12

Objects and Run-Time Efficiency: C++

C++ is an object-oriented extension of the C language. It was originally called *C with classes*, with the name C++ originating around 1984. A Bell Laboratories researcher interested in simulation, Bjarne Stroustrup began the C++ project in the early 1980s. His goal was to add objects and classes to C, using his experience with Simula as the basis for the design. The design and implementation of C++ was originally a one-person effort, with no apparent intent to produce a commercial product. However, as interest in objects and program structure grew over the course of the 1980s, C++ became popular and widely used. In the 1990s, C++ became the most widely used object-oriented language, with good compilers and development environments available for the Macintosh, PC, and Unix-based workstations.

C, discussed in Section 5.2, was originally designed by Dennis Ritchie. The C programming language was used for writing the Unix operating system at Bell Laboratories. The original implementation of C++ was a preprocessor that converted C++ to C.

12.1 DESIGN GOALS AND CONSTRAINTS

The main goal of C++ is to provide object-oriented features in a C-based language without compromising the efficiency of C. In the process of adding objects to C, some other improvements were also made. In more detail, the main design goals of C++ may be summarized as follows:

- data abstraction and object-oriented features,
- better static type checking,
- backwards compatibility with C. In other words, most C code should compile as legal C++, without requiring significant changes to the code,
- efficiency of compiled code, according to the principle "If you do not use a feature, you should not pay for it."

The principle stated in the final goal is significant and may require some thought to appreciate. This principle suggests that C programs should compile as efficiently under the C++ compiler as under the C compiler. It would violate this principle to

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

BJARNE STROUSTRUP

Bjarne Stroustrup is the principal designer of C++. Bjarne came to Bell Laboratories in 1979 after finishing his M.S. degree in Aarhus, Denmark, and his Ph.D. on design of distributed systems in Cambridge, U.K.

Working on distributed computing, in the same Computing Science Center as C and Unix designers Ritchie and Thompson, Stroustrup decided to add object-oriented features from Simula to C. His goal was to further his personal research objectives, which involved simulation.

An occasionally reserved but essentially gregarious and friendly person, Stroustrup was catapulted into the technological limelight by the success of C++, first within AT&T and later in the wider community of practicing software developers. He has written several books on C++, including *The Design and Evolution of C++* (Addison-Wesley, 1994), which contains an interesting history, explanation, and commentary on the language. Bjarne's nonresearch interests include reading books that are not about computer science, photography, hiking and running, travel, and music.

implement C integers as objects, for example, and use dynamic method lookup to find integer functions at run time as in Smalltalk, as this would significantly reduce the performance of C integer calculations. The principle does not mean that C++ statements that also appear in C must be implemented in exactly the same way in both languages, but whatever changes are adopted in C++, they must not slow down execution of compiled code unless some slower features of C++ are also used in the program.

12.1.1 Compatibility with C

The decision to maintain compatibility with C has a pervasive effect on the design of C++. Those familiar with C know that C has a specific machine model, revealing much of the structure of the underlying computer architecture. In particular, C operations that return the address of a variable and place any bit pattern in any location make it possible for C programs to rely on the exact representation of data. Therefore C++ must adhere to the same data representations as C.

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

Most other object-oriented languages, including those designed before C++ and those designed after, use garbage collection to relieve programmers from the task of identifying inaccessible objects and deallocating the associated memory locations. However, there is no inherent reason why objects must be garbage collected. The strongest connection is that with the increased emphasis on abstraction and type correctness, it would have been consistent with other goals of C++ to offer some form of garbage collection where feasible in a form that does not affect the running time of programs that do not use garbage-collected objects. However, because there are features of C that make garbage collection extremely difficult, it would have been difficult to base C++ objects on garbage collection. Not only is garbage collection counter to the C philosophy of providing programmer control over memory, but the similarity between pointers and integers makes it effectively impossible to build a garbage collector that works on C++ programs that use pointer arithmetic or cast integers to pointers.

An important specific decision is to treat C++ objects as a generalization of C structs. This makes it necessary to allow objects to be declared and manipulated in the same way as structs. In particular, objects can be allocated in the activation records of functions or local blocks, as well as on the heap, and can be manipulated directly (i.e., not through pointers). This is one place where C++ deviates from Simula, as Simula objects can be manipulated only through pointers. More specifically, C++ allows a form of object assignment that copies one object into the space previously occupied by another, whereas most other object-oriented languages allow only pointer assignment for objects. Some aspects of C++ objects on the stack are explored in the homework exercises.

12.1.2 Success of C++

C++ is a very carefully designed language that has succeeded admirably, in spite of difficult design constraints. Measured by the number of users, C++ is without question the most successful language of the decade from its development in the mid-1980s until the release of Java in the mid-1990s. However, the design goals and backward compatibility with C do not allow much room for additional aesthetic consideration. Some aspects of C++ have become complex and difficult for many programmers to understand. On the other hand, many C programmers use the C++ compiler and appreciate the benefits of better type checking. Perhaps a fair summary of the success of C++ is that it is widely used, with most users choosing to program in some subset of the language that they feel they understand and that they find suitable for their programming tasks. In other words, C++ is a useful programming tool that allows designers to craft good object-oriented programs, but it does not enforce good programming style in the way that other language designs have attempted to do. This is intended to be a statement of fact and not a count against C++. In fact, much of its success seems attributable to the way that C++ is designed to give programmers choices and not to restrict programmers to a particular style.

There are many published style guides that advocate use of certain features of C++ and caution against the use of others. Those interested in serious C++ programming or interested in understanding how many programmers view the language may wish to visit their library or bookstore and take a look at some of the current guides.

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

12.2 OVERVIEW OF C++

Before looking at the object-oriented features of C++ in Subsection 12.2.2, we will take a quick tour of some of the additions to C that are not related to objects. Evaluation and commentary on C++ appear in Subsection 12.2.3.

12.2.1 Additions to C Not Related to Objects

There are a number of differences between C++ and C that are not related to objects. Although we are primarily interested in the C++ object system, it is worth considering a few of the more significant changes. The most interesting additions are

- type `bool`
- reference types and pass-by-reference
- user-defined overloading
- function templates
- exceptions

There are also changes in memory management calls (`new` and `delete` instead of `malloc` and `free`), changes in stream and file input and output, addition of default parameter values in function declarations, and more minor changes such as the addition of end-of-line comments and elimination of the need for `typedef` keyword with `struct`/`union`/`enum` declarations.

The first three additions, `bool`, pass-by-reference, and overloading, are discussed in the remainder of this subsection. Function templates are discussed in Subsection 9.4.1 and a general discussion of exceptions appears in Subsection 8.2.

Type `bool`

In C, the value of a logical test is an integer. For example, the *C Reference Manual* defines the comparison operator `<` and logical operator `&&` as follows:

- The operator `<` (less than) returns `1` if the first operand is less than the second and `0` otherwise.
- The `&&` operator returns `1` if both its operands are nonzero and `0` otherwise.

This makes it possible to write C statements with expressions such as

```
if ((5 < 3) + 1 && 2 ==3) { . . . }
```

which combine comparison operations and arithmetic.

To make a syntactic distinction between Booleans and integers, C++ has a separate type: `bool` with values `true` and `false`. Because of conversions, this does not completely separate integers and Booleans. In particular, integer values may be assigned to variables of type `bool`, with implicit conversion of nonzero numbers to `true` and `0` to `false`. However, the type `bool` does improve the readability of many programs by indicating that a variable or return value of a function will be used as a Boolean value rather than an integer.

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

The C++ changes can be mimicked in C by the declarations

```
typedef int bool;
#define false 0
#define true 1
```

which define the type bool and values true and false. One difference between built-in Booleans of C++ and Booleans in C is that when C++ Booleans are printed, they print as true and false, not 1 and 0.

Because of all the implicit conversions, a separate Boolean type does not help with one of the most common simple programmer mistakes in C. At least for beginners, conditional statements such as

```
if (a=b) c;
```

are often the result of typographic error; the programmer meant to write

```
if (a==b) c;
```

The reason why the first statement type checks and compiles in C is that an integer assignment a=b has type int and has the value of b. Because a C conditional statement requires an integer (not a Boolean), there is no warning associated with this statement. In most languages with bool different from integer, it would be a type error to write if (a=b) c. However, because C++ Booleans are automatically converted to integers, the statement if (a=b) c remains legal in C++.

Because the introduction of bool into C++ has very little effect in the end, you might wonder why this was done. Before a Boolean type was added, C/C++ often contained definitions of bool, true, and false by macro, as previously illustrated. However, bool could be defined in different ways, with the resulting types having slightly different semantics. For example, bool could be defined as int, unsigned int, or short int. This caused problems when libraries that use slightly different definitions were combined. Therefore it was useful to standardize code by adding a built-in Boolean type.

Reference Type and Pass-By Reference

In C, all parameters are passed by value. If you wish to modify a value that is passed as a parameter, you must pass a pointer to the value. For example, here is C code for a function that increments an integer:

```
void increment(int *p) { /* parameter is pointer to int */
 (*p)++; }               /* modify value pointed to by p */
main() {
 int k = 0;
```

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

```
 increment(&k);          /* pass address of variable k */
 k;                      /* value of k is now 1 */
 ...
}
```

In C++, it is possible to pass-by-reference, as shown here:

```
void increment(int & n) { /* parameter is reference to int */
 n++; }                   /* modify reference param */
main() {
 int k = 0;
 increment(k);           /* no need to explicitly compute address */
 k;                      /* value of k is now 1 */
 ...
}
```

The effect is similar to C's pointer argument, but the calling function does not have to provide the address of an argument and the called function does not have to dereference pointers. A mild inconvenience with C pointer arguments is that the programmer must remember whether a given function uses pass-by-value or the pointer in its parameter list. If pointer arguments are changed to C++ pass-by-reference, then no address calculation needs to be written as part of the call and a programmer using functions from a library can avoid this issue entirely.

A benefit of explicit references is sometimes called *pass-by-constant-reference.* If a function argument is not to be changed by the function, then it is possible to specify that the argument is to be a constant, as in void f (const int &x). In the body of a function with parameter x passed this way, it is illegal to assign to x.

User-Defined Overloading

As discussed in Chapter 6, overloading allows one name to be used for more than one value. In C++, it is possible to declare several functions with the same name, as long as the functions have a different number and type of parameters. For example, here is a C++ program with three types of print function, each called show:

```
#include <stdio.h>
void show(int val) {
     printf("Integer: %d\n", val);
}
void show(double val) {
     printf("Double: %lfn", val);
}
void show(char *val) {
     printf("String: %s\n", val);
}
int main( ) {
```

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

```
    show(12);
    show(3.1415);
    show("Hello World\n!");
}
```

Because the three functions have different types of arguments, the compiler can determine which function to call by the type of the actual parameter used in the function call.

C++ does not allow overloaded functions that have the same number and types of arguments but differ only in their return value because C and C++ functions can be called as statements. When a function is called as a statement, ignoring the return value of the function, the compiler would not have any way of determining which function to call.

One source of potential confusion in C++ arises from combinations of overloading and automatic conversion. In particular, if the actual parameters in a function call do not match any version of a function exactly, the compiler will try to produce a match by promoting and/or converting types. In this example code,

```
void f(int)
void f(int *)
...
f('a')
```

the call f('a') will result in f(int) instead of f(int *) because a char can be promoted to be an int. When a function is overloaded and there are many possible implicit conversions, it can be difficult for a programmer to understand which conversions and call will be used.

12.2.2 Object-Oriented Features

The most significant part of C++ is the set of object-oriented concepts added to C; these are the main concepts:

- *Classes*, which declare the type associated with all objects constructed from the class, the data members of each object, and the member functions of the class.
- *Objects*, which consist of private data and public functions for accessing the hidden data, as in other object-oriented languages.
- *Dynamic lookup*, for member functions that are declared virtual. A virtual function in a derived class (subclass) may be implemented differently from virtual functions of the same name in a base class (superclass).
- *Encapsulation*, based on programmer designations public, private, and protected that determine whether data and functions declared in a class are visible outside the class definition.
- *Inheritance*, using subclassing: One class may be defined by inheriting the data and functions declared in another. C++ allows single inheritance, in which a class

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

has a single base class (superclass) or multiple inheritance in which a class has more than one base class.
- *Subtyping*, based on subclassing. For one class to define a subtype of the type defined by another class, inheritance must be used. However, the programmer may decide whether inheritance results in a subtype or not.

This is only a brief summary; there are many more features, and it would take an entire book to explain all of the interactions between features of C++. Further description of classes, inheritance, and objects appear in Section 12.3.

C++ Terminology. Although C++ terminology differs from Smalltalk (and Java) terminology, there is a fairly close correspondence. The terms class and object are used similarly. Smalltalk instance variables are called member data and methods are called member functions. The term subclass is not usually used in connection with C++. Instead, a superclass is called a base class and a subclass a derived class.The term inheritance has the same meaning in both languages.

12.2.3 Good Decisions and Problem Areas

C++ is the result of an extensive effort involving criticism and suggestions from many experienced programmers. In many respects, the language is as good a design as possible, given the goals of adding objects and better compile-time type checking to C, without sacrificing efficiency or backwards compatibility. Some particularly successful parts of the design are

- *encapsulation:* careful attention to visibility and hiding, including *public, protected,* and *private* levels of visibility and *friend* functions and classes,
- *separation of subtyping and inheritance:* classes may have public or private base classes, giving the programmer some explicit control over the resulting subtype hierarchy,
- *templates* (described in Section 9.4),
- *exceptions* (described in Section 8.2)
- *better static type checking* than C.

There are also a number of smaller successful design decisions in C++. One example is the way the scope resolution operator (written as ::) is used in connection with inheritance and multiple inheritance to resolve ambiguities that are problematic in other languages.

Problem Areas

There are a number of aspects of C++ that programmers occasionally find difficult. Some of the main problem areas are

- *casts and conversions*, which can be complex and unpredictable in certain situations,
- *objects allocated on the stack* and other aspects of object memory management,
- *overloading*, a complex code selection mechanism in C++ that can interact unpredicatably with dynamic lookup (virtual function lookup),
- *multiple inheritance*, which is more complex in C++ than in other languages because of the way objects and virtual function tables are configured and accessed.

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

These problem areas exist not because of oversight but because the goals of C++, followed to their logical conclusion, led to a design with these properties. In other words, these problems were not the result of carelessness or inattention, but a consequence of decisions that were made with other objectives in mind. To be fair, most of them have their roots in C, not the C++ extensions to C. Nonetheless, these features might reasonably lead programmers to prefer other languages when compatibility with C and absolute efficiency of compiled code are not essential to an application.

Some programmers might also say that lack of garbage collection, or lack of a standard interface for writing concurrent programs, is a problem in C++. However, these are facilities that simply lie outside the scope of the language design.

Casts and Conversions. A cast instructs the compiler to treat an expression of one type as if it is an expression of another. For example, `(float) i` instructs the compiler to treat the variable `i` as a float, regardless of its type otherwise, and `(int *)x` causes `x` to be treated as pointer to an integer. In certain situations, C and C++ compilers perform implicit type conversions. For example, when a variable is assigned the value of an expression, the expression may be converted to the type associated with the variable, if needed. In most object-oriented languages, the automatic conversion of an object from one type to another does not change the representation of the object. For example, we have seen from the description of subtyping in Smalltalk in Chapter 11 that `Point` objects can be treated as `ColoredPoint` objects without changing the representation of objects because `ColoredPoint` objects are represented in a way that is compatible with the representation of `Point` objects. If multiple inheritance is used in C++, however, converting an object from a subtype to a supertype may require some change in the value of a pointer to the object. This happens in a way that may introduce hard-to-find bugs and forces programmers to understand the underlying representation of objects. More generally, Stroustrup himself says, "Syntactically and semantically, casts are one of the ugliest features of C and C++" in *The Design and Evolution of C++* (Addison-Wesley, 1994).

Objects Allocated on the Stack. Simula, Smalltalk, and Java allow objects to be created only on the heap, not on the run-time stack. In these other languages, objects can be accessed through pointers, but not through ordinary stack variables that contain space for an object of a certain size. Although C++ on-the-stack objects are efficiently allocated and deallocated as part of entry and exit from local blocks, there are also some disadvantages. The most glaring rough spot is the way that assignment works in combination with subtyping, truncating an object to the size that fits when an assignment is performed. This can change the behavior of an object, as eliminating some of its member data forces the compiler to change the way that virtual functions are selected. Although the C++ language definition explains what happens and why, the behavior of object assignment can be confusing to programmers.

Overloading. Overloading in itself is not a bad programming language feature, but C++ user-defined overloading can be complex. In addition, the interaction with dynamic lookup (virtual functions) can be unpredictable. Because overloading is a compile-time code selection mechanism and dynamic lookup is a run-time code selection mechanism, the two behave very differently. This causes confusion for many programmers.

Multiple Inheritance. There are some inherent complications associated with multiple inheritance that are difficult to avoid. These are discussed in Section 12.5. The C++ language design compounds these problems with a tricky object and method lookup table (vtable) format. Unfortunately, the details of the implementation of multiple inheritance seem to intrude on ordinary programming, sometimes forcing a programmer who is interested in using multiple inheritance to learn some of the implementation details. Even a programmer not using multiple inheritance may be affected if a derived class is defined using multiple inheritance.

12.3 CLASSES, INHERITANCE, AND VIRTUAL FUNCTIONS

The main object-oriented features of C++ are illustrated by a repeat of the point and colored-point classes from Chapter 11.

12.3.1 C++ Classes and Objects

We can define point objects in C++ by declaring the point class, here called Pt, and separately declaring the member functions of class Pt. The class declaration without implementation defines both the interface of point objects and the data used to implement points, but need not include the code for member functions. The declarations that appear in the Pt class are divided into three parts, public members, protected members, and the private members of the class.

For simplicity, Pt is a class of one-dimensional points, each point having only an x coordinate:

```
/* ---------- Point Class Interface -------------------*/
class Pt {
  public:
    Pt(int xv);
    Pt(Pt* pv);
    int getX();
    virtual void move(int dx);
  protected:
    void setX(int xv);
  private:
    int x;
  };
/* ---------- Declarations of Constructors --------*/
  Pt::Pt(int xv) { x = xv; }
  Pt::Pt(Pt* pv) { x = pv->x; }
/* ---------- Declarations of Member Functions --------*/
  int Pt::getX() { return x; }
  void Pt::setX(int xv) { x = xv; }
```

Constructors. A constructor is used to initialize the member data of an object when a program contains a statement or expression indicating that a new object is to

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

be created. When a new point is created, space is allocated, either on the heap or in an activation record on the stack, depending on the statement creating the object. A constructor is then called to initialize the locations allocated for the object. Constructors are declared with the same syntax as that of member functions, except that the "function" name is the same as the class name. In the point class, two constructors are declared. The result is an overloaded function `Pt`, with one constructor called if the argument is an integer and the other called if the argument is a `Pt`.

In Smalltalk terminology, constructors are "class methods," not "instance methods," as a constructor is called without referring to any object that is an instance of the class. In particular, constructors are not member functions.

Visibility. As previously mentioned, the declarations that appear in the `Pt` class are divided into three parts, public members, protected members, and private members of the class. These designations, which affect the visibility of a declaration, may be summarized as follows:

- *public:* members that are visible in any scope in which an object of the class can be created or accessed,
- *protected:* members that are visible within the class and in any derived classes,
- *private:* members visible only within the class in which these members are declared.

The `Pt` class is written following one standard visibility convention that is used by many C++ programmers. Member data is made private, so that if a programmer wishes to change the way that objects of the class are represented, this may be done without affecting the way that other classes (including derived classes) depend on this class. Members that modify private data are made protected, so that derived classes may change the value of member data, but external code is not allowed to do so. Finally, member functions that read the value of member data and provide useful operations on objects are declared public, so that any code with access to an object of this class can manipulate these objects in useful ways.

A feature of C++ that is not illustrated in this example is the *friend* designation, which is used to allow visibility to the private part of a class. A class may declare friend functions and friend classes. If the `Pt` class contained the declaration friend class `A`, then code written as part of class `A` (such as member functions of class `A`) would have access to the private part of class `Pt`. The friend mechanism is used when a pair of classes is closely related, such as matrices and vectors.

Virtual Functions. Member functions may be designated *virtual* or left *nonvirtual*, which is the default for member functions that are not preceded by the keyword virtual. If a method is virtual, it may be redefined in derived classes. Because different objects will then have different member functions of the same name, selection of virtual member functions is by dynamic lookup: There is a run-time code selection mechanism that is used to find and invoke the correct function. These extra steps make calls to virtual functions less efficient than calls to nonvirtual functions.

Nonvirtual methods may not be redefined in derived classes. As a result, calls to nonvirtual methods are compiled and executed in the same way as ordinary function calls not associated with objects or classes.

A common point of confusion is that, syntactically, a redeclaration of a nonvirtual function may appear in a derived class. However, this produces an overloaded function, with code selection done at compile time. We will discuss this phenomenon in Subsection 12.3.3.

12.3.2 C++ Derived Classes (Inheritance)

The following ColorPt class defines an extension of the one-dimensional points defined in the Pt class. As the name suggests, ColorPt objects have a color added. For simplicity, we assume that colors are represented by integers. In addition, to illustrate virtual function definition, moving a colored point causes the color to darken a little:

```
/* ---------- Color Point Class Interface -------------------*/
class ColorPt: public Pt {
  public:
   ColorPt(int xv,int cv);
   ColorPt(Pt* pv,int cv);
   ColorPt(ColorPt* cp);
   int getColor();
   virtual void darken(int tint);
   virtual void move(int dx);
  protected:
   void setColor(int cv);
  private:
   int color;
  };
/* ---------- Declarations of Constructors --------*/
  ColorPt::ColorPt(int xv,int cv)
   : Pt(xv)       /* call parent's constructor */
   { color = cv; }   /* then initialize color */
  ColorPt::ColorPt(Pt* pv,int cv): Pt(pv) { color = cv; }
/* ---------- Declarations of Member Functions --------*/
  int ColorPt::getColor() { return color; }
  void ColorPt::darken(int tint) { color += tint; }
  void ColorPt::move(int dx) {Pt::Move(dx); this->darken(1); }
  void ColorPt::setColor(int cv) { color=cv; }
```

Inheritance. The first line of code declares that the ColorPt class has Pt as a public base class; this is the meaning of the clause : public Pt following the name of the ColorPt class. If the keyword public is omitted, then the base class is called a private base class.

When a class has a base class, the class inherits all of the members of the base class. This means that ColorPt objects have all of the public, protected, and private members of the Pt class. In particular, although the ColorPt class declares only member data color, ColorPt objects also have member data x, inherited from Pt.

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

The difference between public base classes and private base classes is that, when a base class is public, then the declared (derived) class is declared to be a subtype of the base class. Otherwise, the C++ compiler does not treat the class as a subtype, even though it has all the members of the base class. This is discussed in more detail in Section 12.4.

Constructors. The `ColorPt` class has three constructors. As in the `Pt` class, the result is an overloaded function `ColorPt`, with selection between the three functions completed at compile time according to the type of arguments to the constructor.

The two constructor bodies previously discussed illustrate how a derived-class constructor may call the constructor of a base class. As with points, when a new colored point is created, space is allocated, either on the heap or in an activation record on the stack, depending on the statement creating the object, and a constructor is called to initialize the locations allocated for the object. Because the derived class has all of the data members of the base class, most derived-class constructors will call the base class constructor to initialize the inherited data members. If the base class has private data members, which is the case for `Pt`, then the only way for the derived class `ColorPt` to initialize the private members is to call the base-class constructor.

Visibility. When one class inherits from another, the members have essentially the same visibility in the derived class as in the base class. More specifically, public members of the base class become public members of the derived class, protected members of the base class are accessible in the derived class and its derived classes, which is the same visibility as if the member were declared protected in the derived class. Inherited private members exist in the derived class, but cannot be named directly in code written as part of the derived class. In particular, every `ColorPt` object has an integer member `x`, but the only way to assign to or read the value of this member is by calling the protected and public functions from the `Pt` class.

Virtual Functions. As previously mentioned, virtual functions in a base class may be redefined in a derived class. The member function move is declared virtual in the `Pt` class and redefined above for `ColorPt`. In this example, the `move` function for points just changes the x coordinate of the point, whereas the `move` function on colored points changes the x coordinate and the color. If the implementation of `ColorPt::move` were omitted, then the `move` function from `Pt` would be inherited on `ColorPt`. The implementation of virtual functions is discussed in the next subsection.

12.3.3 Virtual Functions

Dynamic lookup is used for C++ virtual functions. A virtual function `f` defined on an object `o` is called by the syntax `o.f(...)` or `p->f(...)` if `p` is a pointer to object `o`. When a virtual function is called, the code for that function is located by a sequence of run-time steps. These steps are similar to the lookup algorithm for Smalltalk, but simpler because of some optimizations that are made possible by the C++ static type system. Each object has a pointer to a data structure associated with its class, called the *virtual function table*, or *vtable* for short (sometimes written as vtbl). The relationship among an object, the vtable of its class, and code for virtual functions is shown in Figure 12.1 with points and colored points.

Virtual Function on Base Class. Suppose that `p` is a pointer to the `Pt` object in Figure 12.1. When an expression of the form `p->move(...)` is evaluated, the code

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

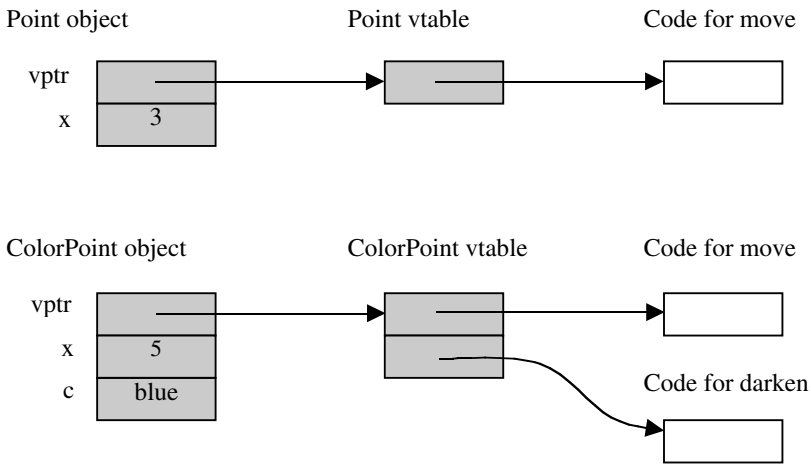


Figure 12.1. Representation of Point and ColoredPoint in C++.

for move must be found and executed. The process begins by following the vtable pointer in p to the vtable for the Pt class. The vtable for Pt is an array of pointers to functions. Because move is the first (and only) virtual function of the Pt class, the compiler can determine that the first location in this array is a pointer to the code for move. Therefore, the run-time code for finding move simply follows the first pointer in the Pt vtable and calls the function reached by this pointer. Unlike in Smalltalk, here there is no need to search the vtable at run time to determine which pointer is the one for move. The C++ type system lets the compiler determine the type of the object pointer at compile time, and this allows the compiler to find the relative position of the virtual function pointer in the vtable at compile time, eliminating the need for run-time search of the vtable.

Virtual Function on Derived Class. Suppose that cp is a pointer to the ColorPt object in Figure 12.1. When an expression of the form cp->move(. . .) is evaluated, the algorithm for finding the code for move is exactly the same as for p->move(. . .): The vtable for ColorPt is an array containing two pointers, one for move and the other for darken. The compiler can determine at compile time that cp is a pointer to a ColorPt object and that move is the first virtual function of the class, so the run-time algorithm follows the first pointer in the vtable without Smalltalk-style run-time search.

Correspondence Between Base- and Derived-Class Vtables. As a consequence of subtyping, a program may assign a colored point to a pointer to a point and call move through the base-class pointer, as in

```
p = cp;
p-> move(. . . );
```

Because p now points to a colored point, the call p->move(. . .) should call the move function from the ColorPt class. However, the compiler does not know at compile time that p will point to a ColorPt object. In fact, the same statement can be executed many times (if this is inside a loop, for example) and p may point to a Pt object on

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

some executions and a ColorPt object on others. *Therefore, the code that the compiler produces for finding* move *when* p *points to a* Pt *must also work correctly when* p *points to a* ColorPt. The important issue is that because move is first in the Pt vtable, move must also be first in the ColorPt vtable. However, the compiler can easily arrange this, as the compiler will first compile the base class Pt before compiling the derived-class ColorPt. In summary, when a derived class is compiled, the virtual functions are ordered in the same way as the base class. This makes the first part of a derived-class vtable look like a base class vtable. Because member data can be accessed by inherited functions, member data in a derived class are also arranged in the same order as member data in a base class.

Unlike in Smalltalk, here there is no link from ColorPt to Point; the derived-class vtable contains a copy of the base-class vtable. This causes vtables to be slightly larger than Smalltalk method dictionaries might be for corresponding programs, but the space cost is small compared with the savings in running time.

Note that nonvirtual functions do not appear in a vtable. Because nonvirtual functions cannot be redefined from base class to derived class, the compiler can determine the location of a nonvirtual function at compile time (just like normal function calls in C).

12.3.4 Why is C++ Lookup Simpler than Smalltalk Lookup?

At run-time, C++ lookup uses indirection through the vtable of the class, with an offset (position in the vtable) that is known at compile time. In contrast, Smalltalk method lookup does a run-time search of one or more method dictionaries. The C++ lookup procedure is much faster than the Smalltalk procedure. However, the C++ lookup procedure would not work for Smalltalk. Let's find out why.

Smalltalk has no static type system. If a Smalltalk program contains the line

```
p selector : parameters
```

sending a message to an object p, then the compiler has no compile-time information about the class of p. Because any object can be assigned to any Smalltalk pointer, p could point to any object in the system. The compiler knows that selector must refer to some method defined in the class of p, but different classes could put the same selector (method name) in different positions of their method dictionaries. As a consequence, the compiler must generate code to perform a run-time search for the method.

The C++ static type system makes all the difference. When a call such as p->move(. . .) is compiled, the compiler can determine a static type for the pointer p. This static type must be a class, and that class must declare or inherit a function called move. The compiler can examine the class hierarchy to see what location a virtual function move will occupy in the vtable for this class. A call

```
p->move(x)
```

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

compiles to the equivalent of the C code

```
(*(p->vptr[1]))(p,x)
```

where the index 1 in the array reference vptr[1] indicates that move is first function in the vtable (represented by the array vptr) for the class of p. The reason why p is passed as an argument to the function (*(p->vptr[1])) is explained in the next subsection.

Arguments to Member Functions and this

There are several issues related to calls to member function, function parameters, and the this pointer. Consider the following code, in which one virtual function calls another:

```
class A {
   public:
     virtual int f (int x);
     virtual int g(int y);
};
int A::f(int x) { . . . g(i) . . . ;}
int A::g(int y) { . . . f(j) . . . ;}
```

If virtual function f is redefined in a derived class B, then a call to the inherited g on class B objects should invoke the new function B::f, not the original function A::f defined for class A. Therefore, calls to one virtual function inside another must use a vtable. However, the call to f inside A::g does not have the form p->f(. . .), and it is not clear at first glance what object p we would use if we wanted to change the call from the simple f(. . .) that appears above to p->f(. . .).

One way of understanding the solution to this problem is to rewrite the code as it is compiled. In other words, the preceding function A::g is compiled as if it were written as

```
int A::g(A* this, int x) { . . . this->f(j) . . . ;}
```

with a new first parameter this to the function, called this, and the call f(j) replaced with this->f(j). Now the call this->f(j) can be compiled in the usual way, by use of the vtable pointer of this to find the code for f, provided that when g is called, the appropriate object is passed as the value of this.

The calling sequence for a member function, whether it is virtual or not, passes the object itself as the this pointer. For example, returning to the Pt and ColorPt example, in code such as

```
ColorPt* q = new ColorPt(3,4);
q->darken(5);
```

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

the call to darken is compiled as if it were a C function call:

```
(*(q->vptr[2]))(q,5);
```

This call shows the offset of darken in the vtable (assumed here to be 2) and the call with the pointer to q passed as the first argument to the compiled code for the member function.

There are several ways that the this pointer is used. As previously illustrated, the this pointer is used to call virtual functions on the object. The this pointer is also used to access data members of the object, whether there are virtual functions or not. The this pointer is also used to resolve overloading, as described in the next subsection.

Scope Qualifiers

Because some of the calling conventions may be confusing, it is worth saying clearly how names are interpreted in C++. There are three *scope qualifiers*. They are :: (double colons), -> (right arrow, consisting of two ASCII characters - and >), and . (period or dot). These are used to qualify a member name with a class name, a pointer to an object, or an object name, respectively.

The following rules are for resolving names:

- A name outside a function or class, not prefixed by :: and not qualified, refers to global object, function, enumerator, or type.
- Suppose C is a class, p is a pointer to an object of class C, and o is an object of class C. These might be declared as follows:

```
Class C : ... { .... }; /* C is a class */
C *p = new C( ...);  /* p is a pointer to an object of class C */
C o( ...)                    /* o is an object of class C */
```

Then the following qualified names can be formed with ::, ->, and .:

```
C::n    /* Class name C followed by member name n */
p->n    /* pointer p to object of class C followed by member name n */
o.n        /* pointer p to object of class C followed by member name n */
```

These refer to a member n of class C, or a base class of C if n is not declared in C but is inherited from a base class.

Nonvirtual and Overloaded Functions

The C++ virtual function mechanism does not affect the way that the address of a nonvirtual or overloaded function is determined by the C++ compiler. For nonvirtual functions, the C++ calls work in exactly the same way as in C, except for the way that the object itself is passed as the this pointer, as described in the preceding subsection.

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

There are also some situations in which overloading and virtual function lookup may interact or be confusing.

Recall that, if a function is not a virtual function, the compiler will know the address of the function at compile time. (Those familiar with linking may realize that this is not strictly true for separately compiled program units. However, linkers effectively make it possible for compilers to be written as if the location of a function is known at compile time.) Therefore, if a C++ program contains a call f(x), the compiler can generate code that jumps to an address associated with the function f.

If f is an overloaded function and a program contains a call f(x), then the compile-time type associated with the parameter x will be used to decide, at compile time, which function code for f will be called when this expression is executed at run time.

The difference between overloading and virtual function lookup is illustrated by the following code. Here, we have two classes, a parent and a child class, with two member functions in each class. One function, called printclass, is overloaded. The other, called printvirtual, is a virtual function that is redefined in the derived class:

```
class parent {
public:
   void printclass() {printf("parent");};
   virtual void printvirtual() {printf("parent");};
};
class child : public parent {
public:
   void printclass() {printf("child");};
   void printvirtual() {printf("child");}; };
main() {
   parent p; child c; parent *q;
   p.printclass(); p.printvirtual(); c.printclass(); c.printvirtual();
   q = &p; q->printclass(); q->printvirtual();
   q = &c; q->printclass(); q->printvirtual();
}
```

The program creates two objects, one of each class. When we invoke the member functions of each class directly through the object identifiers c and p, we get the expected output: The parent class functions print "parent" and the child class functions print "child." When we refer to the parent class object through a pointer of type *parent, then we get the same behavior. However, something else happens when we refer to the child class object through the pointer of type *parent. The call q->printclass() always causes the parent class member to be called; the printclass function is not virtual so the type of the pointer q is used. The static type of q is *parent, so overloading resolution leads to the parent class function. On the other hand, the call q->printvirtual() will invoke a virtual function. Therefore, the output of this program is

```
parent parent child child parent parent parent child.
```

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

The call q->printclass() is effectively compiled as a call printclass(q), passing the object q as the this pointer to printclass. Although the printclass function does not need the this pointer in order to print a string, the argument is used to resolve overloading. More specifically, q->printclass() calls the parent class function because this call is compiled as printclass(q) and the type of the implicit argument q is used by the compiler to choose which version of the overloaded printclass function to call.

12.4 SUBTYPING

In principle, subtyping and inheritance are independent concepts. However, subtyping as implemented in C++ occurs only when inhertance is used. In this section, we look at how subtyping-in-principle might work in C++ and compare this with the form of subtyping used by the C++ type checker. Although the C++ type checker is not as flexible as it conceivably could be, there are also some sound and subtle reasons for some central parts of the C++ design.

12.4.1 Subtyping Principles

Subtyping for Classes. The main principle of subtyping is that, if A <: B, then we should be able to use an A value when a B is required. For classes, subtyping requires a superset relation between public members of the class. For example, we can see that the colored-point class ColorPt defined in Subsection 12.3.2 is a subtype of the point class Pt defined in Subsection 12.3.1 by a comparison of the public members of the classes:

```
Pt:      int getX();
         void move(int);
ColorPt: int getX();
         int getColor();
         void move(int);
         void darken(int tint);
```

The superset relation between sets of public members is guaranteed by C++ inheritance: Because ColorPt inherits from Pt, ColorPt has all of the public members of Pt. Because Pt is a public base class of ColorPt, the C++ type checker will allow ColorPt objects to be assigned to Pt class pointers.

Subtyping for Functions. One type of function is substitutable for another if there is a certain correspondence between the types of arguments and the types of results. More specifically, if functions f and g return the same kind of result and function f can be applied to every kind of argument to which function g can be applied, then we can use function f in place of function g without type error. Similarly, if functions f and g are applicable to the same type of arguments and the result type of f is a subtype of the result type of g, then f can be used in place of g without type error. These two subtyping ideas for functions can be combined in a relatively simple-looking rule

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

that is hard for most people to understand and remember:

> The function type A → B is a subtype of C → D whenever C <: A and B <: D.

It helps to look at a few examples. Let us assume that circle <: shape and consider some function types involving circle and shape.

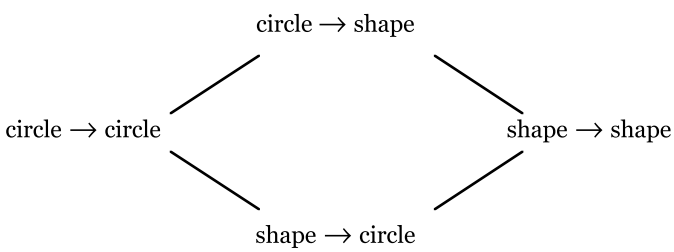
One reasonably straightforward relation is (circle → circle) <: (circle → shape): You can understand this by assuming you have a program that calls a circle → shape function f:

```
shape f (circle x) { ... return(y) ;}
main {
  ... f(circ) ...
}
```

The assertion that circle → circle is a subtype of circle → shape means that it is type safe to replace this f with another function that returns a circle instead of a shape. In other words, where a program requires an circle → shape function, we would not create a type error if we used a circle → circle function instead.

When we change the argument type of a function, the subtyping relation is reversed. For example, (shape → circle) <: (circle → circle): The reasoning involved here is similar to the case we just discussed. If a program uses a function f of type circle → circle correctly, then it may apply f to a circle argument. However, it would work just as well to use a shape → circle function g instead, as our assumption circle <: shape means that g could be applied to any circle argument without causing a type error.

When these two basic ideas are combined, there are four function types involving circle and shape. They are ordered as shown in the following illustration, in which each type shown is a subtype of those above it:



It is worth taking some time to be sure you understand all of the relationships implied by this diagram.

12.4.2 Public Base Classes

When a derived class B has public base class A, the rules of C++ inheritance guarantee that the type of B objects will be a subtype of the type of A objects. The

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

following basic rule guarantees that a derived class defines a subtype of its public base class:

> *Standard C++ Inheritance:* A derived class may redefine virtual members of the base class and may add new members that are not defined in the base class. However, the type and visibility (public, protected or private) of any redefined member must be exactly the same in the base and the derived classes.

This rule was originally used in most implementations of C++, except for the clause about maintaining visibility, which is added here to eliminate some awkward cases.

In principle, if the compiler follows systematic rules for setting the order of member data in objects and the order of virtual function pointers in the vtable, it is possible to define compatible classes of objects without using inheritance. For example, consider the following two classes. The one on the left is the Pt class defined in Subsection 12.3.1, with protected and private members elided for brevity. The class on the right is equivalent (for most compilers) to the ColorPt class defined in Subsection 12.3.2.

```
class Point {
 public:
    int getX();
    void move(int);
 protected:   ...
 private:     ...
};
```

```
class ColorPoint {
 public:
    int getX();
    int getColor();
    void move(int);
    void darken(int);
 protected:   ...
 private:     ...
} :
```

It has exactly the same public, private and protected members, in exactly the same order, producing exactly the same object layout and exactly the same vtable. However, the C++ type checker will not recognize subtyping between this ColorPt class and this Pt class because ColorPt does not use Pt as a public base class.

12.4.3 Specializing Types of Public Members

A more permissive rule than the *standard C++ inheritance* rule stated in Subsection 12.4.2 allows virtual functions to be given subtypes of their original types:

> *Permissive Inheritance:* A derived class may redefine virtual members of the base class and may add new members that are not defined in the base class. The derived-class type of any redefined public member must be a *subtype* of the type of this member in the base class. The visibility (public, protected, or private) of each member must be the same in the derived class as in the base class.

Although early C++ used *standard C++ inheritance*, contemporary implementations of C++ use a rule that is halfway between the standard and the permissive inheritance rules:

> *Contemporary C++ Inheritance:* A derived class may redefine virtual members of the base class and may add new members that are not defined in the

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

base class. The visibility (public, protected, or private) of any redefined member must be the same in the base and the derived classes. If a virtual member function f has type A ⟶ B in a base class, then f may be given type A ⟶ C in a derived class, provided C <: B.

This allows the return type to be replaced with a subtype, but does not allow changes in the argument types of member functions. Because the argument types of functions are used to resolve overloading, but the return types are not, this inheritance rule does not interact with the C++ overloading algorithm.

12.4.4 Abstract Base Classes

In many object-oriented programs, it is useful to define general concepts, such as container, account, shape, or vehicle, that are not implemented themselves, but are useful as generalizations of some set of implemented classes. For example, in the Smalltalk container class library discussed in Section 11.7, all of the classes define the methods of the container interface, but the implemented containers come from more specialized classes such as dictionary, array, set, or bag. Because subtyping follows inheritance in C++, there is a specific C++ construct developed to meet the need for classes that serve an organizational purpose but do not have any implemented objects of their own.

An *abstract class* is a class that has at least one pure virtual member function. A pure virtual member function is one whose implementation is declared empty. It is not possible to construct objects of an abstract class. The purpose of an abstract class is to define a common interface for one or more derived classes that are not abstract.

For example, an abstract-class file can be used to define the operations open and read that must be implemented in every kind of file class that uses file as a base class:

```
class file {
public:
   void virtual Open() = 0; /* pure virtual member function */
   void virtual Read() = 0; /* pure virtual member function */
//..
};
```

Another example appears in the geometry classes in Appendix B.1.5. The class shape is abstract, defining the common interface of various kinds of shapes. Two derived classes of shape are circle and rectangle.

Although it is not possible to create an instance of an abstract class like file or shape, it is possible to declare a pointer with type file* or shape*. If pointer f has type file*, then f can be assigned any object from any derived class of file. Another way that abstract classes are used is as the argument types of functions. If some function can be applied to any type of file, its argument can be declared to have type file*.

In addition to defining an interface, an abstract class establishes a layout for a virtual function table (vtable). In particular, if textFile and streamFile have file as a

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

base class, then both classes will have open and read in the same positions in their respective vtables.

The use of "abstract" in the term "abstract class" is not the same as in "abstract data type." In particular, an abstract class is not a class with a stronger form of information hiding.

12.5 MULTIPLE INHERITANCE

Multiple inheritance is a controversial part of object-oriented programming. Once we have adopted the idea of using inheritance to build new classes, it seems natural to allow a class to be implemented by inheriting from more than one base class.

Unfortunately, multiple inheritance brings with it a set of problems that does not have simple, elegant solutions. For this reason, many language designers tried to avoid putting multiple inheritance in their language (like Java) or postponed adding multiple inheritance as long as possible (like Smalltalk and C++).

Some of the most compelling examples of multiple inheritance involve combining two or more independent kinds of functionality. For example, suppose we are doing some geometric calculations and want to manage the memory allocated to various geometric objects in various ways. We might have a class hierarchy of geometric objects, such as shapes, circles, and rectangles, and a hierarchy of classes of memory management mechanisms. For example, we might have ordinary C++ heap objects that require users to call their destructors and garbage-collected objects that are automatically reclaimed when some function invoking the garbage collector is called. Another possibility that might be useful for some objects would be to associate a reference count with each object (maintaining a count of the number of pointers that refer to this object). This is a simple form of garbage collection that works more efficiently when it is easy to maintain a count accurately. If we have separate geometry and memory management hierarchies, then it will be possible to implement many combinations of shape and memory by multiple inheritance. An example is shown in Figure 12.2.

Here, reference-counted rectangles are implemented by inheriting the geometric aspects of rectangles from the rectangle class and the mechanism for reference counting from the reference-counted class.

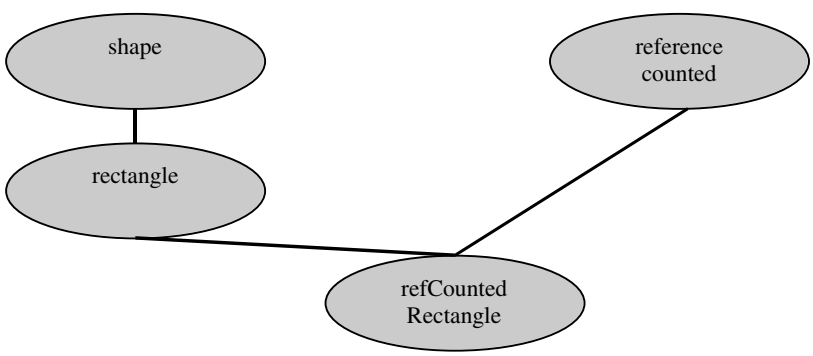


Figure 12.2. Multiple inheritance.

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

12.5.1 Implementation of Multiple Inheritance

The fundamental property of C++ class implementation is that if class B has class A as a public base class, then the initial segment of every B object must look like an A object, and similarly for the B and A virtual function tables. This property makes it possible to access an inherited A member of a B object in exactly the same way we would access the same member of an A object. Although it is fairly easy to guarantee this property with only single inheritance, the implementation of objects and classes with multiple inheritance is more complex.

The implementation issues are best illustrated by example. Suppose class C is defined by inheriting from classes A and B, as in the following code:

```
class A {
  public:
    int x;
    virtual void f();
};
class B {
  public:
    int y;
    virtual void g();
    virtual void f();
};
class C: public A, public B {
  public:
    int z;
    virtual void f();
};
  C *pc = new C;
  B *pb = pc;
  A *pa = pc;
```

In the final three lines of the sample code, we have three pointers to the same object. However, the three pointers have three different static types.

The representation of the C object created from the preceding example class C and the values of the associated pointers are illustrated in Figure 12.3. The illustration shows three pointers to the class C object. Pointers pa and pc point to the top of the object and pointer pb points to a position δ locations below the top of the object. The class C object has two virtual function table pointers and three sections of member data, one for member data declared in the A class, one for member data declared in the B class, and one for member data declared in the C class. The two virtual function table pointers point to two different vtables whose use is subsequently explained. The top vtable, labeled C-as-A, is used when a C object is used as an object of type C or as an object of type A. The lower vtable, labeled C-as-B, is used when a C object is used as an object of type B and to call C member functions that are inherited from B (in this case, function g).

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

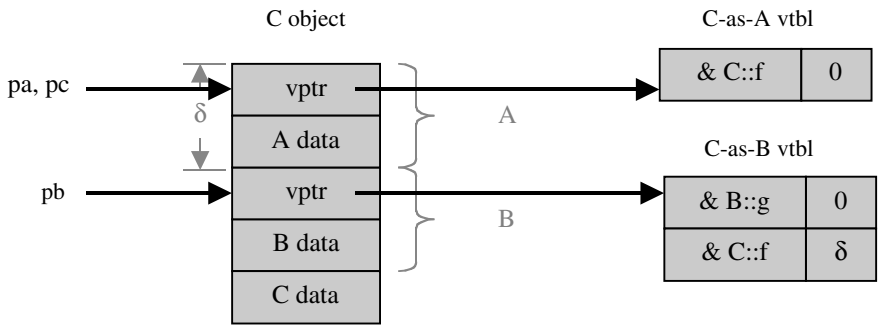


Figure 12.3. Object and vtable layout for multiple inheritance.

Figure 12.3 shows three complications associated with the implementation of multiple inheritance in C++:

- There are multiple virtual function tables for class C, not just one.
- A pointer of static type `B` points to a different place in the object than an `A` or `C` pointer.
- There are some extra numbers stored in the vtable, shown here to the right of the address `& C::f` of a virtual function.

There are two vtables for class `C` because a `C` object may be used as an `A` object or as a `B` object. In symbols, we want `C <: A` and `C <: B`. For this to work, the `C` virtual function table must have an initial segment that looks like an `A` virtual function table and an initial segment that looks like a `B` virtual function table. However, because `A` and `B` are independent classes, there is no reason why `A` and `B` should have similar vtables. The solution is to provide two vtables, one that allows a `C` object to be used as an `A` object, and another that allows a `C` object to be used as a `B` object. In principle, if a derived class has `n` base classes, the class will have `n+1` vtables, one for each base class and one for the derived class itself. However, in practice, only `n` vtables are needed, as the vtable for the derived class can follow the order of the vtable for one of the base classes, allowing the vtable for the derived class to be the same as the vtable used for viewing objects of the derived class as objects of one of the base classes.

There is a similar issue surrounding the arrangement of data members within a C object. As we can see from Figure 12.3, the pointer `pb`, referring to a `C` object as a `B` object, points to a different place in the object than `pa` and `pc`. When a `C` object is used as a `B` object, the member data must be accessed in exactly the same way as if the object had been created as a `B` object. This is accomplished when the member data inherited from `B` are arranged in a contiguous part of the object, below the `A` data. When a pointer to a `C` object is converted or cast to type `B`, the pointer must be incremented by some amount δ to skip over the `A` part and point directly to the `B` part of the object.

The number δ, which is the numeric difference `pb - pa`, is stored in the `B` table, as it is also used in calling some of the member functions. The offset δ in this vtable is used in call to `pb->f()`, as the virtual function `f` defined in `C` may refer to `A` member data

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

that are above the pointer pb. Therefore, on the call to inherited member function f, the this pointer passed to the body of f must be δ less than the pointer pb.

12.5.2 Name Clashes, Diamond Inheritance, and Virtual Base Classes

Some interesting and troubling issues arise in connection with multiple inheritance. The simplest and most pervasive problem is the problem of name clashes, which arises when two base classes have members of the same name. A variant of the name clash problem arises when a class C inherits from two classes, A and B, that inherit the same member from a common base class, D. In this case, the inherited members of C with the same name will have the same definition. This eliminates some problems. However, there is still the issue of deciding whether to place two copies of doubly inherited member data in a derived class. By default, C++ duplicates members that are inherited twice. The virtual base-class mechanism provides an alternative way of sharing common base classes in which doubly inherited members occur only once in the derived class.

Name Clashes

If class C inherits from classes A and B and A and B have members of the same name, then there is a *name clash*. There are three general ways of handling name clashes:

- *Implicit resolution:* The language resolves name conflicts with an arbitrary rule.
- *Explicit resolution:* The programmer must explicitly resolve name conflicts in the code in some way.
- *Disallow name clashes:* Programs are not allowed to contain name clashes.

There is no systematic best solution to the problem of name clashes. The simplest solution is to disallow name clashes. However, there are some situations, such as the diamond inheritance patterns subsequently described, for which this seems unnecessarily restrictive.

Implicit resolution also has its problems. Python and the CLOS, for example, have mechanisms that give priority to one base class over another. This approach does not work well in some situations. For example, suppose class C inherits from classes A and B and suppose that both A and B have a member function named f. If class C is declared so that A has priority over B, then class C will inherit the member function f from A and not inherit the member f from B. However, suppose that class B has another member function g that calls member function f and relies on a particular definition of f in order to work properly. Then, in effect, inheriting f from class A will cause the inherited member g from B not to work properly.

C++ allows name clashes and requires explicit resolution in the program. If a data member m is inherited from two bases classes A and B, then there will be two data members in the derived class. If a member m is inherited from two bases classes A and B, then the only way to refer to member m is by a fully qualified name. This is illustrated in the following C++ code samples.

The following code contains a compile-time error because the call to an inherited member function is ambiguous:

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

```
class A {
   public:
     virtual void f() { ... }
};
class B {
   public:
     virtual void f() { ... }
};
class C : public A, public B { };
...
       C* p;
       p->f();    // error since call to member function is ambiguous
```

In C++, it is not an error to inherit from two classes containing the same member names. The error occurs only when the compiler sees a reference to the doubly inherited member. The preceding program fragment can be rewritten to avoid the error, by use of fully qualified names. Classes A and B remain the same; only the definition of class C needs to be changed to specify which inherited member function f should be called:

```
class C : public A, public B {
public:
       virtual void f( ) {
              A::f( );    // Call member function f from A, not B::f();
       }
...
};
```

Here, class C explicitly defines a member function f. However, the definition of C::f is simply to call A::f. This makes the definition of C::f unambiguous and has the same effect as inheriting A::f by some other name resolution rule.

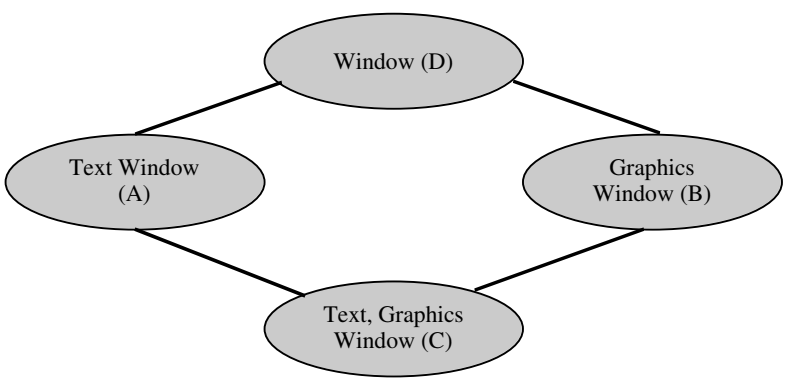


Figure 12.4. Diamond inheritance.

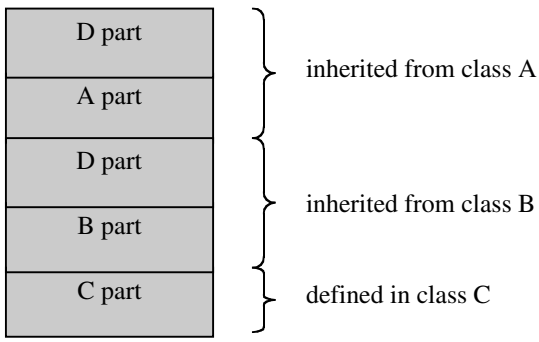
https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

Diamond Inheritance

An interesting kind of name clash occurs when two classes share a common base class. In this situation, a member that is inherited twice will have the same definition in both classes. The complications associated with this situation are commonly called the *diamond inheritance problem*, as this situation involves a diamond-shaped inheritance graph. For example, consider the four classes shown in Figure 12.4.

The classes are labeled `A`, `B`, `C`, and `D` because these shorter labels fit better in subsequent illustrations.

Under ordinary multiple inheritance, there are two copies of window members in a text graphics window. More specifically, the data members of a class `C` object have the following form:



A class `C` object has all of these parts because a class `C` object has parts for both of the base classes, `A` and `B`. However, classes `A` and `B` both inherit from `D`, so `A` objects have a copy of each data member inherited from `D` and `B` objects have a copy of each data member inherited from `D`.

For window classes, it does not make sense for a text and graphics window to have two copies of each window-class member. In a windowing system with this design, the window class might contain data members to store the bounding box of the window, data members to determine the background color and border color of the window, and so on. The text window class might have additional data members and member functions to store and display text inside a window, and similarly the graphics window class would contain declarations and code for displaying graphics inside a window. If we want a window that can contain graphics and text within the same window, then C++ multiple inheritance as described so far will not work. Instead, the C++ implementation of multiple inheritance will produce objects that consist of two windows, one capable of displaying text and the other capable of displaying graphics.

The diamond inheritance problem is the basic problem that there is no "right" way to handle multiple inheritance when two base classes of a class `C` both have the same class `D` as a base class. This is called the diamond inheritance problem because the classes `A`, `B`, `C`, and `D` form a diamond shape, as shown in Figure 12.4. The biggest problem with diamond inheritance is what to do with member data. One option is to duplicate the `D` class data members. This can give useful behavior in some cases, but is undesirable in situations like the window classes in Figure 12.4. If data members are duplicated, then there is also a naming problem: If you select an inherited `D` member

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

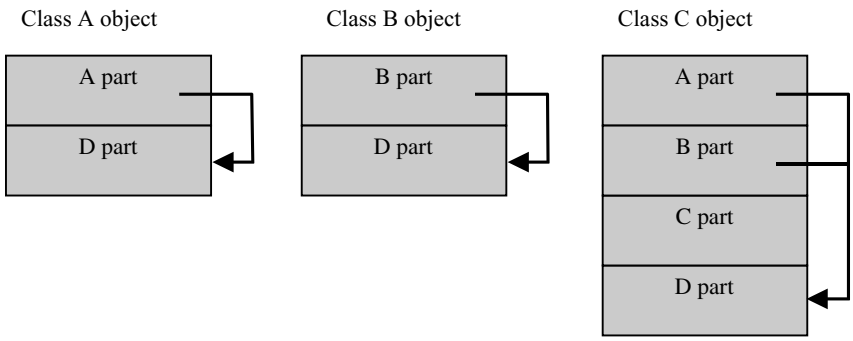


Figure 12.5. Virtual base class.

by name, which copy should you get? The C++ solution is to require a fully qualified name.

The alternative to duplicating data members is to put only one copy of each `D` member into a `C` object and have the inherited `A` and `B` class member functions both refer to the shared copy. This also can lead to problems if the `A` and `B` classes may use `D` data members differently. For example, some `A` member function may be written assuming an inherited `D` class data member is an even number, whereas some `B` class member function may set it to an odd number. Diamond inheritance leads to problems that do not have a simple, generally applicable solution.

Virtual Base Classes

C++ has a mechanism for eliminating multiple copies of duplicated base-class members, called *virtual base classes*. If `D` is declared a virtual base class of `A` and `B`, and class `C` has `A` and `B` as base classes, then `A`, `B`, and `C` class objects will have the form illustrated in Figure 12.5.

The main idea is that if class `A` has a virtual base class `D`, then access to `D` data members of an `A` object is by indirection through a pointer in the object. As a result, the location of the `D` part does not matter. The `D` part could immediately follow the `A` part of the object or reside someplace else in memory. This indirection through a pointer allows `C` objects to have an `A` part and a `B` part that share a common `D` part.

Inheritance with virtual base classes has its complications. One annoyance is that the choice between virtual and nonvirtual base classes is made once for all uses of a class. More specifically, suppose that one class, `A`, has another class, `D`, as a base class. Then the designer of `A` must decide whether `D` should be a virtual base class or not, even though this affects only other classes that inherit from `A`. Moreover, if some classes that inherit from `A` need `D` to be a virtual base class of `A` and others require the opposite, then the only solution is to make two versions of the `A` class, one with `D` as a virtual base class and the other with `D` as a nonvirtual base class. An oddity of virtual base classes is that it is possible to convert a pointer to an instance of a class that has a virtual base class to a pointer to an object of that virtual base class. However, the opposite conversion is not allowed, making this form of type conversion irreversible. For this reason, some style guides recommend against casts when virtual base classes are used. This can be a complex guideline to follow, as users of a class library must

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

pay careful attention to when virtual base classes are used in the implementation of the library.

12.6 CHAPTER SUMMARY

The main goal of C++ is to provide object-oriented features in a C-based language without compromising the efficiency of C. C++ is a very carefully designed language that has succeeded admirably in spite of difficult design constraints.

Some features of C++ that are not part of the object system are type `bool`, reference types and pass-by-reference, user-defined overloading, function templates, and exceptions. C++ also has better static type checking than C.

The focus on C++ efficiency is articulated in the design principle,"If you do not use a feature, you should not pay for it." This principle suggests that C programs should compile as efficiently under the C++ compiler as under the C compiler. It would violate this principle to implement C integers as objects, for example, and use dynamic method lookup to find integer functions at run-time as in Smalltalk, as this would significantly reduce the performance of C integer calculations.

The most significant part of C++ is the set of object-oriented concepts added to C; these are the main concepts:

- *Classes*, which declare the type associated with all objects constructed from the class, the data members of each object, and the member functions of the class.
- *Objects*, which consist of private data and public functions for accessing the hidden data, as in other object-oriented languages.
- *Dynamic lookup*, for member functions that are declared virtual. A virtual function in a derived class (subclass) may be implemented differently from virtual functions of the same name in a base class (superclass).
- *Encapsulation*, based on programmer designations public, private, and protected that determine whether data and functions declared in a class are visible outside the class definition.
- *Inheritance*, using subclassing, in which one class may be defined by inheriting the data and functions declared in another. C++ allows single inheritance, in which a class has a single base class (superclass) or multiple inheritance in which a class has more than one base class.
- *Subtyping*, based on subclassing. For one class to define a subtype of the type defined by another class, inheritance must be used. However, the programmer may decide whether inheritance results in a subtype or not by specifying that the base class is public or not.

At run time, C++ virtual function lookup uses indirection through the virtual function table (vtable) of the class. The C++ static type system makes it possible to determine the offset (position of a pointer in the vtable) of virtual function at compile time. When a call like `p->move(...)` is compiled, the compiler can determine a static type for the pointer `p`. This static type must be a class, and this class must declare or inherit a function called `move`. The compiler can examine the class hierarchy to see what location the virtual function move will occupy in the vtable for the class.

A fundamental property of C++ class implementation is that if class `B` has class `A` as a public base class, then the initial segment of every `B` object must look like an

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

A object, and similarly for the B and A virtual function tables. This property makes it possible to access an inherited A member of a B object in exactly the same way we would access the same member of an A object. Although it is fairly easily to guarantee this property with only single inheritance, the implementation of objects and classes with multiple inheritance is more complex.

There are a number of aspects of C++ that programmers occasionally find difficult. Some of the main problem areas are

- *casts and conversions* that can be complex and unpredictable in certain situations,
- *objects allocated on the stack* and other aspects of object memory management,
- *overloading*, a complex code selection mechanism in C++ that can interact unpredicatably with dynamic lookup (virtual function lookup),
- *multiple inheritance*, which is more complex in C++ than in other languages because of the way objects and virtual function tables are configured and accessed.

These problem areas exist not because of oversights but because the goals of C++, followed to their logical conclusion, led to a design with these properties.

EXERCISES

12.1 Assignment and Derived Classes

This problem exams the difference between two forms of object assignment. In C++, local variables are stored on the run-time stack, whereas dynamically allocated data (created with the new keyword) are stored on the heap. A local object variable allocated on the stack has an object L-value and an R-value. Unlike many other object-oriented languages, C++ allows object assignment into object L-values.

A C++ programmer writes the following code:

```
class Vehicle {
public:
    int x;
    virtual void f();
    void g();
} ;

class Airplane : public Vehicle {
public:
    int y;
    virtual void f();
    virtual void h();
} ;

void inHeap() {
    Vehicle *b1 = new Vehicle;          // Allocate object on the heap
    Airplane *d1 = new Airplane;        // Allocate object on the heap
    b1->x = 1;
    d1->x = 2;
    d1->y = 3;
    b1 = d1;                    // Assign derived class object to base class pointer
```

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

```
}

void onStack() {
    Vehicle b2;                    // Local object on the stack
    Airplane d2;                   // Local object on the stack
    b2 . x = 4;
    d2 . x = 5;
    d2 . y = 6;
    b2 = d2;                // Assign derived class object to base class variable
}

int main() {
   inHeap();
   onStack();
}
```

(a) Draw a picture of the stack, heap, and vtables that result after objects `b1` and `d1` have been allocated (but *before* the assignment `b1=d1`) during the call to `inHeap`. Be sure to indicate where the instance variables and vtable pointers of the two objects are stored before the assignment `b1=d1` and to which vtables the respective vtable pointers point.

(b) Redraw your diagram from (a), showing the changes that result after the assignment `b1=d1`. Be sure to clearly indicate where `b1`'s vtable pointer points after the assignment `b1=d1`. Explain why `b1`'s vtable pointer points where it does after the assignment `b1=d1`.

(c) Draw a picture of the stack, heap, and vtables that result after objects `b2` and `d2` have been allocated (but *before* the assignment `b2=d2`) during the call to `onStack`. Be sure to indicate where the instance variables and vtable pointers of the two objects are stored before the assignment `b2=d2` and to which vtables the respective vtable pointers point.

(d) In C++, assignment to objects (such as `b2=d2`) is performed by copying into all the left object's data members (`b2`'s in our example) from the right object's (`d2`'s) corresponding data members. If the right object contains data members not present in the left object, then those data simply are not copied. In `b2=d2` this means overwriting all `b2`'s data members with the corresponding value from `d2`.

   i. Re-draw your diagram from (c), showing the changes that result after the assignment `b2=d2`.

   ii. Explain why it is not sensible to copy all of `d2`'s member data into `b2`'s record.

   iii. Explain why `b2`'s vtable pointer is not changed by the assignment.

(e) We have used assignment statements `b1=d1` and `b2=d2`. Why are the opposite statements `d1=b1` and `d2=b2` not allowed?

12.2 Function Objects

The C++ Standard Template Library (STL) was discussed in an earlier chapter. In STL terminology, a *function object* is any object that can be called as if it is a function. Any object of a class that defines `operator()` is a function object. In

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

addition, an ordinary function may be used as a function object, as f() is meaningful is f is a function and similarly if f is a function pointer.

Here is a C++ template for combining an argument type, return type, and operator() together into a function object. Every object from every subclass of FuncObj<A,B>, for any types A and B is a function object, but not all function objects come from such classes:

```
template <typename Arg, typename Ret>
class FuncObj {
public:
  typedef Arg argType;
  typedef Ret retType;
  virtual Ret operator()(Arg) = 0;
};
```

Here are two example classes of function objects. In the first case, the constructor stores a value in a protected data field so that different function objects from this class will divide by different integers. In a sense to be explored later in this problem, instances of DivideBy are similar to closures as they may contain hidden data:

```
class DivideBy : public FuncObj<int, double> {
protected:
  int divisor;
public:
  DivideBy(int d) {
    this->divisor = d;
  }
  double operator()(int x) {
    return x/((double)divisor);
  }
};

class Truncate : public FuncObj<double, int> {
public:
  int operator()(double x) {
    return (int) x;
  }
};
```

Because DivideBy uses the FuncObj template as a public base class, DivideBy is a subtype of FuncObj<int, double>, and similarly for Truncate.

(a) Fill in the blanks to complete the following Compose template. The Compose constructor takes two function objects f and g and creates a function object that computes their composition $\lambda x.\ f(g(x))$.

You will need to fill in type declarations on lines 9 and 13 and code on lines 10, 11, and 14. If you do not know the correct C++ syntax, you may use short English descriptions for partial credit. (We have double-checked the parentheses to make sure they are correct.)

To give you a better idea of how Compose is supposed to work, you may want to look at the following sample code that uses Compose to compose DivideBy and Truncate:

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

```
1:    template < typename Ftype , typename Gtype >
2:    class Compose :
3:      public FuncObj < typename Gtype : : argType,
4:                       typename Ftype : : retType > {
5:    protected:
6:      Ftype *f;
7:      Gtype *g;
8:    public:
9:      Compose( _____ f, _____ g) {
10:       _____ = f ;
11:       _____ = g ;
12:     }
13:     _____ operator()( _____ x) {
14:      return ( _____ )(( _____ )( _____ ));
15:     }
16:   };

void main() {
  DivideBy *d = new DivideBy(2);
  Truncate *t = new Truncate();
  Compose<DivideBy, Truncate> *c1
    = new Compose<DivideBy, Truncate>(d,t);
  Compose<Truncate, DivideBy> *c2
    = new Compose<Truncate, DivideBy>(t,d);

  cout << (*c1)(100.7) << endl; // Prints 50.0
  cout << (*c2)(11) << endl;    // Prints 5
}
```

(b) Consider code of the following form, in which A, B, C, and D are types that might not all be different (i.e., we could have A = B = C = D = int or A, B, C, and D might be four different types):

```
class F : public FuncObj<A, B> {
  ...
};
class G : public FuncObj<C, D> {
  ...
};
F *f = new F( . . . );
G *g = new G( . . . );
Compose<F, G> *h
 = new Compose<F, G>(f,g);

cout << (*h)( . . . ) << endl; // call compose function
```

What will happen if the return type D of g is not the same as the argument type A of f? If it is possible for an error to occur and possible for an error not to occur, say which conditions will cause an error and which will not. If it is possible for an error to occur at compile time or at run time, say when the error will occur and why.

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

(c) Our Compose template is written assuming that Ftype and Gtype are classes that have argType and retType type definitions. If you wanted to define a Compose template that works for all function objects, what arguments would your template have and why? Your answer should be in the form of "the types of variables ..., the arguments of functions ..., and the return values of ..." Assume that the code in lines 10, 11, and 14 stays the same. All we want to change are the template parameters on line 1 and possibly the parts of the body of the template that refer to template parameters.

The rest of this problem asks about the similarities and differences between C++ function objects and function closures in languages like Lisp, ML, and Scheme. In studying scope and activation records, we looked at a function makeCounter that returns a counter, initialized to some integer value. Here is makeCounter function in Scheme:

```
(define makeCounter
  (lambda (val)
    (let (( counter (lambda (inc) (set! val (+ val inc)) val)) )
      counter)))
```

Here is part of an interactive session in which makeCounter is used:

```
==> (define c (makeCounter 3))
#<unspecified>
==> (c 2)
5
==> (c 2)
7
```

The first input defines a counter c, initialized to 3. The second input line adds 2, producing value 5, and the third input line adds 2 again, producing value 7.

A general idea for translating Lisp functions into C++ function objects is to translate each function f into a class A so that objects from class A are function objects that behave like f. If f is defined within some nested scope, then the constructor for A will put the global variables of f into the function object, so that an instance of A behaves like a closure for f.

Because makeCounter does not have any free variables, we can translate makeCounter into a class MAKECOUNTER that has a constructor with no parameters:

```
class MAKECOUNTER {
    public:
        class COUNTER {
            protected:
                int val;
            public:
                COUNTER(int init){val = init;}
                int operator()(int inc) {
                    val = val + inc;
                    return val;
                }
        };
        MAKECOUNTER(){}
```

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

```
        COUNTER* operator()(int val) {
                return new COUNTER(val);
        }
};
```

We can create a MAKECOUNTER function object and use it to create a COUNTER function object as follows:

```
MAKECOUNTER *m = new MAKECOUNTER();
MAKECOUNTER::COUNTER * c = (*m)(3);

cout << (*c)(2) << endl; // Prints 5
```

(d) Thinking generally about Lisp closures and C++ function objects, describe one way in which C++ function objects might serve some programming objectives better than Lisp closures.

(e) Thinking generally about Lisp closures and C++ function objects, describe one way in which Lisp closures might serve some programming objectives better than C++ function objects.

(f) Do you think this approach sketched in this problem will allow you to translate an arbitrary nesting of Lisp or ML functions into C++ function objects? You may want to consider the following variation of MAKECOUNTER in which the counter value val is placed in the outer class instead of in the inner class:

```
class MAKECOUNTER {
protected:
  int val;

public:
  class COUNTER {
  private:
    MAKECOUNTER *mc;

  public:
    COUNTER(MAKECOUNTER* mc, int init) {
      this->mc = mc;
      mc->val = init;
    }

    int operator()(int inc) {
      mc->val = mc->val + inc;
      return mc->val;
    }
  };
  friend class COUNTER;

  MAKECOUNTER(){}
  COUNTER* operator()(int val) {
    return new COUNTER(this,val);
  }
};
```

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

12.3 Function Subtyping

Assume that A <: B and B <: C. Which of the following subtype relationships involving the function type B ⟶ B hold in principle?

(i) (B ⟶ B) <: (B ⟶ B)
(ii) (B ⟶ A) <: (B ⟶ B)
(iii) (B ⟶ C) <: (B ⟶ B)
(iv) (C ⟶ B) <: (B ⟶ B)
(v) (A ⟶ B) <: (B ⟶ B)
(vi) (C ⟶ A) <: (B ⟶ B)
(vii) (A ⟶ A) <: (B ⟶ B)
(viii) (C ⟶ C) <: (B ⟶ B)

12.4 Subtyping and Public Data

This question asks you to review the issue of specializing the types of public member functions and consider the related issue of specializing the types of public data members. To make this question concrete, we use the example of circles and colored circles in C++. Colored circles get darker each time they are moved. We assume there is a class Point of points and class ColPoint of colored points with appropriate operations. Class Point is a public base class of ColPoint. The basic definition for circle is:

```
class Circle {
public:
    Circle(Point* c, float r) { center=c; radius=r; }

    Point* center;
    float radius;
    virtual Circle* move(float dx, float dy)
                {center->move(dx,dy); return(this);}
};
```

For each of the following definitions of colored circle, explain why a colored circle should or should not be considered a subtype of a circle, in principle. If a colored circle should not be a subtype, give some fragment of code that would be type correct if we considered colored circles to be a subtype of circles, but would lead to a type error at run time:

(a)
```
class ColCircle : public Circle {
public:
    ColCircle(Point* c, float r, color cl) : Circle(c,r) { col=cl; }

    color col;
    virtual Circle* move(float dx, float dy)
                {center->move(dx,dy); darken(); return(this);}
    virtual void darken()
                {if (col==green) col=darkgreen;}
};
```

(b) Suppose we change the definition of ColCircle so that move returns a colored circle:

```
class ColCircle : public Circle {
```

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

```
public:
    ColCircle(Point* c, float r, color cl) : Circle(c,r) { col=cl; }

    color col;
    virtual ColCircle* move(float dx, float dy)
                {center->move(dx,dy); darken(); return(this);}
    virtual void darken()
                {if (col==green) col=darkgreen;}
};
```

(c) Suppose that instead of representing the color of a circle by a separate data member, we replace points with colored points and use a colored point for the center of the circle. One advantage of doing so is that we can use all of the color operations provided for colored points, as illustrated by the darken member function below. This could be useful if we wanted to have the same color operations on a variety of geometric shapes:

```
class ColCircle : public Circle {
public:
    ColCircle(ColPoint* c, float r) : Circle(c,r) { }

    ColPoint* center;

    virtual ColCircle* move(float dx, float dy)
                {center->move(dx,dy); darken(); return(this);}
    virtual void darken()
                {center->darken();}
};
```

The important issue is the redefinition of the type of center. Assume that each ColCircle object has center and radius data fields at the same offset as the center and radius fields of a Circle object. In the modified version of C++ considered in this question, the declaration ColPoint* center does *not* mean that a ColCircle object has two center data fields.

12.5 Phantom Members

A C++ class may have virtual members that may be redefined in derived classes. However, there is no way to "undefine" a virtual (or nonvirtual) member. Suppose we extend C++ by adding another kind of member, called a *phantom* member, that is treated as virtual, but only defined in derived classes if an explicit definition is given. In other words, a "phantom" function is not inherited unless its name is listed in the derived class. For example, if we have two classes

```
class A {
...
public:
    phantom void f(){...}
    ...
};
class B : public A {
...
```

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

```
public:
    .../* no definition of f */
};
```

then f would appear in the vtbl for A objects and, if x is an A object, x . f() would be allowed. However, if f is not declared in B, then f might not need to appear in the vtbl for B objects and, if x is a B object, x . f() would not be allowed. Is this consistent with the design of C++, or is there some general property of the language that would be destroyed? If so, explain what this property is and why it would be destroyed.

12.6 Subtyping and Visibility

In C++, a virtual function may be given a different access level in a derived class. This produces some confusing situations. For example, this is legal C++, at least for some compilers:

```
class Base {
   public:
       virtual int f();
};
class Derived: public Base {
   private:
       virtual int f();
};
```

This question asks you to explain why this program conflicts with some reasonable principles, yet somehow does not completely break the C++ type system.

(a) In C++, a derived class D with public base class B is treated as a subtype of B. Explain why this is generally reasonable, given the definition of subtyping from class.

(b) Why would it be reasonable for someone to argue that it is *incorrect* to allow a public member inherited from a public base class to be redefined as private?

(c) A typical use of subtyping is to apply a function that expects an B argument to a D when D <: B. For example, here is a simple "toy" program that applies a function defined for base-class objects to a derived-class object. Explain why this program *compiles* and *executes* for the preceding Base and Derived classes (assuming we have given an implementation for f):

```
int g(Base &x) {
  return(x.f()+1);
}

int main() {
  Base b;
  cout << "g(b) = " << g(b) << endl;
  Derived d;
  cout << "g(d) = " << g(d) << endl;
}
```

(d) Do you think there is a mistake here in the design of C++ ? Briefly explain why or why not.

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

12.7 Private Virtual Functions

At first thought, the idea of a private virtual function might seem silly: Why would you want to redefine a function that is not visible outside the class? However, there is a programming idiom that makes use of private virtual functions in a reasonable way. This idiom is illustrated in the following code, which uses a public function to call a private virtual function:

```
class Computer {
  public:
  void retailPrice(void) {
     int p = 2 * manufactureCost();
     printf("$ %d \ n", p); // Print p
  }
  private:
  virtual int manufactureCost(void) {return 1000;}
};

class IBMComputer: public Computer {
  private:
  virtual int manufactureCost(void) {return 1500;}
};

int main(void) {
  Computer *cPtr = new Computer();
  IBMComputer *ibmPtr = new IBMComputer();
  Computer *cibmPtr = new IBMComputer();

  cPtr->retailPrice();
  ibmPtr->retailPrice();
  cibmPtr->retailPrice();
  return 0;
}
```

This question asks about how the function calls to retailPrice() are evaluated.

(a) Explain which version of manufactureCost() is used in the call cibmPtr-> retailPrice() and why.

(b) If the virtual keyword is omitted from the declaration of manufactureCost in the derived class IBMComputer, the preceding code will still compile without error and execute. Will manufactureCost be implemented as a virtual function in class IBMComputer? Use your knowledge of how C++ is implemented to explain why or why not.

(c) It is possible for the private base-class function to be declared public in the derived class. Does this conflict with subtyping principles? Explain why or why not in a few words.

12.8 "Like Current" in Eiffel

Eiffel is a statically typed object-oriented programming language designed by Bertrand Meyer and his collaborators. The language designers did not intend the language to have any type loopholes. However, there are some problems

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

surrounding an Eiffel type expression called like current. When the words like current appear as a type in a method of some class, they mean, "the class that contains this method." To give an example, the following classes were considered statically type correct in the language Eiffel.

```
Class Point
    x : int
    method equals (pt : like current) : bool
        return self.x == pt.x

class ColPoint inherits Point
    color : string
    method equals (cpt : like current) : bool
        return self.x == cpt.x and self.color == cpt.color
```

In Point, the expression like current means the type Point, whereas in ColPoint, like current means the type ColPoint. However, the type checker accepts the redefinition of method equals because the declared parameter type is like current in both cases. In other words, the declaration of equals in Point says that the argument of p.equals should be of the same type as p, and the declaration of equals in ColPoint says the same thing. Therefore the types of equals are considered to match.

(a) Using the basic rules for subtyping objects and functions, explain why ColPoint should not be considered a subtype of Point "in principle."

(b) Give a short fragment of code that shows how a type error can occur if we consider ColPoint to be a subtype of Point.

(c) Why do you think the designers of Eiffel decided to allow subtyping in this case? In other words, why do you think they wanted like current in the language?

(d) When this error was pointed out (by W. Cook after the language had been in use for several years), the Eiffel designers decided not to remove like current, as this would "break" lots of existing code. Instead, they decided to modify the type checker to perform whole-program analysis. More specifically, the modified Eiffel type checker examined the whole Eiffel program to see if there was any statement that was likely to cause a type error.

   i. What are some of the disadvantages of whole-program analysis? Do not just say, "it has to look at the whole program." Instead, think about trying to debug a program in a language in which the type checker uses whole-program analysis. Are there any situations in which the error messages would not be as useful as in traditional type checking in which the type of an expression depends only on the types of its parts?

   ii. Suppose you were trying to design a type checker that allows safe uses of like current. What kind of statements or expressions would your type checker look for? How would you distinguish a type error from a safe use of like current?

12.9 Subtyping and Specifications

In the Eiffel programming language, methods can have preconditions and post-conditions. These are Boolean expressions that must be true before the method

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

is called and true afterwards. For example, the pop method in the following Stack class has a precondition size > 0. This means that in order for the pop method to execute properly, the stack should be nonempty beforehand. The postcondition size′ = size - 1 means that the size after executing pop will be one less than the size before this method is called. The syntactic convention here is that a "primed variable," such as size′, indicates the value of that variable after execution of the method:

```
class Stack {
   ...
   int size
   ...
   void pop() pre: size > 0
              post: size′ = size - 1 {
         ...
   }
 ...
 }
```

This question asks you about subtyping when preconditions and postconditions are considered part of the type of a method.

(a) In an implementation of stacks in which each stack has a maximum size, given by a constant MAX_SIZE, the push method might have the following form:

```
class FixedStack {
   ...
   int size
   ...
   void push() pre: size < MAX_SIZE {
         ...
   }
 ...
 }
```

whereas without this size restriction, we could have a stack class

```
class Stack {
   ...
   int size
   ...
   void push() pre: true {
         ...
   }
 ...
 }
```

with the precondition of push always satisfied. (For simplicity, there are no postconditions in this example.) If this is the only difference between the two classes, which one should be considered a subtype of the other? Explain briefly.

(b) Suppose that we have two classes that are identical except for the preconditions and postconditions on one method:

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

```
class A {
   ...
   int size
   ...
   void f() pre: PA, post: TA {
         ...
   }
...
}
class B {
   ...
   int size
   ...
   void f() pre: PB, post: TB {
         ...
   }
...
}
```

What relationships among PA, TA, PB, and TB should hold in order to have B <: A? Explain briefly.

12.10 C++ Multiple Inheritance and Casts

An important aspect of C++ object and virtual function table (vtbl) layout is that if class D has class B as a public base class, then the initial segment of every D object must look like a B object, and similarly for the D and B virtual function tables. The reason is that this makes it possible to access any B member data or member function of a D object in exactly the same way we would access the B member data or member function of a B object. Although this works out fairly easily with only single inheritance, some effort must be put into the implementation of multiple inheritance to make access to member data and member functions uniform across publicly derived classes.

Suppose class C is defined by inheriting from classes A and B:

```
class A {
    public:
        int x;
        virtual void f();
};
class B {
    public:
        int y;
        virtual void f();
        virtual void g();
};
class C : public A, public B {
    public:
        int z;
        virtual void f();
};
C *pc = new C;   B *pb = pc;   A *pa = pc;
```

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

and pa, pb, and pc are pointers to the same object, but with different types. The representation of this object of class C and the values of the associated pointers are illustrated in this chapter.

(a) Explain the steps involved in finding the address of the function code in the call pc->f(). Be sure to distinguish what happens at compile time from what happens at run time. Which address is found, &A::f(), &B::f(), or &C::f()?

(b) The steps used to find the function address for pa->f() and to then call it are the same as for pc->f(). Briefly explain why.

(c) Do you think the steps used to find the function address for and to call pb->f() have to be the same as the other two, even though the offset is different? Why or why not?

(d) How could the call pc->g() be implemented?

12.11 Multiple Inheritance and Thunks

Suppose class C is defined by inheriting from classes A and B:

```
class A {
    public:
        virtual void g();
        int x;
};
class B {
    public:
        int y;
        virtual B* f();
        virtual void g();
};
class C : public A, public B {
    public:
        int z;
        virtual C* f();
        virtual void g();
};
C *pc = new C;   B *pb = pc;   A *pa = pc;
```

and pa, pb, and pc are pointers to the same object, but with different types.

Then, pa and pc will contain the same value, but pb will contain a different value; it will contain the address of the B part of the C object. The fact that pb and pc do not contain the same value means that in C++, a cast sometimes has a run-time cost. (In C this is never the case.)

Note that in our example B::f and C::f do not return the same type. Instead, C::f returns a C* whereas B::f returns a B*. That is legal because C is derived from B, and thus a C* can always be used in place of a B*.

However, there is a problem: When the compiler sees pb->f() it does not know whether the call will return a B* or a C*. Because the caller is expecting a B*, the compiler must make sure to return a valid pointer to a B*. The solution is to have the C-as-B vtable contain a pointer to a *thunk*. The thunk calls C::f, and then adjusts the return value to be a B*, before returning.

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

(a) Draw all of the vtables for the classes in these examples. Show to which function each vtable slot points.

(b) Does the fact that casts in C++ sometimes have a run-time cost, whereas C never does, indicate that C++ has not adhered to the principles given in class for its design? Why or why not?

(c) Because thunks are expensive and because C++ charges programmers only for features they use, there must be a feature, or combination of features, that are imposing this cost. What feature or features are these?

12.12 Dispatch on State

One criticism of dynamic dispatch as found in C++ and Java is that it is not flexible enough. The operations performed by methods of a class usually depend on the state of the receiver object. For example, we have all seen code similar to the following file implementation:

```
class StdFile {
   private:
      enum { OPEN, CLOSED } state; /* state can only be either OPENor CLOSED */

   public:
      StdFile() { state = CLOSED; }          /* initial state is closed */
      void Open() {
         if (state == CLOSED) {
            /* open file . . . */
            state = OPEN;
         } else {
            error "file already open";
         }
      }
      void Close() {
         if (state == OPEN ) {
            /* close file . . . */
            state = CLOSED;
         } else {
            error "file not open";
         }
   }
}
```

Each method must determine the state of the object (*i.e.,* whether or not the file is already open) before performing any operations. Because of this, it seems useful to extend dynamic dispatch to include a way of dispatching not only on the class of the receiver, but also on the state of the receiver. Several object-oriented programming languages, including BETA and Cecil, have various mechanisms to do this. In this problem we examine two ways in which we can extend dynamic dispatch in C++ to depend on state. First, we present dispatch on three pieces of information:

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

- the name of the method being invoked
- the type of the receiver object
- the explicit state of the receiver object

As an example, the following declares and creates objects of the File class with the new dispatch mechanism:

```
class File {
   state in { OPEN, CLOSED };    /* declare states that a File object may be in */

   public:
      File() { state = CLOSED; }     /* initial state is closed */
      switch(state) {

       case CLOSED: {
         void Open() {           /* 1 */
            /* open file ... */
            state = OPEN;
         }
         void Close() {
            error "file not open";
         }
      }

       case OPEN: {
         void Open() {          /* 2 */
            error "file already open";
         }
         void Close() {
            /* close file ... */
            state = CLOSED;
         }
      }
   }
}

File* f = new File();
f->Open();      /* calls version 1 */
f->Open();      /* calls version 2 */
...
```

The idea is that the programmer can provide a different implementation of the same method for each state that the object can be in.

(a) Describe one advantage of having this new feature, i.e., are there any advantages to writing classes like File over classes like StdFile. Describe one disadvantage of having this new feature.

(b) For this part of the problem, assume that subclasses cannot add any new states to the set of states inherited from the base class. Describe an object representation that allows for efficient method lookup. Method call should be as fast as virtual method calls in C++, and changing the state of an object

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press

should be a constant time operation. (*Hint*: you may want to have a different vtable for each state). Is this implementation acceptable according to the C++ design goal of only paying for the features that you use?

(c) What problems arise if subclasses are allowed to extend the set of possible states? For example, we could now write a class such as

```
class SharedFile: public File {
   state in { OPEN, CLOSED, READONLY };    /* extend the set of states */
   ...
}
```

Do not try to solve any of these problems. Just identify several of them.

(d) We may generalize this notion of dispatch based on the state of an object to dispatch based on any predicate test. For example, consider the following Stack class:

```
class Stack {
   private:
      int n;
      int elems[100];

   public:
      Stack() { n = 0; }

      when(n == 0) {
         int Pop() {
            error "empty";
         }
      }

      when(n > 0) {
         int Pop() {
            return elems[--n];
         }
      }
      ...
}
```

Is there an easy way to extend your proposed implementation in part (b) to handle dispatch on predicate tests? Why or why not?

https://doi.org/10.1017/CBO9780511804175.013 Published online by Cambridge University Press