12

Derived Classes

Do not multiply objects without necessity.
– W. Occam

Concepts and classes — derived classes — member functions — construction and destruction — class hierarchies — type fields — virtual functions — abstract classes — traditional class hierarchies — abstract classes as interfaces — localizing object creation — abstract classes and class hierarchies — advice — exercises.

12.1 Introduction [derived.intro]

From Simula, C++ borrowed the concept of a class as a user-defined type and the concept of class hierarchies. In addition, it borrowed the idea for system design that classes should be used to model concepts in the programmer's and the application's world. C++ provides language constructs that directly support these design notions. Conversely, using the language features in support of design concepts distinguishes effective use of C++. Using language constructs only as notational props for more traditional types of programming is to miss key strengths of C++.

A concept does not exist in isolation. It coexists with related concepts and derives much of its power from relationships with related concepts. For example, try to explain what a car is. Soon you'll have introduced the notions of wheels, engines, drivers, pedestrians, trucks, ambulances, roads, oil, speeding tickets, motels, etc. Since we use classes to represent concepts, the issue becomes how to represent relationships between concepts. However, we can't express arbitrary relationships directly in a programming language. Even if we could, we wouldn't want to. Our classes should be more narrowly defined than our everyday concepts – and more precise. The notion of a derived class and its associated language mechanisms are provided to express hierarchical relationships, that is, to express commonality between classes. For example, the concepts of a circle and a triangle are related in that they are both shapes; that is, they have the concept of a shape in common. Thus, we must explicitly define class ***Circle*** and class ***Triangle*** to have class ***Shape*** in

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

common. Representing a circle and a triangle in a program without involving the notion of a shape would be to lose something essential. This chapter is an exploration of the implications of this simple idea, which is the basis for what is commonly called object-oriented programming.

The presentation of language features and techniques progress from the simple and concrete to the more sophisticated and abstract. For many programmers, this will also be a progression from the familiar towards the less well known. This is not a simple journey from ''bad old techniques'' towards ''the one right way.'' When I point out limitations of one technique as a motivation for another, I do so in the context of specific problems; for different problems or in other contexts, the first technique may indeed be the better choice. Useful software has been constructed using all of the techniques presented here. The aim is to help you attain sufficient understanding of the techniques to be able to make intelligent and balanced choices among them for real problems.

In this chapter, I first introduce the basic language features supporting object-oriented programming. Next, the use of those features to develop well-structured programs is discussed in the context of a larger example. Further facilities supporting object-oriented programming, such as multiple inheritance and run-time type identification, are discussed in Chapter 15.

12.2 Derived Classes [derived.derived]

Consider building a program dealing with people employed by a firm. Such a program might have a data structure like this:

```
struct Employee {
    string first_name, family_name;
    char middle_initial;
    Date hiring_date;
    short department;
    // ...
};
```

Next, we might try to define a manager:

```
struct Manager {
    Employee emp;              // manager's employee record
    set<Employee*> group;      // people managed
    short level;
    // ...
};
```

A manager is also an employee; the ***Employee*** data is stored in the ***emp*** member of a ***Manager*** object. This may be obvious to a human reader – especially a careful reader – but there is nothing that tells the compiler and other tools that ***Manager*** is also an ***Employee***. A ***Manager**** is not an ***Employee****, so one cannot simply use one where the other is required. In particular, one cannot put a ***Manager*** onto a list of ***Employee***s without writing special code. We could either use explicit type conversion on a ***Manager**** or put the address of the ***emp*** member onto a list of ***employee***s. However, both solutions are inelegant and can be quite obscure. The correct approach is to explicitly state that a ***Manager*** *is* an ***Employee***, with a few pieces of information added:

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

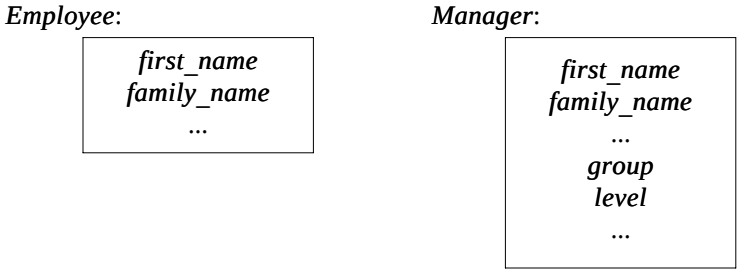
```
struct Manager : public Employee {
        set<Employee*> group;
        short level;
        // ...
};
```

The ***Manager*** is *derived* from ***Employee***, and conversely, ***Employee*** is a *base class* for ***Manager***. The class ***Manager*** has the members of class ***Employee*** (***name***, ***age***, etc.) in addition to its own members (***group***, ***level***, etc.).

Derivation is often represented graphically by a pointer from the derived class to its base class indicating that the derived class refers to its base (rather than the other way around):

A derived class is often said to inherit properties from its base, so the relationship is also called *inheritance*. A base class is sometimes called a *superclass* and a derived class a *subclass*. This terminology, however, is confusing to people who observe that the data in a derived class object is a superset of the data of an object of its base class. A derived class is larger than its base class in the sense that it holds more data and provides more functions.

A popular and efficient implementation of the notion of derived classes has an object of the derived class represented as an object of the base class, with the information belonging specifically to the derived class added at the end. For example:



Deriving ***Manager*** from ***Employee*** in this way makes ***Manager*** a subtype of ***Employee*** so that a ***Manager*** can be used wherever an ***Employee*** is acceptable. For example, we can now create a list of ***Employee***s, some of whom are ***Manager***s:

```
void f(Manager m1, Employee e1)
{
        list<Employee*> elist;

        elist.push_front(&m1);
        elist.push_front(&e1);
        // ...
}
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

A ***Manager*** is (also) an ***Employee***, so a ***Manager**** can be used as a ***Employee****. However, an ***Employee*** is not necessarily a ***Manager***, so an ***Employee**** cannot be used as a ***Manager****. In general, if a class ***Derived*** has a public base class (§15.3) ***Base***, then a ***Derived**** can be assigned to a variable of type ***Base**** without the use of explicit type conversion. The opposite conversion, from ***Base**** to ***Derived****, must be explicit. For example:

```
void g(Manager mm, Employee ee)
{
    Employee* pe = &mm;      // ok: every Manager is an Employee
    Manager* pm = &ee;       // error: not every Employee is a Manager

    pm->level = 2;           // disaster: ee doesn't have a 'level'

    pm = static_cast<Manager*>(pe);     // brute force: works because pe points
                                        // to the Manager mm

    pm->level = 2;       // fine: pm points to the Manager mm that has a 'level'
}
```

In other words, an object of a derived class can be treated as an object of its base class when manipulated through pointers and references. The opposite is not true. The use of ***static_cast*** and ***dynamic_cast*** is discussed in §15.4.2.

Using a class as a base is equivalent to declaring an (unnamed) object of that class. Consequently, a class must be defined in order to be used as a base (§5.7):

```
class Employee;      // declaration only, no definition

class Manager : public Employee {   // error: Employee not defined
    // ...
};
```

12.2.1 Member Functions [derived.member]

Simple data structures, such as ***Employee*** and ***Manager***, are really not that interesting and often not particularly useful. We need to give the information as a proper type that provides a suitable set of operations that present the concept, and we need to do this without tying us to the details of a particular representation. For example:

```
class Employee {
    string first_name, family_name;
    char middle_initial;
    // ...
public:
    void print() const;
    string full_name() const
        { return first_name + ' ' + middle_initial + ' ' + family_name; }
    // ...
};
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
class Manager : public Employee {
        // ...
public:
        void print() const;
        // ...
};
```

A member of a derived class can use the public – and protected (see §15.3) – members of its base class as if they were declared in the derived class itself. For example:

```
void Manager::print() const
{
        cout << "name is " << full_name() << '\n';
        // ...
}
```

However, a derived class cannot use a base class' private names:

```
void Manager::print() const
{
        cout << " name is " << family_name << '\n';     // error!
        // ...
}
```

This second version of ***Manager*::*print*()** will not compile. A member of a derived class has no special permission to access private members of its base class, so ***family_name*** is not accessible to ***Manager*::*print*()**.

This comes as a surprise to some, but consider the alternative: that a member function of a derived class could access the private members of its base class. The concept of a private member would be rendered meaningless by allowing a programmer to gain access to the private part of a class simply by deriving a new class from it. Furthermore, one could no longer find all uses of a private name by looking at the functions declared as members and friends of that class. One would have to examine every source file of the complete program for derived classes, then examine every function of those classes, then find every class derived from those classes, etc. This is, at best, tedious and often impractical. Where it is acceptable, ***protected*** – rather than ***private*** – members can be used. A protected member is like a public member to a member of a derived class, yet it is like a private member to other functions (see §15.3).

Typically, the cleanest solution is for the derived class to use only the public members of its base class. For example:

```
void Manager::print() const
{
        Employee::print();   // print Employee information
        cout << level;       // print Manager-specific information
        // ...
}
```

Note that **::** must be used because ***print*()** has been redefined in ***Manager***. Such reuse of names is typical. The unwary might write this:

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
void Manager::print() const
{
    print();    // oops!

    // print Manager-specific information
}
```

and find the program involved in an unexpected sequence of recursive calls.

12.2.2 Constructors and Destructors [derived.ctor]

Some derived classes need constructors. If a base class has constructors, then a constructor must be invoked. Default constructors can be invoked implicitly. However, if all constructors for a base require arguments, then a constructor for that base must be explicitly called. Consider:

```
class Employee {
    string first_name, family_name;
    short department;
    // ...
public:
    Employee(const string& n, int d);
    // ...
};

class Manager : public Employee {
    set<Employee*> group;    // people managed
    short level;
    // ...
public:
    Manager(const string& n, int d, int lvl);
    // ...
};
```

Arguments for the base class' constructor are specified in the definition of a derived class' constructor. In this respect, the base class acts exactly like a member of the derived class (§10.4.6). For example:

```
Employee::Employee(const string& n, int d)
    : family_name(n), department(d)          // initialize members
{
    // ...
}

Manager::Manager(const string& n, int d, int lvl)
    : Employee(n,d),          // initialize base
      level(lvl)              // initialize members
{
    // ...
}
```

A derived class constructor can specify initializers for its own members and immediate bases only; it cannot directly initialize members of a base. For example:

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
Manager::Manager(const string& n, int d, int lvl)
    : family_name(n),    // error: family_name not declared in manager
      department(d),     // error: department not declared in manager
      level(lvl)
{
    // ...
}
```

This definition contains three errors: it fails to invoke ***Employee´s*** constructor, and twice it attempts to initialize members of ***Employee*** directly.

Class objects are constructed from the bottom up: first the base, then the members, and then the derived class itself. They are destroyed in the opposite order: first the derived class itself, then the members, and then the base. Members and bases are constructed in order of declaration in the class and destroyed in the reverse order. See also §10.4.6 and §15.2.4.1.

12.2.3 Copying [derived.copy]

Copying of class objects is defined by the copy constructor and assignments (§10.4.4.1). Consider:

```
class Employee {
    // ...
    Employee& operator=(const Employee&);
    Employee(const Employee&);
};

void f(const Manager& m)
{
    Employee e = m;      // construct e from Employee part of m
    e = m;               // assign Employee part of m to e
}
```

Because the ***Employee*** copy functions do not know anything about ***Manager***s, only the ***Employee*** part of a ***Manager*** is copied. This is commonly referred to as *slicing* and can be a source of surprises and errors. One reason to pass pointers and references to objects of classes in a hierarchy is to avoid slicing. Other reasons are to preserve polymorphic behavior (§2.5.4, §12.2.6) and to gain efficiency.

12.2.4 Class Hierarchies [derived.hierarchy]

A derived class can itself be a base class. For example:

```
class Employee { /* ... */ };
class Manager : public Employee { /* ... */ };
class Director : public Manager { /* ... */ };
```

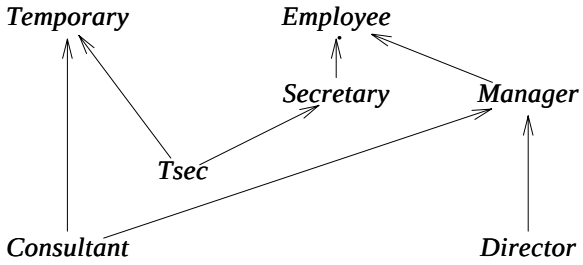
Such a set of related classes is traditionally called a *class hierarchy*. Such a hierarchy is most often a tree, but it can also be a more general graph structure. For example:

```
class Temporary { /* ... */ };
class Secretary : public Employee { /* ... */ };
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
class Tsec : public Temporary, public Secretary { /* ... */ };
class Consultant : public Temporary, public Manager { /* ... */ };
```

Or graphically:



Thus, as is explained in detail in §15.2, C++ can express a directed acyclic graph of classes.

12.2.5 Type Fields [derived.typefield]

To use derived classes as more than a convenient shorthand in declarations, we must solve the following problem: Given a pointer of type *base**, to which derived type does the object pointed to really belong? There are four fundamental solutions to the problem:

[1] Ensure that only objects of a single type are pointed to (§2.7, Chapter 13).
[2] Place a type field in the base class for the functions to inspect.
[3] Use ***dynamic_cast*** (§15.4.2, §15.4.5).
[4] Use virtual functions (§2.5.5, §12.2.6).

Pointers to base classes are commonly used in the design of *container classes* such as set, vector, and list. In this case, solution 1 yields homogeneous lists, that is, lists of objects of the same type. Solutions 2, 3, and 4 can be used to build heterogeneous lists, that is, lists of (pointers to) objects of several different types. Solution 3 is a language-supported variant of solution 2. Solution 4 is a special type-safe variation of solution 2. Combinations of solutions 1 and 4 are particularly interesting and powerful; in almost all situations, they yield cleaner code than do solutions 2 and 3.

Let us first examine the simple type-field solution to see why it is most often best avoided. The manager/employee example could be redefined like this:

```
struct Employee {
        enum Empl_type { M, E };
        Empl_type type;

        Employee() : type(E) { }

        string first_name, family_name;
        char middle_initial;

        Date hiring_date;
        short department;
        // ...
};
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
struct Manager : public Employee {
    Manager() { type = M; }

    set<Employee*> group;     // people managed
    short level;
    // ...
};
```

Given this, we can now write a function that prints information about each *Employee*:

```
void print_employee(const Employee* e)
{
    switch (e->type) {
    case Employee::E:
        cout << e->family_name << '\t' << e->department << '\n';
        // ...
        break;
    case Employee::M:
    {   cout << e->family_name << '\t' << e->department << '\n';
        // ...
        const Manager* p = static_cast<const Manager*>(e);
        cout << " level " << p->level << '\n';
        // ...
        break;
    }
    }
}
```

and use it to print a list of *Employee*s, like this:

```
void print_list(const list<Employee*>& elist)
{
    for (list<Employee*>::const_iterator p = elist.begin(); p!=elist.end(); ++p)
        print_employee(*p);
}
```

This works fine, especially in a small program maintained by a single person. However, it has the fundamental weakness in that it depends on the programmer manipulating types in a way that cannot be checked by the compiler. This problem is usually made worse because functions such as *print_employee()* are organized to take advantage of the commonality of the classes involved. For example:

```
void print_employee(const Employee* e)
{
    cout << e->family_name << '\t' << e->department << '\n';
    // ...
    if (e->type == Employee::M) {
        const Manager* p = static_cast<const Manager*>(e);
        cout << " level " << p->level << '\n';
        // ...
    }
}
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

Finding all such tests on the type field buried in a large function that handles many derived classes can be difficult. Even when they have been found, understanding what is going on can be difficult. Furthermore, any addition of a new kind of ***Employee*** involves a change to all the key functions in the system – the ones containing the tests on the type field. The programmer must consider every function that could conceivably need a test on the type field after a change. This implies the need to access critical source code and the resulting necessary overhead of testing the affected code. The use of an explicit type conversion is a strong hint that improvement is possible.

In other words, use of a type field is an error-prone technique that leads to maintenance problems. The problems increase in severity as the size of the program increases because the use of a type field causes a violation of the ideals of modularity and data hiding. Each function using a type field must know about the representation and other details of the implementation of every class derived from the one containing the type field.

It also seems that the existence of any common data accessible from every derived class, such as a type field, tempts people to add more such data. The common base thus becomes the repository of all kinds of ''useful information.'' This, in turn, gets the implementation of the base and derived classes intertwined in ways that are most undesirable. For clean design and simpler maintenance, we want to keep separate issues separate and avoid mutual dependencies.

12.2.6 Virtual Functions [derived.virtual]

Virtual functions overcome the problems with the type-field solution by allowing the programmer to declare functions in a base class that can be redefined in each derived class. The compiler and loader will guarantee the correct correspondence between objects and the functions applied to them. For example:

```
class Employee {
        string first_name, family_name;
        short department;
        // ...
public:
        Employee(const string& name, int dept);
        virtual void print() const;
        // ...
};
```

The keyword ***virtual*** indicates that ***print***`()` can act as an interface to the ***print***`()` function defined in this class and the ***print***`()` functions defined in classes derived from it. Where such ***print***`()` functions are defined in derived classes, the compiler ensures that the right ***print***`()` for the given ***Employee*** object is invoked in each case.

To allow a virtual function declaration to act as an interface to functions defined in derived classes, the argument types specified for a function in a derived class cannot differ from the argument types declared in the base, and only very slight changes are allowed for the return type (§15.6.2). A virtual member function is sometimes called a *method*.

A virtual function *must* be defined for the class in which it is first declared (unless it is declared to be a pure virtual function; see §12.3). For example:

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
void Employee::print() const
{
    cout << family_name << '\t' << department << '\n';
    // ...
}
```

A virtual function can be used even if no class is derived from its class, and a derived class that does not need its own version of a virtual function need not provide one. When deriving a class, simply provide an appropriate function, if it is needed. For example:

```
class Manager : public Employee {
    set<Employee*> group;
    short level;
    // ...
public:
    Manager(const string& name, int dept, int lvl);
    void print() const;
    // ...
};

void Manager::print() const
{
    Employee::print();
    cout << "\tlevel " << level << '\n';
    // ...
}
```

A function from a derived class with the same name and the same set of argument types as a virtual function in a base is said to *override* the base class version of the virtual function. Except where we explicitly say which version of a virtual function is called (as in the call **Employee::print()**), the overriding function is chosen as the most appropriate for the object for which it is called.

The global function **print_employee()** (§12.2.5) is now unnecessary because the **print()** member functions have taken its place. A list of **Employee**s can be printed like this:

```
void print_list(set<Employee*>& s)
{
    for (set<Employee*>::const_iterator p = s.begin(); p!=s.end(); ++p) // see §2.7.2
        (*p)->print();
}
```

or even

```
void print_list(set<Employee*>& s)
{
    for_each(s.begin(),s.end(),mem_fun(&Employee::print));    // see §3.8.5
}
```

Each **Employee** will be written out according to its type. For example:

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
int main()
{
    Employee e("Brown",1234);
    Manager m("Smith",1234,2);
    set<Employee*> empl;
    empl.push_front(&e);      // see §2.5.4
    empl.push_front(&m);
    print_list(empl);
}
```

produced:

```
Smith 1234
    level 2
Brown 1234
```

Note that this will work even if ***Employee*::*print_list*()** was written and compiled before the specific derived class ***Manager*** was even conceived of! This is a key aspect of classes. When used properly, it becomes the cornerstone of object-oriented designs and provides a degree of stability to an evolving program.

Getting ''the right'' behavior from ***Employee***'s functions independently of exactly what kind of ***Employee*** is actually used is called *polymorphism*. A type with virtual functions is called a *polymorphic type*. To get polymorphic behavior in C++, the member functions called must be ***virtual*** and objects must be manipulated through pointers or references. When manipulating an object directly (rather than through a pointer or reference), its exact type is known by the compilation so that run-time polymorphism is not needed.

Clearly, to implement polymorphism, the compiler must store some kind of type information in each object of class ***Employee*** and use it to call the right version of the virtual function ***print*()**. In a typical implementation, the space taken is just enough to hold a pointer (§2.5.5). This space is taken only in objects of a class with virtual functions – not in every object, or even in every object of a derived class. You pay this overhead only for classes for which you declare virtual functions. Had you chosen to use the alternative type-field solution, a comparable amount of space would have been needed for the type field.

Calling a function using the scope resolution operator `::` as is done in ***Manager*::*print*()** ensures that the virtual mechanism is not used. Otherwise, ***Manager*::*print*()** would suffer an infinite recursion. The use of a qualified name has another desirable effect. That is, if a ***virtual*** function is also ***inline*** (as is not uncommon), then inline substitution can be used for calls specified using `::`. This provides the programmer with an efficient way to handle some important special cases in which one virtual function calls another for the same object. The ***Manager*::*print*()** function is an example of this. Because the type of the object is determined in the call of ***Manager*::*print*()**, it need not be dynamically determined again for the resulting call of ***Employee*::*print*()**.

It is worth remembering that the traditional and obvious implementation of a virtual function call is simply an indirect function call (§2.5.5), so efficiency concerns should not deter anyone from using a virtual function where an ordinary function call would be acceptably efficient.

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

12.3 Abstract Classes [derived.abstract]

Many classes resemble class ***Employee*** in that they are useful both as themselves and also as bases for derived classes. For such classes, the techniques described in the previous section suffice. However, not all classes follow that pattern. Some classes, such as class ***Shape***, represent abstract concepts for which objects cannot exist. A ***Shape*** makes sense only as the base of some class derived from it. This can be seen from the fact that it is not possible to provide sensible definitions for its virtual functions:

```
class Shape {
public:
    virtual void rotate(int) { error("Shape::rotate"); } // inelegant
    virtual void draw() { error("Shape::draw"); }
    // ...
};
```

Trying to make a shape of this unspecified kind is silly but legal:

```
Shape s; // silly: ''shapeless shape''
```

It is silly because every operation on ***s*** will result in an error.

A better alternative is to declare the virtual functions of class ***Shape*** to be *pure virtual functions*. A virtual function is ''made pure'' by the initializer **= *0***:

```
class Shape {          // abstract class
public:
    virtual void rotate(int) = 0;     // pure virtual function
    virtual void draw() = 0;          // pure virtual function
    virtual bool is_closed() = 0;     // pure virtual function
    // ...
};
```

A class with one or more pure virtual functions is an *abstract class*, and no objects of that abstract class can be created:

```
Shape s; // error: variable of abstract class Shape
```

An abstract class can be used only as an interface and as a base for other classes. For example:

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

class Circle : public Shape {
public:
    void rotate(int) { }                  // override Shape::rotate
    void draw();                          // override Shape::draw
    bool is_closed() { return true; }     // override Shape::is_closed

    Circle(Point p, int r);
private:
    Point center;
    int radius;
};
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

A pure virtual function that is not defined in a derived class remains a pure virtual function, so the derived class is also an abstract class. This allows us to build implementations in stages:

```
class Polygon : public Shape {                  // abstract class
public:
        bool is_closed() { return true; }       // override Shape::is_closed
        // ... draw and rotate not overridden ...
};

Polygon b;      // error: declaration of object of abstract class Polygon

class Irregular_polygon : public Polygon {
        list<Point> lp;
public:
        void draw();                    // override Shape::draw
        void rotate(int);               // override Shape::rotate
        // ...
};

Irregular_polygon poly(some_points);    // fine (assume suitable constructor)
```

An important use of abstract classes is to provide an interface without exposing any implementation details. For example, an operating system might hide the details of its device drivers behind an abstract class:

```
class Character_device {
public:
        virtual int open(int opt) = 0;
        virtual int close(int opt) = 0;
        virtual int read(char* p, int n) = 0;
        virtual int write(const char* p, int n) = 0;
        virtual int ioctl(int ...) = 0;
        virtual ~Character_device() { }      // virtual destructor
};
```

We can then specify drivers as classes derived from ***Character_device***, and manipulate a variety of drivers through that interface. The importance of virtual destructors is explained in §12.4.2.

With the introduction of abstract classes, we have the basic facilities for writing a complete program in a modular fashion using classes as building blocks.

12.4 Design of Class Hierarchies [derived.design]

Consider a simple design problem: provide a way for a program to get an integer value from a user interface. This can be done in a bewildering number of ways. To insulate our program from this variety, and also to get a chance to explore the possible design choices, let us start by defining our program's model of this simple input operation. We will leave until later the details of implementing it using a real user-interface system.

The idea is to have a class ***Ival_box*** that knows what range of input values it will accept. A program can ask an ***Ival_box*** for its value and ask it to prompt the user if necessary. In addition, a program can ask an ***Ival_box*** if a user changed the value since the program last looked at it.

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

Because there are many ways of implementing this basic idea, we must assume that there will be many different kinds of ***Ival_boxes***, such as sliders, plain boxes in which a user can type a number, dials, and voice interaction.

The general approach is to build a ''virtual user-interface system'' for the application to use. This system provides some of the services provided by existing user-interface systems. It can be implemented on a wide variety of systems to ensure the portability of application code. Naturally, there are other ways of insulating an application from a user-interface system. I chose this approach because it is general, because it allows me to demonstrate a variety of techniques and design tradeoffs, because those techniques are also the ones used to build ''real'' user-interface systems, and – most important – because these techniques are applicable to problems far beyond the narrow domain of interface systems.

12.4.1 A Traditional Class Hierarchy [derived.traditional]

Our first solution is a traditional class hierarchy as is commonly found in Simula, Smalltalk, and older C++ programs.

Class ***Ival_box*** defines the basic interface to all ***Ival_box***es and specifies a default implementation that more specific kinds of ***Ival_box***es can override with their own versions. In addition, we declare the data needed to implement the basic notion:

```
class Ival_box {
protected:
        int val;
        int low, high;
        bool changed;
public:
        Ival_box(int ll, int hh) { changed = false; val = low = ll; high = hh; }

        virtual int get_value() { changed = false; return val; }
        virtual void set_value(int i) { changed = true; val = i; }          // for user
        virtual void reset_value(int i) { changed = false; val = i; }       // for application
        virtual void prompt() { }
        virtual bool was_changed() const { return changed; }
};
```

The default implementation of the functions is pretty sloppy and is provided here primarily to illustrate the intended semantics. A realistic class would, for example, provide some range checking.

A programmer might use these ''*ival* classes'' like this:

```
void interact(Ival_box* pb)
{
        pb->prompt(); // alert user
        // ...
        int i = pb->get_value();
        if (pb->was_changed()) {
                // new value; do something
        }
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
        else {
                // old value was fine; do something else
        }
        // ...
}

void some_fct()
{
        Ival_box* p1 = new Ival_slider(0,5);        // Ival_slider derived from Ival_box
        interact(p1);

        Ival_box* p2 = new Ival_dial(1,12);
        interact(p2);
}
```

Most application code is written in terms of (pointers to) plain ***Ival_box***es the way ***interact***( ) is. That way, the application doesn't have to know about the potentially large number of variants of the ***Ival_box*** concept. The knowledge of such specialized classes is isolated in the relatively few functions that create such objects. This isolates users from changes in the implementations of the derived classes. Most code can be oblivious to the fact that there are different kinds of ***Ival_box***es.

To simplify the discussion, I do not address issues of how a program waits for input. Maybe the program really does wait for the user in ***get_value***( ), maybe the program associates the ***Ival_box*** with an event and prepares to respond to a callback, or maybe the program spawns a thread for the ***Ival_box*** and later inquires about the state of that thread. Such decisions are crucial in the design of user-interface systems. However, discussing them here in any realistic detail would simply distract from the presentation of programming techniques and language facilities. The design techniques described here and the language facilities that support them are not specific to user interfaces. They apply to a far greater range of problems.

The different kinds of ***Ival_box***es are defined as classes derived from ***Ival_box***. For example:

```
class Ival_slider : public Ival_box {
        // graphics stuff to define what the slider looks like, etc.
public:
        Ival_slider(int, int);

        int get_value();
        void prompt();
};
```

The data members of ***Ival_box*** were declared ***protected*** to allow access from derived classes. Thus, ***Ival_slider***::***get_value***( ) can deposit a value in ***Ival_box***::***val***. A ***protected*** member is accessible from a class' own members and from members of derived classes, but not to general users (see §15.3).

In addition to ***Ival_slider***, we would define other variants of the ***Ival_box*** concept. These could include ***Ival_dial***, which lets you select a value by turning a knob; ***flashing_ival_slider***, which flashes when you ask it to ***prompt***( ); and ***popup_ival_slider***, which responds to ***prompt***( ) by appearing in some prominent place, thus making it hard for the user to ignore.

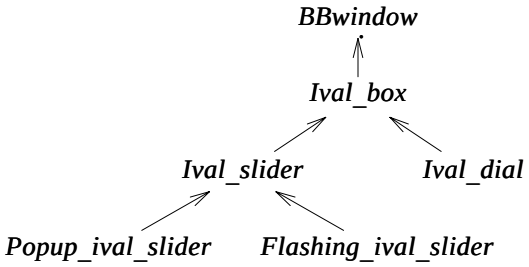
From where would we get the graphics stuff? Most user-interface systems provide a class defining the basic properties of being an entity on the screen. So, if we use the system from ''Big

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

Bucks Inc.,'' we would have to make each of our ***Ival_slider***, ***Ival_dial***, etc., classes a kind of ***BBwindow***. This would most simply be achieved by rewriting our ***Ival_box*** so that it derives from ***BBwindow***. In that way, all our classes inherit all the properties of a ***BBwindow***. For example, every ***Ival_box*** can be placed on the screen, obey the graphical style rules, be resized, be dragged around, etc., according to the standard set by the ***BBwindow*** system. Our class hierarchy would look like this:

```
class Ival_box : public BBwindow { /* ... */ }; // rewritten to use BBwindow
class Ival_slider : public Ival_box { /* ... */ };
class Ival_dial : public Ival_box { /* ... */ };
class Flashing_ival_slider : public Ival_slider { /* ... */ };
class Popup_ival_slider : public Ival_slider { /* ... */ };
```

or graphically:



12.4.1.1 Critique [derived.critique]

This design works well in many ways, and for many problems this kind of hierarchy is a good solution. However, there are some awkward details that could lead us to look for alternative designs.

We retrofitted ***BBwindow*** as the base of ***Ival_box***. This is not quite right. The use of ***BBwindow*** isn't part of our basic notion of an ***Ival_box***; it is an implementation detail. Deriving ***Ival_box*** from ***BBwindow*** elevated an implementation detail to a first-level design decision. That can be right. For example, using the environment defined by ''Big Bucks Inc.'' may be a key decision of how our organization conducts its business. However, what if we also wanted to have implementations of our ***Ival_box***es for systems from ''Imperial Bananas,'' ''Liberated Software,'' and ''Compiler Whizzes?'' We would have to maintain four distinct versions of our program:

```
class Ival_box : public BBwindow { /* ... */ };      // BB version
class Ival_box : public CWwindow { /* ... */ };      // CW version
class Ival_box : public IBwindow { /* ... */ };      // IB version
class Ival_box : public LSwindow { /* ... */ };      // LS version
```

Having many versions could result in a version-control nightmare.

Another problem is that every derived class shares the basic data declared in ***Ival_box***. That data is, of course, an implementation detail that also crept into our ***Ival_box*** interface. From a practical point of view, it is also the wrong data in many cases. For example, an ***Ival_slider*** doesn't need the value stored specifically. It can easily be calculated from the position of the slider when someone executes ***get_value***`()`. In general, keeping two related, but different, sets of data is

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

asking for trouble. Sooner or later someone will get them out of sync. Also, experience shows that novice programmers tend to mess with protected data in ways that are unnecessary and that cause maintenance problems. Data is better kept private so that writers of derived classes cannot mess with them. Better still, data should be in the derived classes, where it can be defined to match requirements exactly and cannot complicate the life of unrelated derived classes. In almost all cases, a protected interface should contain only functions, types, and constants.

Deriving from ***BBwindow*** gives the benefit of making the facilities provided by ***BBwindow*** available to users of ***Ival_box***. Unfortunately, it also means that changes to class ***BBwindow*** may force users to recompile or even rewrite their code to recover from such changes. In particular, the way most C++ implementations work implies that a change in the size of a base class requires a recompilation of all derived classes.

Finally, our program may have to run in a mixed environment in which windows of different user-interface systems coexist. This could happen either because two systems somehow share a screen or because our program needs to communicate with users on different systems. Having our user-interface systems ''wired in'' as the one and only base of our one and only ***Ival_box*** interface just isn't flexible enough to handle those situations.

12.4.2 Abstract Classes [derived.interface]

So, let's start again and build a new class hierarchy that solves the problems presented in the critique of the traditional hierarchy:

[1] The user-interface system should be an implementation detail that is hidden from users who don't want to know about it.

[2] The ***Ival_box*** class should contain no data.

[3] No recompilation of code using the ***Ival_box*** family of classes should be required after a change of the user-interface system.

[4] ***Ival_box***es for different interface systems should be able to coexist in our program.

Several alternative approaches can be taken to achieve this. Here, I present one that maps cleanly into the C++ language.

First, I specify class ***Ival_box*** as a pure interface:

```
class Ival_box {
public:
        virtual int get_value() = 0;
        virtual void set_value(int i) = 0;
        virtual void reset_value(int i) = 0;
        virtual void prompt() = 0;
        virtual bool was_changed() const = 0;
        virtual ~Ival_box() { }
};
```

This is much cleaner than the original declaration of ***Ival_box***. The data is gone and so are the simplistic implementations of the member functions. Gone, too, is the constructor, since there is no data for it to initialize. Instead, I added a virtual destructor to ensure proper cleanup of the data that will be defined in the derived classes.

The definition of ***Ival_slider*** might look like this:

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
class Ival_slider : public Ival_box, protected BBwindow {
public:
        Ival_slider(int, int);
        ~Ival_slider();

        int get_value();
        void set_value(int i);
        // ...
protected:
        // functions overriding BBwindow virtual functions
        // e.g. BBwindow::draw(), BBwindow::mouse1hit()
private:
        // data needed for slider
};
```

The derived class ***Ival_slider*** inherits from an abstract class (***Ival_box***) that requires it to implement the base class' pure virtual functions. It also inherits from ***BBwindow*** that provides it with the means of doing so. Since ***Ival_box*** provides the interface for the derived class, it is derived using ***public***. Since ***BBwindow*** is only an implementation aid, it is derived using ***protected*** (§15.3.2). This implies that a programmer using ***Ival_slider*** cannot directly use facilities defined by ***BBwindow***. The interface provided by ***Ival_slider*** is the one inherited by ***Ival_box***, plus what ***Ival_slider*** explicitly declares. I used ***protected*** derivation instead of the more restrictive (and usually safer) ***private*** derivation to make ***BBwindow*** available to classes derived from ***Ival_slider***.

Deriving directly from more than one class is usually called *multiple inheritance* (§15.2). Note that ***Ival_slider*** must override functions from both ***Ival_box*** and ***BBwindow***. Therefore, it must be derived directly or indirectly from both. As shown in §12.4.1.1, deriving ***Ival_slider*** indirectly from ***BBwindow*** by making ***BBwindow*** a base of ***Ival_box*** is possible, but doing so has undesirable side effects. Similarly, making the ''implementation class'' ***BBwindow*** a member of ***Ival_box*** is not a solution because a class cannot override virtual functions of its members (§24.3.4). Representing the window by a ***BBwindow**** member in ***Ival_box*** leads to a completely different design with a separate set of tradeoffs (§12.7[14], §25.7).

Interestingly, this declaration of ***Ival_slider*** allows application code to be written exactly as before. All we have done is to restructure the implementation details in a more logical way.

Many classes require some form of cleanup for an object before it goes away. Since the abstract class ***Ival_box*** cannot know if a derived class requires such cleanup, it must assume that it does require some. We ensure proper cleanup by defining a virtual destructor ***Ival_box::~Ival_box()*** in the base and overriding it suitably in derived classes. For example:

```
void f(Ival_box* p)
{
        // ...
        delete p;
}
```

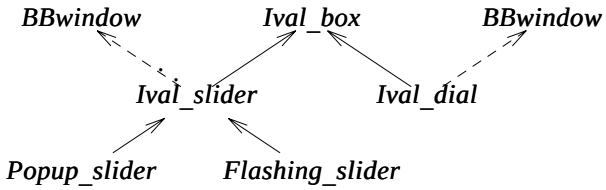
The ***delete*** operator explicitly destroys the object pointed to by ***p***. We have no way of knowing exactly to which class the object pointed to by ***p*** belongs, but thanks to ***Ival_box***'s virtual destructor, proper cleanup as (optionally) defined by that class' destructor will be called.

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

The ***Ival_box*** hierarchy can now be defined like this:

```
class Ival_box { /* ... */ };
class Ival_slider : public Ival_box, protected BBwindow { /* ... */ };
class Ival_dial : public Ival_box, protected BBwindow { /* ... */ };
class Flashing_ival_slider : public Ival_slider { /* ... */ };
class Popup_ival_slider : public Ival_slider { /* ... */ };
```

or graphically using obvious abbreviations:



I used a dashed line to represent protected inheritance. As far as general users are concerned, doing that is simply an implementation detail.

12.4.3 Alternative Implementations [derived.alt]

This design is cleaner and more easily maintainable than the traditional one – and no less efficient. However, it still fails to solve the version control problem:

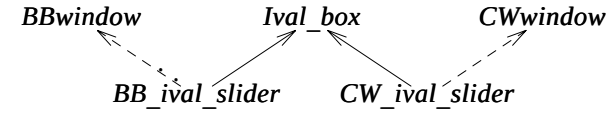
```
class Ival_box { /* ... */ };      // common
class Ival_slider : public Ival_box, protected BBwindow { /* ... */ }; // for BB
class Ival_slider : public Ival_box, protected CWwindow { /* ... */ }; // for CW
// ...
```

In addition, there is no way of having an ***Ival_slider*** for ***BBwindow***s coexist with an ***Ival_slider*** for ***CWwindow***s, even if the two user-interface systems could themselves coexist.

The obvious solution is to define several different ***Ival_slider*** classes with separate names:

```
class Ival_box { /* ... */ };
class BB_ival_slider : public Ival_box, protected BBwindow { /* ... */ };
class CW_ival_slider : public Ival_box, protected CWwindow { /* ... */ };
// ...
```

or graphically:



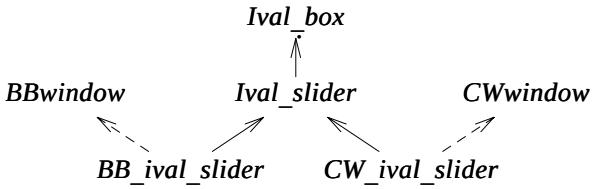
To further insulate our application-oriented ***Ival_box*** classes from implementation details, we can derive an abstract ***Ival_slider*** class from ***Ival_box*** and then derive the system-specific ***Ival_sliders*** from that:

```
class Ival_box { /* ... */ };
class Ival_slider : public Ival_box { /* ... */ };
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
class BB_ival_slider : public Ival_slider, protected BBwindow { /* ... */ };
class CW_ival_slider : public Ival_slider, protected CWwindow { /* ... */ };
// ...
```

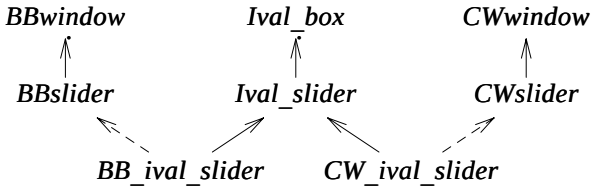
or graphically:



Usually, we can do better yet by utilizing more-specific classes in the implementation hierarchy. For example, if the ''Big Bucks Inc.'' system has a slider class, we can derive our ***Ival_slider*** directly from the ***BBslider***:

```
class BB_ival_slider : public Ival_slider, protected BBslider { /* ... */ };
class CW_ival_slider : public Ival_slider, protected CWslider { /* ... */ };
```

or graphically:



This improvement becomes significant where – as is not uncommon – our abstractions are not too different from the ones provided by the system used for implementation. In that case, programming is reduced to mapping between similar concepts. Derivation from general base classes, such as ***BBwindow***, is then done only rarely.

The complete hierarchy will consist of our original application-oriented conceptual hierarchy of interfaces expressed as derived classes:

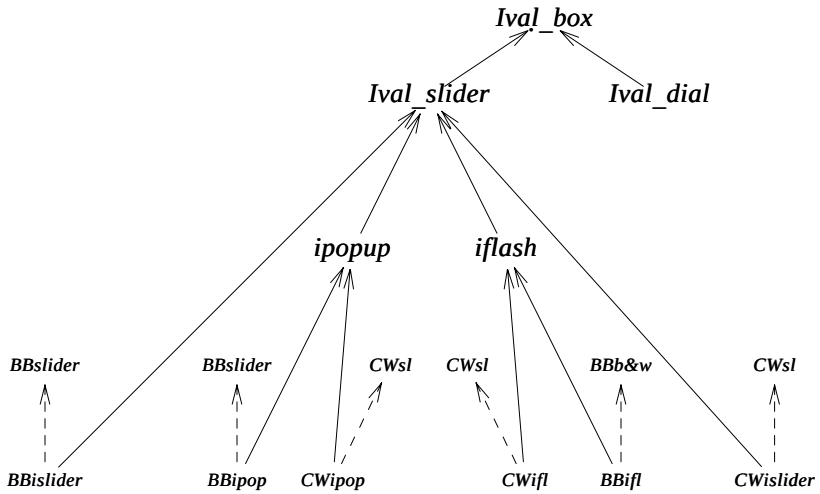
```
class Ival_box { /* ... */ };
class Ival_slider : public Ival_box { /* ... */ };
class Ival_dial : public Ival_box { /* ... */ };
class Flashing_ival_slider : public Ival_slider { /* ... */ };
class Popup_ival_slider : public Ival_slider { /* ... */ };
```

followed by the implementations of this hierarchy for various graphical user-interface systems, expressed as derived classes:

```
class BB_ival_slider : public Ival_slider, protected BBslider { /* ... */ };
class BB_flashing_ival_slider : public Flashing_ival_slider,
        protected BBwindow_with_bells_and_whistles { /* ... */ };
class BB_popup_ival_slider : public Popup_ival_slider, protected BBslider { /* ... */ };
class CW_ival_slider : public Ival_slider, protected CWslider { /* ... */ };
// ...
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

Using obvious abbreviations, this hierarchy can be represented graphically like this:



The original ***Ival_box*** class hierarchy appears unchanged surrounded by implementation classes.

12.4.3.1 Critique [derived.critique2]

The abstract class design is flexible and almost as simple to deal with as the equivalent design that relies on a common base defining the user-interface system. In the latter design, the windows class is the root of a tree. In the former, the original application class hierarchy appears unchanged as the root of classes that supply its implementations. From the application's point of view, these designs are equivalent in the strong sense that almost all code works unchanged and in the same way in the two cases. In either case, you can look at the ***Ival_box*** family of classes without bothering with the window-related implementation details most of the time. For example, we would not need to rewrite ***interact*****()** from §12.4.1 if we switched from the one class hierarchy to the other.

In either case, the implementation of each ***Ival_box*** class must be rewritten when the public interface of the user-interface system changes. However, in the abstract class design, almost all user code is protected against changes to the implementation hierarchy and requires no recompilation after such a change. This is especially important when the supplier of the implementation hierarchy issues a new ''almost compatible'' release. In addition, users of the abstract class hierarchy are in less danger of being locked into a proprietary implementation than are users of a classical hierarchy. Users of the ***Ival_box*** abstract class application hierarchy cannot accidentally use facilities from the implementation because only facilities explicitly specified in the ***Ival_box*** hierarchy are accessible; nothing is implicitly inherited from an implementation-specific base class.

12.4.4 Localizing Object Creation [derived.local]

Most of an application can be written using the ***Ival_box*** interface. Further, should the derived interfaces evolve to provide more facilities than plain ***Ival_box***, then most of an application can be written using the ***Ival_box***, ***Ival_slider***, etc., interfaces. However, the creation of objects must be

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

done using implementation-specific names such as ***CW_ival_dial*** and ***BB_flashing_ival_slider***. We would like to minimize the number of places where such specific names occur, and object creation is hard to localize unless it is done systematically.

As usual, the solution is to introduce an indirection. This can be done in many ways. A simple one is to introduce an abstract class to represent the set of creation operations:

```
class Ival_maker {
public:
    virtual Ival_dial* dial(int, int) =0;                    // make dial
    virtual Popup_ival_slider* popup_slider(int, int) =0; // make popup slider
    // ...
};
```

For each interface from the ***Ival_box*** family of classes that a user should know about, class ***Ival_maker*** provides a function that makes an object. Such a class is sometimes called a *factory*, and its functions are (somewhat misleadingly) sometimes called *virtual constructors* (§15.6.2).

We now represent each user-interface system by a class derived from ***Ival_maker***:

```
class BB_maker : public Ival_maker {            // make BB versions
public:
    Ival_dial* dial(int, int);
    Popup_ival_slider* popup_slider(int, int);
    // ...
};

class LS_maker : public Ival_maker {            // make LS versions
public:
    Ival_dial* dial(int, int);
    Popup_ival_slider* popup_slider(int, int);
    // ...
};
```

Each function creates an object of the desired interface and implementation type. For example:

```
Ival_dial* BB_maker::dial(int a, int b)
{
    return new BB_ival_dial(a,b);
}

Ival_dial* LS_maker::dial(int a, int b)
{
    return new LS_ival_dial(a,b);
}
```

Given a pointer to a ***Ival_maker***, a user can now create objects without having to know exactly which user-interface system is used. For example:

```
void user(Ival_maker* pim)
{
    Ival_box* pb = pim->dial(0,99);      // create appropriate dial
    // ...
}
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
BB_maker BB_impl; // for BB users
LS_maker LS_impl; // for LS users

void driver()
{
    user(&BB_impl);     // use BB
    user(&LS_impl);     // use LS
}
```

12.5 Class Hierarchies and Abstract Classes [derived.hier]

An abstract class is an interface. A class hierarchy is a means of building classes incrementally. Naturally, every class provides an interface to users and some abstract classes provide significant functionality to build from, but ''interface'' and ''building block'' are the primary roles of abstract classes and class hierarchies.

A classical hierarchy is a hierarchy in which the individual classes both provide useful functionality for users and act as building blocks for the implementation of more advanced or specialized classes. Such hierarchies are ideal for supporting programming by incremental refinement. They provide the maximum support for the implementation of new classes as long as the new class relates strongly to the existing hierarchy.

Classical hierarchies do tend to couple implementation concerns rather strongly with the interfaces provided to users. Abstract classes can help here. Hierarchies of abstract classes provide a clean and powerful way of expressing concepts without encumbering them with implementation concerns or significant run-time overheads. After all, a virtual function call is cheap and independent of the kind of abstraction barrier it crosses. It costs no more to call a member of an abstract class than to call any other ***virtual*** function.

The logical conclusion of this line of thought is a system represented to users as a hierarchy of abstract classes and implemented by a classical hierarchy.

12.6 Advice [derived.advice]

[1] Avoid type fields; §12.2.5.
[2] Use pointers and references to avoid slicing; §12.2.3.
[3] Use abstract classes to focus design on the provision of clean interfaces; §12.3.
[4] Use abstract classes to minimize interfaces; §12.4.2.
[5] Use abstract classes to keep implementation details out of interfaces; §12.4.2.
[6] Use virtual functions to allow new implementations to be added without affecting user code; §12.4.1.
[7] Use abstract classes to minimize recompilation of user code; §12.4.2.
[8] Use abstract classes to allow alternative implementations to coexist; §12.4.3.
[9] A class with a virtual function should have a virtual destructor; §12.4.2.
[10] An abstract class typically doesn't need a constructor; §12.4.2.
[11] Keep the representations of distinct concepts distinct; §12.4.1.1.

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

12.7 Exercises [derived.exercises]

1. (∗1) Define

```
class base {
public:
    virtual void iam() { cout << "base\n"; }
};
```

   Derive two classes from ***base***, and for each define ***iam()*** to write out the name of the class. Create objects of these classes and call ***iam()*** for them. Assign pointers to objects of the derived classes to ***base**** pointers and call ***iam()*** through those pointers.
2. (∗3.5) Implement a simple graphics system using whatever graphics facilities are available on your system (if you don't have a good graphics system or have no experience with one, you might consider a simple ''huge bit ASCII implementation'' where a point is a character position and you write by placing a suitable character, such as * in a position): ***Window(n,m)*** creates an area of size ***n*** times ***m*** on the screen. Points on the screen are addressed using (x,y) coordinates (Cartesian). A ***Window w*** has a current position ***w.current()***. Initially, ***current*** is ***Point(0,0)***. The current position can be set by ***w.current(p)*** where ***p*** is a ***Point***. A ***Point*** is specified by a coordinate pair: ***Point(x,y)***. A ***Line*** is specified by a pair of ***Point***s: ***Line(w.current(),p2);*** class ***Shape*** is the common interface to ***Dot***s, ***Line***s, ***Rectangle***s, ***Circle***s, etc. A ***Point*** is not a ***Shape***. A ***Dot***, ***Dot(p)*** can be used to represent a ***Point p*** on the screen. A ***Shape*** is invisible unless ***draw()***n. For example: ***w.draw(Circle(w.current(),10))***. Every ***Shape*** has 9 contact points: ***e*** (east), ***w*** (west), ***n*** (north), ***s*** (south), ***ne***, ***nw***, ***se***, ***sw***, and ***c*** (center). For example, ***Line(x.c(),y.nw())*** creates a line from ***x***'s center to ***y***'s top left corner. After ***draw()***ing a ***Shape*** the current position is the ***Shape***'s ***se()***. A ***Rectangle*** is specified by its bottom left and top right corner: ***Rectangle(w.current(),Point(10,10))***. As a simple test, display a simple ''child's drawing of a house'' with a roof, two windows, and a door.
3. (∗2) Important aspects of a ***Shape*** appear on the screen as a set of line segments. Implement operations to vary the appearance of these segments: ***s.thickness(n)*** sets the line thickness to ***0***, ***1***, ***2***, or ***3***, where ***2*** is the default and ***0*** means invisible. In addition, a line segment can be ***solid***, ***dashed***, or ***dotted***. This is set by the function ***Shape::outline()***.
4. (∗2.5) Provide a function ***Line::arrowhead()*** that adds arrow heads to an end of a line. A line has two ends and an arrowhead can point in two directions relative to the line, so the argument or arguments to ***arrowhead()*** must be able to express at least four alternatives.
5. (∗3.5) Make sure that points and line segments that fall outside the ***Window*** do not appear on the screen. This is often called ''clipping.'' As an exercise only, do not rely on the implementation graphics system for this.
6. (∗2.5) Add a ***Text*** type to the graphics system. A ***Text*** is a rectangular ***Shape*** displaying characters. By default, a character takes up one coordinate unit along each coordinate axis.
7. (∗2) Define a function that draws a line connecting two shapes by finding the two closest ''contact points'' and connecting them.
8. (∗3) Add a notion of color to the simple graphics system. Three things can be colored: the background, the inside of a closed shape, and the outlines of shapes.
9. (∗2) Consider:

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
class Char_vec {
        int sz;
        char element[1];
public:
        static Char_vec* new_char_vec(int s);
        char& operator[](int i) { return element[i]; }
        // ...
};
```

Define ***new_char_vec***( ) to allocate contiguous memory for a ***Char_vec*** object so that the elements can be indexed through ***element*** as shown. Under what circumstances does this trick cause serious problems?

10. (∗2.5) Given classes ***Circle***, ***Square***, and ***Triangle*** derived from a class ***Shape***, define a function ***intersect***( ) that takes two ***Shape**** arguments and calls suitable functions to determine if the two shapes overlap. It will be necessary to add suitable (virtual) functions to the classes to achieve this. Don't bother to write the code that checks for overlap; just make sure the right functions are called. This is commonly referred to as *double* ***dispatch*** or a *multi-method*.
11. (∗5) Design and implement a library for writing event-driven simulations. Hint: <***task.h***>. However, that is an old program, and you can do better. There should be a class ***task***. An object of class ***task*** should be able to save its state and to have that state restored (you might define ***task***::***save***( ) and ***task***::***restore***( )) so that it can operate as a coroutine. Specific tasks can be defined as objects of classes derived from class ***task***. The program to be executed by a task might be specified as a virtual function. It should be possible to pass arguments to a new task as arguments to its constructor(s). There should be a scheduler implementing a concept of virtual time. Provide a function ***task***::***delay***(***long***) that ''consumes'' virtual time. Whether the scheduler is part of class ***task*** or separate will be one of the major design decisions. The tasks will need to communicate. Design a class ***queue*** for that. Devise a way for a task to wait for input from several queues. Handle run-time errors in a uniform way. How would you debug programs written using such a library?
12. (∗2) Define interfaces for ***Warrior***, ***Monster***, and ***Object*** (that is a thing you can pick up, drop, use, etc.) classes for an adventure-style game.
13. (∗1.5) Why is there both a ***Point*** and a ***Dot*** class in §12.7[2]? Under which circumstances would it be a good idea to augment the ***Shape*** classes with concrete versions of key classes such as ***Line***.
14. (∗3) Outline a different implementation strategy for the ***Ival_box*** example (§12.4) based on the idea that every class seen by an application is an interface containing a single pointer to the implementation. Thus, each "interface class" will be a handle to an "implementation class," and there will be an interface hierarchy and an implementation hierarchy. Write code fragments that are detailed enough to illustrate possible problems with type conversion. Consider ease of use, ease of programming, ease of reusing implementations and interfaces when adding a new concept to the hierarchy, ease of making changes to interfaces and implementations, and need for recompilation after change in the implementation.

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.