Mapping Files

The compiler automatically generates its own name mapping files. However, users can override these name mapping files for special cases by writing their own. For instance, suppose that use of class *A* requires that both *A.h* and *B.h* be included, in that order, for the declaration to compile. Then the user-specified name mapping file must read:

```
@dec A
<A.h>
<B.h>
```

Users can also override the rule that the name of the template definition file is the name of the template declaration file, except with a *.c* suffix, by placing a *@def* directive in the name mapping file:

```
@dec List
"List.h"
@def List
"List_impls.c"
```

Here template *List* is declared in *List.h*, but the definitions of its members can be found in *List_impls.c*. The compiler never generates *@def* directives automatically.

User-specified type information is given in *nmapXXX* files (such as *nmap037*), while automatically generated information is in *defmap* ones. *nmapXXX* files are considered in alphabetical order.

Auto-generated name mapping files have only the basename of the header file. User-specified ones may have arbitrary pathnames, though the standard *-I* convention is followed and recommended for ease of moving application code.

User-specified map files have precedence over the auto-generated one, if the application basenames match at type lookup time. The exact order of type lookup is:

- Try to find matching type/basename in user-specified files.
- Look for a default *@dec* entry in user-specified files.
- Try to find matching type/basename in auto-generated map file.
- Look for a default *@dec* in auto-generated map file.
- If not a template type, generate a forward declaration.
- If a template type *T*, assume it is declared in *T.h* and defined in *T.c*. Issue a warning.

Link–Time Actions

At link time, the name mapping file must be up–to–date and available to guide instantiation. Before linking, the compiler needs to determine what, if any, instantiations are necessary, and generate them as object files to be fed into the actual link. If the repository is empty or does not exist, no pre–linking is done, and the non–template case therefore incurs no extra processing time.

The first step in the process is to determine what external symbols are unresolved in the application. Standard UNIX™ linking calls for object files to be unconditionally linked, while object files in an archive are linked only if symbols from them are needed. It is therefore not enough to scan all objects and archives looking for unresolved symbols, since that would result in instantiations being done for symbols that are never linked. One solution to the problem is to actually link the program and see what symbols show up as undefined. This is expensive.

So, instead of linking, we simulate the link, by running – on all objects and archives, and then performing the symbol table actions such as would be done in a real link. This approach is much faster than linking, because only the symbol tables of objects and archives must be read; no relocation is done.

Partitioning Symbols and Granularity

The output of the process so far is a set of symbols that are unresolved and for which we must provide instantiations. We screen the set to eliminate non–template symbols, which are assumed to be defined somewhere else. Function templates are encoded just like C++ functions, and so are considered for instantiation only if found in the map files.

Given a list of symbols, we pick apart the symbol names, which encode the template name and argument types, and partition the list according to a desired level of granularity.

By default, all needed symbols for a given template class are instantiated into a single object file. Alternatively, one can specify that each member function for a template class be instantiated in its own object file, with all virtual tables and virtual functions for the class being instantiated into a single object file as well.

Instantiating all symbols in a single object file results in much shorter instantiation times, especially if there is extensive template use in an application. By contrast, instantiating each member function in a separate object file makes it easier to build libraries that can be shared, but can easily cause instantiations that involve hundreds of compiles, especially when the repository is populated for the first time.

Creating the Instantiation File

For each partition of the symbols we create the *instantiation file* with an internally generated name. It is a file that includes all the headers necessary to fully define a template class or function. That is, the file will *#include* the template declaration file, then the template definition file, and then the argument declaration files, in the order that the arguments appear from left to right. (Exception: no file is included more than once). For example, given the name mapping file:

```
@dec Vector app4
<Vector.h>
@def Vector app4
<Vector2.c>
@dec A app4
<A.h>
@dec B app4
<A.h>
```

and an external unresolved symbol

```
Vector<B>::f(int,B)
```

we would come up with a file:

```
#include <Vector.h>
#include <Vector2.c>
```

```
#include <A.h>
```

If the type is not found in the name mapping files, it is assumed to be an incomplete type used as the base type of a pointer or reference. For example, for the template class *Vector<A*>*, with *A* not previously seen, the instantiation file would be:

```
#include <Vector.h>
#include <Vector2.c>
struct A;
```

Naming the Instantiation File

Object files stored in the repository are the result of instantiations that are done for particular combinations of templates and arguments. Each object file has a checksum file stored with it. Depending on naming conventions used, an object file and its associated file containing the checksum will have names like:

```
Vector__pt__4_i1A.o
Vector__pt__4_i1A.cs
```

or:

```
pt06717175.o
pt06717175.cs
```

for a template class *Vector<int,A>*.

The first case is used when the operating system supports long names, while the latter convention is used for ones that do not; a hash code of the encoded template class name is used.

Dependency Management

When a template definition or declaration file, or an argument declaration file, is changed, the instantiations that used these files become out of date. We must have some way of detecting this case and redoing the instantiation. The alternative of recompiling all such files at each link is too expensive. On the other hand, a naive scheme, such as not recompiling any instantiation file for which an object already exists, is prone to mysterious errors if an instantiation becomes out of date and is not detected.

For each instantiation file there is a set of unresolved symbols that came from the partitioning described above. If there is no object file in the repository for this instantiation file, or if that object file does not define all the symbols that are needed, the instantiation file is recompiled, with the result replacing the old version in the repository.

If, on the other hand, there is an object file that defines all needed symbols, we then have to decide whether it is up to date, that is, whether any of the header files it included when it was made have changed. To detect changes, a header cache scheme is used. The preprocessor is run on the instantiation file to compute a list of all headers upon which the instantiation depends. The list is stored in the repository with the instantiation, and updated as needed when headers change. The instantiation object is out of date if its timestamp is older than any of the headers upon which it depends.

To determine whether the instantiation defines all the needed symbols, a sorted list of symbols (that will be used for directed instantiation) is included in the checksum file.

Directed Instantiation

If the dependency analysis determines that recompilation is necessary, the instantiation file is compiled in directed mode, with user−specified *−I* and *−D* options to name header file directories. The C++ compiler is passed the instantiation file as a normal source file, along with a list of symbols to be instantiated. No code is generated for other symbols; in a typical case the resulting object file will contain a subset of the functions defined for a particular template class.

Growing Instantiation Files

When using class−level granularity (all the needed members of a class template are in a single object

When using class−level granularity (all the needed members of a class template are in a single object file), a situation can arise where a subset of the members is initially instantiated, but a later iteration of the process expands the set of members that are required. In this case, the instantiator will remove the old object file from the repository, and reinstantiate the entire object file, including the members that were in the original, plus the new members that needed to be added.

Actual Linking

After all the instantiation files have been compiled, we are left with a set of objects for them plus all the application objects and archives. Actual linking cannot yet be done, because the instantiations may have created a need for *additional* symbols. To handle such a case, the symbol tables of the object files are extracted and the whole process repeats. When the pre−linker is satisfied that all the required symbols are present, then the actual link is performed.

Multiple Repositories

To support large projects, the above scheme is extended to handle more than one repository. Team projects often have a central set of standard versions of source files and headers, with the ability for team members to override these versions with local copies of files. The multiple repository scheme is intended to support such usage.

At compile−time, only one repository is recognized. At link time, the first repository specified is considered writeable, while all others are read−only. Any default repository is ignored if a repository is explicitly named.

Type lookup in map files follows the algorithm in the section above on user overriding of name mapping files, except that the algorithm is applied in total to each repository left to right until the type is found.

For instantiation object files, the search is also left to right until an object file is found that defines all required symbols and that passes the dependency checks. If the object file is not found, the instantiation is done into the first, that is writeable, repository.

Repository Locking

A repository may be shared by several people in a team of programmers. If several of them simultaneously try to link, and instantiation takes place in the repository, chaos will result. Therefore a locking scheme using standard UNIX™ system calls is implemented, with stub functions provided for custom locking implementations. If a repository is locked other users wait in an idle loop until it is free. Only the first repository (the writeable one) is locked.

Performance

Performance at compile−time (updating the map files) is not significantly affected by this scheme. Extracting type information and updating the repository is an efficient operation.

There is substantial cost in determining which template symbols are unresolved. However, since the alternative might be several actual links, this is a reasonable expense.

Binary sizes are optimal, except for virtual functions in a template class; they are always instantiated, because there is no static way to tell if they are used. This is no different from the non−template case.

The biggest performance issue (beyond instantiation) is that of iteration at link time as instantiations create demand for more symbols. The initial scan of an application might determine that *Vector<A>::f* is needed, and when it is instantiated, it requires *Vector<A>::g*, and so on. There is not a completely clean solution to this problem, except to note that once templates and the headers describing template arguments are in a steady state, the repositories will be up to date, so a full−blown instantiation will rarely be necessary.

Limitations of Our Approach

The most obvious limitation is type name mapping. The fully automated scheme requires that a type be described in a single self−contained header or a header that includes the other headers that it needs.

The user can override the default and specify arbitrary lists of header files for a single type.

This scheme does not handle local types, that is, types defined within a function. The problem is that reestablishing the context at link time for such a type is difficult at best. There is no way to recreate the scoping and other information for the function that the type appeared in.

Finally, our approach does not handle static data defined in template definition files. For example:

```
static int x = 83;
template <class T> void Vector<T>::f() {int y = x;}
template <class T> void Vector<T>::g() {int z = x;}
```

In this example, two template functions wish to share a file static *x*. Since the functions may be split up at instantiation time, there is no way to handle this other than by making the file static into a global with some internal naming convention.

Other Approaches to Instantiation

Replaying Source Files

A variant to the above approach is replaying source files. That is, at link time determine what application source file uses a particular template symbol. Such a file must by definition have all the required type information available to it except for the template definition file. If the template definition file can be provided, the source file can be replayed, that is, all the functions and data ignored and only the types accessed.

This scheme is somewhat more flexible in its handling of types, with name mapping files no longer required. However, a different set of problems arises. One has to be able to determine the source file that the object file was created from. Another problem is performance; a source file after preprocessing may be a megabyte or more in length, and it is expensive to replay.

Preprocessor Macros and Pragmas

These manual schemes suffer from the disadvantages listed in the section above on manual instantiation.

Carrying Headers Along

One way to recover type information at link time is to carry all the required information along in the repository. That way, one does not have to worry about whether headers are available at link time. However, this approach has a very significant size cost. An alternative would be to "digest" the header and abstract its type information, but even this would result in large files.

Summary

We are aware of no template instantiation scheme that does not involve many tradeoffs. The above scheme, with its slight restrictions on header file usage, seems to offer the best compromise between ease of use, ease of implementation, portability, and performance.

Chapter 8

Template Instantiation Environment

This chapter presents detailed information on use of the template instantiation environment with C++ Release 3.0.1. It assumes familiarity with the template section of *The Annotated C++ Reference Manual* (M.A. Ellis and B. Stroustrup, Addison−Wesley, 1990) and **Chapter 7, "Automatic Template Instantiation."**

This chapter contains the following sections:

- **"Getting Started"**

- **"Coding Conventions"**
- **"Map Files"**
- **"CC Options"**
- **"Usage Scenarios"**
- **"Performance"**
- **"Streamlining the Non−template Case"**
- **"Interpreting Errors"**
- **"What Can Go Wrong"**
- **"Tools"**
- **"Examples"**

Note: This chapter is from a paper by Glen McCluskey.

Getting Started

Suppose that you want to use vectors in an application, but find that the built−in ones in C++ are too restrictive. For example, they do not grow dynamically as new elements are added.

A first attempt to define a vector template might look like this:

```
template <class T> class Vector {
      T* data;
      int size;
public:
      Vector();
      T& operator[](int);
};
```

This declaration has two private data members *data* and *size* and two public functions *operator*[] and the constructor. There is one argument *T* to the template.

Assume that the declaration of the *Vector* template is in a file *Vector.h*. Then the implementation would be in a file *Vector.c* and would look like:

```
template <class T> Vector<T>::Vector()
{
      // start off with 10 elements
      size = 10;
      data = new T[size];
}
template <class T> T& Vector<T>::operator[](int n)
{
      int os;
      int i;
      T* newdata;
      // grow if have to
      if (n >= size) {
          os = size;
          while (size <= n)
              size *= 2;
          newdata = new T[size];
          for (i = 0; i < os; i++)
              newdata[i] = data[i];
          delete data;
          data = newdata;
      }
      // return reference to data slot
      return data[n];
}
```

Note that the implementation looks much like regular C++ code. The one difference is that the code is *parameterized*, that is, the implementation is relative to a type that is unknown but represented by *T*.

Finally, there is an application that uses this template:

```
#include <stream.h>
#include "Vector.h"
main()
{
      Vector<int> v;
      int i;
      // put data into vector
      for (i = 1; i <= 15; i++)
          v[i] = i * i;
      // pull it back out
      for (i = 1; i <= 15; i++)
          cout << i << "  " << v[i] << "\n";
}
```

Note that the *Vector* template has a type int substituted for the *T* we saw earlier. Note also that the vector *v* can be used transparently, without having to worry about increasing its size.

To compile this application, one would say:

```
$ CC appl.c
```

After *appl.c* is compiled, the CC compiler will go on to create an object file for the template class (template plus particular arguments) *Vector<int>*. The object will contain the members *Vector<int>:: Vector*() and *Vector<int>::operator*[](*int*). This process is known as *instantiation*.

To get an idea of the actions the compiler took, you can look in the *repository*, the area where the compiler stored the object file:

```
$ ls ./ptrepository
```

Besides the object file, there will be a *.c* (instantiation file), a *.cs* (checksum file), and the name mapping file *defmap*.

Coding Conventions

This section gives recommendations about how an application's files should be structured to make best use of templates.

Argument Declaration Files

An argument declaration file is used to declare types used as arguments to a template. For example, for the template class *Vector<A,B**>*, A and B are the underlying argument types. Fundamental types require no special declarations; for example, the types *int* or unsigned short*.

An argument type should be declared in one header file that is either self−contained or that includes other headers that it needs. If this is not possible then a map file needs to be written (see **"Map Files"** below). It is acceptable to have several types defined in one header. An example of a self−contained header would be:

```
#ifndef INCL_A
#define INCL_A
class A {
      int x;
public:
      void f();
      void g() {}
};
```

```
#endif
```

while one with other includes might look like:

```
#ifndef INCL_A
#define INCL_A
#include "Point.h"
class A {
      Point p[10];
public:
      void rotate(int);
};
#endif
```

INCL_A is an *include guard*, used to prevent the same file from being included more than once. Use of include guards is strongly recommended when writing template header files.

The C++ compiler extracts type information from headers and remembers it so that the instantiation process can get it back when needed. If a type has not been previously seen it is possible to use it only as a pointer or reference type, for example *Vector<A*,B&>*.

Template Declaration Files

A template declaration file is used to declare a template. It is like a class declaration in that the function and data members are laid out in the normal way. For example, a declaration file could contain:

```
template <class T> class AAA {
      T x;
      int y;
public:
      void f();
      void g(T&);
};
```

For function templates a forward declaration is used:

```
template <class T> void sort(T*, int n);
```

Function template external names resulting from instantiation are encoded the same as C++ functions. As such, a map file entry is the only way the instantiation system knows that an unresolved symbol might represent a template needing expansion, and the map file entry is created when a forward declaration is seen.

The template declaration file is the only template header file included by the user application; template definition files discussed in the next section are automatically included at instantiation time.

Like argument declaration files, a template declaration file should include header files it needs for types it uses. However, headers for types used as template arguments or the definition of the template itself should *not* be included, since these are handled automatically by the instantiation system.

Template Definition Files

The template definition file contains the implementation of a template. By convention, it is assumed that the definition file has the same name as the template declaration file, but with *.c* substituted for *.h* or *.l1* in the name. The −pte option may be used to change the assumed suffix. (See the CC man page).

This convention can be overridden by map files (see **"Map Files"** below). If the example of the previous section was declared in *AAA.h*, then the definition file would be *AAA.c*, and would look like:

```
template <class T> void AAA<T>::f() { /* ... */ }
template <class T> void AAA<T>::g(T&) { /* ... */ }
```

A definition file should not include the declaration file that matches it or the argument files that declare any template argument types. However, if include guards are consistently used, such inclusion is harmless. Including a guarded template definition file in a template declaration file will cause the definition file to be type−checked at application compile−time, at the expense of a slower compile.

There must be a definition file for each declaration one, or else a map file written to override the standard convention. For a template type *T*, the declaration and definition files *T.h* and *T.c* are assumed if the files cannot be otherwise found. If a template definition file does not exist along the *−I* path, a warning is given and the file not included. All other missing files will cause a preprocessor error at instantiation time.

Inline Functions

Inline template member functions are treated similarly to their class counterparts. An inline can be declared and defined in the template declaration:

```
template <class T> struct A {
      void f() { /* ... */ }
};
```

or in the definition file:

```
template <class T> inline void A<T>::f() { /* ... */ }
```

The *inline* keyword is mandatory if the inline is defined outside the template declaration; if not used, the member will not be inlined. In release 3.0, the inline must be defined in the template class body.

Types Defined in Application Sources

The instantiation system is best suited for types defined in header rather than source files. If you want to define types in source files, it is best to do it all or none. That is, fully define all templates and template argument types in one source file, or else define them all in headers.

Data and Functions in Template Files

Template declaration and definition files *are* header files, and should be treated as such. This means, for example, that a function or data item defined in such a header will probably be laid down in several instantiation files, leading to linker conflicts.

A static function (local to the file) will be duplicated in each instantiation. This is functionally correct but wasteful of space. Static data items, as in:

```
static int x;
template <class T> int A<T>::f()
{
      int y = x++;
      return y;
}
```

are not supported, in part because it is not clear what such usage really means. The template definition file may be used several times, and the *x* variable is no longer local to the file. Static class members are a better choice in this case.

Summary of Coding Conventions

- Headers should be guarded against multiple inclusion.
- A header describing a type should be self−contained or include with include guards other headers that it needs.
- The name of a template definition file should be the same as the template declaration file, with *.h* replaced by *.c*.
- Function templates need a forward declaration.

All the following restrictions can be gotten around by use of map files.

Map Files

A map file can be used to overcome the restrictions noted in the previous section. The default map file is *defmap* in the repository, with *defmap.old* the previous version that is created every time the map file is rewritten. User−specified files start with the string *nmap* and are also placed in the repository. User−specified files take precedence over the default and are considered in alphabetical order. For example, *nmap001* is looked at before *nmap2*.

A map file entry might look like this:

```
@dec Vector app1 app2
<Vector2.h>
@def Vector app1 app2
<Vector2a.c>
<Vector2b.c>
```

This says that type *Vector* is declared in *Vector2.h* and defined (implemented) in *Vector2a.c* and *Vector2b.c*; the headers have standard *#include* semantics as specified by the type of quotes on the header name. The type is valid for the application files *app1* and *app2* (whether in source or object form, for example *app1.c* or *app2.o*).

Application files are recorded to handle the case where there are distinct applications sharing one repository. It is also possible to have map file entries with no application files specified; these entries serve as a last resort if the type cannot otherwise be found. Long lists of application names can be continued onto multiple lines by using the backslash character at the end of the line.

If only one of the *@dec* and @def entries for a type is specified in the file, then standard naming rules are used to derive the other header names. For example, without the *@def* entry above, the instantiation system would infer that *Vector* is defined in *Vector2.c*.

When the instantiation system writes the default map file, it compresses it by using a string table at the top of the file:

```
@tab
app1
app2
app3
app4
@etab
@dec A @0 @2
<A.h>
```

and then references application names using the notation *@nnn*. This notation is not required; you can spell out the application names.

In map files, operator function templates are encoded as described on page 125 of *The Annotated C++ Reference Manual*. For example, *operator<<* comes out as *__ls*. Note that function template types are recorded without argument information. This effectively coalesces overloaded functions into a single map file entry.

CC Options

The template instantiation system adds several options to *CC*. These are specified on the *CC* line or by setting the environment variable *PTOPTS*. For example, to permanently enable verbose mode, you would say:

```
export PTOPTS=−ptv (SysV)
setenv PTOPTS −ptv (BSD)
```

−pta says to instantiate a whole template class rather than only those members that are needed. There are performance issues around this discussed below.

−ptn changes the default instantiation behavior for one−file programs to that of larger programs, where instantiation is broken out separately and the repository updated. One−file programs normally have instantiation optimized so that instantiation is done into the application object itself. The process is described below in the section on usage scenarios for simple programs.

−ptrpathname specifies a repository, with *./ptrepository* the default. If several repositories are given, only the first is writeable, and the default repository is ignored unless explicitly named.

−pts causes instantiations to be split into separate object files, with one function per object (including overloaded functions), and all class static data and virtual functions grouped into a single object. In release 3.0, *−pts* and *−pta* cannot be used together, that is, *−pts* can only be used to split up *needed* functions rather than *all* functions.

−ptt was used in Release 3.0 to alter dependency checking. It is now (Release 3.0.1 and later) an obsolete option.

−ptv turns on verbose or verify mode, which displays each phase of instantiation as it occurs, together with the elapsed time in seconds that phase took to complete. Use of this option is recommended if you are new to templates. With verbose mode, the reason an instantiation is done and the exact *CC* command used are displayed.

The preprocessor directives *−I* and *−D* work as they normally do, but must also be specified at link time, to pick up the various template and application type header files.

+i is not changed from previous usage. It causes the instantiation system to leave *..c* files in the repository.

Usage Scenarios

This section describes how the instantiation scheme can be used for different types of projects.

Simple Programs

By default, a one−file program that is to be compiled and linked (no *−c* option) causes the C++ compiler to instantiate everything it can into the object file for the program. This means that the link−time instantiation system is bypassed, if all templates and argument types are found within the program itself. This behavior can be disabled via the *−ptn* option, that is, the instantiation system will kick in even for simple programs.

Small and Medium Projects

A small project often operates out of one directory and with a single developer. Suppose that such a project wanted to use some templates from a directory of template headers */usr/local/template/incl*. This would be done by saying:

```
$ CC −I/usr/local/template/incl −c file1.c
```

and at link time:

```
$ CC −I/usr/local/template/incl file1.o file2.o −o prog
```

with the *−I* directive required for the instantiation mechanism. The repository used here would be the default *./ptrepository*.

If there is more than one project in a directory, it is better to use an explicitly−named repository:

```
$ mkdir rep1
$ CC −I/usr/local/template/incl −ptrrep1 −c file1.c
```

as a means of better separating one project from another.

Repository Permissions

When the default repository is created, it is given the same permissions as the directory it is created within. This is done with the *chmod* system call, with a fatal error given if it cannot be done. The instantiation system also tries to change the group of the created directory using *chown*, but it is not a fatal error if this cannot be done. Permissions of already existing repositories are never changed.

After possibly setting directory permissions, the file creation mode mask is set using the system call *umask*; this is done by taking the 1's complement of the directory permissions.

What all this means is that a repository will be created with the same access as its parent directory, and files that are created in the repository will reflect this access. If a repository is to be shareable it must be explicitly changed to be so:

```
$ chmod 775 ptrepository
```

or it must be implicitly created in a directory with such permissions. The instantiation system deletes files in the repository before rewriting them, so if a repository has files in it and then the repository's permissions are changed, no access problems will come up.

Another approach is for team members to set the default creation mask at the shell level:

```
$ umask 002
```

Large Projects and Multiple Repositories

A large project often has the notion of a centralized set of files (sources, objects, libraries) plus a local work area for each developer. The idea is that the developer can use a combination of files from the central area plus the local one.

The best way to model this type of development is by multiple repositories. The instantiation system looks first in the local repository and then the central one, both for map files and instantiation objects. With such a scheme, a typical compilation line would read:

```
$ CC –I/usr/jones/tincl –I/usr/proj/tincl –I/usr/jones/incl \
       –I/usr/proj/incl –ptr/usr/jones/rep –ptr/usr/proj/rep \
       –c file.c
```

Repository Management

The instantiation system adds to the repository but does not delete from it. Maintenance must be done by the user. For example, a *make* rule to completely delete the repository after a compile and link would look like:

```
appl:   appl1.o appl2.o
          CC appl1.o appl2.o –o appl
          rm –rf ./ptrepository
```

Of course, with this rule, instantiation must be repeated at every link.

Sharing Code and Use of Archives

Instantiations in a repository are simply object files, that are easily exportable into an archive. For example, with the default repository one can say:

```
$ ar cr projlib.a ./ptrepository/*.o
```

Such an archive may or may not be useful to other projects. By default, the system instantiates only what an application needs, and thus the object files will not contain all members of template classes. Another project with different needs will not be able to use such objects. Use of the *–pta* option (instantiate everything) will solve this problem at the expense of wasted binary size. A reasonable strategy might be to initially use *–pta* and turn it off later in a project cycle.

The option *–pts* may also be useful. It causes the instantiation system to split up instantiations, one function per object file. This reduces problems with object files clashing because they contain different but overlapping subsets of symbols.

Libraries

By the term *library* is meant a collection of object files, also known as an *archive*. This term is also used to denote collections of template headers, but such usage is confusing and not further discussed here.

Suppose that one has a library that uses templates, but end users of the library do not know or care about templates. How can the instantiation process be avoided for those users? The answer is to pre−instantiate or form the *closure* of the library, that is, instantiate everything into object files and add the objects to the library.

To do this for a library */usr/proj/lib.a*, one would say:

```
$ mkdir scr
$ cd scr
$ ar x /usr/proj/lib.a
$ CC −pts −I/usr/proj/tincl −I/usr/proj/incl *.o
$ rm −f /usr/proj/lib.a
$ ar cr /usr/proj/lib.a *.o ./ptrepository/*.o
```

This process will cause a link error from *CC* (because there is no *main* function) that can be ignored. *−pts* is specified so that the instantiation objects are split apart into separate files, which minimizes external symbol collisions when objects are linked with an application or another library.

Note that object file names in the repository may be longer than the 14 characters that *ar* will handle. The solution is to rename object files; a tool for this purpose is described in the section below on tools.

Dependency Checking

The instantiation system computes for each instantiation object the set of headers upon which it depends. This is done by running the preprocessor on the instantiation file. This set is called a *header cache* and is stored in the repository. It is updated when headers change.

An instantiation object is considered out of date if its timestamp is older than any of the headers in the cache for that object. This scheme is very similar to *make*, but does not require the user to express instantiation object dependencies in a makefile.

Sometimes it is desirable to get around dependency checking. To force reinstantiation, it is sufficient to simply delete all object files in the repository. The standard *make* trick of using *touch* to update the source or object modification times can also be used. For example, touching all object files in the repository will force off instantiation.

Specializations

A *specialization* is a means of overriding the standard version of a template class or a particular member of the class. This is done to get slightly different semantics or for performance. To illustrate how specializations work, consider this example:

```
// t1.h
// A template
template <class T> struct A {
      int f();
};
// t2.h
// B template
template <class T> struct B {
      int f();
      int g();
};
// t3.h
// f template
template <class T> int f(T);
// t1.c
// implementation of A
template <class T> int A<T>::f()
{
      return 37;
}
// t2.c
```

```
// implementation of B
template <class T> int B<T>::f()
{
      return 37;
}
template <class T> int B<T>::g()
{
      return 37;
}
// t3.c
// implementation of f
template <class T> int f(T t)
{
      return 83;
}
// A.h
// class specialization of A
struct A<int> {
      int f();
};
// A.c
// implementation of A<int> specialization
#include "t1.h"
#include "A.h"
int A<int>::f()
{
      return 47;
}
// B.c
// specialization of B<int>::g() - B<T>::f() unaffected
#include "t2.h"
int B<int>::g()
{
      return 47;
}
// f.c
// specialization of f
#include "t3.h"
int f(int t)
{
      return 57;
}
// main.c
#include "t1.h"
#include "A.h"
#include "t2.h"
#include "t3.h"
extern "C" void exit(int);
main()
{
      A<int> a;
      B<int> b;
      if (a.f() != 47)
          exit(1);
      b.f();
      if (b.g() != 47)
          exit(2);
      f(1234L);
```

```
        if (f(83) != 57)
            exit(3);
        exit(0);
}
```

There are three templates *A, B*, and *f*. The template class *A<int>* is completely specialized by the declaration and definition in *A.h* and *A.c*. *B.c* defines a specialization for *B<int>::g*, and *f.c* a specialization *f(int)*. Note that when a complete template class is specialized, the class must be defined before use.

To compile and link this application, one would say:

```
$ CC −c A.c
$ CC −c B.c
$ CC −c f.c
$ CC main.c A.o B.o f.o
```

The compiled specializations must be placed on the link line to prevent the general versions from being instantiated at link time. For example, if *f.o* had not been placed on the *CC* line, the specialized version of *f(int)* would not have been used.

Specialization of static template class data members is done in a similar way. A template declaration may provide a general template initializer:

```
template <class T> int A<T>::x = 97;
```

To specialize this, one would say:

```
int A<int>::x = 52;
```

somewhere in the application.

Debugging Instantiations

A debug option such as *−g* is passed by *CC* through to *ptlink* to the call to *CC* to instantiate. Debugging is therefore automatically enabled for instantiations.

One thing can go wrong with debugging, however. If a repository pathname is specified, and it is not fully qualified, then the debugger will fail to find the source files if the application binary is moved relative to the repositories used to build it. This is easily fixed by giving full pathnames for repositories. For example:

```
$ CC −ptr/usr/xxx/ptrepository appl.c
```

There may be problems with individual debuggers, however. If you set a breakpoint on a source line of a template definition file, this sets up a many−to−one relationship between the instantiations and the source. For example, the instantiation objects for *Vector<A>* and *Vector<B>* will both refer to the header file *Vector.c*. Another way of saying it is that there is no current way to debug individual instantiations.

Performance

Performance of the instantiation scheme is almost purely a function of how much instantiation is done. Manipulating map files and other housekeeping contributes little to the total cost.

The simplest way to cut down instantiation costs is to avoid instantiation. This is done by *anticipating* the need for template classes, and supplying them in archive (library) form at link time; object files could be migrated out of the repository into an archive for this purpose. This implies that such template classes change infrequently. If template classes *do* change frequently, there is an intrinsic cost to be paid in reinstantiation. As a rule of thumb, migration should be considered if there are more than about 15 – 20 objects in the repository.

Another important aspect of performance is minimizing iteration during instantiation. For example, with the default behavior of instantiating only what is needed, a member *A<int>::f* might be needed in an application. This member may in turn need *A<int>::g*, causing another iteration to occur in the instantiation loop. In general there is no way to determine in advance what the instantiation dynamics will be, short of an intimate understanding of one's application.

The compiler option *−pta* can be used to instantiate a whole template class rather than simply those members that are needed. This will prevent iteration on the class members, but may cause another type of iteration, where unused members cause a demand for members of some other template class. Some experimentation will be necessary to determine the best option to use in a given case.

Streamlining the Non−template Case

There is no easy way to tell whether an application uses templates, so the instantiation mechanism must cast its net widely. If the default repository exists and has a non−empty *defmap* file in it, the pre−linker is called.

The pre−linker in turn calls *nm* for each object and archive. If this does not turn up any unresolved template symbols, linking is then done. So in the worst non−template case *nm* is called once for each file.

The simplest way to avoid any overhead at all is to delete the default repository from the current directory.

Interpreting Errors

Errors and warnings coming from the instantiation system will have either the string:

```
CC[ptcomp]
```

or the string:

```
CC[ptlink]
```

in front of them. Other errors will be from the C++ compiler itself or the linker.

What Can Go Wrong

Timestamps on Networks of Workstations

Like *make*, dependency analysis relies on modification timestamps to determine if an object file (an instantiation) is out of date. If you have a network of workstations, it is possible that timestamps will be out of sync because file modification times may be set from different clocks on different file servers. In such a case timestamps will not compare and dependency checking will not work correctly.

There is nothing that the instantiation system can do about this problem. It must be solved by system administration. There are often schemes available to continually synchronize workstation clocks with each other.

External Name Length Limitations

The instantiation scheme examines the symbol tables of object files to get a list of template symbols used to drive the instantiation process. These symbol names must fully describe the template class used by a given function or data item. Some systems have a name length limit of 8 or 32, which will not work. Other systems have a limit of 256, which is usually adequate.

Note that typedefing a long class name or template class name to shorten it will not fix this problem, since the typedef name is expanded to the underlying types when external names are encoded.

Map File Problems

A map file contains type/header pairs for one or more programs. For example, an entry such as:

```
@dec T app1 app2
```

```
"U.h"
```

says that type *T* is defined in *U.h* and is valid for the files *app1* and *app2*.

If you have many distinct programs that use the same type names, then the default map file will become very large. This is only likely to occur if there are many programs in one directory, for example, test cases that all use the type *T*. The map file can be compressed using a string table (see **"Map Files"** above).

Another problem is slowly accumulating garbage in map files. An effort is made to delete out−of−date information when a file is recompiled, but this process is not perfect.

Violation of the One Definition Rule

Because of separate compilation, the C++ compiler will accept usage such as:

```
// file 1
struct A {};
// file 2
template <class T> struct A {};
```

even though this is a violation of the One Definition Rule. Because type mapping information is collected into one file, the instantiation system will catch many such errors. The form of the error is:

```
fatal error: type A defined twice in map files
```

Picking Up the Wrong Versions of Headers

Some source code control and configuration management systems support named versions of source files and headers, and program compilation is done with particular sets of versions of files (a configuration). Template instantiation does not cause any problems with this, but one must be sure that the same versions of files are specified via *−I* at *link* time as are given at *compile−time*.

Replaying Source Files

If a source file looks like:

```
// main.c
#include <Vector.h>
struct A {};
main()
{
      Vector<A> a;
      a.f();
}
```

and *Vector.h* does not have include guards, then it will end being included twice, once to get at the type *Vector* and once as an indirect result of including *main.c* to get at the type *A*.

The workaround for this is either to use include guards or else completely define the types in *main.c* or completely define them in header files.

Function Templates

A function template is encoded just like a C++ function. At instantiation time there is no way to tell them apart. Therefore, the instantiation system tries to instantiate function templates only if an entry is found for them in the map files. This entry will not be there unless a forward declaration:

```
template <class T> void f(T);
```

has been seen.

Another problem occurs if only a function *definition* is given in a single−file application, and then *−ptn* or *−c* is used to tell the instantiation system not to instantiate on the fly:

```
template <class T> void f(T) {}
main()
{
      f(37);
}
$ CC -ptn prog.c
```

Because there is no *declaration*, no entry is made in the map file, resulting in an unresolved global *f*(*int*) at link time. The workaround is to use a declaration or not use *−ptn*.

Finally, a problem can occur with specializations of function templates and argument matching. Suppose that there is a function template:

```
template <class T> void f(Vector<T>&);
```

and a declaration of a specialization:

```
void f(char*);
```

Further suppose that the specialization is not defined anywhere and so is found to be unresolved by the pre−linker. The pre−linker will look for *f* in the map files and find it, and will therefore conclude that it is a template to be instantiated. It will then attempt to instantiate with a *char** argument to the *f*(*Vector<T>&*) template, with disastrous results.

Static Data Member Initialization

The instantiation system considers that the tentative definition (global common) that the C++ compiler emits for each static data member of a template class represents an *undefined* external symbol that must be defined and initialized somewhere. For example:

```
template <class T> struct A {
      static int x;
};
```

by itself would result in an unresolved external.

This usage follows the C++ standard, but the C++ compiler has not enforced it up to now.

An initializer might look like:

```
template <class T> int A<T>::x = 47;
```

or:

```
int A<char*>::x = 89;
```

The first of these is a general template initializer, the second a specialization.

Type Checking of Template Members

By default, only members of a template class that are used are instantiated. Other members are not type−checked and therefore legally could contain errors. For overloaded functions, all versions of the function are instantiated so long as at least one of them is used. So if one overloaded function is called and the other contains errors, instantiation will not succeed. All virtual functions are instantiated because there is no way to tell whether they are needed.

If you use the *−pta* option, or compile a one−file program, the C++ compiler will try to instantiate all members of needed template classes, with potential errors.

nm Problems

The utility *nm* is used to extract the symbol tables from objects and archives. There is at least one case where *nm* can fail without it being a fatal error, and that is for an object file containing no functions or data. An *nm* failure therefore elicits only a warning and not an error, with the contents of the object or archive ignored.

Another obscure problem comes up if *main* is defined in an archive rather than an object file; the start up program */lib/crt0.o* is not considered in pre-linking, and thus there is no demand for *main*.

Renaming Object Files

The basename of an object file is used to validate type entries in map files. If the name changes, the type entry will be invalid unless other object files specified along with the renamed one are also found on the basename list in the map file.

The simplest solution to this problem is to write a map file with a type entry containing no list of basenames (see **"Map Files"** for a description of map files above).

Debug and Large Binaries

The instantiation system creates one object file for each template class. With some debug formats, the linker does not merge duplicated strings and other debug information occurring in several object files. This can cause a large blowup in binary size. The problem has no easy solution.

Source File Extensions

When the instantiation system is built, the extensions for source, header, and object files can be specified, with defaults of *.c*, *.h*, and *.o* respectively. Once configured, these extensions are fixed, with one exception.

The exception is the rule that case is preserved when replacing a header extension (for a template declaration file) with a source extension (for a template definition file). For example, if a template *Vector* is declared in *Vector.H*, then the definition is assumed to be in *Vector.C*.

Tools

Because the repository is a UNIX™ directory and the files in it are not special in any way, it is possible to use standard utilities in various ways to get at information. For example, consider a system that has only 14-character filenames. Hash codes are used to name files in place of complete mangled names, and it would be nice to come up with a correspondence list showing which hash code maps to what template name.

A shell script to do this is:

```
#!/bin/sh
# display the template class for each instantiation file
# in the repository
PATH=/bin:/usr/bin:/usr/ucb
pn=`basename $0`
rep=$1
if [ "$rep" = "" -o ! -d "$rep" ]
then
      echo "usage: $pn repository" 1>&2
      exit 1
fi
cd $rep
ls *.c |
while read fn
do
      n=`sed -n ’1s/^\/\* \(.*\) \*\/$/\1/p’ $fn`
      echo "$fn  -->  $n"
done
exit 0
```

Another tool can be used to package the object files in a repository into an archive, with renaming to short names for *ar*.

```
#!/bin/sh
# export contents of repository into an archive
PATH=/bin:/usr/bin:/usr/ucb
pn=`basename $0`
t=/tmp/$pn.$$
trap "rm -rf $t; exit 2" 1 2 3 15
rm -rf $t
mkdir $t
if [ $# -ne 2 -o ! -d "$1" ]
then
      echo "usage: $pn repository archive"
      exit 1
fi
n=1
for i in $1/*.o
do
      cp $i $t/${n}.o
      n=`expr $n + 1`
done
rm -f $2
ar cr $2 $t/*.o
if [ -x /bin/ranlib -o -x /usr/bin/ranlib ]
then
      ranlib $2
fi
rm -rf $t
exit 0
```

These tools are included in the distribution.

Examples

This section will walk through a few small example cases.

Single File (Hello World) Case

In the simplest case, the template definition and the application code that uses it are all in the same file:

userapp.c:

```
#include "String.h"
template <class T>
class Stack {
      T* head;
public:
      Stack() : head(0) {}
      T pop();
      void push(T&);
};
template <class T> T Stack<T>::pop()
{
/* ... */
}
template <class T> void Stack<T>::push(T& arg)
{
/* ... */
}
main()
{
```

```
      Stack<String> s;
      /* Code that uses push and pop */
}
```

The following steps describe how the instantiation works in this case. However, all this is done behind the scenes; in the normal case, the user will simply say

```
CC userapp.c
```

and the right thing will happen automatically.

1. When *userapp.c* is compiled to an object file, the references to *Stack<String>::push*(*String&*) and *Stack<String>::pop*() will be compiled as normal function calls. Since *Stack<String>::Stack*() is inline, no reference to that function is generated.
2. As a side effect of compiling *userapp.c* to an object file, the name mapping file is updated to show the declaration of templates and classes:

   Name mapping file (defmap):

   ```
   @dec String userapp
   "String.h"
   @dec Stack userapp
   "userapp.c"
   ```

3. Because a single file is being compiled and linked, instantiation will occur by default into the application file, and the repository will not be updated.

Separate Compilation

More commonly, the template will be declared in a header file, with the definitions provided in a separate file:

```
Stack.h:
template <class T>
class Stack {
      T* head;
public:
      Stack() : head(0) {}
      T pop();
      void push(T&);
};
Stack.c:
template <class T> T Stack<T>::pop()
{
/* ... */
}
template <class T> void Stack<T>::push(T& arg)
{
/* ... */
}
userapp.c:
#include "String.h"
#include "Stack.h"
main()
{
      Stack<String> s;
      /* Code that uses push and pop */
}
```

Here, the scenario is the same as before, except that the template declaration and definition will be gotten from a different place. The name mapping file after compiling *userapp.c* will be:

```
Name mapping file (defmap):
@dec String userapp
"String.h"
@dec Stack userapp
"Stack.h"
```

Since the template declaration file is *Stack.h*, the definition file will be *Stack.c*. The automatically generated instantiation file will be:

```
#include "Stack.h"  // Template decl file
#include "Stack.c"  // Template defn file
#include "String.h" // Arg decl file
```

Of course, *Stack.c* must be available along the *−I* path in order for the instantiation to succeed.

An advantage of this structure is that the template definition file (*Stack.c*) does not have to be parsed with user application code; it need only be parsed when templates are being instantiated.

The first steps in the procedure are as before, and then the pre−linker does its work.

1. At link time, the pre−linker will determine that the following functions must be instantiated:

   ```
   Stack<String>::push(String&)
   Stack<String>::pop()
   ```

2. The repository is checked for an object file that contains these instantiations. If there is one, and it is up to date with regard to its headers, we add that file to the list of files to be linked, and go to step 4.
3. If the repository does not contain an up−to−date object file with these instantiations, we must instantiate them now. Since we are using the default granularity, both members of *Stack<String>* will be instantiated into the same object file. The template declaration file (from the name mapping file) is *Stack.h*. The template definition file has the same name as the template declaration file, except that the suffix (if any) is changed to *.c*; here, the template definition file is *Stack.c*. The argument declaration file (also from the name mapping file) is *String.h*.
4. Directed instantiation is used; the compiler builds an object file that contains the definitions of *Stack<String>::push(String&)* and *Stack<String>::pop*(), plus any virtual functions in *Stack<String>*.
5. The resulting object file is put in the repository, and the pre−linker iterates (since the member functions of *Stack<String>* might themselves refer to other templates). If there are any new instantiations needed, we repeat the above process.
6. If the pre−linker is satisfied that all required object files are available, the linker is called to complete the link.

Again, remember that this all happens behind the scenes; for this simple case, users need not be aware of the template instantiation details, except that any *−I* flags passed at compile−time must also be passed at link time.

Separate Compilation, Special Case Provided At Link Time

It is legal for a special case of a template member to be discovered at link time. For example, in the previous case, suppose this additional file were provided at link time:

```
stringpop.c:
#include "String.h"
#include "Stack.h"
/* Special case version of Stack<String>::pop */
void Stack<String>::pop()
{
/* ... */
}
```

Here, the pre−linker would determine that only *Stack<String>::push*(*String&*) was required, and only that function would be instantiated. Note that this will work properly even if the general version of

Stack<String>::pop() would not compile; since we did not need the general version, we did not try to compile it.

Chapter 9
Type−Safe Linkage for C++

This chapter describes the problems involved in generating names for overloaded functions in C++ and in linking to C programs. It also discusses how these problems relate to library building. It presents a solution that provides a degree of type−safe linkage. This eliminates several classes of errors from C++ and allows libraries to be composed more freely than has hitherto been possible. Finally the current encoding scheme for C++ names is presented.

This chapter contains the following sections:

- **"Introduction"**
- **"The Original Problem"**
- **"The Original Solution"**
- **"Problems with the Original Solution"**
- **"A General Solution"**
- **"Experience"**
- **"Details"**
- **"Alternative Solutions"**
- **"Syntax Alternatives"**
- **"Conclusions"**

Note: This chapter is from a paper by Bjarne Stroustrup.

Introduction

This paper describes the type−safe linkage scheme used by the 2.1 release of C++ and the mechanism provided to allow traditional (unsafe) linkage to non−C++ functions. It describes the problems with the scheme used by previous releases, the alternative solutions considered, and the practicabilities involved in converting from the old linkage scheme to the new.

The new scheme makes the *overload* keyword redundant, simplifies the construction of tools operating on C++ object code, makes the composition of C++ libraries simpler and safer, and enables reliable detection of subtle program inconsistencies. The scheme does not involve any run−time costs and does not appear to add measurably to compile and link time.

The scheme is compatible with older C++ implementations for pure C++ programs but requires explicit specification of linkage requirements for linkage to non C++ functions.

The Original Problem

C++ allows overloading of function names; that is, two functions may have the same name provided their argument types differ sufficiently for the compiler to tell them apart. For example,

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

Naturally, these functions must have different names in the object code produced from a C++ program. This is achieved by suffixing the name the user chose with an encoding of the argument types (the *signature* of the function). Thus the names of the two *sqrt*() functions become:

```
sqrt__Fd          // the sqrt that takes a double argument
sqrt__F7complex   // the sqrt that takes a complex argument
```

Some details of the encoding scheme are described under **"The Function Name Encoding Scheme."**

When experiments along this line began five years ago it was immediately noticed that for many sets of overloaded functions there was exactly one function of that name in the standard C library. Since C does not provide function name overloading there could not be two. It was deemed essential for C++ to be able to use the C libraries without modification, recompilation, or indirection. Thus the problem became to design an overloading facility for C++ that allowed calls to C library functions such as *sqrt*() even when the name *sqrt* was overloaded in the C++ program.

The Original Solution

The solution, as used in all non−experimental C++ implementations up to now, was to let the name generated for a C++ function be the same as would be generated for a C function of the same name wherever possible. Thus *open*() gets the name *open* on systems where C doesn't modify its names on output, the name *_open* on systems where C prepends an underscore, etc.

This simple scheme clearly isn't sufficient to cope with overloaded functions. The keyword *overload* was introduced to distinguish the hard case from the easy one and also because function name overloading was considered a potentially dangerous feature that should not be accidentally or implicitly applied. In retrospect this was a mistake.

To allow linkage to C functions the rule was introduced that only the second and subsequent version of an overloaded function had their names encoded. Thus the programmer would write

```
overload sqrt;
double sqrt(double);        // sqrt
complex sqrt(complex);      // sqrt__F7complex
```

and the effect would be that the C++ compiler generated code referring to *sqrt* and *sqrt__F7complex*. This enabled a C++ programmer to use the C libraries. This trick solves the problems of name encoding, linkage to C, and protection against accidental overloading, but it is clearly a hack. Fortunately, it was only documented in the *"BUGS"* section of the C++ manual page.

Problems with the Original Solution

There are at least three problems with this scheme:

- how to name *overloaded* functions so that one may be a C function
- how to detect errors caused by inconsistent function declarations
- how to specify libraries so that several libraries can be easily used together

The overload Linkage Problem

Consider a program that uses an overloaded function *print*() to output *glob*s and *widget*s. Naturally *glob*s are defined in *glob.h* and *widget*s in *widget.h*. A user writes

```
// file1.c:
#include <glob.h>
#include <widget.h>
```

but this elicits an error message from the C++ compiler since *print*() is declared twice with different argument types. The user then modifies the program to read

```
// file1.c:
overload print;
#include <glob.h>
#include <widget.h>
```

and all is well until someone in some other part of the program writes

```
// file2.c:
overload print;
#include <widget.h>
#include <glob.h>
```

This fails to link since *file1.c*'s output refers to *print* (meaning *print(glob)*) and *print__F6widget*, whereas *file2.c*'s output refers to *print* (meaning *print*(*widget*)) and *print__F4glob*. This is of course a nuisance, but at least the program fails to link and the programmer can—after some detective work based on relatively uninformative linker error messages—fix the problem. The nastier variation of this will happen to the conscientious programmer who knows that *print*() is overloaded and inserts the appropriate *overload* declarations, but happens to use only one variation of *print*() in each of two source files:

```
// file1.c:
overload print;
#include <glob.h>
// file2.c:
overload print;
#include <widget.h>
```

The output from *file1.c* and *file2.c* now both refer to *print*. Unfortunately, in the output from file1.c *print* means *print*(*glob*) whereas *print* refers to *print*(*widget*) in the output from file2.c. One might expect linkage to fail because *print*() has been defined twice. However, on most systems this is not what happens in the important case where the definitions of *print*(*glob*) and *print*(*widget*) are placed in libraries. Then, the linker simply picks the first definition of *print*() it encounters and ignores the second. The net effect is that calls (silently) go to the wrong version of *print*(). If we are lucky, the program will fail miserably (core dump); if not, we will simply get wrong results.

The requirement that the *overload* keyword must be used explicitly and the non−uniform treatment of overloaded functions ("the first overloaded function has C linkage") is a cause of complexity in C++ compilers and in other tools that deal with C++ program text or with object code generated by a C++ compiler.

The General Linkage Problem

This problem of inconsistent linkage is a variation of the general problem that C provides only the most rudimentary facilities for ensuring consistent linkage. For example, even in ANSI C and in C++ (until now) the following example will compile and link without warning:

```
#include <stdio.h>
extern int sqrt(int);
main()
{
      printf("sqrt(%d) == %d\m",2,sqrt(2));
}
```

and produce output like this

```
sqrt(2) == 0
```

because even though the user clearly specified that an integer *sqrt*() was to be used, the C compiler/linker uses the double−precision floating−point *sqrt*() from the standard library. This problem can be handled by consistent and comprehensive use of correct and complete header files. However, that is not an easy thing to achieve reliably and is not standard practice. The traditional C and C++ compiler/linker systems do not provide the programmer with any help in detecting errors, oversights, or dangerous practices.

These linkage problems are especially nasty because they increase disproportionately with the size of programs and with the amount of library use.

Combining Libraries

The standard header *complex.h* overloads *sqrt*():

```
// complex.h:
overload sqrt;
#include <math.h>
complex sqrt(complex);
```

Some other header, *3d.h*, declares *sqrt*() without overloading it:

```
// 3d.h:
#include <math.h>
```

Now a user wants both the 3d and the complex number packages in a program:

```
#include <3d.h>
#include <complex.h>
```

Unfortunately this does not compile because of this sequence of operations:

```
double sqrt(double);    // from <3d.h>
overload sqrt;    // from <math.h> via <complex.h>
```

A function must be overloaded before its first declaration is processed. So the programmer, who really did not want to know about the internals of those headers, must reorder the #include directives to get the program to compile:

```
#include <complex.h>
#include <3d.h>
```

This will work unless *3d.h* overloads some function, say *atan*(), that *complex.h* does not. Even in that case the programmer can cope with the problem by adding sufficient *overload* declarations where *3d.h* and *complex.h* are included:

```
overload sqrt;
overload atan;
#include <3d.h>
#include <complex.h>
```

This reordering and/or adding of *overload* declarations is work that is really quite spurious and in any case irrelevant to the job the programmer is trying to do. Worse, if the extra *overload* declarations were placed in a header file the programmer has now set the scene for the users of the new package to have exactly the same problems when they try combining this new library with other libraries. It becomes tempting to overload all functions or at least to provide header files that overload all interesting functions. This again defeats any real or imagined benefits of requiring explicit *overload* declarations.

A General Solution

The overloading scheme used for C++ (until now) interacts with the traditional C linkage scheme in ways that bring out the worst in both. Overloading of function names that was introduced to provide notational convenience for programmers is becoming a noticeable source of extra work and complexity for builders and users of libraries. Either the idea of overloading is bad or else its implementation in C++ is deficient. The insecure C linkage scheme is a source of subtle and not−so−subtle errors. In summary:

- lack of type checking in the linker causes problems
- use of the *overload* keyword causes problems
- we must be able to link C++ and C program fragments

A solution to 1 is to augment the name of *every* function with an encoding of its signature. A solution to 2 is to cease to require the use of *overload* (and eventually abolish it completely). A solution to 3 is to require a C++ programmer to state explicitly when a function is supposed to have C−style linkage.

The question is whether a solution based on these three premises can be implemented without noticeable overhead and with only minimal inconvenience to C++ programmers. The ideal solution would

- require no C++ language changes
- provide type−safe linkage
- allow for simple and convenient linkage to C
- not break existing C++ code
- allow use of (ANSI style) C headers
- provide good error detection and error reporting

- be a good tool for library building
- not impose run−time overhead
- not impose compile−time overhead

We have not been able to devise a scheme that fulfills all of these criteria strictly, but the adopted scheme is a good approximation.

Type−safe C++ Linkage

First of all, every C++ function name is encoded by appending its signature. This ensures that a program will only load provided every function that is called has a definition and that the type specified at the point of call is the same as the type specified at the point of definition. For example, given:

```
f(int i) { ... }         // f__Fi
f(int i, char* j) { ... }        // f__FiPc
```

These examples will cause correct linkage:

```
extern f(int);         // f__Fi   - links to f(int)
f(1);
extern f(int,char*);         // f__FiPc - links to f(int,char*)
f(1,"asdf");
```

These examples will cause linkage errors independent of where in the program they occur because no *f*() with a suitable signature has been defined:

```
// no declaration of f() in this file
// (this is only legal in C programs)
f(1);        // f      - links to ???
extern f(char*);        // f__FPc - links to ???
f("asdf");
extern f(int ...);        // f__Fie - links to ???
f(1,"asdf");
```

One might consider extending this encoding scheme to include global variables, etc., but this does not appear to be a good idea since that would introduce at least as many problems as it would solve. For example:

```
// file1.c:
int aa = 1;
extern int bb;
//file2.c:
char* aa = "asdf";  // error: aa is declared int in file1.c
extern char* bb;    // error: bb is declared int in file1.c
```

Under the current C scheme, the double definition of *aa* will be caught and the inconsistent declarations of *bb* will not. Using an encoding scheme, the double definition of *aa* would not be caught since the difference in encoding would cause *two* differently named objects to be created—contrary to the rules of C and C++. The fact that the inconsistent declarations of *bb* would be caught by some linkers (not all) does not compensate for the incorrect linkage of *aa*. Consequently only functions are encoded using their signatures.

This linkage scheme is much safer than what is currently used for C, but it is not meant to solve all linkage problems. For example, if two libraries each provides a function *f*(*int*) as part of their public interface there is no mechanism that allows the compiler to detect that there are supposed to be two different *f*(*int*) functions. If the *.o* files are loaded together the linker will detect the error, but where a library search mechanism is employed the error may go undetected.

Note that this linking scheme simply enforces the C++ rules that every function must be declared before it is called and that every declaration of an external name in C++ must have exactly the same type.

In essence, we use the name encoding scheme to "trick" the linker into doing type checking of the separately compiled files. More comprehensive solutions can be achieved by modifying the linker to understand C++ types. For example, a linker could check the types of global data objects and might also be able to provide features for ensuring the consistency of global constants and classes. However, getting an improved linker into use is typically a hard and slow process. The scheme presented here is portable across a great range of systems and can be used immediately.

Implicit Overloading

If a function is declared twice with different argument types it is overloaded. For example:

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

is accepted without any explicit *overload* declaration. Naturally, *overload* declarations will be accepted in the foreseeable future; they are simply not necessary any more.

Does this relaxation of the C++ rules cause new problems? It does not appear to be the case. For example, originally I imagined that obvious mistakes such as

```
double sqrt(double);        // sqrt__Fd
double d = sqrt(2.3);
double sqrt(int d) { ... }  // sqrt__Fi
```

would cause hard−to−find errors. It certainly would with the traditional C linkage rules, but with type−safe linkage the program simply will not link because there is no function called *sqrt__Fd* defined anywhere. Even the standard library function will not be found because its name is *sqrt* as always.

Another imagined problem was that a call

```
f(x);
```

would suddenly change its meaning when a function became overloaded by the inclusion of a new header file containing the declaration of another function *f*(). This is not the case, because the C++ ambiguity rules ensure that the introduction of a new *f*() will either leave the meaning of f(x) unchanged (the new *f*() was unrelated to the type of *x*) or will cause a compile−time error because an ambiguity was introduced.

C Linkage

This leaves the problem of how to call a C function or a C++ function "masquerading" as a C function. To do this a programmer must state that a function has C linkage. Otherwise, a function is assumed to be a C++ function and its name is encoded. To express this an extension of the "extreme" declaration is introduced into C++:

```
extern "C" {
      double sqrt(double);    // sqrt(double) has C linkage
}
```

This linkage specification does not affect the semantics of the program using *sqrt*() but simply tells the compiler to use the C naming conventions for the name used for *sqrt*() in the object code. This means that the name of *this* *sqrt*() is *sqrt* or *_sqrt* or whatever is required by the C linkage conventions on a given system. One could even imagine a system where the C linkage rules were the type−safe C++ linkage rules as described above so that the name of *sqrt*() was *sqrt__Fd*. Linkage specifications nest, so that if we had other linkage conventions such as Pascal linkage we could write:

```
              // default: C++ linkage here
extern "C" {
              // C linkage here
      extern "Pascal" {
              // Pascal linkage here
          extern "C++" {
              // C++ linkage here
          }
              // Pascal linkage here
      }
              // C linkage here
```

```
}
              // C++ linkage here
```

Such nestings will typically only occur as the result of nested *#include* s.

The *{}* in a linkage specification does *not* introduce a new scope; the braces are simply used for grouping. This strongly resembles the use of *{}* in enumerations.

The keyword *extern* was used because it is already used to specify linkage in C and C++. Strings (for example, "*C*" and "C++") were chosen as linkage specifiers because identifiers (e.g., *C* and *Cplusplus*) would de facto introduce new keywords into the language and because a larger alphabet can be used in strings.

Naturally, only one of a set of overloaded functions can have C linkage, so the following causes a compile−time error:

```
extern "C" {
      double sqrt(double);
      complex sqrt(complex);
}
```

Note that C linkage can be used for C++ functions intended to be called from C programs as well as for C functions. In particular, it is necessary to use C linkage for C++ functions written to implement standard C library functions for use by C programs. However, using the encoded C++ name from C preserves type−safety at link time. This technique can be valuable in other languages too. I have already seen an example of the C++ scheme applied to assembly code to prevent nasty link errors for low−level routines. One might consider using this C++ linkage scheme for C also, but I suspect that the sloppy use of type information in many C programs would make that too painful.

In an "all C++" environment no linkage specifications would be needed. The linkage mechanism is intended to ease integration of C++ code into a multi−lingual system.

Caveat

One could extend this linkage specification mechanism to other languages such as Fortran, Lisp, Pascal, PL/1, etc. The way such an extension is done should be considered very carefully because one "obvious" way of doing it would be to build into a C++ compiler the full knowledge of the type structure and calling conventions of such "foreign" languages. For example, a C++ compiler *might* handle conversion of zero terminated C++ strings into Pascal strings with a length prefix at the call point of function with Pascal linkage and *might* use Fortran call by reference rules when calling a function with Fortran linkage, etc.

There are serious problems with this approach:

- The complexity and speed of a C++ compiler could be seriously affected by such extensions.
- Unless an extension is widely available, accepted programs using it will not be portable.
- Two implementations might "extend" C++ with a linkage specification to the same "foreign" language, say Fortran, in different ways so as to make identical C++ programs have subtly different effects on different implementations.

These problems are not unique to linkage issues or to this approach to linkage specification. In most cases linkage from C++ to another language is best done simply by using a common and fairly simple convention such as "C linkage" plus some standard library routines and/or rules for argument passing, format conversion, etc., to avoid building knowledge of non−standard calling conventions into C++ compilers. This ought to be simpler from C++ than from most other languages. For example, reference type arguments can be used to handle Fortran argument passing conventions in many cases and a Pascal string type with a constructor taking a C style string can trivially be written. Where extensions are unavoidable, however, C++ now provides a standard syntax for expressing them.

Experience

The first reaction to this scheme is to look for a way of handling linkage and overloading without requiring explicit linkage specifications. We have not been able to come up with a system that enabled C linkage to be implicit without serious side effects. The advantages and objections of the adopted scheme are discuss below. **"Alternative Solutions"** describes alternative schemes that were considered and rejected.

Making Linkage Specifications Invisible

One obvious advantage of this scheme is that it allows a programmer to give a set of functions C linkage with a single linkage specification without modifying the individual function declarations. This is particularly useful when standard C headers are used. Given a C header (that is, an ANSI C header with function prototypes, etc.)

```
// C header:
// C declarations
```

one can trivially modify the header for use from C++:

```
// C++ header:
extern "C" {
      // C header:
      // C declarations
}
```

This creates a C++ header that cannot be shared with C.

Sharing with C can be achieved using *#ifdef*:

```
// C and C++  header:
#ifdef __cplusplus
extern "C" {
#endif
      // C header:
      // C declarations
#ifdef __cplusplus
}
#endif
```

where *__cplusplus* is defined by every C++ compiler.

In cases where one for some reason cannot or should not modify the header itself one can use an indirection:

```
// C++ header:
extern "C" {
#include "C_header"
}
```

Fortunately, such transformations can be done by trivial programs so that most of the effort in converting C headers need not be done by hand.

It was soon discovered that even though programmers tend to scatter function declarations throughout the C++ program text, most C functions actually come from well defined C libraries for which there are—or ought to be—standard header files.

Placing all of the necessary linkage specifications in standard header files means that they are not seen by most users most of the time. Except for programmers studying the details of C library interfaces, programmers installing headers for new C libraries for C++ users, and programmers providing C++ implementations for C interfaces, the linkage specifications are invisible.

Error Handling

The linker detects errors, but reports them using the names found in the object code. This can be compensated for by adding knowledge about the C++ naming conventions to the linker or (simpler) by providing a filter for processing linker error messages. This output was produced by such a filter:

```
C++ symbol mapping:
PathListHead::~PathListHead()          __dt__12PathListHeadFv
Path_list::sepWork()                   sepWork__9Path_listFv
Path::pathnorm()                       pathnorm__4PathFv
Path::operator&(Path&)                 __ad__4PathFR4Path
Path::first()                          first__4PathFv
Path::last()                           last__4PathFv
Path::rmfirst()                        rmfirst__4PathFv
Path::rmlast()                         rmlast__4PathFv
Path::rmdots()                         rmdots__4PathFv
Path::findpath(String&)                findpath__4PathFR6String
Path::fullpath()                       fullpath__4PathFv
```

Bringing this filter into use had the curious effect of replacing the usual complaint about "ugly C++ names" with complaints that the linker didn't provide sufficient information about C functions and global data objects. The reason for presenting the encoded and unencoded names of undefined functions side by side is to help users who use tools, such as debuggers, that haven't yet been converted to understand C++ names.

A plain C debugger such as *sdb*, *dbx*, or *codeview* can be used for C++ and will correctly refer to the C++ source, but it will use the encoded names found in the object code. This can be avoided by employing a routine that "reverses" the encoding, that is, reads an encoded name and extracts information from it.

This is the same function used to write the linker output filter. The examples here are based on the name decoding routine written by Steve Brandt and used to modify the UNIX System V C debugger *sdb* into *sdb++*.

The encoding scheme is described under **"The Function Name Encoding Scheme."** A standard C++ name decoder should be generally available for use by debugger writers and others who deal directly with object code. Until such decoders are in widespread use the programmer must have at least a minimal understanding of the encoding scheme.

Upgrading Existing C++ Programs

Decorating the standard header files with the appropriate linkage specifications had two effects. First, most of the declarations scattered in the program text that were referring to C functions were either redundant (because the function had already been declared in a header) or at least potentially incorrect (because they differed from the declaration of that header file on some commonly used system). The second phenomenon observed was that every non−trivial program converted to the new linkage system contained inconsistent function declarations. A noticeable number of declarations found in the program text were plain wrong, that is, different from the ones used in the function definition. This was caused in part by sloppiness, for example, where a programmer had declared a function

```
f(int ...);
```

to shut up the compiler instead of looking up the type of the second argument. A more common problem was that the "standard" header files had changed since the function declaration was placed in the text so that the "local" declaration didn't match any more; this often happens when a file is transferred from one system to another, say from a BSD to a System V.

Introducing the new linkage system involved adding linkage specifications. Typically, these linkage specifications were only needed in standard header files. The process of introducing linkage specifications invariably revealed errors in the programs—even in programs that had been considered correct for years. The process strongly resembles trying *lint* on an old C program.

As was expected, some programmers first tried to get around the requirements for explicit C linkage by enclosing their entire program in a linkage directive. This might have been considered a fine way of converting old C++ programs with minimum effort had it not had the effect of ensuring that every program that uses facilities provided by such a program would also have to use the unsafe C linkage. To achieve the benefits from the new linkage scheme most C++ programs must use it. The requirement that at most one of a set of overloaded functions can have C linkage defeats this way of converting programs. The slightly slower and more involved method of using standard header files (already containing the necessary linkage specifications) and adding a few extra linkage specifications in local headers where needed must be used. This also has the benefit of unearthing unexpected errors.

Details

The scope of C function declarations has always been a subject for debate. In the context of C++ with linkage specifications and overloaded functions it seems prudent to answer some variations of the standard questions.

Default Linkage

Consider:

```
extern "C" {
      int f(int);
}
int f(int);          // default: f() has C++ linkage
```

Is it the same *f*() that was defined with C linkage above and does it have C or C++ linkage? It is the same *f*() and it does (still) have C linkage. The first linkage specification "wins" provided the second declaration has "only" default (that is, C++) linkage.

Where linkage is explicitly specified for a function, that specification must agree with any previous linkage. For example:

```
extern "C" {
      int f(int);     // f() has C linkage
}
int g();         // default: g() has C++ linkage
extern "C++" {
      int f(int);     // error: inconsistent linkage specification
      int g();    // fine
}
```

The reason to require agreement of explicit linkage specifications is to avoid unnecessary order dependencies. The reason to allow a second declaration with implicit C++ linkage to take on the linkage from a previous explicit linkage specification is to cope with the common case where a declaration occurs both in a *.c* file and in a standard header file.

Declarations in Different Scopes

Consider:

```
extern "C" {
      int f(int);
}
void g1()
{
      int f(int);
      f(1);
}
```

Is the *f*() declared local to *g1* the same as the global *f*() and does the function called in *g1*() have C linkage? It is the same *f*() and it does have C linkage.

Consider:

```
extern "C" {
      int f(int);
}
void g2()
{
      int f(char*);
      f(1);
```

```
      f("asdf");
}
```

Does the local declaration of *f*() overload the global *f*() or does it hide it? In other words, is the call *f*(*1*) legal? That call is an error because the local declaration introduces a new *f*(). In the tradition of C, the declaration of *f*(*char**) also draws an warning. Consider:

```
void g3()
{
      int ff(int);
};
void g4()
{
      int ff(char*);
      ff("asdf");
      ff(1);
};
```

Does the second declaration of *ff*() overload the first? In other words, is the call *ff*(*1*) legal? The call is an error and a warning is issued about the two declarations of *ff*() because (as in the example above) overloading in different scopes is considered a likely mistake.

Local Linkage Specification

Linkage specifications are *not* allowed inside function definitions. For example:

```
void g5()
{
      extern "C" {  // error: linkage specification in function
          int h();
      }
}
```

The reason for this restriction is to discourage the use of local declarations of C functions and to simplify the language rules.

Alternative Solutions

So, the linkage specification scheme works, but isn't there a better way of achieving the benefits of that scheme? Several schemes were considered. This section presents the first two or three alternatives people usually come up with and explains why we rejected them. Naturally, we also considered more and weirder solutions, but all the plausible ones were variations of the ones presented here.

The Scope Trick

The first attempt to provide type-safe linkage involved the use of *overload* and the C++ scope rules. All overloaded function names were encoded, but non-overloaded function names were not. This scheme had the benefit that the linkage rules for most functions were the C linkage rules—and had the problem that those rules are unsafe. The most obvious problem was that at first glance there is no way of linking an overloaded function to a standard C library function. This problem was handled using a "scope trick":

```
overload sqrt;
complex sqrt(complex);
inline double sqrt(double d)
{
      extern double sqrt(double);    // A completely new sqrt()
                                    // not overloaded
      return sqrt(d);               // not a recursive call
                                   // but a call of the C function
                                   // sqrt
```

```
}
```

In effect, we provided a C++ calling stub for the C function *sqrt*(). The snag is that having thus *defined sqrt*(*double*) in a standard header a user cannot provide an alternative to the standard version. The problems with library combination in the presence of *overload* are not addressed in this scheme, and are actually made worse by the proliferation of definitions of overloaded functions in header files. In particular, if two "standard" libraries each overload a function then these two libraries cannot be used together since that function will be defined twice: once in each of the two standard headers.

There is also a compile−time overhead involved. In retrospect, I consider this scheme somewhat worse than the original "the first overloaded has C linkage" scheme.

C "Storage Class"

It is clear that the definitions providing a calling stub are redundant. We could simply provide a way of stating that a member of a set of overloaded functions should be a C function. For example:

```
complex sqrt(complex);
cdecl double sqrt(double);     // sqrt(double) has C linkage
```

This is equivalent to

```
complex sqrt(complex);
extern "C" {
      double sqrt(double);
}
```

but less ugly. However, it involves complicating the C++ language with yet another keyword. Functions from other languages will have to be called too and they each have separate requirements for linkage so the logical development of this idea would eventually make *ada*, *fortran*, *lisp*, *pascal*, etc., keywords. Using a keyword also requires modification of the declarations of the C functions and those are exactly the declarations we would want *not* to touch since they will typically live in header files shared with an ANSI C compiler. In some cases we would even like not to touch a file in which such declarations reside.

Overload "Storage Class"

The use of a keyword to indicate that a function is a C function is logically very similar to the linkage specification solution, though inferior in detail. An alternative is to have a keyword indicate that a function should have its signature added. The keyword *overload* might be used. For example:

```
overload complex sqrt(complex); // use C++ linkage
double sqrt(double);            // C linkage by default
```

This has the disadvantage that the programmer has to add information to gain type safety rather than having it as default and would de facto ensure that the C++ type−safe linkage rules would only be used for overloaded functions. Furthermore, this would mean that libraries could only be combined if the designers of these libraries had decorated all the relevant functions with *overload*. This scheme also invalidates all old C++ programs without providing significant benefits.

Calling Stubs

One way of dealing with C linkage would be *not* to provide any facilities for it in the C++ language, but to require every function called to be a C++ function. To achieve this one would simply re−compile all libraries and have one version for C and another for C++. This is a lot of work, a lot of waste, and not feasible in general. In the cases where recompilation of a C program as a C++ program is not a reasonable proposition (because you don’t have the source, because you cannot get the program to compile, because you don’t have the time, because you don’t have the file space to hold the result, etc.) you can provide a small dummy C++ function to call the C function. Such a function would be written in C (for portability) or in assembler (for efficiency). For example:

```
double sqrt__Fd(d) double d;
         /* C calling stub for sqrt(double): */
```

```
{
      extern double sqrt();
      return sqrt(d);
}
```

A program can be provided to read the linker output and produce the required stubs.

This scheme has the advantage that the user works in what appears to be an "all C++" environment (but so does the adopted scheme once a few C libraries have been recompiled with C++ and/or a few header files have been decorated with linkage specifications). It does, however, also suffer from a few severe disadvantages. A "C calling stub maker" program cannot be written portably. Therefore, it would become a bottleneck for porting C++ implementations and C++ programs and thus a bottleneck for the use of C++. It is also not clear that this approach can be implemented everywhere without loss of efficiency since it requires large numbers of functions to have two names (a C name and a C++ name). This takes up code space and introduces large numbers of extra names that would slow down programs reading object files such as linkers, loaders, debuggers, etc. The C calling interfaces would also be ubiquitous and available for anyone to use by mistake, thus re-introducing the C linkage problems in a new guise.

Encode Only C++ Functions

The fundamental problem with all but the last scheme outlined above is that they require the programmer to decorate the source code with directives to help the compiler determine which functions are C functions. Ideally, the compiler would simply look at the program and determine the linkage necessary for each individual function based on its type. Could the compiler be that smart? Unfortunately, no. There is no way for the compiler to know whether

```
extern double sqrt(double);
```

is written in C or C++. However, one might handle most cases by the heuristic that if a function is clearly a C++ function it gets C++ linkage and if it isn't it gets C linkage. For example:

```
complex sqrt(complex);     // clearly C++: sqrt__F7complex
double sqrt(double);       // could be C:    sqrt
```

Since *complex* is a class, *sqrt*(*complex*) is clearly a C++ function and it is encoded. The other *sqrt*() might be C so it isn't.

Applying this heuristic would mean that most functions would not have type-safe linkage—but we are used to that. It would also mean that overloading a function based on two C types would be impossible or require special syntax:

```
int max(int,int);
double max(double,double);
```

Such overloading *must* be possible because there are many such examples and several of those are important, especially when support for both single and double-precision floating-point arithmetic becomes widespread:

```
float sqrt(float);
double sqrt(double);
```

This implies that either *overload* or linkage specifications must be introduced to handle such cases. The heuristic nature of the specification of where these directives are needed will lead to confusion, overuse, and errors.

If *overload* is re-introduced, the cautious programmer will use it systematically wherever a relatively simple class is used (in case a revision of the system should turn it into a plain C struct), wherever an argument is typedef'd (because that *typedef* might some day refer to a plain C type), and wherever there is any doubt. This will lead to the now well known problems of combining libraries. Similarly, if linkage specifications are required anywhere, they will proliferate because of doubts about where they are needed.

It does not seem wise to refrain from checking linkage in a large number of cases and to introduce a rather arbitrary heuristic into the linking rules for C++ without being able to reduce the complexity of the language or to reduce the burden on the programmer somewhere.

Nothing

Naturally, while considering these alternative schemes the easy option of doing nothing was regularly re−considered. However, the original scheme still suffers from the problems described in **"The Original Problem"** : insecure linkage, spurious *overload* declarations, and overloading rules that complicate the life of library writers and library users.

Syntax Alternatives

The scheme of giving all C++ functions type−safe linkage and providing a syntax for expressing that a given function is to have C linkage was thus chosen and tried. However, there were still several alternatives for expressing C linkage for this general scheme.

Why extern?

Instead of employing the existing keyword *extern* we might have introduced a new one such as *linkage* or *foreign*. The introduction of a new keyword always breaks some programs (though usually not in any serious way and for a well chosen new keyword not many programs) and *extern* already has the right meaning in C and C++. In almost all cases *extern* is redundant since external linkage is the default for global names and for locally declared functions. When used, *extern* simply emphasizes the fact that a name should have external linkage. The use of *extern* introduced here merely allows the programmer to tag an *extern* declaration with information of *how* that linkage is to be established.

Linkage for Individual Functions

One obvious alternative is to add the linkage specification to each individual function:

```
extern "C" double sqrt(double);   // sqrt(double) has C linkage
```

The problem with this is that it does not serve the need to be able to give a set of C functions C linkage with one declaration and requires the declaration of every C function to be modified. In particular, it does not allow a C header (that is, an ANSI C header) to be used from a C++ program in such a way that all the functions declared in it get C linkage.

This notation for linkage specification of individual functions is not just an alternative to the linkage "block" adopted but also an obvious extension to the adopted syntax. I intend to review the situation after the current scheme has been used a while longer to see if the use of linkage specifications warrants this extension.

Linkage Pragmas

The original implementation of the linkage specifications used a *#pragma* syntax:

```
#pragma linkage C
double sqrt(double);         // sqrt(double) has C linkage
#pragma linkage
```

This was considered too ugly by many but did appear to have significant advantages. For example, it can be argued that linkage to "foreign languages" is not part of the language proper. Such linkage cannot be specified once and for all in a language manual since it involves the *implementations* of *two* languages on a given system. Such implementation specific concepts are exactly what pragmas were introduced into Ada and ANSI C to handle. The *#pragma* syntax was trivial to implement and easy to read. It was also ugly enough to discourage overuse and to encourage hiding of linkage specifications in header files.

There are problems with this view, though. For example, it is most often assumed that any *#pragma* can be ignored without affecting the meaning of a program. This would not be the case with linkage pragmas. Another problem is that for the moment many C implementations do not support a pragma mechanism and it is not certain that those that do can be relied upon to "do the right thing" for linkage pragmas used by a C++ compiler.

Linkage to a particular foreign language does not belong in C++ because such linkage will in principle be local to a given system and non−portable. However, the fact that linkage to other languages occurs is a general concept that can and ought to be supported by a language intended to be used in multi−language environments. In practice, one can assume that at least C and Fortran will be available on most systems where C++ is used and that a large group of users will need to call functions written in these languages. Consequently, one would expect C++ implementations to support C and Fortran linkage.

The fact that C (like most other languages) does not provide a concept of linkage to program fragments written in other languages led to the absence of an explicit linkage mechanism in C++ and to the problems of link safety and overloading.

Special Linkage Blocks

Another approach would be to introduce a new keyword, say *linkage*, and use it to specify both the start and the end of a linkage block:

```
linkage("C");
double sqrt(double);    // sqrt(double) has C linkage
linkage("");
```

This avoids introducing yet another meaning for {}, allows setting and restoring of linkage to be two separate operations, allows all linkage directives to be found by simple pattern matching in a line oriented editor, and allows all linkage directives to be suppressed by a single macro

```
#define linkage(a)
```

The problem with this seems to be that it tempts people to think of linkage as a compiler "mode" that can be switched on and off at random times and doesn't obey block structure. For example:

```
linkage("C");
double sqrt(double); // sqrt(double) has C linkage
f() {
      extern g();       // g() has C linkage
linkage("");
      extern h();       // h() has C++ linkage
      ...
}
```

It also becomes hard to convince people that linkage specifications come in pairs and can be nested.

The same approach, with the same educational problems, can be tried without introducing a new keyword:

```
extern "C";
double sqrt(double);    // sqrt(double) has C linkage
extern "";
```

Note that whatever syntax was chosen, linkage specifications were intended to obey block structure to be fit cleanly into the language. In particular, if linkage "blocks" and ordinary blocks were not obliged to nest the job of writers of tools manipulating C++ source text, such as a C++ incremental compilation environment, would be needlessly complicated.

Conclusions

The use of function name encodings involving type signatures provides a significant improvement in link safety compared to C and earlier C++ implementations. It enables the (eventual) abolition of the redundant keyword *overload* and allows libraries to be combined more freely than before. The use of linkage specifications enables relatively painless linkage to C and eventually to other languages as well. The scheme described here appears to be better than any alternative we have been able to devise.

The Function Name Encoding Scheme

The (revised) C++ function name encoding scheme was originally designed primarily to allow the function and class names to be reliably extracted from encoded class member names. It was then modified for use for *all* C++ functions and to ensure that relatively short encodings (less than 31 characters) could be achieved reliably for systems with limitations on the length of identifiers seen by the linker. The description here is just intended to give an idea of the technique used, not as a guide for implementers.

The basic approach is to append a function's signature to the function name. The separator __ is used so a decoder could be confused by a name that contained __ except as an initial sequence, so don't use names such as *a__b__c* in a C++ program if you like your debugger and other tools to be able to decompose the generated names.

The encoding scheme is designed so that it is easy to determine

- if a name is an encoded name
- what (unencoded) name the user wrote
- what class (if any) the function is a member of
- what are the types of the function arguments

The basic types are encoded as

`void`	`v`
`char`	`c`
`short`	`s`
`int`	`i`
`long`	`l`
`float`	`f`
`double`	`d`
`long double`	`r`

A global function name is encoded by appending *__F* followed by the signature so that *f*(*int,char,double*) becomes *f__Ficd*. Since *f*() is equivalent to *f*(*void*) it becomes *f__Fv*.

Names of classes are encoded as the length of the name followed by the name itself to avoid terminators. For example, *x::f*() becomes *f__1xFv* and *rec::update*(*int*) becomes *update__3recFi*.

Type modifiers are encoded as

`unsigned`	`U`
`const`	`C`
`volatile`	`V`
`signed`	`S`

so *f*(*unsigned*) becomes *f__FUi*. If more than one modifier is used they will appear in alphabetical order so *f*(*const signed char*) becomes *f__FCSc*.

The standard modifiers are encoded as

`pointer`	`*`	`P`
`reference`	`&`	`R`
`array`	`[10]`	`A10_`
`function`	`()`	`F`
`ptr to member`	`S::*`	`M1S`

So *f*(*char**) becomes *f__FPc* and *"printf*(*const char* ...*)*"* becomes *printf__FPCce*.

To shorten encodings repeated types in an argument list are not repeated in full; rather, a reference to the first occurrence of the type in the argument list is used. For example:

```
f(complex,complex);    // f__F7complexT1
      // the second argument is of the same
      // type as argument 1
f(record,record,record,record);    // f__F6recordN31
      // the 3 arguments 2, 3, and 4 are of
      // the same type as argument 1
```

A slightly different encoding is used on systems without case distinction in linker names. On systems where the linker imposes a restriction on the length of identifiers, the last two characters in the longest legal name are replaced with a hash code for the remaining characters. For example, if a 45 character name is generated on a system with a 31 character limit, the last 16 characters are replaced by a 2 character hash code yielding a 31 character name.

Naturally, the encoding of signatures into identifier of limited length cannot be perfect since information is destroyed. However, experience shows that even truncation at 31 characters for the old and less dense encoding was sufficient to generate distinct names in real programs. Furthermore, one can often rely on the linker to detect accidental name clashes caused by the hash coding. The chance of an undetected error is orders of magnitude less than the occurrence of known problems such as C programmers accidentally choosing identical names for different objects in such a way that the problem isn't detected by the compiler or the linker.

Chapter 10

Access Rules for C++

This paper defines the C++ access rules, as they relate to the various protection methods, and explains some of the reasoning for these rules.

This chapter contains the following sections:

- **"Introduction"**
- **"Access Rules"**
- **"Explanation"**
- **"Base Member Declarations"**
- **"Examples (Not Interdependent)"**

Note: This chapter is from a paper by Phil Brown.

Introduction

One feature of C++ is the provision for function and data protection through a combination of the following:

- *public*, *protected,* and *private* class members - Every class member has an associated level of protection. *public* indicates no protection. *private* indicates access is limited to members and friends. *protected* is similar to *private* except that it allows access additionally to derived classes.
- inheritance - Derived classes are defined in terms of base classes. Inheritance is the name and description of the process by which a derived class acquires the data and functions of its base classes. The *private* members of the base classes are not accessible in the derived class. The protection of other members is dependent on the type of the derivation. *public* and *protected* members of *public* base classes will have the same protection in the derived class. These same members from a *private* base class will be *private* in the derived class.
- friendship - Friendship overrides all protections within a class. A friend declaration within a class denotes another class or function as a *potential* friend.

Denoting a class as a friend, in effect, denotes each function member of that class as a friend.

Access Rules

The following rules define when a *potential* friend is considered a friend:

1. Any visible non–class–member is accessible.
2. If an object is accessible, then:
   - *public* members of the object's class type are accessible.
   - *potential* friends of the object's class type will be considered friends.

- The same level of access applies to the *public* base classes of the object's class type.

3. All members of a class, and *public* and *protected* members of its base classes, are accessible by member and friend functions of the class.

Rules #2b and #3 can be combined to override Rule #2c.

Explanation

1. Any visible non−class−member is accessible.

   The first of the access rules is the starting point for many references. In the following:

   ```
   int i;
   void
   f() {
         i = 1;    // OK - Rule #1
   }
   ```

   the variable *i* is accessible since it is not a class member and is visible in the function *f.*

2. If an object is accessible, then:
   - *public members of the object's class type are accessible.* The first part of the second rule is a restatement of its condition. Access to *public* members of a class object is the minimal amount of accessibility (excluding *no access*).

     ```
     class B {
     public:
           int i;
     };
     void
     f() {
           B b;
           b.i = 1;    // OK - Rule #1, #2a
     }
     ```

     In this case, the variable *b* is accessible by Rule #1. Since *b* is accessible, the *public* member *i* of *"class B"* will be accessible (Rule #2a).
   - potential *friends of the object's class type will be considered friends*. One way to view this is to consider a friend declaration as a *public* member which will not be honored unless that friend declaration is accessible. Once friendship *has* been established, access is described by Rule #3.

     ```
     class B {
     private:        // unnecessary
           int i;
           friend void f();
     };
     class D : private B {
     };
     void
     f() {
           B b;
           b.i = 1;    // OK - Rule #1, #2b, #3
           D d;
           d.i = 1;    // ERROR - Rule #1, #2a, -fail-
     }
     ```

     In this example, both variables *b* and *d* are accessible according to Rule #1. However, in the first case, the function *f* is a friend of *"class B"* since, by Rule #2b, *b* is accessible and *"class B"* has a friend declaration for the function *f.* Rule #3 states that, as a friend, *f* will have access to all of the members of *"class B."* The assignment to *b.i* is thus valid. In the second case, the *public* members of *d* are accessible according to Rule #2a. Since function *f* is not a friend of *"class D"* and *"class B"* is not a *public* base class of *"class D,"* there are no other access rules to apply. The assignment to *d.i* is invalid.

- *the same level of access applies to the public base* classes of the object's class type. This rule applies when Rule #3 cannot (access is not by a member or friend). Note that there is *no access* to *private* base classes.

```
class B {
public:
int i;
};
class D : public B {
private:    // unnecessary
int j;
};
void
f() {
D d;
d.i = 1;    // OK - Rule #1, #2a, #2c
d.j = 1;    // ERROR - Rule #1, #2a, -fail-
}
```

  In this example, the variable *d* is accessible according to Rule #1. According to Rule #2a, the *public* members of *"class D"* are thus accessible. Since *j* is a *private* member of *"class D,"* it will not be accessible. However, by Rule #2c, since *"class B"* is a *public* base class of *"class D,"* the *public* members of *"class B"* will also be accessible. The assignment to *d.i* is valid.

3. *All members of a class, and public and protected members of its base classes, are accessible by member and friend functions of the class.* (self–explanatory)

The reasoning for the rules as they apply to inheritance is illustrated by **Figure 10–1** below.

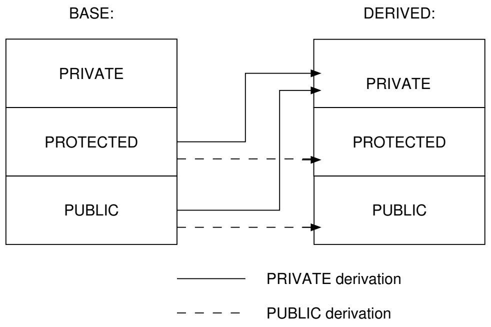

Figure 10–1 Derivation Relationship

This diagram shows the level of protection of a base class member when referenced through a derived class. As indicated in Rule #3, since friends and members of the derived class have access to all members of the derived class, they will also have access to the *public* and *protected* members of any base class.

When neither a friend nor member of the derived class, access to base class members will be determined by the type of derivation. If it is a *private* derivation, the base class members will be *private* in the derived class. As such, the base class members will not be accessible. However, in a *public* derivation, the same level of access will apply for base class members as applies within the derived class.

Base Member Declarations

public and *protected* base member declarations in a derived class (of the form *base_class::member)* can be used to alter the accessibility of class members. When given in a *private* derived class, a base member declaration will make the designated base member appear to be a member of the derived class.

A *public* base member declaration must appear in a *public* section of the derived class. Similar logic applies to the *protected* case.

Thus, accessibility of the member will be determined at the level of the derived class.

A superfluous base member declaration (i.e., one given in a *public* derived class) is ignored. This is necessary since an inaccessible base member declaration can conceivably hide a validly accessible base member.

```
class A {
protected:
      int i;
      friend f();
};
class B : public A {
protected:
      A::i;
};
void f() {
      B* p; p->i = 1; // This would be illegal if the base
                      // member declaration was not ignored
}
```

Examples (Not Interdependent)

```
//------ start of example 01 ------
class B {
      int i;
      friend void f();
};
class D : public B {
};
void
f() {
      B* p = new B;
      D* q = new D;
      int fi1 = p->i;     // OK - Rule #1, #2b, #3
      int fi2 = q->i;     // OK - Rule #1, #2a, #2c, #2b, #3
}
//------ start of example 02 ------
class B {
      int i;
};
class D : public B {
};
void
f() {
      B* p = new B;
      D* q = new D;
      int fi1 = p->i;     // ERROR - Rule #1, #2a, -fail-
      int fi2 = q->i;     // ERROR - Rule #1, #2a, #2c, -fail-
}
//------ start of example 03 ------
class B {
      int i;
      friend C;
};
```

```
class C : private B {
      friend D;
      void f1() {
      int fi1 = i;    // OK - Rule #3, #2b, #3
      }
};
class D : public C {
      void f2() {
      int fi2 = i;    // ERROR - Rule #3, #2b, #3, -fail-
      }
};
//------ start of example 04 ------
class B {
      int i;
      friend D;
};
class C : private B {
};
class D : public C {
      void f() {
      int fi1 = i;    // ERROR - Rule #3, -fail-
      }
};
//------ start of example 05 ------
class B {
      int i;
      friend D;
};
class C : public B {
};
class D : private C {
      void f() {
      int fi1 = i;    // OK - Rule #3, #2c, #2b, #3
      }
};
//------ start of example 06 ------
class B {
      int i;
      friend D;
};
class D {
      void f() {
      B* p = new B;
      int fi1 = p->i;    // OK - Rule #1, #2b, #3
      }
};
//------ start of example 07 ------
class B {
protected:
      int a;
};
class D : public B {
      friend void f();
public:
      int b;
};
void
f() {
      D* p;
```

```
      p->a = 1;     // OK - Rule #1, #2b, #3
      p->b = 2;     // OK - Rule #1, #2a

      B* pp;
      pp->a = 1;     // ERROR - Rule #1, #2a, -fail-
      pp->b = 1;     // ERROR - Rule #1, #2a, -fail-

      pp = p;
      pp->a = 1;     // ERROR - Rule #1, #2a, -fail-
      pp->b = 2;     // ERROR - Rule #1, #2a, -fail-
}
//------ start of example 08 ------
class A {
protected:
      int a;
};
class B : public A {
};
class C : public B {
      void f(B* p);
};
void
C::f(B* p) {
      a = 1;     // OK - Rule #3, #2c
      p->a = 2;     // ERROR - Rule #1, #2a, #2c, -fail-
}
//------ start of example 09 ------
class A {
      int a;
      friend void f();
};
class B : public A {
};
void
f() {
      B* p;
      p->a = 1;     // OK - Rule #1, #2a, #2c, #2b, #3
      A* p2;
      p2->a = 2;     // OK - Rule #1, #2b, #3
}
//------ start of example 10 ------
class B {
      friend void f1();
public:
      int a;
};
class C : private B {
      friend void f2();
};
class D : public C {
};
void
f1() {
      D* p1;
      p1->a = 1;     // ERROR - Rule #1, #2a, #2c, -fail-
}
void
f2() {
```

```
      D* p2;
      p2->a = 1;    // OK - Rule #1, #2a, #2c, #2b, #3
}
//------ start of example 11 ------
class B {
      friend void f1();
      int a;
};
class C : private B {
      friend void f2();
};
class D : public C {
};
void
f1() {
      D* p1;
      p1->a = 1;    // ERROR - Rule #1, #2a, #2c, -fail-
}
void
f2() {
      D* p2;
      p2->a = 1;    // ERROR - Rule #1, #2a, #2c, #2b, #3, -fail-
}
//------ start of example 12 ------
class B {
      friend void f1();
public:
      int a;
};
class C : public B {
      friend void f2();
};
class D : public C {
};
void
f1() {
      D* p1;
      p1->a = 1;    // OK - Rule #1, #2a, #2c, #2c
}
void
f2() {
      D* p2;
      p2->a = 1;    // OK - Rule #1, #2a, #2c, #2b, #3
}
//------ start of example 13 ------
class B {
      friend void f1();
      int a;
};
class C : public B {
      friend void f2();
};
class D : public C {
};
void
f1() {
      D* p1;
      p1->a = 1;    // OK - Rule #1, #2a, #2c, #2c, #2b, #3
```

```
}
void
f2() {
      D* p2;
      p2->a = 1;    // ERROR - Rule #1, #2a, #2c, #2b, #3, -fail-
}
//---------------------------------
```

Chapter 11

Inline Functions in C++

Inline functions in C++ permit the programmer to increase the execution time efficiency of a program. There are some places, however, where inline functions should not be used—for functions that the compiler is incapable of expanding inline.

This chapter contains the following sections:

- **"Introduction"**
- **"Regular Inline Functions"**
- **"Inline Member Functions"**
- **"Inline Constructors and Destructors"**
- **"Examples of Inline Function Errors and Warnings"**
- **"How to Find Inline Functions That Are Not Expanded"**
- **"Summary"**

Note: This chapter is from a paper by Dennis Mancl.

Introduction

Inline functions are a useful feature in the C++ programming language. When they are used judiciously, the run−time efficiency of the programs where they are used can be improved considerably. But not all uses of inline functions will cause programs to run faster. The purpose of this chapter is to describe the proper use of C++ inline functions and to point out some of the pitfalls in their use.

This paper discusses the implementation of inline functions in the AT&T C++ Language System Release 2.1. Other C++ compilation systems exist, and these have subtle differences with the information presented here.

Regular Inline Functions

C++ functions are declared inline by putting the keyword *inline* before the definition of the function. For example:

```
// example program inline1.c:
inline int min(int a, int b)
{
      return ((a < b) ? a : b);
}
```

Inline functions increase the execution speed of a program by getting rid of the normal subroutine call overhead. In a program calling a normal function, the actual code for the function is generated by the compiler in one place, with a return from subroutine instruction at the end. Each call to the function is translated into instructions that place the function arguments on the stack, place the return address on the stack, jump to the start of the subroutine, and clean up the stack after the return.

Inline functions improve execution speed by eliminating the steps of stacking the arguments, jumping to the subroutine, and cleaning up the stack. Instead, the function's code is repeated in each place where the function is called. This may create a very large program if an inline function is large and is called in many places in a program—the use of inline functions is a space−time trade−off. The inline function feature is used most often for very simple functions.

In order to make a decision on whether a function should be inline or not, five pieces of information must be estimated:

1. the object code size of the function
2. the object code size of the stacking, jumping, and cleanup code
3. the execution time of the stacking, jumping, and cleanup code
4. the number of places in the program where the function is called
5. the average number of times the function is actually executed in a single run of the program

Item 1 can be determined easily by compiling the function separately. Items 2 and 3 are more difficult to determine—it is necessary to look at the assembly code generated by the C compiler. Item 4 can be counted directly from the code, and item 5 can be found by running the program with a profiler such as *prof, lprof,* or *gprof.* With this information, the cost in space and the benefit in time can be determined. (Space cost = [(1) – (2)] * (4), time savings = (3) * (5). Notice that for very small functions, the space cost can be negative, that is, certain inline functions save both time and space.) In real life this kind of formal analysis is rarely performed—but programmers use their own intuitive estimates of these quantities to decide where *inline* should be used.

Many normal functions can be made into inline functions. According to the definition of the C++ *inline* keyword, *inline* is treated as a "hint" to the compiler—the compiler may choose to ignore the hint. See the *AT&T C++ Language System Release 2.1 Product Reference Manual,* page 36, for details about the *inline* keyword. In the AT&T C++ Language System some circumstances prevent inline expansion—for instance, loops are not allowed in an inline function. A more complete (but not totally exhaustive) list of the translator's expansion rules for inline functions appears below. In some cases where the translator can't expand a function, it produces an error message and the function must be rewritten. In other cases, the translator gives a warning message indicating that the function will not be expanded inline, but the function will be converted to a static function—this means that no execution speed improvement will occur and there may be a possible program size penalty if the inline function is included in multiple source files. Finally, there are cases where the translator produces no warning message, but a static copy of the inline function is added to the object file.

Inline Member Functions

There are two different ways of defining C++ inline functions that are member functions of a class. The first and most common way is to include the definition of the function in the declaration of the class. In this kind of inline function definition, the keyword *inline* is optional.

```
// example program inline2.c:
class example {
      int hrs;
      int mins;
public:
      void settime(int hval, int mval) { hrs = hval; mins =
                                                   minval;}
};
```

The second way uses the *inline* keyword before the definition of a member function outside of the class declaration.

```
// example program inline3.c:
class example {
      int hrs;
      int mins;
public:
      int seconds_after_midnight();
};
inline int example::seconds_after_midnight() {
```

```
      return (60 * (mins + 60*hrs));
}
```

Both methods of defining inline member functions are okay, but the second is very useful in the case when it is necessary to relocate the definition of an inline function to another part of the file in order to eliminate forward references to other inline functions. (But the function definition can't be moved from the *.h* file containing the definition of the class to a single *.c* file, because all modules using the function must contain the definition of the function.)

Inline Constructors and Destructors

Constructors and destructors can also be inline functions. Inline constructors or destructors cause the space−time trade−off analysis to be more complicated—for two reasons. First, it is more difficult to count the number of places in the program where a constructor or a destructor is called than to count the number of calls to a normal function. Constructor functions are called for declarations, conversions, and invocations of the *new* operator, and destructor function calls are at the end of a block that contains a declaration of a class object or where the *delete* operator is invoked. Second, the code generated by an inline constructor or destructor function in a derived class will contain the code from the inline constructor or destructor in the base class, if the corresponding base class constructor or destructor is also an inline function.

Inline constructors and destructors are very useful in reducing the execution time for initialization and cleanup of class objects. Estimating the cost in space is easier to do by experimentation than by analysis.

Examples of Inline Function Errors and Warnings

When an inline function is declared, it is not always possible to do an inline expansion of the function. An inline function that cannot be expanded is converted to a "static" function—a normal non−inline function whose scope is limited to the *.c* file in which it is contained.

If an inline function cannot be expanded, as is the case in some of the examples below, the function can have a large negative impact on the amount of space used by a program. Most inline function definitions are found in header files (*.h* files) which define variables and function prototypes for many different parts of a larger C++ program. In the compilation process, the C++ compilation system will put a duplicate static copy of the inline function in many different object files.

In an extreme case, in a C++ program that consists of 100 *.c* files, each of which *#include s* the header file defining an inline function, there is a possibility that 100 static copies of the inline function may appear in the final executable file. In a couple of the examples given below (example programs *inline9.c* and *inline11.c*), a static copy of an inline function is produced even though there is no inline function invoked in the program. These "non−inline inline" constructs should be avoided. The following is a list of some examples where inline functions are converted to static functions:

- An inline function definition contains a loop.

  ```
  // example program inline4.c:

  inline char *find_next(char *string, char objective)
  {
        while (*string && (*string != objective)) string++;
        return ((*string == objective) ? string : NULL);
  }
  ```

  In this case, the translator will produce the following warning message (in C++ Release 2.1, only if the *+w* option is used).

  ```
  warning: "inline" ignored,  find_next() contains loop
  ```

  It will proceed to generate code—in this case, the function is converted to a static function.
- An inline function contains multiple returns.

```
// example program inline5.c:

#include <string.h>
inline char *my_strcpy(char *dest, char *src)
{
      if (dest == 0) return (0);
      return (strcpy(dest, src));
}
```

This program causes a translator error message:

```
sorry, not implemented: cannot expand inline function my_strcpy() with
 statement after "return."
```

This problem can be sidestepped by making a single return.

```
// example program inline6.c:

#include <string.h>
inline char *my_strcpy(char *dest, char *src)
{
      char *retval;
      if (dest == 0) retval = 0;
      else retval = strcpy(dest, src);
      return (retval);
}
```

- An inline function contains a local array variable.

```
// example program inline7.c:

#include <string.h>
inline void basename(char *filename) {
      char basename_buf[15];
      char *p;
      if ((p = strrchr(filename, '/')) != 0) {
          strncpy(basename_buf, p + 1, 15);
          basename_buf[14] = '\0';
          strcpy(filename, basename_buf);
      }
}
```

This program causes a translator error message:

```
sorry, not implemented: cannot expand inline function needing temporary
 variable of vector type at the place where the inline function is called.
```

A similar message would be produced if the local array were declared "static char *basename_buf[15]*". In order to make this function work, the array needs to be made static and it must be moved outside of the function definition.

- An inline function is defined recursively.

```
// example program inline8.c:

inline int factorial(int n) {
      return ((n == 0) ? 1 : (n * factorial(n - 1)));
}
```

This example compiles with no warning messages from the translator (unless the *+w* option is used). In fact, any call to this function is expanded inline, but the recursive call within the function definition is not expanded. For example, the statements

```
int n = 5;
m = factorial(n);
```

are translated into

```
int n = 5;
m = ((n == 0) ? 1 : (n * factorial(n - 1)));
```

- There are nested calls to an inline function in another inline function.

```
// example program inline9.c:

inline int min(int a, int b)
{
      return ((a < b) ? a : b);
}
inline int min3(int a, int b, int c)
{
      return (min(a, min(b, c))); // An inline function that
            // makes nested calls to another inline function
}
```

In this example, the function *min*() becomes a static function. This occurs even if there is no call to the *min3*() function anywhere in the program. The translator only gives a warning message to indicate that a static copy of *min*() has been created if the *+w* option is present. The translator will only expand an inline function once in an expression. This can be sidestepped by splitting this into two expressions:

```
// example program inline10.c:

inline int min3(int a, int b, int c)
{
      int temp = min(b, c);    // this works okay -- no static
      return (min(a, temp));    // copy of min() is created
}
```

- An inline function is called by a previously defined inline function.
- This case can only happen within a class definition, because in any other case, the compiler gives an error message for a forward reference from an inline function to another inline function.

```
// example program inline11.c:

class example {
      int hrs;
      int mins;
public:
      void roundup() { clearmins(); hrs++; };
          // roundup() contains a forward reference
          // to the inline function clearmins()
      void clearmins() { mins = 0; }
};
```

The function *clearmins*() is called before it is defined. In this example, the function *clearmins*() becomes a static function. This occurs even if there is no call to the *roundup*() function anywhere in the program. The translator gives a warning message to indicate that a static copy of *clearmins*() has been created only if the *+w* option is present.

This problem can be sidestepped by changing the order of the definitions of the functions *roundup* () and *clearmins*().

- The address of an inline function is needed.

```
// example program inline12.c:

inline int min(int a, int b)
{
      return ((a < b) ? a : b);
}
```

```
extern void do_function(int (*)(int, int));
main() {
      do_function(&min);
}
```

A static copy of the inline function is added to the file so that the function pointer can point to a real function. Normal calls to the *min*() function are expanded inline—only calls through the function pointer will call the static copy of the function.

This program and the next two are examples of the static copy of the inline function being triggered by the use of the function rather than by the definition of the function.

- An inline function is also virtual.

  This is a result of the implementation of virtual functions. The virtual function table must have a pointer to an actual function, so a static copy of the function is created. The static copy of the function is usually added to only one file—that's the file where the translator puts the virtual function table, which is usually the file that contains the definition of the first non–inline virtual function.

```
// example program inline13.c:
class baseclass {
      int data;
public:
      virtual inline void clrdata() { data = 0; };
};
main() {
      baseclass b, *baseptr;
      b.clrdata();    // inline only in C++ 2.0 or 2.1
          //    (not in C++ 1.2)
      baseptr->clrdata();    // never inline
      baseptr->baseclass::clrdata();    // always inline
}
```

  The "virtual inline" function definition is sometimes useful, because of the execution speed efficiency of inline expansion in the cases where the function can be expanded inline. The choice of whether to make a virtual function into an inline virtual function is based on the same space–time tradeoffs as with normal functions, but in the analysis it is necessary to remember that function calls that use the "virtual" mechanism (as is the case in the example program above: (*baseptr–>clrdata*()) never get the benefit of inline expansion.

How to Find Inline Functions That Are Not Expanded

In a C++ *library,* a collection of C++ classes and functions that will be used by many programmers in many places, a simple check should be made for functions that are not expanded inline. One way to find the "non–inline" inlines is to compile the library with the *+w* option. This option will give warnings about all of the examples given in the previous section. Another way to find functions that are being converted to static functions by the C++ compilation system is to look at the symbol table of a test program. (The *nm (1)* command prints a list of the names in the symbol table of a *.o* file.) If the symbol table contains the names of "inline" functions, then those functions are being converted to static functions, because inline functions normally don't turn up in the symbol table.

Here's an example.

```
// example program inline14.h:
#include <memory.h>
class SimpleStr {
      char *stringbuffer;
      int length;
public:
      SimpleStr() { stringbuffer = 0; }
      SimpleStr(int str_length) {
          stringbuffer = new char[str_length];
```

```
        length = str_length;
        clrbuf();
    }
    void clrbuf() { memset(stringbuffer, 0, length); }
};
```

In order to test this code to determine if there is a "non−inline" inline function, a simple *.c* has to be compiled:

```
// example program inline14.c:
#include "inline14.h"
SimpleStr s;
```

The "non−inline" inline functions can be found by applying the following commands to the object file:

```
nm inline14.o | grep "static" | grep "\.text" | c++filt
```

Running this set of commands produces:

```
SimpleString::clrbuf() | 52|static|    |    |    |.text
```

C++ symbol mapping demangled:

```
SimpleString::clrbuf()
```

mangled:

```
_clrbuf__12SimpleStringFv
```

A more complicated tool could be written to search the source file for inline function definitions and to search the object file symbol table for the matching function name. A more complete test program would include calling each of the inline functions.

Summary

In conclusion, the inline function construct in the C++ programming language is a useful feature. It needs to be used selectively—keeping in mind the space−time tradeoffs in using them and the places where inline definitions add no execution speed improvement to the program.

Chapter 12

As Close as Possible to C—But No Closer

The purpose of this chapter is to summarize the remaining differences between the ANSI C standard and C++, explain their motivation, and point out cases where these differences are less important than they might appear at first.

This chapter contains the following sections:

- **"Introduction"**
- **"Name Spaces and Nesting"**
- **"Linkage"**
- **"Linkage Consistency"**

Note: This chapter is from a paper by Andrew Koenig and Bjarne Stroustrup.

Introduction

ANSI C and the C subset of C++ serve subtly different purposes.

The purpose of ANSI C is to provide a standard: to codify existing practice and resolve inconsistencies among existing implementations. The purpose of C++ is to provide C programmers with a tool they can use to shape their thinking in fundamentally different ways. Both aimed at compatibility with "Classic C" and both came close to hitting their mark.

The two goals have necessarily resulted in some fundamental differences of approach between the two languages. In a few cases, C++ departed slightly from "Classic C"— always with knowledge of the cost of doing so and always with the aim to gain something well worth that cost. X3J11 did the same according to its aims and constraints. Wherever possible, C++ has adopted the X3J11 modifications and resolutions and in a few noteworthy cases ANSI C adopted C++ features.

The purpose of this chapter is to summarize the remaining differences between the ANSI C standard and C++, explain their motivation, and point out cases where these differences are less important than they might appear at first.

Below, C refers to C as defined by the December 7, 1988 ANSI C standard and C++ refers to C++ as defined in the February 1990 draft *C++ Reference Manual.* We use the word "difference" to refer to something that is C but not C++. Things that can be done in C++ but not C are not interesting in this context unless they also somehow restrict C++ from expressing something that is C.

Note that in this context pure extensions of C provided by C++ are *not* incompatibilities.

Name Spaces and Nesting

Structure Name Spaces

C puts variables and structure tags in separate name spaces; C++ uses a single name space. The reason for this, of course, is that abstract data types – classes are a crucial part of the foundation of C++ and it is important to be able to use them as naturally as if they were built–in types. Essentially every C++ program depends on this.

The place where it matters most – at least the place where people have complained the most – is when a library function deals with a structure with the same name. For instance:

```
struct stat {
      /* member declarations */
};
int stat(const char * , struct stat *);
```

The C++ language definition therefore allows precisely this kind of thing. We believe this will smoothly accommodate most existing C usages while still allowing the economical expression C++ programmers have come to appreciate and depend on.

The only remaining difference between C and C++ in the name space area is that C++ does not allow a name to be declared as both a structure tag and a (different) *typedef* name in the same scope. For example:

```
struct stat {
      /* member declarations */
};
typedef int stat;
```

Allowing this construct in C++ would create serious problems with composition of header files describing libraries since declarations of functions, variables, and classes then could undetectably change meaning as the result of header file inclusions. Note that C++ specifically does allow the following common C usage:

```
typedef struct A {
      /* member declarations */
} A;
```

Structure Nesting

In both C and C++, a structure may be declared inside another:

```
struct Outer {
      int a;
      int b;
      struct Inner {
          int x;
          int y;
      } c;
      int d;
};
```

In C, this is merely a notational convenience: the declaration above is precisely equivalent to:

```
struct Inner {
      int x;
      int y;
};
struct Outer {
      int a;
      int b;
      struct Inner c;
      int d;
};
```

In C++, structures are classes, classes can have member functions, and member functions obey the same scope rules as any other functions. As a result, it is necessary for classes to obey the normal scope rules as well, so that in the first example above, the name *Inner* is only directly visible inside the body of the *Outer* class declaration and its associated member functions. For example:

```
struct Outer {
      int a;
      int b;
      struct Inner {
          int x;
          int y;
      } c;
      int d;
};
void f()
{
      Outer limits;          /* legal C and C++ */
      Inner sanctum;          /* legal C, illegal C++ */
      Outer::Inner view;          /* legal C++, illegal C */
}
```

This difference should not cause trouble in practice because a considerate C++ implementation will detect invalid C++ usages that are valid C, issue an appropriate warning, and use the C treatment. Of course, good programming practice argues against *declaring* a name like *Inner* in a way that makes it look nested and then *using* it in a way that requires that it be externally visible.

Similarly, the C++ notion that a class establishes a scope is useful for declaring enumerations. They are kept in the scope of their class and accessible elsewhere – subject to access control – by explicit qualification with their class name:

```
class X {
public:
      enum state { good, bad, fail };
      state readstate();
      // ...
private:
      some_operation() { state s = good; /* ... */ }
      // ...
```

```
};
void f(X& x)
{
      while (x.readstate() == X::good) // ...
}
```

Linkage

The major difference here is that C allows

```
extern void f();
/* no declaration of g */
main()
{
      f(3,4);     /* legal C, illegal C++ */
      g(5,6);     /* legal C, illegal C++ */
}
```

and C++ does not.

In C++, a function prototype with no arguments means that the function has no arguments, and it is an error to call it with arguments. The ANSI C standard lists this (mis)use of function declarations as obsolescent. We think C++ is an excellent place to institute this disappearance.

To declare a function *f*() with no arguments in a way that unambiguously means the same thing in both C and C++, say something like this:

```
int getcount(void);
```

Furthermore, in C++, a prototype is required for *any* call; an undeclared function cannot be called. This is absolutely fundamental to the type safety of C++; C requires a prototype only for functions, such as *printf* (), that accept variable numbers of arguments.

Linkage Consistency

C++ requires the types of all objects and functions with external linkage to be consistent across separate compilations – and enforces this requirement. This implies that all prototypes for a function must agree (exactly) and that any structure tag used in the type of any object or function with external linkage must refer to the same structure (with the same name) in all files.

We are not convinced that this requirement is actually different from C but we are convinced that actual C practice is such that enforcing the requirement would be unacceptable because it would break too much code.

C++ relies on name equivalence; that is, two types are considered the same if and only if they have the same name – and there must be exactly one definition of a given type in given scope. ANSI C allows certain examples of structural equivalence; that is, two types are in a few cases considered the same provided their declarations are sufficiently similar. For example:

```
// file1:
struct S { int a, int b; };
extern int f(struct S);
// file2:
int f(struct { int a, int b; } arg) { /* ... */ };
```

This example is ANSI C because the declarations of the structures used as the argument type for *f*() are sufficiently similar. In C++, the *f*() in *file2* is not considered a definition of the *f*() declared in *file1* because they have different argument types.

Linkage of const

Global variables declared *const* have external linkage by default in C; in C++ such *consts* have internal linkage by default. The reason for this is to avoid having to allocate memory for things that, in our experience, are usually intended as the equivalent of preprocessor macros and to allow systematic use of integer *consts* in constant expressions. For example:

```
const SIZE = 100;
int table[SIZE];    /* legal C++, illegal C */
```

Again this is not as much of a problem as one might expect. It makes no difference, of course, if a constant is only defined and used in a single file or if the *const* is local. If the same constant is defined in several files, the programmer will have to declare it *static* anyway to avoid multiple definition errors from the linker or declare it *extern* in all files but one. All uses of *const* where explicit *static* or *extern* is used are compatible as are all uses of local *consts.* The only case to look out for is:

```
file1:
      const a = 1;
file2:
      extern const a;
```

which will not link in C++ because no definition will be found for the *a* referenced in *file2.* The following modification makes the C program acceptable to a C++ compiler:

```
file1:
      extern const a = 1;
file2:
      extern const a;
```

Keywords

C++ has a few extra keywords: *asm*, *catch*, *class*, *delete*, *friend*, *inline*, *new*, *operator*, *private*, *protected*, *public*, *template*, *this*, *try*, and *virtual*. This can't be helped; one cannot add fundamental concepts to a language in a reasonable way without introducing words to refer to those concepts. To avoid chaos, such words must be reserved in languages such as C and C++.

Miscellaneous

The differences mentioned above are the most important because they affect the interface between C and C++ programs and limit – if only insignificantly – what can be expressed in header files shared by the the two languages. The relative insignificance of these limitations can be seen from the fact that all the ANSI C standard library header files are also legal C++ header files.

Other differences are limited to individual source files and are less important since no significant C++ program can pass a C compiler anyway.

Assignment of void

C not only allows any object pointer to be assigned to a *void** but also a *void** to be assigned to any object pointer. This opens a blatant hole in the type system that was not present in classic C.

We surmise that the reason this is considered acceptable is that C already has a worse hole in its type system in the form of unchecked function arguments and because it provides the C programmer the convenience of not having to cast the results of calling *malloc*(), *calloc*(), etc. Opening this hole is not acceptable in C++ where reliance on the type system is greater. In C++, only the harmless assignment of any object pointer to a *void** (and not the opposite assignment) is accepted. Furthermore, since C++ programmers use operator *new* in preference to using *malloc*(), etc., directly, the hole in the type system would provide no extra convenience in C++.

In a similar vein, C++ does not allow a pointer to a *const* object to be assigned to *void*:* the program must use a *const void** instead. This is to catch things like:

```
extern "C" {
      int read(int, void, int);
      int write(int, const void, int);
```

```
}
const char filename[] = "/etc/passwd";
void f()
{
      read(0, filename, sizeof(filename));   // read int a const?
}
```

If an arbitrary *const* pointer could be freely assigned to *void*,* there would be no way for the compiler to detect that this program fragment tries to read into a constant array.

Of course a C++ program can use a cast to convert a *void** to or from any other kind of pointer.

Type of ’a’

The type of character constants in C++ is *char* instead of *int.* However, since the rules for determining the integer value of a character constant are identical in C and C++ the only way to detect this in a C program is with an expression like *sizeof(’a’).* In C++, however, it is essential for the overloaded function resolution mechanism to resolve *’a’* as *a char* so that

```
cout << ’a’;
```

can print *a* instead of *97.*

For the same reason, the type of an enumerator is the type of its enumeration. This *may* lead to an incompatibility since, given

```
enum e { A, B };
```

sizeof(*A*)*==sizeof*(*enum e*) and *sizeof*(*enum e*) are not guaranteed to be equal to *sizeof*(*int*) in either C or C++.

Repeated Definition

In C++, a "plain" global object declaration (without *extern* or an initializer) is a definition. Two of those in a file give a double definition error. For example:

```
int i;
int i;
```

In C, this is accepted. The reason for the difference is again the uniform treatment of built−in and user−defined types. Suppose *Int* is a class for which a constructor taking no arguments has been declared:

```
Int i;
Int i;
```

Each declaration requires a call of the constructor and each places that call in the sequence of such calls to be executed for the file. Deciding that only one of the declarations is "real" and ignoring the other not only adds complexity to C++ compilers and/or linkers but can also introduce dependency errors into the dynamic initialization.

Character Array Initialization

For some reason C has come to allow the previously illegal initialization

```
char v[3] = "asd";
```

Allowing this violates the rule that strings are terminated by *’\0’* so C++ still rejects it. The way to initialize a bounded character array that is not intended to be used as a string is to initialize the individual elements:

```
char v[3] = { ’a’, ’s’, ’d’ };
```

goto Skipping Initialization

The following is legal C, but not C++:

```
void f()
{
      /* ... */
      goto ll;
      {
          int a = 7;
          String b = "asdf";
      ll:
          /* ... */
      }
      /* ... */
}
```

In C, this is merely dangerous and bad style. In C++, it would with great regularity cause core dumps; that innocuous looking *String* might be a class for which a destructor will be called on exit from *f*().

enum Assignment

In C, an *int* may be assigned to a variable of an enumeration type. For example:

```
enum e { A, B };
enum e obj1 = 7;
enum e obj2 = 257; /* what if an e is represented by a char? */
```

C leaves the meaning of many such assignments implementation dependent and the ANSI C manual recommends a warning against all such assignments. C++ prohibits them. As usual, casting can defeat type checking:

```
enum e obj3 = (enum e) 7;    /* caveat emptor */
```

Comments

It was believed that the introduction of *//* comments in C++ did not lead to any incompatibilities. Here is a counter example:

```
main()
{
      int a = 4;
      int b = 8//* divide by a*/a;
      +a;
}
```

Note that the use of a prefix operator starting the line is essential, as is the absence of whitespace. We do not consider this incompatibility serious enough to abandon either style of comments.

We note in passing that it is important to cater to *//* comments in the preprocessor too, lest the following evoke surprising preprocessor complaints:

```
#include <stdio.h>
int flag;    // remember if we have called getc()
```

Conclusion

We have tried to summarize the differences between ANSI C and C++. In the design of C++, we have been trying to keep the differences as minor as possible. We believe that we have succeeded in this beyond reasonable expectations and that differences that remain are unimportant to C programmers, essential to C++ programmers, and stem from the somewhat different purposes behind C and C++.

Glossary

archive

A collection of object files typically grouped using the *ar* command. At link time, only object files that have needed symbols are extracted from the archive.

argument declaration file

A file containing the declaration of a class, struct, union, or enum type.

basename

The part of a UNIX™ pathname after the last slash (/).

class template

A template that describes a family of types.

directed mode

A compiler mode that is used to generate an object file with certain specified template symbols defined in it.

external symbol

A name of a function or data item in an object file that is available to other object files to link against.

function template

A template that describes a family of functions.

header cache

A file in the repository used to store the list of headers needed by each instantiation.

instantiate

To form an instantiation by binding a template to particular argument types.

instantiation

A generated class or function that is the result of binding a template to particular argument types.

instantiation file

A source file that is compiled in directed mode to produce an object file instantiation.

library

Same as *archive*.

name mapping file

A text file that describes which header files in an application define particular class, enum, and template types.

pre–linker

A tool that examines unresolved template references in object files and archives and creates an additional set of object files that resolve those references; the additional objects are fed into the link step.

repository

A directory that stores instantiations as object files, together with name mapping files.

specialization

An instantiation of a template class or function template that overrides the standard version.

template

A skeleton or description for a family of types or functions.

template argument

A type or constant specified to a template to distinguish a particular usage of the template.

template class

An instantiation of a class template for particular argument types.

template declaration file
A header file that declares a template interface.

template definition file
An implementation of a template's functions.

template function
An instantiated function template.

timestamp
The date and time a file was last changed.