5
Inheritance

Inheritance: base and derived classes

Without inheritance, each class had to be created from scratch. Within the {curly braces}, we had to declare each and every member of the new class:

```
class newclass {
    declaration for member 1;
    declaration for member 2;
    declaration for member 3;
    //etc.
};
```

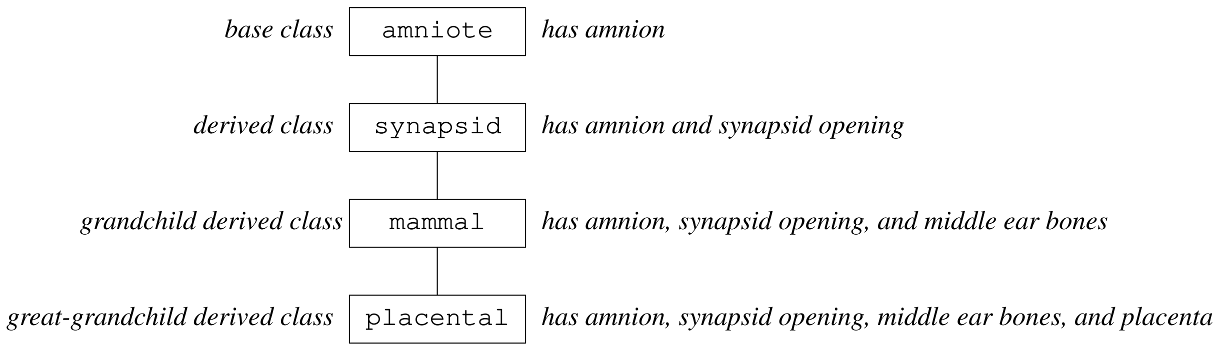
With inheritance, we can create a new class with a head start. The new class will automatically have all the members of an existing class, plus whatever additional members we'd like to add. It will therefore have all the functionality (i.e., the "look and feel") of the existing class, plus more.

The existing class is called the *base class;* the new one is called the *derived class.* (Java calls them the *superclass* and *subclass* respectively. But that's confusing, because the subclass has more members than the superclass.) In a diagram, the base class is always drawn above its derived class(es).

Pages 163−179 presented four reasons to package a chunk of code or functionality as a class. A fifth reason is because a class is the unit of syntax from which a derived class can inherit functionality.

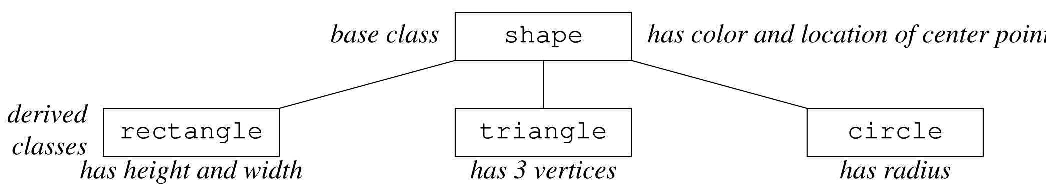
A tall, narrow tree

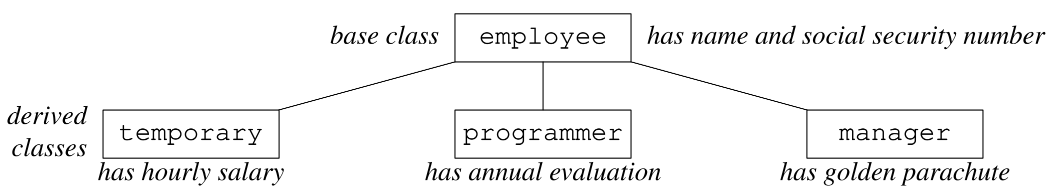
One use of inheritance is to build up a big class in layers, gradually adding more and more members. Consider the fossil halls on the fourth floor of the American Museum of Natural History, and their two movies narrated by Meryl Streep. The animals in each box in the diagram have all of the features of the ones in the boxes above it, plus more. For example, a synapsid animal has a synapsid opening in its skull. But it also inherits an amnion, at least if it's female, and is therefore also an amniote animal. This is the celebrated "is-a" relationship between a base class and its derived class: every synapsid is an amniote.



A wide, bushy tree

Another use of inheritance is to make specialized versions of an existing class. A drawing program might have a class for each kind of shape that can be displayed; a personnel program might have one for each kind of employee.





5.1 Inheritance without Virtual Functions

The following class will be our base class. It could have any name—it doesn't have to be named `base`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/base/base.h`

```
#ifndef BASEH
#define BASEH
#include <iostream>
#include "obj.h"
using namespace std;

class base {
    obj o1;
    obj o2;
public:
    base() {
        cout << "default construct base ";
        print();
        cout << "\n";
    }
```

```

    base(int initial_o1, int initial_o2): o1(initial_o1), o2(initial_o2) {
        cout << "construct base ";
        print();
        cout << "\n";
    }

    ~base() {
        cout << "destruct base ";
        print();
        cout << "\n";
    }

    void f() const {}

    void print() const {
        o1.print();
        cout << ", ";
        o2.print();
    }
};
#endif
```

The header file for a derived class must always `#include` the header file for its base class (line 3). It then `#include`'s the header files for the classes of its own data members (line 4). Our program would still happen to compile even without line 4, because fortunately the `obj.h` in line 4 has already been included by `base.h` in line 3. But we include line 4 because a professional never relies on luck.

The keyword `public` in line 6 announces that we are doing *public inheritance.* In public inheritance, the public members of the base class become public members of the derived class. With *private* inheritance, the public members of the base class become private members of the derived class (p. 581). For the time being, we'll stick with public inheritance.

The `print` member function of class `base` has therefore become a public member function of class `derived`. But this function, while adequate to print a `base`, will print only 50% of the data in a `derived` object. For this reason we must provide class `derived` with a bigger and better `print` function of its own, in line 15. And this is where all our trouble will begin.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/derived/derived.h`

```
 1 #ifndef DERIVEDH
 2 #define DERIVEDH
 3 #include "base.h"
 4 #include "obj.h"
 5
 6 class derived: public base {
 7     obj o3;
 8     obj o4;
 9 public:
10     derived();
11     derived(int initial_o1, int initial_o2, int initial_o3, int initial_o4);
12     ~derived();
13
14     void g() const {}
15     void print() const;
16 };
```

```
17 #endif
```

Some of the member functions of class `derived` are not inline (lines 10−12 and 15 of the above header file `derived.h`), so we also need the following `derived.C` *implementation file.*

The constructor for the derived class always begins by calling the constructor for the base class. If the latter requires arguments, they are passed with the colon in line 13. If the constructor for the base class requires no arguments, it can be called implicitly in line 5½.

The `derived::print` in lines 28–37 is commented out because it will not compile. The `o1` and `o2` members of class `base` are private, so they can be mentioned only in the member functions and friends of that class. Our workaround is the definition in lines 40–47, which begins by calling `base::print` to do half of its work. It is no sin for a member function of a derived class to call upon a member function of the base class, if we're happy with the member function of the base class as far as it goes. In fact, there is no other way for `derived::print` to print `o1` and `o2`.

Without the `base::`, line 42 would call `derived::print` and we'd go into an infinite loop. More about this shortly. An `operator<<` function for class `derived` will also come later.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/derived/derived.C`

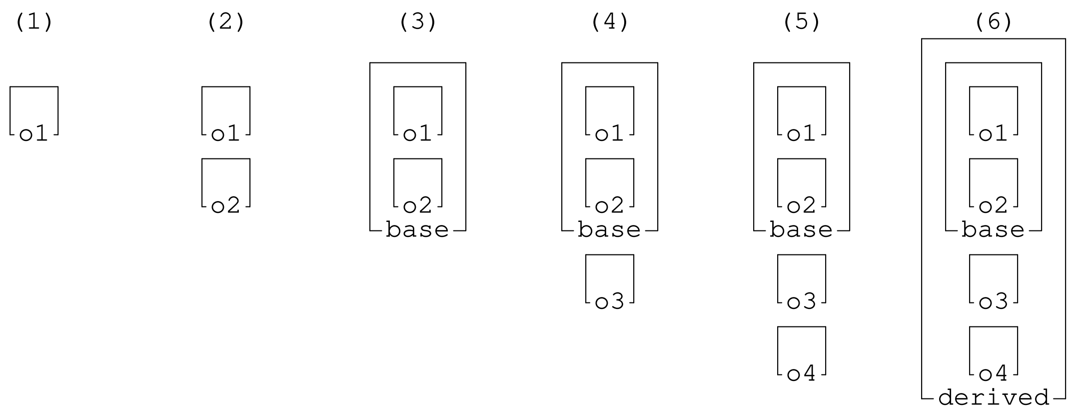
```
 1 #include <iostream>
 2 #include "derived.h"
 3 using namespace std;
 4
 5 derived::derived()
 6 {
 7     cout << "default construct derived ";
 8     print();
 9     cout << "\n";
10 }
11
12 derived::derived(int initial_o1, int initial_o2, int initial_o3, int initial_o4)
13     : base(initial_o1, initial_o2), o3(initial_o3), o4(initial_o4)
14 {
15     cout << "construct derived ";
16     print();
17     cout << "\n";
18 }
19
20 derived::~derived()
21 {
22     cout << "destruct derived ";
23     print();
24     cout << "\n";
25 }
26
27 /*
28 void derived::print() const
29 {
30     o1.print();   //won't compile, because o1 is private member of class base
31     cout << ", ";
32     o2.print();
33     cout << ", ";
34     o3.print();
35     cout << ", ";
36     o4.print();
```

```
37 }
38 */
39
40 void derived::print() const
41 {
42     base::print();   //will compile, because print is a public member of class base
43     cout << ", ";
44     o3.print();
45     cout << ", ";
46     o4.print();
47 }
```

A movie of a derived object being constructed

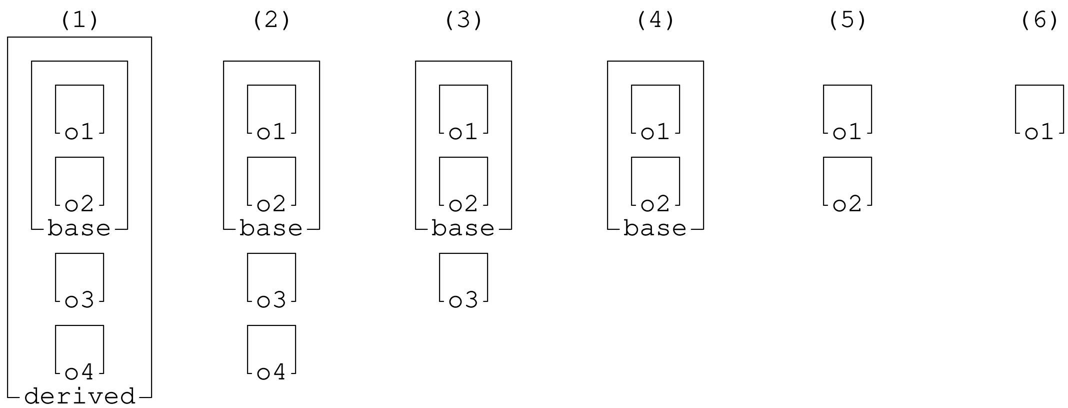
When we construct a `derived` object, the bodies of six constructors are executed. Once again, we make a series of detours before executing the body of the constructor for the derived object. First we call the constructor for the base object (steps 1 to 3), which makes two detours of its own (steps 1 and 2). Then we call the constructors for the additional data members introduced in the derived class (steps 4 and 5). Finally we execute the body of the constructor for the derived object (step 6). As with aggregation, the outermost object is always constructed last.

`o3` and `o4` are constructed in the order in which they are declared in lines 7–8 of the above `derived.h`. The order in which `o3` and `o4` are listed after the colon in line 13 of the above `derived.C` is irrelevant. Note one peculiarity of that line: `base` is the name of a *class,* while `o3` and `o4` are the names of *data members.*



A movie of a derived object being destructed

Six destructors are called, in exactly the reverse order, when we destruct a `derived` object. The outermost object is destructed first:



The output of lines 8 and 26 verifies the above diagrams:

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/derived/main.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "derived.h"
 4 using namespace std;
 5
 6 int main()
 7 {
 8     derived d(10, 20, 30, 40);
 9     cout << "\n";
10
11     d.g();              //Can use any public member of class derived.
12     d.f();              //Can also use any public member of class base.
13
14     d.print();          //Call the print member function of class derived.
15     cout << "\n";
16     d.base::print();    //Call the print member function of class base.
17     cout << "\n\n";
18
19     const derived *const p = &d; //same examples, but with p-> instead of d.
20
21     p->print();          //Call the print member function of class derived.
22     cout << "\n";
23     p->base::print();    //Call the print member function of class base.
24     cout << "\n\n";
25
26     return EXIT_SUCCESS;
27 }
```

`construct 10`	*Line 8 constructs* `d` *(six lines of output).*
`construct 20`	
`construct base 10, 20`	
`construct 30`	
`construct 40`	
`construct derived 10, 20, 30, 40`	
`10, 20, 30, 40`	*Line 14 calls* `derived::print`.
`10, 20`	*Line 16 calls* `base::print`.
`10, 20, 30, 40`	*Line 21 calls* `derived::print`.
`10, 20`	*Line 23 calls* `base::print`.
`destruct derived 10, 20, 30, 40`	*Line 26 destructs* `d` *(six lines of output).*
`destruct 40`	
`destruct 30`	
`destruct base 10, 20`	
`destruct 20`	
`destruct 10`	

Two groups of names in scope after the dot or arrow

If the name of a variable, function, enumeration, or typedef can be mentioned at a certain point in the program, we say that the name is *in scope* at that point. After the `d.`'s in lines 11, 12, 14 and 16 of the above `main.C`, and after the `p->`'s in lines 21 and 23, the following two groups of names are in scope:

(1) The members of the derived class.

(2) the members of the base class.

When identifying a name after a dot or arrow, the computer considers the members of the derived class before the members of the base class. This is significant when the base and derived classes have a member with the same name. For example, `d` has the old `print` inherited from the base class, and it also has the new `print` in the derived class. Since the members of the derived class are considered before those of the base class, the `print` member of the derived class will hide the `print` member of the base class in the above lines 14 and 21.

To prevent the hiding, we use the scope resolution operator `::` in the above lines 16 and 23. It has higher precedence than the two neighboring operators, dot and `(`function call`)`, so it need no parentheses of its own.

```
d . base :: print ()
```

Since both groups of names are in scope after the `d.`'s, we say that the object `d` is of class `base` as well as of class `derived`: it belongs to two different data types simultaneously. Of these two types, `derived` has everything that `base` has, plus more. We therefore say that `derived` is the *most derived* (i.e., biggest and best) type of `d`.

5.2 Scoping Rules for a Derived Class

Four groups of variables in scope in a member function of a derived class

In pp. 122−124, we saw that two groups of names are in scope in the body of a non-member function, and three groups are in scope in the body of a member function. In the body of a member function of a derived class, four groups of names are in scope:

(1) The local variables, typedefs, enumeraions, etc.

(2) The members of the derived class.

(3) The members of the base class.

(4) The variables that are neither local nor members of the derived or base classes, i.e., the globals.

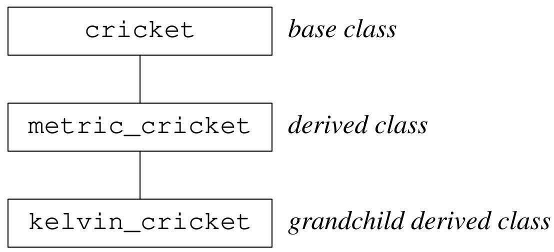
When identifying a variables in the body of a member function of a derived class, the computer first considers the locals, then the members of the derived class, then the members of the base class, and finally the globals. If two things have the same name, the local will therefore hide the member of the derived class (line 20), the member of the derived class will hide the member of the base class (line 23), and the member of the base class will hide the global (line 26). We would need the scope operator `::` to access the member of the derived class (line 21), the member of the base class (line 24), or the global (line 27).

```
 1 int k = 10;
 2
 3 class base {
 4 public:
 5     int j;
 6     int k;
 7 };
 8
 9 class derived: public base {
10     int i;
11     int j;
12 public:
13     void f() const;
14 };
15
16 void derived::f() const
17 {
18     int i = 20;
19
20     cout << i << "\n"              //the local i in line 18
21         << derived::i << "\n";     //the i member of class derived in line 10
22
23     cout << j << "\n"              //the j member of class derived in line 11
24         << base::j << "\n";        //the j member of class base in line 5
25
26     cout << k << "\n"              //the k member of class base in line 6
27         << ::k << "\n";            //the global k in line 1
28 }
```

In the body of a member function of a ‘‘grandchild’’ derived class, five groups of variables would be in scope. Et cetera.

A simple example of inheritance

A cricket will tell us the temperature if we count how fast it chirps. We will then build a series of bigger and better crickets.



—On the Web at
`http://i5.nyu.edu/~mm64/book/src/cricket/cricket.h`

```
#ifndef CRICKETH
#define CRICKETH

class cricket {
    unsigned chirps;   //per 15 seconds
public:
    cricket(unsigned initial_chirps): chirps(initial_chirps) {}
    double fahrenheit() const {return chirps + 39;}
};
#endif
```

We now derive a class that can do everything that `cricket` can do, plus more: it will give results in Celsius as well as Fahrenheit. Because it is derived with public inheritance, the public member `fahrenheit` of class `cricket` is also a public member of class `metric_cricket`.

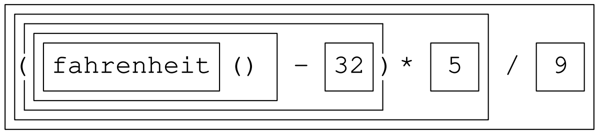
—On the Web at
`http://i5.nyu.edu/~mm64/book/src/cricket/metric_cricket.h`

```
 1 #ifndef METRIC_CRICKETH
 2 #define METRIC_CRICKETH
 3 #include "cricket.h"
 4
 5 class metric_cricket: public cricket {
 6 public:
 7     metric_cricket(unsigned initial_chirps): cricket(initial_chirps) {}
 8     double celsius() const {return (fahrenheit() - 32) * 5 / 9;}
 9 };
10 #endif
```

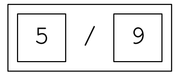
The above line 8 multiplies by `5/9` because a Celsius degree is wider than a Fahrenheit degree. To span the distance from freezing to boiling takes 180 Fahrenheit degrees (that’s 212 − 32), but only 100 Celsius degrees. But doesn’t line 8 have a bug? Won’t the integer division `5/9` truncate to zero?

No. In fact, there is no integer division in line 8. The `fahrenheit` function returns a `double`, causing the subtraction to yield a `double` result. The multiplication comes next (since multiplication and division have left-to-right associativity in C and C++), yielding a `double` product. The `/` therefore performs `double` division, which does not truncate.

The box diagram shows that there is no `5/9` subexpression of `(fahrenheit() - 32) * 5 / 9`.



Had 5/9 been a subexpression, it would have been boxed:



Finally, we derive a grandchild class:

—On the Web at
http://i5.nyu.edu/~mm64/book/src/cricket/kelvin_cricket.h

```
#ifndef KELVIN_CRICKETH
#define KELVIN_CRICKETH
#include "metric_cricket.h"

class kelvin_cricket: public metric_cricket {
public:
    kelvin_cricket(unsigned initial_chirps)
        : metric_cricket(initial_chirps) {}

    double kelvin() const {return celsius() + 273.15;}
};
#endif
```

—On the Web at
http://i5.nyu.edu/~mm64/book/src/cricket/main1.C

```
#include <iostream>
#include <cstdlib>
#include "kelvin_cricket.h"
using namespace std;

int main()
{
    cricket buddy(33);
    cout << "Fahrenheit == " << buddy.fahrenheit() << "\n\n";

    metric_cricket mc(33);
    cout << "Fahrenheit == " << mc.fahrenheit() << "\n"
        << "Celsius == " << mc.celsius() << "\n\n";

    kelvin_cricket kc(33);
    cout << "Fahrenheit == " << kc.fahrenheit() << "\n"
        << "Celsius == " << kc.celsius() << "\n"
        << "Kelvin == " << kc.kelvin() << "\n";

    return EXIT_SUCCESS;
}
```

```
Fahrenheit == 72        line 9

Fahrenheit == 72        line 12
Celsius == 22.2222      line 13; defaults to six significant digits

Fahrenheit == 72        line 16
Celsius == 22.2222      line 17
Kelvin == 295.372       line 18; still six significant digits
```

Add an extra data member to the derived class

In the above example, each derived class got an additional member function. In the next one, each derived class will get an additional data member. We saw the diagram on pp. 473–474.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/amniote/amniote.h`

```
#ifndef AMNIOTEH
#define AMNIOTEH

typedef int amnion_t;

class amniote {
    amnion_t amnion;
public:
    amniote(const amnion_t& initial_amnion): amnion(initial_amnion) {}
};
#endif
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/amniote/synapsid.h`

```
#ifndef SYNAPSIDH
#define SYNAPSIDH
#include "amniote.h"

typedef int opening_t;    //synapsid opening

class synapsid: public amniote {
    opening_t opening;
public:
    synapsid(const amnion_t& initial_amnion, const opening_t& initial_opening)
        : amniote(initial_amnion), opening(initial_opening) {}
};
#endif
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/amniote/mammal.h`

```
#ifndef MAMMALH
#define MAMMALH
#include "synapsid.h"

typedef int bones_t;      //middle ear bones: incus, malleus, stapes

class mammal: public synapsid {
    bones_t bones;
```

```
public:
    mammal(const amnion_t& initial_amnion,
        const opening_t& initial_opening,
        const bones_t& initial_bones)
        : synapsid(initial_amnion, initial_opening), bones(initial_bones) {}
};
#endif
```

—On the Web at
http://i5.nyu.edu/~mm64/book/src/amniote/placental.h

```
#ifndef PLACENTALH
#define PLACENTALH
#include "mammal.h"

typedef int placenta_t;

class placental: public mammal {
    placenta_t placenta;
public:
    placental(const amnion_t& initial_amnion,
        const opening_t& initial_opening,
        const bones_t& initial_bones,
        const placenta_t& initial_placenta)

        : mammal(initial_amnion, initial_opening, initial_bones),
        placenta(initial_placenta) {}
};
#endif
```

A manipulator for output and input

On pp. 361–362, we saw that classes `ostream` and `istream` have the member functions in lines 4 and 12.

```
 1 class ostream {
 2     //etc.
 3 public:
 4     ostream& operator<<(ostream& (*p)(ostream&)) {return p(*this);}
 5     ostream& operator<<(ios_base& (*p)(ios_base&)) {p(*this); return *this;}
 6     //etc.
 7 };
 8
 9 class istream {
10     //etc.
11 public:
12     istream& operator>>(istream& (*p)(istream&)) {return p(*this);}
13     istream& operator>>(ios_base& (*p)(ios_base&)) {p(*this); return *this;}
14     //etc.
15 };
```

The argument `p` in the above line 4 could point to a function such as the following.

```
16 ostream& hex(ostream& ost)
17 {
18     ost.setf(ios_base::hex, ios_base::basefield);
19     return ost;
```

```
20 }
```

The address of `hex` can be passed to the `operator<<` in line 4 by writing

```
21     cout << hex;                    //cout.operator<<(hex);
```

Similarly, the `p` in line 12 could point to another `hex` function.

```
22 istream& hex(istream& ist)
23 {
24     ist.setf(ios_base::hex, ios_base::basefield);
25     return ist;
26 }
```

The address of this `hex` can be passed to the `operator>>` in line 12 by writing

```
27     cin >> hex;                     //cin.operator>>(hex);
```

But there is no need to write the two `hex` functions in lines 16 and 22. Classes `ostream` and `istream` are derived from class `ios_base`, as we saw in our first inheritance diagram on pp. 383–385. We can therefore define a single `hex` function that accepts both types of stream.

```
28 ios_base& hex(ios_base& io)
29 {
30     io.setf(ios_base::hex, ios_base::basefield);
31     return io;
32 }
```

The address of this `hex` function can be passed to the `operator<<` and `operator>>` in the above lines 5 and 13.

```
33     cout << hex;        //cout.operator<<(hex); the operator<< in line 5
34     cin >> hex;         //cin.operator>>(hex);  the operator>> in line 13
```

Note that the `operator<<` in line 5 and the `operator>>` in line 13 ignore the return value of the `hex` in line 28. This `hex` returns an object of the base class `ios_base`, but the `operator<<` and `operator>>` must return an object of the derived class. They therefore return `*this`, the object they belong to.

▼ Homework 5.2a: input a point object in either coördinate system

Our `polar` and `cartesian` i/o manipulators in pp. 362–366 can be "output" to an `ostream`. Let them be "input" to an `istream` as well.

```
35     point A;
36     cin >> polar >> A >> cartesian;
37     cout << polar << A << cartesian << "\n";
```

Each object of class `istream` has the same expandable array that we had in class `ostream`. In fact, the array and its attendant functions `xalloc` and `iword` are actually members of class `ios_base`, and inherited by classes `ostream` and `istream`. To acknowledge its origin, and to avoid favoritism, change the `ostream::xalloc` to `ios_base::xalloc` in line 6 of the `point.C` in p. 364.

Our original `polar` and `cartesian` friend functions took and returned an `ostream`, just like our original `hex` in the above line 16. The expression `cout << polar` therefore called the `operator<<` in the above line 4. But now, `polar` and `cartesian` should accept either an `istream` or an `ostream`. Change the argument and return value of `polar` and `cartesian` to `ios_base`, the common ancestor of `ostream` and `istream`, as we did for the `hex` in the above line 28. The expression `cout << polar` will now call the `operator<<` in the above line 5, and `cin >> polar` will call the `operator<<` in the above line 13.

Our `operator>>` friend of class `point` will call the `iword` member function of the `istream` object, just as our `operator<<` called the `iword` member function of the `ostream` object. To store polar input into the `x`, `y` data members of the `point`, the `operator>>` friend of class `point` should use these conversions:

$$x = r \cos \theta$$

$$y = r \sin \theta$$

▲

5.3 Virtual Functions

Example 1: we know in advance which object is pointed to.

The `d` in line 1 is both a `derived` and a `base`. Thus the expression `&d` in line 5 is both a pointer to a `derived` and a pointer to a `base`. And since it is a pointer to a `base`, it can be stored into `p`.

In this simple example, it would be more natural to declare `p` to be a pointer to a `derived`. After all, we know in advance that it points to the `derived` object `d`. (“In advance” means when we write and compile the program.) But in a more realistic example, we might not know until runtime which object is pointed to by a pointer. In fact, we may not even know until runtime which class of object is pointed to. Our application might create one kind of object in response to a mouse move, another kind of object in response to a keystroke. We can’t predict what the user will do at runtime, so can’t predict which classes of object we’ll have to deal with.

`p` is declared to be a pointer to a `base` to allow it to point to any object of class `base`, or of any class derived from `base`. Had `p` been declared to be a pointer to a `derived`, it could not point to an object that was merely a `base`.

Will line 6 call `base::print` or `derived::print`? Does the name `print` in line 6 represent `base::print` or `derived::print`? When a name represents a function, we say that the name is *bound* to the function. To which function will the name `print` in line 6 be bound?

A case could be made for either binding. The `p` in line 6 is a“pointer to `base`”, suggesting that the name `print` in 6 should be bound to `base::print`. But the pointed-to object `d` in line 6 is a `derived`, suggesting that the `print` in 6 should be bound to `derived::print`.

Unfortunately, the definition of the language says that the binding of the name `print` in line 6 is determined by the data type of `p`, not the data type of `d`. The name `print` is bound to `base::print`, and line 6 calls this function. This is bad news, since `base::print` prints only half of the data in `d`.

In this dismal scenario, the binding—the decision as to which function is represented by the name `print`—is performed when the program is compiled. It is therefore called *early* or *static binding.* Line 6 is held hostage to the data type of `p`, and the data type of `d` is ignored. Is there any way to bind the name based on the data type of `d`?

```
1       derived d(10, 20, 30, 40);
2       d.print();    //Calls derived::print.
3       cout << "\n";
4
5       base *p = &d;
6       p->print();   //Calls base::print, but derived::print would be better.
7       cout << "\n";
8
9       //Exactly the same example, but with a reference instead of a pointer.
10      base& r = d;
11      r.print();    //Calls base::print, but derived::print would be better.
12      cout << "\n";
```

Virtual vs. non-virtual member functions

To make the above line 6 call `derived::print`, we must prefix the keyword `virtual` to the declaration of `base::print` in line 31 of `base.h` on p. 475:

```
31      virtual void print() const {
```

(We will also need the keyword `virtual` on the destructor at line 23 of `base.h`; see pp. 493–494.) This will cause the binding of the name `print` in line 6 to be determined by the data type of the pointed-to object `d`, not the data type of `p`. In addition, the binding will be performed at runtime, as line 6 is executed. This is called *late* or *dynamic binding.* If line 6 is executed more than once, the decision will be made afresh each time, based on the data type of whatever object `p` is pointing to during each execution. We will see this repeated execution in examples 3 and 4 below.

We could also prefix a `virtual` to the declaration of `derived::print` in line 15 of `derived.h`:

```
15      virtual void print() const;
```

but don't—it's not necessary and nobody does it. Since the two functions have the same name, arguments, and return type, the second function is automatically `virtual` too.

Example 2: we don't know in advance which object is pointed to.

From now on, we will assume that the declaration for `base::print` has the keyword `virtual`.

Almost every function call in C is statically bound: we can predict in advance which function will be called.

```
 1      printf("hello\n");
```

But the function call in line 9 is dynamically bound: we can't predict which function it will call. The decision has been deferred until runtime. In C this situation would be exotic, requiring a "pointer to a function". In C++, however, it is standard operating procedure. Be patient a moment and you'll see what it's for.

```
 2  #include <cstdlib>      //for rand
 3  using namespace std;
 4
 5      base b(10, 20);
 6      derived d(30, 40, 50, 60);
 7
 8      base *p = rand() % 2 == 0 ? &b : &d;
 9      p->print();          //Could call base::print or derived::print.
10      cout << "\n";
11
12      //Exactly the same example, but with a reference instead of a pointer.
13      base& r = rand() % 2 == 0 ? b : d;
14      r.print();           //Could call base::print or derived::print.
15      cout << "\n";
```

Example 3:
we don't know in advance which object is pointed to, and the statement is executed more than once.

Lines 9–12 construct objects of different data types. We'd like to store these objects in a container: an array, `vector`, or `list`. We can't quite do that, however, because all the items in a container must be of the same data type.

But we can do the next best thing: lines 14–19 create a *container of pointers* to the objects. All the pointers can be of the same data type, because as we've just seen, a pointer to a `base` can hold the address of either a `base` or a `derived`. (By the way, we can have an array of pointers but not an array of references. See p. 80.)

The loop in lines 22–25 prints all the objects. Each time line 23 is executed, it selects the appropriate `print` function for the object that `a[i]` points to. It will call `base::print` during the first two iterations, and `derived::print` during the next two.

When we write line 23, we may have no idea what object, or even what class of object, will be pointed to by `a[i]`. But—and this is the big idea—we don't need to know. We can rely on the ''virtual'' machinery to select the correct `print` function for us. (See p. 1012 for another use of this same scenario.)

Some people think of a virtual function as a ''polymorphic'' function: one which automatically changes its shape (i.e., the contents of its body) based on the class of the pointed-to object. But of course there is no such thing. A virtual function is actually a set of functions* that share the same name, argument types, return type, etc.: `base::print` and `derived::print`. Because of this agreement, every function in the family can be called by writing the same expression: `a[i]->print()`. When we write this expression, we don't need to know which function will be called at runtime. One member of the family will be selected for us automatically, determined by the data type of the object that `a[i]` points to at runtime.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/polymorphic/polymorphic3.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "base.h"
 4 #include "derived.h"
 5 using namespace std;
 6
 7 int main()
 8 {
 9     base b1(10, 20);
10     base b2(30, 40);
11     derived d1(50, 60, 70, 80);
12     derived d2(90, 100, 110, 120);
13
14     base *const a[] = {   //base is the "greatest common denominator"
15         &b1,
16         &b2,
17         &d1,
18         &d2
19     };
20     const size_t n = sizeof a / sizeof a[0];
21
22     for (size_t i = 0; i < n; ++i) {
23         a[i]->print();
24         cout << "\n";
25     }
26
27     cout << "\n";
28
29     //The same loop, but with a pointer p instead of a size_t i.
30     for (const base *const *p = a; p < a + n; ++p) {
31         (*p)->print();
32         cout << "\n";
33     }
34
```

* A template function will also be defined as a set of functions. See pp. 664–665.

```
35     return EXIT_SUCCESS;
36 }
```

```
10, 20                    not bothering to show output of constructors and destructors
30, 40
50, 60, 70, 80
90, 100, 110, 120

10, 20
30, 40
50, 60, 70, 80
90, 100, 110, 120
```

Without virtual functions, we'd have to write the following chain of `else-if`'s in place of line 23 (and line 31):

```
37         if (a[i] points to an object that is merely of class base) {
38             //call the base::print that belongs to that object
39             a[i]->print();
40         }
41
42         else if (a[i] points to an object of class derived) {
43             //call the derived::print that belongs to that object
44             reinterpret_cast<const derived *>(a[i])->print();
45         }
46
47         else {
48             output a runtime error message;
49         }
```

and you'd have to insert another `else if` every time you derived another class from class `base`.

We can't always anticipate which member functions should be declared `virtual`; more on this later. So why not be on the safe side and make *every* member function `virtual`, as in Java? (Since they're *all* virtual in Java, there's no keyword for "virtual" in that language.) Well, as the above list of `else-if`'s shows, a call to a virtual function does more work than a call to a non-virtual one. It is said that a call to a virtual function takes 1.6 times as long as a normal one. We're programming in C++ because we want speed.

Example 4: same moral as example 3.

Will line 28 (and 34) call `base::print` or `derived::print`? And when will the decision be made, i.e., when will the name `print` in line 28 (and 34) be bound?

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/polymorphic/polymorphic4.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "base.h"
 4 #include "derived.h"
 5 using namespace std;
 6
 7 void f(const base *p);
 8 void g(const base& r);
 9
10 int main()
11 {
```

```
12     base b(10, 20);
13     derived d(30, 40, 50, 60);
14
15     f(&b);
16     f(&d);
17
18     cout << "\n";
19
20     g(b);
21     g(d);
22
23     return EXIT_SUCCESS;
24 }
25
26 void f(const base *p)
27 {
28     p->print();
29     cout << "\n";
30 }
31
32 void g(const base& r)   //same function, but with a reference argument
33 {
34     r.print();
35     cout << "\n";
36 }
```

`10, 20`	*line 15 (not bothering to show output of constructors and destructors)*
`30, 40, 50, 60`	*line 16*
`10, 20`	*line 20*
`30, 40, 50, 60`	*line 21*

Warning: use pass-by-reference to avoid slicing

The function `f` in line 22 will accept a `base` or a `derived` via pass-by-value. But when line 14 gives it the `derived` object `d`, `f` will be aware only of the `base` that forms the core of `d`. The rest of `d` will be sliced off. `d` itself will be undamaged, but the derived part of `d` will be agnored by `f`. To remedy this, use the pass-by-reference in lines 28 and 34.

This kind of slicing is bad. There's a totally different kind of slicing that is good. See pp. 901–902.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/polymorphic/slice.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "derived.h"
 4 using namespace std;
 5
 6 void f(base b);
 7 void g(const base *b);
 8 void h(const base& b);
 9
10 int main()
11 {
12     derived d(10, 20, 30, 40);
```

```
13
14     f(d);
15     cout << "\n";
16     g(&d);
17     h(d);
18
19     return EXIT_SUCCESS;
20 }
21
22 void f(base b)
23 {
24     b.print();
25     cout << "\n";
26 }
27
28 void g(const base *p)
29 {
30     p->print();
31     cout << "\n";
32 }
33
34 void h(const base& r)   //same function; this time, argument is a reference
35 {
36     r.print();
37     cout << "\n";
38 }
```

We didn't bother to show the output of the constructor in line 12 and the destructor in line 19. We do show the output of the copy constructor and the destructor for the `base` object in line 22.

`copy construct 10`	*Line 14 didn't print all of* `d`*: it called* `base::print`.
`copy construct 20`	
`10, 20`	
`destruct base 10, 20`	
`destruct 20`	
`destruct 10`	
`10, 20, 30, 40`	*Line 16 printed all of* `d`*: it called* `derived::print`.
`10, 20, 30, 40`	*Line 17 printed all of* `d`*: it called* `derived::print`.

Object-oriented programming

Object-oriented programming will help you when

(1) You are working with objects of many different classes, and expect to add new classes in the future.

(2) When you write the program, you can't predict the exact (i.e., "most derived") class that each object will belong to.

(3) You are accessing the objects via pointers or references, rather than by name, as in the four examples. In fact, many objects have no name. Only objects created by declarations have names; ones created by `new` (the C++ equivalent of `malloc`) have none.

If all of these classes are derived from a common base class named `base`, you can declare a pointer that can point to an object belonging to any of them:

```
    base *p;
```

Then you can say `p->print()` and the correct function will be selected and called.

Object-oriented programming is the use of late binding to let line 23 on p. 488 do the work of the list of `if` statements in lines 37–49. Line 23 will decide as it runs which function to call: `base::print`, `derived::print`, etc. It could call a different function each time it is executed. And if additional classes are derived from the same base class, each with their own `print` member function, line 23 will call these new functions even without recompilation.

Late binding in C++ is performed with inheritance and virtual functions. If you use objects without inheritance and virtual functions, your programming is merely *object-based,* not *object-oriented.* Don't feel guilty: object-orientation is not always necessary. See the three conditions above.

Why not use aggregation instead of inheritance?

Thanks to inheritance, the expression in the celebrated line 23 above works for objects of either class `base` or of class `derived`. But suppose we had built class `derived` from class `base` using aggregation instead of inheritance:

```
1 //Alternative version of derived.h.
2 #ifndef DERIVED
3 #define DERIVED
4
5 #include <iostream>
6 #include "base.h"
7 #include "obj.h"
8 using namespace std;
9
10 class derived {
11 public:
12     base b;    //a data member instead of a base class, public for simplicity
13 private:
14     obj o3;
15     obj o4;
16 public:
17     derived(int initial_o1, int initial_o2, int initial_o3, int initial_o4);
18     void g() const {}
19     void print() const;
20 };
21 #endif
```

To call the `g` member function of each object (line 18 of the above `base.h`), we'd have to write the two different expressions in lines 37 and 39 below. We would therefore need the chain of `if`'s in lines 36–42 instead of the single expression in the above line 23.

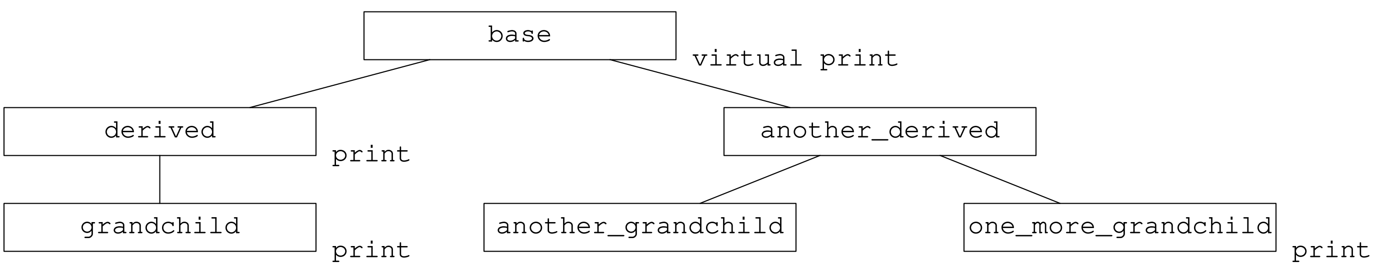
```
22     base b1(10, 20);
23     base b2(30, 40);
24     derived d1(50, 60, 70, 80);
25     derived d2(90, 100, 110, 120);
26
27     void *const a[] = {   //the greatest common denominator is now merely void *
28         &b1,
29         &b2,
30         &d1,
31         &d2
32     };
33     const size_t n = sizeof a / sizeof a[0];
34
35     for (size_t i = 0; i < n; ++i) {
```

```
        if (a[i] points to an object of class base) {
            a[i]->f();              //simplified pseudo-code
        } else if (a[i] points to an object of class derived) {
            a[i]->b.f();            //simplified pseudo-code
        } else {
            output a runtime error message;
        }

        cout << "\n";
    }
```

A family of functions

A virtual function is not a function. It is a family of functions, sharing the same name, argument types, return type, etc. One of these functions, marked with the keyword `virtual`, belongs to a base class; the others belong to classes derived therefrom.

If `base::print` is not adequate to print a derived object, we can provide a bigger and better `print` function in the derived class. That's what the following diagram does for class `derived`. But if there's a derived class for which `base::print` is adequate, you don't have to write a `print` function for that derived class. In the diagram, classes `another_derived` and `another_grandchild` rely on the original `base::print` function.



Five requirements for a virtual function

(1) All the functions that constitute a virtual function must have the same name.

(2) All the functions that constitute a virtual function must have the same argument types, although not necessarily the same default values for the arguments.

(3) All the functions that constitute a virtual function must either all be `const` or all be non-`const`. In other words, they must agree in the data type of their implicit argument, as well as their explicit arguments.

(4) All the functions that constitute a virtual function must have the same return type (with the exception on p. 523). If a base class has a `virtual` function and a derived class has a function with the same name, same argument types, but different return value type, you get an error message at compile time.

The functions that constitute a virtual function do not have to agree on their level of publicity. On p. 497 we will see an example where the function in the derived class is private, while the one in the base class is not private.

(5) A base class with a virtual function must have a virtual destructor if objects of the base class or of derived classes will be allocated dynamically. Will they be so allocated? No one knows yet. To be on the safe side, we prefix the keyword `virtual` to line 23 of `base.h` on p. 475. (There is no need for the keyword `virtual` on the destructor for class `derived`.) If the base class has no destructor, we write an empty one just to carry the keyword `virtual`.

```
    virtual ~base() {}
```

If class `base` has a virtual destructor, the `delete` in line 9 will call the correct destructor, either the one for `base` or the one for `derived`. If class `base` does not have a virtual destructor, the `delete` in line 9 would always call the destructor for class `base`, never the one for class `derived`. We would be held hostage to the data type of the expression `p` in line 9.

```
2     base *const p = rand() % 2 == 0
3         ? new base(10, 20)
4         : new derived(30, 40, 50, 60);
5
6     p->print();
7     cout << "\n";
8
9     delete p;
```

See pp. 501−503 for another situation in which the destructor for a base class must be virtual.

What happens if you don't fulfill the above requirement (2)

If a base class has a `virtual` function and a derived class has a function with the same name, same return type, but different argument types, you get no error message. The function in the derived class merely hides the function in the base class because of the scoping rules.

Here's an example. The base class has a function `f` that accepts an `int`. The user wants to give the derived class another function `f` that will accept a `char`. A worthy goal, but line 19 ends up calling line 12. Line 12 has eclipsed line 7.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/polymorphic/hide.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 class base {
 6 public:
 7     virtual void f(int i) const {cout << i << "\n";}        //Print in decimal.
 8 };
 9
10 class derived: public base {
11 public:
12     void f(char c) const {cout << "'" << c << "'" << "\n";} //Print a character.
13 };
14
15 int main()
16 {
17     derived d;
18     d.f('A');    //Calls derived::f.
19     d.f(66);     //I wish it called base::f, but it calls derived::f.
20
21     return EXIT_SUCCESS;
22 }
```

`'A'`	*line 18*
`'B'`	*line 19*

How to fix it

This is the one case where you *wouldn't* write the keyword `virtual` in front of the first of a pair of member functions with the same names, arguments, and return values, such as the ones in lines 7 and 12.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/polymorphic/supplement.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 class base {
 6 public:
 7     void f(int i) const {cout << i << "\n";}                 //Print in decimal.
 8 };
 9
10 class derived: public base {
11 public:
12     void f(int i) const {base::f(i);}                        //call-through
13     void f(char c) const {cout << "'" << c << "'" << "\n";} //Print a character.
14 };
15
16 int main()
17 {
18     derived d;
19     d.f('A');    //Calls the derived::f in line 13.
20     d.f(66);     //Calls the derived::f in line 12, which calls base::f.
21     return EXIT_SUCCESS;
22 }
```

`'A'`	*line 19*
`66`	*line 20*

See the more elegant solution on pp. 1025−1026.

Protected members

We already know that a member of a class can be public or private. It can also be *protected:* mentionable only by the member functions or friends of the class to which it belongs, and of any class derived therefrom, including grandchildren, great-grandchildren, etc. In public inheritance, the protected members of the base class become protected members of the derived class.

A non-`const` data member should never be protected, for then its value could be changed by indefinitely many functions throughout the program. The only protected members should be things that are intrinsically unchangeable: a member function, enumeration, `const` data member, or typedef or other data type.

There is one subtlety in the definition of a protected member. An object of a derived class can usually mention a protected member of its base class (line 16). We can even do this when the member belongs to a different object of the same derived class (line 18). But we cannot do this when the member belongs to an object that is *not* of the same derived class. Lines 21 and 23 try to mention the `f` that belongs to objects of classes `derived1` and `base`, but these `f`'s are unmentionable in a member function of class `derived2`. (Line 24 fails for the same reason as line 23. It's ironic, because 24 is only doing the same thing we did in 16.) This restriction will come back to haunt us on p. 579.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/polymorphic/protected.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 class base {
 6 protected:
 7     void f() const {cout << "base::f\n";}
 8 };
 9
10 class derived1: public base {
11 };
12
13 class derived2: public base {
14 public:
15     void g() const {
16         f();            //will compile
17         derived2 d2;
18         d2.f();         //will compile
19
20         derived1 d1;
21         //d1.f();       //won't compile
22         base b;
23         //b.f();        //won't compile
24         //static_cast<const base *>(this)->f();   //won't compile
25     }
26 };
27
28 int main()
29 {
30     derived2 d2;
31     d2.g();
32     return EXIT_SUCCESS;
33 }
```

```
base::f
base::f
```

There are no virtual friends.

We now provide the long overdue `operator<<` friend for classes `base` and `derived`. No one wants to have to call a member function named `print`.

Each of the following classes has different data members, so each requires a different `operator<<` function. It sounds like they should be a family of virtual functions. But there's a problem. Only a member function can be virtual, and an `operator<<` is not a member function of the class that it outputs. If the `operator<<` needs to mention the private members of the class, it must get that access by being a friend, not a member, of that class.* (If the `operator<<` does not need to mention the private members, it should be neither a member function nor a friend.)

The workaround is to write one non-virtual `operator<<` (lines 13–16) that will call a virtual member function to do all its work (the `print` in line 10). The derived class can then override the virtual

* Remember why? If an `operator<<` (or any other `operator` function) were a member function, the language would require it to be a member function of its left operand. But the object that we want to output is always the right operand of the <<.

function (line 21) but it won't have to override the `operator<<`.

The `base` argument of the `operator<<` in line 13 must be passed by reference. Were it passed by value, it would be sliced (pp. 490–491) and line 14 would always call the `base::print` in line 10, never the `derived::print` in line 21.

`base::print` must be public or protected because it is mentioned in line 21, a point outside the member functions and friends of class `base`. But `derived::print` can be private even though it may be called from line 14, a point outside the member functions and friends of class `derived`. Line 14 does not actually mention `derived::print`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/polymorphic/virtualfriend.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 class base {
 6     int i;
 7 protected:
 8     virtual void print(ostream& ost) const {ost << i;}
 9 public:
10     base(int initial_i): i(initial_i) {}
11     virtual ~base() {}
12
13     friend ostream& operator<<(ostream& ost, const base& b) {
14         b.print(ost);
15         return ost;
16     }
17 };
18
19 class derived: public base {
20     int j;
21     void print(ostream& ost) const {base::print(ost); ost << ", " << j;}
22 public:
23     derived(int initial_i, int initial_j): base(initial_i), j(initial_j) {}
24 };
25
26 int main()
27 {
28     base b(10);
29     const base *p = &b;
30     cout << *p << "\n";   //operator<<(cout, *p) << "\n";
31
32     derived d(20, 30);
33     p = &d;
34     cout << *p << "\n";   //operator<<(cout, *p) << "\n";
35
36     return EXIT_SUCCESS;
37 }
```

`10`	*lines 28–30*
`20, 30`	*lines 32–34*

▼ **Homework 5.3a: allocate an array of derived objects**

Even if the `base` class has a virtual destructor, why must the pointer `p` in lines 10–11 be declared as a pointer to a `derived`, not a pointer to a `base`? Assume that a `derived` object is larger than a `base` object.

```
1     base *p = new base(10);           //a base object
2     delete p;                         //okay
3
4     p = new base[3];                  //an array of base objects
5     delete[] p;                       //okay
6
7     p = new derived(20, 30);          //a derived object
8     delete p;                         //okay
9
10    p = new derived[3];               //an array of derived objects
11    delete[] p;                       //blows up
```

If we had written an `operator new[]` and `operator delete[]` member function for class `base`, lines 4 and 5 would have called them. We didn't, so these lines called the global `operator new[]` and `operator delete[]` in the C++ Standard Library.
▲

5.4 Hidden Pointers I: the Virtual Function Table (vtbl)

Back on p. 488 we saw the celebrated `->` operator in line 23 of `polymorphic3.C`. How can the expression `a[i]->print()` call two different member functions? How does it interrogate the target object and decide which `print` function to call?

My platform has a typical implementation. Every class that has virtual functions (including classes derived from those having virtual functions) has a table in memory called the *virtual table,* or *vtbl,* for that class. The class `base` in line 5 has a vtbl because it has virtual functions; the class `derived` in line 29 has a vtbl because it has virtual functions inherited from class `base`.

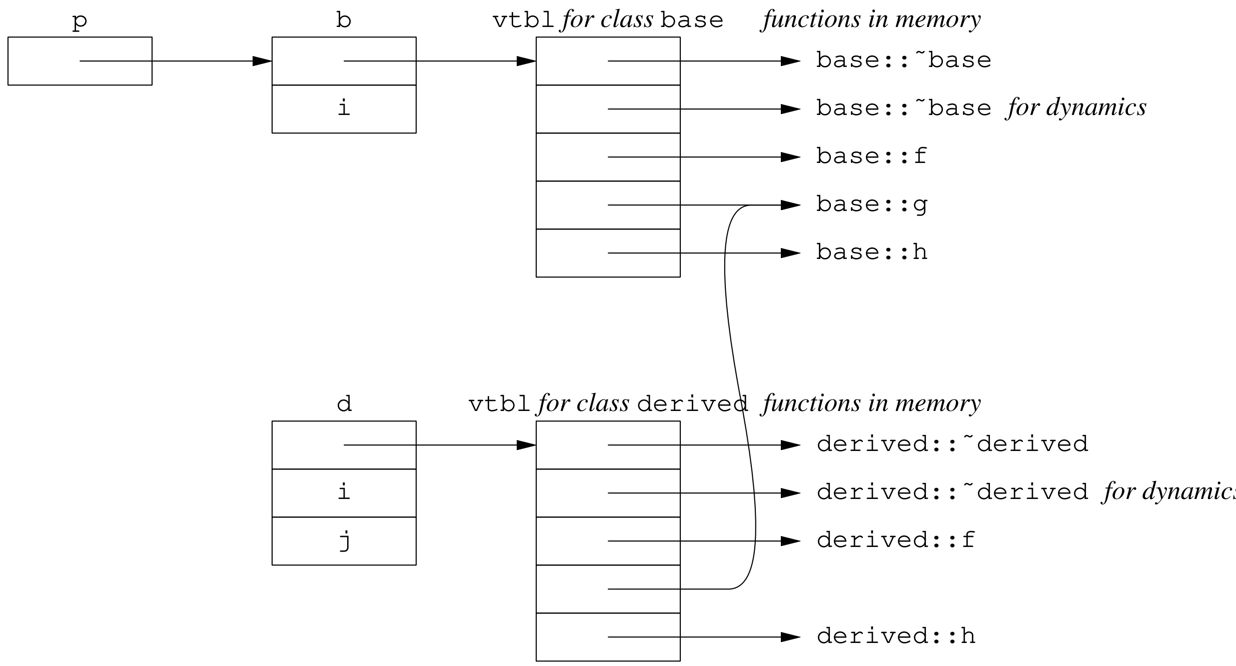
There is exactly one vtbl for each class that has virtual functions, and the vtbl for the class is shared by all the objects of the class. For example, all the object of class `base` share one vtbl, and all the objects of class `derived` share another vtbl.

Every object that has virtual functions begins with a pointer to the vtbl for the most derived class of that object. For example, the object `b` in line 41 is merely a `base`; it begins with a pointer to the vtbl for class `base`. The object `d` in line 45 is both a `base` and a `derived`; it begins with a pointer to the vtbl for class `derived`. See the following diagrams of `b` and `d`, and the value of the `sizeof`'s in lines 42 and 46.

At first glance, the vtbl looks like an array of pointers. But the pointers may be of different types, so the vtbl actually has to be a structure whose fields are pointers. For each virtual function in the class, there is a field containing a pointer to the function that is most appropriate for the class. For example, in the vtbls for class `base` and every class derived therefrom, the third field points to functions that belong to the virtual function `f`. The third field of the vtbl for class `base` points to `base::f`, and the third field in the vtbl for class `derived` points to `derived::f`. The fourth field in both vtbls points to `base::g`, since this function was never overridden in class `derived`; see the spline in the diagram.

Now we can trace how the celebrated `->` operator did its work in the `a[i]->print()` in line 23 of `polymorphic3.C` on p. 488. We will use the simpler example `p->f()` in line 54. The first time this expression is executed, `p` points to the `base` object `b`. See the following diagram.

The `->` performs three dereferences. First we dereference the `p`, which gets us to the object `b` (or to `d`, the second time this expression is executed). Then we dereference the pointer in the object, which gets us to the vtbl for the most derived class of the object. Finally, we dereference the pointer in the third field of the vtbl, the field for the virtual function `f`, which gets us to either `base::f` or `derived::f`.



C++ is sufficiently low-level to let us read an object's vtbl and manually call the functions to which the vtbl points. The structure in line 16 shows the layout of the vtbl for class `base` on my platform. We can use the same layout for any derived class that has no additional virtual functions. Class `derived`, for example, has no virtual functions other than the `f`, `g`, `h`, and destructor that it inherits from class `base`.

The structure in line 16 describes the vtbls for classes `base` and `derived`. It will be seen that the first (or only) argument in lines 17–21 is declared to be a pointer to `base`. In a call to a function in the `base` vtbl , this argument will point to a `base` object. In a call to a function in the `derived` vtbl , this argument will point to a `derived` object. No explicit casting is necessary to make this work.

The four virtual functions are of different types, so the corresponding fields in the `layout` structure had to be pointers to functions of different types. For example, the `base::f` in line 11 and `derived::f` in 33 take a read/write pointer to a `base` and return `void`, and the `ptr_to_f` in line 19 is a pointer to this type of function. On the other hand, the `base::g` in line 12 takes a read-only pointer to a base and returns `void`, and the `ptr_to_g` in line 20 is a pointer to this type of function. On my platform, a vtbl begins with pointers to two different implementations of the destructor: one for objects that are not dynamically allocated, and one for objects that are.

The structure in line 24 shows the layout of a `base` object in memory on my platform. Its first field is a pointer to the vtbl for class base; its second field is the data member `i`.

The `p` in line 52 is a pointer to a `base` object (which might also be a `derived` object). The `*p` in line 59 is the base object itself. To give us a clean notation for accessing the fields of the object, line 59 lets line 60 pretend that there is a `layout` structure named `lay` in memory exactly where the object is. (`lay` is merely a reference. Had `lay` been an actual structure, we would have incurred the expense of copying the imagined structure into `lay`.)

To give us a clean notation for accessing the fields of the vtbl, line 60 lets lines 63–65 pretend that there is a `vtbl` structure named `v` in memory exactly where the vtbl is. Lines 63–65 call the functions whose addresses are in the vtbl. They do the same thing as lines 54–56, in the sense that touching an electrode to the leg of a dissected frog makes the muscles do the same thing as when a live frog jumps. To make it easy to tally the three dereferences, we wrote them with explicit asterisks in lines 59, 60 and 63. The last asterisk can be implicit, as in the comment in line 63. The -> operator in line 54 performs these three dereferences.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/vtbl/vtbl.C`

```
#include <iostream>
#include <cstdlib>
using namespace std;

class base {
    int i;
public:
    base(int initial_i): i(initial_i) {}
    virtual ~base() {}

    virtual void f() {cout << "base::f\n";}
    virtual void g() const {cout << "base::g\n";}
    virtual int h(int n) const {cout << "base::h\n"; return i + n;}
};

struct vtbl {     //of a base object
    void (*ptr_to_destructor)(base *);
    void (*ptr_to_dynamic_destructor)(base *);
    void (*ptr_to_f)(base *);    //ptr_to_f is a pointer to a function
    void (*ptr_to_g)(const base *);
    int  (*ptr_to_h)(const base *, int);
};

struct layout {   //of a base object
    const vtbl *ptr_to_vtbl;
    int i;
};

class derived: public base {
    int j;
public:
    derived(int initial_i, int initial_j): base(initial_i), j(initial_j) {}
    void f() {cout << "derived::f\n";}
    int h(int n) const {cout << "derived::h\n"; return j + n;}
};

void print(base *p);

int main()
{
    base b(10);
    cout << "sizeof b == " << sizeof b << "\n";
    print(&b);

    derived d(20, 30);
    cout << "sizeof d == " << sizeof d << "\n";
    print(&d);

    return EXIT_SUCCESS;
}

void print(base *p)
{
    p->f();
```

```
55      p->g();
56      cout << p->h(40) << "\n";
57
58      //Unofficial; not portable.
59      const layout& lay = reinterpret_cast<const layout &>(*p);
60      const vtbl& v = *lay.ptr_to_vtbl;
61
62      //This is what the calls in lines 54–56 actually do.
63      (*v.ptr_to_f)(p);                          //v.ptr_to_f(p);
64      (*v.ptr_to_g)(p);                          //v.ptr_to_g(p);
65      cout << (*v.ptr_to_h)(p, 40) << "\n\n";    //v.ptr_to_h(p, 40)
66  }
```

```
sizeof b == 8     sizeof (vtbl *) + sizeof (int)
base::f           Line 43 passes a base object to print.
base::g
base::h
50
base::f
base::g
base::h
50

sizeof d == 12    sizeof (vtbl *) + sizeof (int) + sizeof (int)
derived::f        Line 47 passes a derived object to print.
base::g
derived::h
70
derived::f
base::g
derived::h
70
```

Of course, the computer does not always need to use the vtbl. When an object is mentioned by name, rather than accessed through a pointer, we can see at compile time which `f` we are calling. There is no need at runtime to look up the address of the appropriate `f` in the object's vtbl.

```
    base b(10);
    b.f();

    derived d(20, 30);
    d.f();
```

5.5 Dynamic Allocation of Base and Derived Objects

In pp. 415–419 we wrote an `operator new` and `operator delete` function for allocating objects of one specific class. We now provide the same functions for class `base`, in lines 18 and 24. Our functions produce tracing output, but they defer the actual allocation and deallocation to the global `operator new` and `operator delete`. To call these global functions, we need the unary scope operator `::` in lines 19 and 26. Without it, our functions would call themselves and go into an infinite loop (p. 476).

An `operator delete` for a base class can have an extra argument that we didn't have before, the `size_t n` in line 24. Like the `size_t n` in line 18, it gives the size of the object being allocated and deallocated. But now these arguments give the *total* size of the object, including the size of any derived object in which it is embedded. To ensure that the correct sizes are passed to these functions, the base class must have a virtual destructor. The output shows that on my platform, a `base` is eight bytes (the four-byte `i` plus four bytes of overhead) and a `derived` is 12 (`i` and `j`, plus the overhead).

Our simple `operator new` and `operator delete` merely print these numbers. A more sophisticated pair of functions, like the ones in pp. 415–419, could use them to perform their own allocation.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/polymorphic/new.C`

```
#include <iostream>
#include <cstdlib>
#include <new>
using namespace std;

const char *progname;
void my_new_handler();

class base {
    int i;
public:
    base(int initial_i = 0): i(initial_i) {
        cout << "construct base " << i << "\n";
    }

    virtual ~base() {cout << "destruct base " << i << "\n";}

    void *operator new(size_t n) {
        void *const p = ::operator new(n);
        cout << "base::operator new(" << n << ") returns " << p << "\n";
        return p;
    }

    void operator delete(void *p, size_t n) {
        cout << "base::operator delete(" << p << ", " << n << ")\n";
        ::operator delete(p);
    }
};

class derived: public base {
    int j;
public:
    derived(int initial_i = 0, int initial_j = 0)
        : base(initial_i), j(initial_j) {
        cout << "construct derived " << initial_i << " " << j << "\n";
    }

    ~derived() {cout << "destruct derived " << j << "\n";}
};

int main(int argc, char **argv)
{
    progname = argv[0];
    set_new_handler(my_new_handler);
```

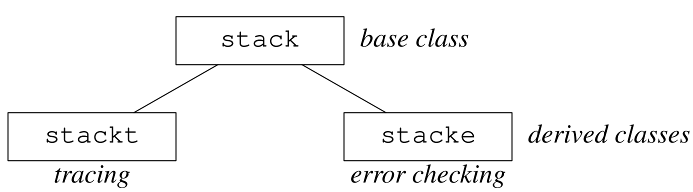
```
45
46     base *p = new base(10);
47     delete p;
48
49     cout << "\n";
50
51     p = new derived(10, 20);
52     delete p;
53
54     return EXIT_SUCCESS;
55 }
56
57 void my_new_handler()
58 {
59     cerr << progname << ": out of store\n";
60     exit(EXIT_FAILURE);
61 }
```

`base::operator new(8) returns 0x22280`	*line 46 allocates a* `base` *object*
`construct base 10`	
`destruct base 10`	*line 47 deallocates the* `base` *object*
`base::operator delete(0x22280, 8)`	
`base::operator new(12) returns 0x23a98`	*line 51 allocates a* `derived` *object*
`construct base 10`	
`construct derived 10 20`	
`destruct derived 20`	*line 52 deallocates the* `derived` *object*
`destruct base 10`	
`base::operator delete(0x23a98, 12)`	

▼ **Homework 5.5a: does the base class destructor have to be virtual?**

Let the destructor for the above class `base` in line 16 be non-virtual. Will line 52 still call the destructor for class `derived`? Are the correct `n` arguments still passed to `base::operator delete`?
▲

A simple example of inheritance with virtual functions



Here's a bare-bones version of the stack we saw on pp. 149–154 and 172–174. We'll use inheritance to build classes with additional features: tracing for debugging, and error checking. (I concede that in real life, no one would write the base class stack without error checking.)

For the data type `size_t` in lines 7 and 9, see p. 66. For the initialization of the static data member `max_size` in line 7, see p. 238.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/stackt/stack.h`

```
1 #ifndef STACKH
```



```
#define STACKH
#include <cstddef>   //for size_t
using namespace std;

class stack {
    static const size_t max_size = 100;
    int a[max_size];
    size_t n;        //stack pointer: subscript of next free element
public:
    stack(): n(0) {}
    virtual ~stack() {}

    virtual void push(int i) {a[n++] = i;}
    virtual int pop() {return a[--n];}

    size_t size() const {return n;}
    static size_t capacity() {return max_size;}
};
#endif
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/stackt/main1.C`

```
#include <iostream>
#include <cstdlib>
using namespace std;
#include "stack.h"

int main()
{
    cout << "To hire a person, type their social security number.\n"
        "To fire the most recently hired person, type a zero.\n"
        "To quit, type a negative number.\n";

    ::stack s;   //Call the constructor for s with no arguments.

    for (;;) {
        int ss;            //uninitialized variable
        cin >> ss;
        if (ss < 0) {      //quit
            break;
        }

        if (ss > 0) {      //hire
            s.push(ss);
        } else {           //fire
            cout << "Firing number " << s.pop() << ".\n";
        }
    }

    return EXIT_SUCCESS;   //Call the destructor for s.
}
```

```
To hire a person, type their social security number.
To fire the most recently hired person, type a zero.
To quit, type a negative number.
10                         You type the numbers in italics.
20
30
0
Firing number 30.
0
Firing number 20.
40
0
Firing number 40.
0
Firing number 10.
-1
```

With classes `stackt` and `stacke` we can add functionality to class `stack` without having to change or duplicate the code in that class. For example, the author of `stackt` has no need to agonize again over whether the ++ should be prefix or postfix. We can let this sleeping dog lie in the base class `stack`.

In line 10, the constructor for `stackt` begins by calling the constructor for `stack` and passing it no arguments. This would still happen even if we didn't write the `stack(),`, so cross it out.

`stackt::push` begins by calling `stack::push`. As we have seen, there is no stigma attached to having a member function of the derived class call a member function of the base class to do part of its work, if we're happy with the member function of the base class as far as it goes. Remember, the member functions in lines 13 and 14 would go into an infinite loop if we forget the `::stack::` (p. 476.)

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/stackt/stackt.h`

```
 1 #ifndef STACKTH
 2 #define STACKTH
 3 #include <iostream>
 4 #include "stack.h"
 5 using namespace std;
 6
 7 class stackt: public ::stack {   //stack with tracing output
 8     ostream& ost;
 9 public:
10     stackt(ostream& initial_ost): stack(), ost(initial_ost) {ost << "stackt()\n";}
11     ~stackt() {ost << "~stackt()\n";}
12
13     void push(int i) {::stack::push(i); ost << "push(" << i << ")\n";}
14     int pop() {const int i = ::stack::pop(); ost << "pop(" << i << ")\n"; return i;}
15 };
16 #endif
```

We can remove the following line 7 entirely. Even without it, class `stacke` would still have a constructor that takes no arguments, which would call the constructor for class `stack` with no arguments.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/stackt/stacke.h`

```
 1 #ifndef STACKEH
 2 #define STACKEH
 3 #include "stack.h"
```

```

class stacke: public ::stack {   //stack with error checking
public:
    stacke(): stack() {}
    ~stacke();

    void push(int i);
    int pop();
};
#endif
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/stackt/stacke.C`

```
#include <iostream>
#include <cstdlib>
#include "stacke.h"
using namespace std;

stacke::~stacke()
{
    if (size() != 0) {
        cerr << "Warning: stack still contains " << size()
            << " values.\n";
    }
}

void stacke::push(int i)
{
    if (size() >= capacity()) {
        cerr << "Can’t push when size " << size() << " >= capacity "
            << capacity() << ".\n";
        exit(EXIT_FAILURE);
    }

    ::stack::push(i);
}

int stacke::pop()
{
    if (size() == 0) {
        cerr << "Can’t pop when size " << size() << " == 0.\n";
        exit(EXIT_FAILURE);
    }

    return ::stack::pop();
}
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/stackt/main2.C`

```
#include <iostream>
#include <cstdlib>

#include "stackt.h"
#include "stacke.h"
```

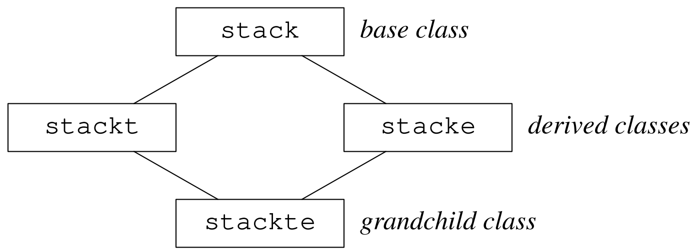
```
 6 using namespace std;
 7
 8 void f(::stack *p);
 9 void g(::stack& r);
10
11 int main()
12 {
13     ::stack s;
14     stackt st(cout);
15     stacke se;
16
17     f(&s);
18     f(&st);
19     f(&se);
20
21     cout << "\n";
22
23     g(s);
24     g(st);
25     g(se);
26
27     return EXIT_SUCCESS;
28 }
29
30 void f(::stack *p)
31 {
32     p->push(10);
33     cout << p->pop() << "\n";
34 }
35
36 void g(::stack& r)   //Exactly the same function, but with a reference argument.
37 {
38     r.push(20);
39     cout << r.pop() << "\n";
40 }
```

`stackt()`	*line 14*
`10`	*line 17*
`push(10)`	*line 18*
`pop(10)`	*line 18*
`10`	*line 18*
`10`	*line 19*
`20`	*line 23*
`push(20)`	*line 24*
`pop(20)`	*line 24*
`20`	*line 24*
`20`	*line 25*
`~stackt()`	*line 27*

A preview of multiple inheritance

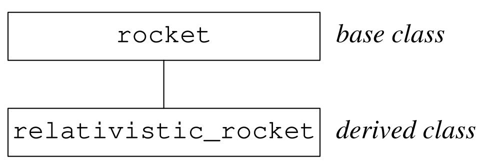
I'd like to create the grandchild class `stackte`, which would inherit debugging from class `stackt` and error checking from class `stacke`. Having two or more parents is called *multiple inheritance* as opposed to *single inheritance.* Java has only single inheritance.

To ensure that the grandchild `stackte` will inherit only a single copy of its grandparent `stack`, we will have to let `stackt` and `stacke` be *virtual base classes.* See pp. 554−557.



5.6 Partition the Code into Member Functions

Which member functions need to be marked as virtual?



It would seem that the biggest difficulty with object-oriented programming is to decide which member functions must be marked as virtual. We cannot always identify the member functions which are adequate to service the class they belong to, but which may be inadequate to service a derived class that no one has dreamt of yet.

The following class `rocket` illustrates this problem. It was written on the eve of Einstein's Special Theory of Relativity. No one suspected that the `length` member function in line 13 would become obsolete in a derived class, so how could they have known to mark it as virtual?

A class can't have a data member and a member function with the same name. That's why the names of the data members have underscores.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/relative/rocket.h`

```
 1 #ifndef ROCKETH
 2 #define ROCKETH
 3
 4 class rocket {
 5     double _length;   //in meters
 6     double _v;        //velocity in meters per second
 7 public:
 8     rocket(double initial_length, double initial_v)
 9         : _length(initial_length), _v(initial_v) {}
10
11     virtual ~rocket() {}
12
13     virtual double length() const {return _length;}
14     double v() const {return _v;}
```

```
};
#endif
```

To complete the example, we show a class derived after the publication of relativity theory. The speed of light is represented by the letter c (celerity); it is a member function in line 12, rather than a data member, because I wanted it to be public. Nothing can travel faster than light, and the constructor checks for this.

An object becomes shorter as it approaches the speed of light. The square root in the `length` function, $\sqrt{1 - \dfrac{v^2}{c^2}}$, has the value 1 when the rocket is stationary ($v == 0$), and shrinks toward zero as the rocket speeds up (v approaches c).

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/relative/relativistic_rocket.h`

```
 1 #ifndef RELATIVISTIC_ROCKETH
 2 #define RELATIVISTIC_ROCKETH
 3 #include <cmath>       //for sqrt
 4 #include "rocket.h"
 5 using namespace std;
 6
 7 class relativistic_rocket: public rocket {
 8 public:
 9     relativistic_rocket(double initial_length, double initial_v);
10
11     //speed of light in vacuum (meters per second)
12     static double c() {return 2.99792458e8;}
13
14     double length() const {
15         return rocket::length() * sqrt(1 - v() * v() / (c() * c()));
16     }
17 };
18 #endif
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/relative/relativistic_rocket.C`

```
#include <iostream>  //for cerr and <<
#include <cstdlib>   //for EXIT_FAILURE
#include <cmath>     //for abs
#include "relativistic_rocket.h"
using namespace std;

relativistic_rocket::relativistic_rocket(double initial_length, double initial_v)
    : rocket(initial_length, initial_v)
{
    if (abs(v()) >= c()) {
        cerr << "Velocity " << v() << " can't be >= the speed of light "
            << c() << ".\n";
        exit(EXIT_FAILURE);
    }
}
```

At $\dfrac{\sqrt{3}}{2}$ of the speed of light, a rocket shrinks to half of its original length. At $\dfrac{\sqrt{15}}{4}$ of the speed of light, it shrinks to one fourth; at $\dfrac{\sqrt{63}}{8}$, to one eighth.

The C++ Standard Library has three `sqrt` functions, taking arguments of type `float`, `double`, and `long double`. The computer would not have known which one to call had we written an argument of type `int` in lines 12–14.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/relative/main.C`

```
 1 #include <iostream>
 2 #include <iomanip>
 3 #include <cstdlib>
 4 #include <cmath>
 5 #include "relativistic_rocket.h"
 6 using namespace std;
 7
 8 int main()
 9 {
10     const double a[] = {   //fraction of the speed of light
11         0,
12         sqrt( 3.0) / 2,
13         sqrt(15.0) / 4,
14         sqrt(63.0) / 8,
15         1
16     };
17     const size_t n = sizeof a / sizeof a[0];
18
19     for (const double *p = a; p < a + n; ++p) {
20         const relativistic_rocket r(1, *p * relativistic_rocket::c());
21
22         cout << "velocity == " << scientific << r.v()
23             << resetiosflags(ios_base::floatfield) //turn off scientific
24             << ", length == " << r.length() << "\n";
25     }
26
27     return EXIT_SUCCESS;
28 }
```

```
velocity == 0.000000e+00, length == 1
velocity == 2.596279e+08, length == 0.5
velocity == 2.902728e+08, length == 0.25
velocity == 2.974411e+08, length == 0.125
Velocity 2.99792e+08 can't be >= the speed of light 2.99792e+08.
```

Divide the code of a class into member functions

As shown above, there may be no way to tell in advance which member functions should be marked as virtual. But the real difficulty is much worse. There may be no way to tell in advance how the code in a class should be divided up into member functions. The correct partitioning becomes obvious only when it is too late, after the incorrect design has been engraved in granite.

It will take a multi-part example to illustrate a problem as complicated as this. We will build up a date class that knows which years are leap years and which are not. In real life, we would write this as a single class. But to illustrate how to create software in layers, we will write it as a base class and a derived class.

The class `date` that we will start with does not know which years are leap. It assumes they all are. But it is intended to be a base class for a smarter class that does know which are leap. (It would seem to make more sense for the base class to assume by default that all years are non-leap. We'll see why it has to assume thaht all years are leap on p. 518, when we see the derived class.)

There are two constructors, with three arguments in line 30 and no arguments in line 34. Their common code has been factored out into a separate member function, the `install` in line 13.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/virtual1/date.h`

```
 1 #ifndef DATEH
 2 #define DATEH
 3 #include <iostream>
 4 #include <ctime>     //for time and localtime
 5 using namespace std;
 6
 7 class date {
 8     int year;    //Must construct data members in this order.
 9     int month;   //date::january to date::december inclusive
10     int day;     //1 to length[month] inclusive
11
12     static const int length[];
13     virtual void install(int m, int d, int y);
14 public:
15     enum month_t {   //indices into the length array
16         january = 1,
17         february,
18         march,
19         april,
20         may,
21         june,
22         july,
23         august,
24         september,
25         october,
26         november,
27         december
28     };
29
30     date(int initial_month, int initial_day, int initial_year) {
31         install(initial_month, initial_day, initial_year);
32     }
33
34     date();
35     virtual ~date() {}
36
37     int get_month() const {return month;}
38     int get_day()   const {return day;}
39     int get_year()  const {return year;}
40
41     virtual date& operator++();
42     virtual date& operator--();
43
44     friend ostream& operator<<(ostream& o, const date& d) {
45         return o << d.month << "/" << d.day << "/" << d.year;
46     }
47 };
48 #endif
```

The biggest member functions are `install` and the prefix `operator++` `operator--`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/virtual1/date.C`

```
#include <iostream>
#include <cstdlib>
#include "date.h"
using namespace std;

const int date::length[] = {
     0,   //dummy
    31,   //january
    29,   //february
    31,   //march
    30,   //april
    31,   //may
    30,   //june
    31,   //july
    31,   //august
    30,   //september
    31,   //october
    30,   //november
    31    //december
};

void date::install(int m, int d, int y)   //called by each constructor
{
    year = y;

    if (m < january || m > december) {
        cerr << "bad month " << m << "/" << d << "/" << y << "\n";
        exit(EXIT_FAILURE);
    }
    month = m;

    if (d < 1 || d > length[month]) {
        cerr << "bad day " << m << "/" << d << "/" << y << "\n";
        exit(EXIT_FAILURE);
    }
    day = d;
}

date::date()                              //Initialize to the current date.
{
    const time_t t = time(0);

    if (t == static_cast<time_t>(-1)) {
        cerr << "time failed\n";
        exit(EXIT_FAILURE);
    }

    const tm *const s = localtime(&t);
    install(s->tm_mon + 1, s->tm_mday, s->tm_year + 1900);
}

```

```
date& date::operator++()
{
    if (++day > length[month]) {
        day = 1;
        if (++month > december) {
            month = january;
            ++year;
        }
    }

    return *this;
}

date& date::operator--()
{
    if (--day < 1) {
        if (--month < january) {
            month = december;
            --year;
        }
        day = length[month];
    }

    return *this;
}
```

The above class `date` thinks every year is a leap year.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/virtual1/main.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "date.h"
 4 using namespace std;
 5
 6 int main()
 7 {
 8     date d(date::february, 28, 2014);
 9     cout << ++d << "\n";
10
11     return EXIT_SUCCESS;
12 }
```

The above line 9 behaves as if we had written

```
13     cout << d.operator++() << "\n";
```

which behaves as if we had written

```
14     operator<<(cout, d.operator++()) << "\n";
```

which behaves as if we had written

```
15     operator<<(operator<<(cout, d.operator++()), "\n");
```

As we have already seen, operator overloading gives us a nice, linear notation for deeply nested function calls.

```
2/29/2014
```

Reuse more of the base class

Some of the member functions of the above class are good enough to be inherited by a derived class that is responsible for knowing about leap years. An example is the default constructor in lines 39–50 of the above `date.C`: nothing in it would become obsolete when we have to handle leap years. But all three of the biggest member functions would have to be rewritten to handle leap years. That's why they were virtual:

(1) `install`

(2) prefix `operator++`

(3) prefix `operator--`

We could easily mark these functions as virtual, and override them in the derived class with ones that know about leap years. But these were the three biggest functions of the base class. And many more would have had to be virtual had we bothered to write them: `operator+=`, `operator-`, etc. The intent of inheritance is to let us *reuse* the base class in the derived class, not force us to *rewrite* the base class in the derived class. Apparently we have not yet achieved this goal.

Were we too hasty in resigning ourselves to rewriting the three big member functions in their entirety down in the derived class? Can any part of them be salvaged? In fact, almost every line can be. The only thing wrong with the prefix `operator++` in lines 52–63 of the above `date.C` is the expression `length[month]` in line 54. No other part of this function would become obsolete in a derived class responsible for knowing about leap years. Similarly, only one small would become obsolete—once again, the `length[month]`—in the prefix `operator--` and `install`.

To avoid rewriting the three big member functions, we simply excise the diseased tissue—the expression `length[month]`—and package it as a separate member function. We can reuse more of the base class code if we create a new member function to hold each piece of code that will become obsolete in the derived class. The following version of the base class still thinks that every year is a leap year, but now only one small member function (not counting the destructor) has to be virtual (line 39). It will be the only part of the base class that will have to be rewritten in the derived class.

To minimize the code that has to be rewritten in the derived classes, keep the job of the virtual function as simple as possible. Our `length` function merely returns a value; the `if`'s that use this value are in the non-virtual member functions of the base class. Other examples will be on pp. 519 and 534.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/virtual2/date.h`

```
 1 #ifndef DATEH
 2 #define DATEH
 3 #include <iostream>
 4 #include <ctime>
 5 using namespace std;
 6
 7 class date {
 8     int year;        //Must construct data members in this order.
 9     int month;       //date::january to date::december inclusive
10     int day;         //1 to length[month] inclusive
11
12     void install(int m, int d, int y);
13 public:
14     enum month_t {   //indices into the length array
15         january = 1,
16         february,
17         march,
18         april,
```

```
        may,
        june,
        july,
        august,
        september,
        october,
        november,
        december
    };

    date(int initial_month, int initial_day, int initial_year) {
        install(initial_month, initial_day, initial_year);
    }

    date();
    virtual ~date() {}

    int get_month() const {return month;}
    int get_day()   const {return day;}
    int get_year()  const {return year;}
    virtual int length() const;

    date& operator++();
    date& operator--();

    friend ostream& operator<<(ostream& o, const date& d) {
        return o << d.month << "/" << d.day << "/" << d.year;
    }
};
#endif
```

—On the Web at
http://i5.nyu.edu/~mm64/book/src/virtual2/date.C

```
#include <iostream>
#include <cstdlib>
#include "date.h"
using namespace std;

int date::length() const
{
    static const int a[] = {
         0,   //dummy
        31,   //january
        29,   //february
        31,   //march
        30,   //april
        31,   //may
        30,   //june
        31,   //july
        31,   //august
        30,   //september
        31,   //october
        30,   //november
        31    //december
```

```
    };

    return a[month];
}

void date::install(int m, int d, int y)
{
    year = y;

    if (m < january || m > december) {
        cerr << "bad month " << m << "/" << d << "/" << y << "\n";
        exit(EXIT_FAILURE);
    }
    month = m;

    if (d < 1 || d > length()) {
        cerr << "bad day " << m << "/" << d << "/" << y << "\n";
        exit(EXIT_FAILURE);
    }
    day = d;
}

date::date()   //Initialize to the current date.
{
    const time_t t = time(0);

    if (t == static_cast<time_t>(-1)) {
        cerr << "time failed\n";
        exit(EXIT_FAILURE);
    }

    const tm *const s = localtime(&t);
    install(s->tm_mon + 1, s->tm_mday, s->tm_year + 1900);
}

date& date::operator++()
{
    if (++day > length()) {
        day = 1;
        if (++month > december) {
            month = january;
            ++year;
        }
    }

    return *this;
}

date& date::operator--()
{
    if (--day < 1) {
        if (--month < january) {
            month = december;
            --year;
```

```
        }
        day = length();
    }

    return *this;
}
```

Now we can derive a class that knows about leap years, without having to rewrite most of the base class.

In line 8, the constructor for `leapdate` begins by calling the constructor for `date` and passing it no arguments. This would still happen even if we don't write the `: date()`, so don't bother to write the it. We still have to keep the `leapdate() {}` in line 8, however. The computer will not supply a default constructor for us if we have written another constructor with arguments (line 7).

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/virtual2/leapdate.h`

```
 1 #ifndef LEAPDATEH
 2 #define LEAPDATEH
 3 #include "date.h"
 4
 5 class leapdate: public date {
 6 public:
 7     leapdate(int initial_month, int initial_day, int initial_year);
 8     leapdate(): date() {}
 9
10     int length() const;
11 };
12 #endif
```

As we have already seen, it is no sin for a member function of a derived class to call upon a member function of the base class (line 31).

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/virtual2/leapdate.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "leapdate.h"
 4 using namespace std;
 5
 6 leapdate::leapdate(int initial_month, int initial_day, int initial_year)
 7     : date(initial_month, initial_day, initial_year)
 8 {
 9     if (initial_day > length()) {
10         cerr << "bad day " << initial_month << "/" << initial_day << "/"
11             << initial_year << "\n";
12         exit(EXIT_FAILURE);
13     }
14 }
15
16 int leapdate::length() const
17 {
18     const int y = get_year();
19
20     bool is_leap;                //uninitialized; true if this is a leap year
21
```

```
22     if (y % 400 == 0) {          //2000 and 2400 are leap years
23         is_leap = true;
24     } else if (y % 100 == 0) {   //1700, 1800, 1900, and 2100 are not leap years
25         is_leap = false;
26     } else if (y % 4 == 0) {
27         is_leap = true;
28     } else {
29         is_leap = false;
30     }
31
32     return !is_leap && get_month() == february ? 28 : date::length();
33 }
```

To make the variable `is_leap` a `const`, and make the code run faster by putting the most common case first, condense the above lines 20–30 to

```
34     const bool is_leap = y % 4 == 0 && (y % 100 != 0 || y % 400 == 0);
```

We now get correct output from the above `main.C`, if we change the object to a `leapdate`:

```
3/1/2014
```

Why does the base class `date` think that every year is a leap year? Wouldn't it have been more natural for the base class to assume that every year is non-leap? Well, suppose we declare

```
35     leapdate ld(date::february, 29, 2004);
```

The constructor for `leapdate` begins by passing these three arguments to the constructor for the base class `date` in the above line 7. The constructor for `date` must therefore be able to accept February 29th.

Why does the constructor for the derived class have to compare `initial_day` to the length of the month in the above line 9? Wasn't this check already performed by the constructor for the base class when it called `install`? Well, we have to do it again because `install` was calling `date::length`. The constructor for `leapdate` will call `leapdate::length`.

Superhuman foresight and godlike omniscience

When we write a class that may later be used as a base for other classes, can we anticipate every expression and statement that may have to be overridden in the derived classes, cut them out, and isolate them in one or more virtual member functions? This is a much, much harder problem than merely deciding which member functions need to be marked as virtual.

To see how hard this is, can you see any statements still in the non-virtual member functions of the above class `date` that might need to be isolated in virtual member functions because of a derived class that no one has dreamt of yet? Please do not read farther until you have tried this.

Let's call the last class `date` the "base class". The base class believes there was a Year Zero between 1 B.C. and 1 A.D.. Suppose we had to derive a class that was smart enough to know that there was no Year Zero. The `install` member function of the base class now is obsolete because of line 29 of `date.C`: it has to do more to the `year` data member than just the `year = y;`. The prefix `operator++` member function of the base class is obsolete because of line 50: it can't just blindly add 1 to `year`. Similarly, the prefix `operator--` member function is obsolete because of line 62.

With the benefit of hindsight, we should have given the base class the following virtual member function. It can be private since it will be called only by the member functions of the base class.

```
 1     virtual bool is_legal_year() const {return true;}
```

It always returns true since the base class believes that any number is a valid year number. It believes that there was a Year Zero.

In the `install` member function of the base class, line 29 should have been

```
    year = y;
    if (!is_legal_year()) {
        cerr << "bad year " << m << "/" << d << "/" << year << "\n";
        exit(EXIT_FAILURE);
    }
```

In the prefix `operator++` member function of the base class, line 50 should have been

```
    ++year;
    if (!is_legal_year()) {
        ++year;
    }
```

In the prefix `operator--` member function of the base class, line 62 should have been

```
    --year;
    if (!is_legal_year()) {
        --year;
    }
```

Finally, the derived class should have a smarter version of the `is_legal_year` virtual member function. It can, and therefore should, be private, because it is called only when the member functions of the base class call the virtual function `is_legal_year`. In general, however, a virtual function in a derived class does not necessarily have to have the same level of privacy as the function in the base class.

```
    bool is_legal_year() const {return get_year() != 0;}
```

Once again, we have simplified the job of the virtual function in order to minimize the code that has to be rewritten in the derived classes. The virtual function merely returns true or false; the `if`'s that use these values are in the non-virtual member functions of the base class. See p. 514.

As a test of your perspicuity, are there any more statements still in the non-virtual member functions of the base class that might need to be isolated in virtual member functions? Or have we caught them all?

▼ Homework 5.6a: the Julian to Gregorian switch-over

The English-speaking world switched from the Julian to the Gregorian calendar in September, 1752. Eleven days were removed from that month to synchronize the new calendar with the seasons. It was the Y2K problem of the Eighteenth Century.

```
   September 1752
 S  M Tu  W Th  F  S
       1  2 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

Suppose we have to derive a date class that is smart enough to know about this. the `install` member function of the base class is now obsolete because of line 41 of `date.C`: not every day is legal. The prefix `operator++` member function is obsolete because of line 46: it can't blindly add 1, because the day after September 2, 1752 was September 14th. Similarly, the prefix `operator--` member function is obsolete because of line 59.

With the benefit of hindsight, how shall we fix this? Was the `is_legal_year` virtual member function a good idea?

▲

▼ Homework 5.6b: have we isolated all the potentially obsolete bits of code yet?

Are there any other statements still in the non-virtual member functions of the base class that might need to be isolated in virtual member functions?

▲

5.7 Abstract Base Classes and Pure Virtual Functions

A base class for two implementations of class date

No birds were flying overhead—
There were no birds to fly.

—Lewis Carroll, *Through the Looking-Glass*, Chapter IV

To illustrate pure virtual functions and abstract base classes, let's go back to a simpler class `date` that knows nothing of leap years, Julian vs. Gregorian, or the absence of a year 0. We won't even bother with an `operator--` or any other function to move the `date` backwards.

The original class `date` had three data members:

```
    int year;           //Must construct the data members in this order.
    int month;          //date::january to date::december inclusive
    int day;            //1 to length[month] inclusive
```

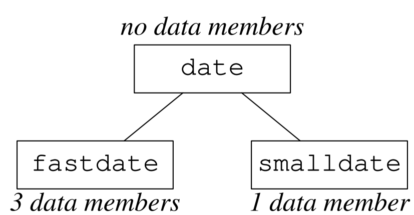
We can save space by changing them to one data member:

```
    int day;            //number of days before or after January 1, 0 A.D.
```

Unfortunately, the class `date` with one data member is slower. Its constructor has more work to do: it must combine its three integer arguments into one big integer. Conversely, its `operator<<` friend also has more work: it must render its integer data member back into three separate integers (`m/d/y`).

Let's name the two implementations after their virtues: class `fastdate` (with three data members) and class `smalldate` (with one data member). We will derive them from a common base class `date`, containing the members needed by both derived classes. But none of these members will be data members: the two derived classes have no data members in common. It will be our first class with no data members.

Class `date` is intended only as building block for the two derived classes. No one will ever construct an object whose most derived class is `date`, i.e., an object that is merely a `date` and nothing else. Such an object would be hollow—it would have no data members.



The `print` member function on p. 497, line 10 of `virtualfriend.C` had to be public because it was called by a function that was neither a member nor a friend of its class. But the `print` member function in line 7 of the following `date.h` can be private because it is called only by a friend of its class.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/pure/date.h`

```
1 #ifndef DATEH
2 #define DATEH
3 #include <iostream>
4 using namespace std;
5
```

```
 6 class date {
 7     virtual void print(ostream& ost) const;
 8 public:
 9     enum month_t {
10         january = 1,
11         february,
12         march,
13         april,
14         may,
15         june,
16         july,
17         august,
18         september,
19         october,
20         november,
21         december
22     };
23
24     static const int length[];   //no non-static data members
25
26     date(int initial_month, int initial_day, int initial_year);
27     virtual ~date() {}
28
29     virtual date& operator++();
30     virtual date& operator+=(int count);
31
32     friend ostream& operator<<(ostream& ost, const date& d) {
33         d.print(ost);
34         return ost;
35     }
36 };
37 #endif
```

The constructor for class `date` in lines 21−34 performs the error checking for all the derived classes. But it does not initialize any data members: this class has no data members to initialize.

It would be premature to attempt to increment or print an object with no data members, so the bodies of the `operator++` and `print` in lines 45−55 contain only an error message and an exit. (Some compilers would warn you that they fail to `return` a value.) Most of the other member functions would be the same way, so I didn't bother to define them. Oddly enough, though, there is one member function that we can define even though we have no data members yet: the `operator+=` in lines 36–43. (It's virtual because there will be better [i.e., faster] ones in the derived classes.) We are able to define it because it defers most of its work to an `operator++` function in line 39. To which `operator++`? We haven't written any working `operator++` yet, but we will. The best `operator++` for the object at hand will be selected, since `operator++` is virtual.

But let's go back to the "premature" member functions `operator++` and `print`. Why did we even declare them in class `date`? Primarily because we had to provide a place to hang the keyword `virtual`. Without writing the keyword in this class, the `operator++` and `print` member functions in the derived classes would not be virtual.

We also had to declare an `operator++` in class `date` because it is called by one of the member functions of this class (`operator+=`). And we had to declare a `print` in class `date` because it is called by one of the friends of this class (`operator<<`). Without these declarations, our `operator+=` and `operator<<` would not compile.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/pure/date.C`

```
#include <cstdlib>
#include "date.h"
using namespace std;

const int date::length[] = {
    0,    //dummy entry so that january will have subscript 1
    31,   //january
    28,   //february
    31,   //march
    30,   //april
    31,   //may
    30,   //june
    31,   //july
    31,   //august
    30,   //september
    31,   //october
    30,   //november
    31    //december
};

date::date(int initial_month, int initial_day, int initial_year)
{
    if (initial_month < january || initial_month > december) {
        cerr << "bad month " << initial_month << "/" << initial_day
            << "/" << initial_year << "\n";
        exit(EXIT_FAILURE);
    }

    if (initial_day < 1 || initial_day > length[initial_month]) {
        cerr << "bad day " << initial_month << "/" << initial_day
            << "/" << initial_year << "\n";
        exit(EXIT_FAILURE);
    }
}

date& date::operator+=(int count)
{
    while (--count >= 0) {
        ++*this;   //(*this).operator++();
    }

    return *this;
}

date& date::operator++()
{
    cerr << "can't call date::operator++\n";
    exit(EXIT_FAILURE);
}

void date::print(ostream& ost) const
{
```

```
    cerr << "can’t call date::print\n";
    exit(EXIT_FAILURE);
}
```

The only thing we can do with a `date` is to pass a zero to its `operator+=` member function. Any other argument, or any other member function, will give us an error message at runtime. We can’t even print it.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/pure/main1.C`

```
#include <cstdlib>
#include "date.h"
using namespace std;

int main()
{
    date d(date::january, 1, 2014);
    d += 0;   //d.operator+=(0);

    return EXIT_SUCCESS;
}
```

The derived classes

The virtual functions in lines 19–20 can return a `fastdate&` even though the corresponding functions in the base class returned a `date&`: a virtual function in a derived class can have a return type that is derived from the return type of the function in the base class. This is the exception in p. 493, ¶ (4). Unfortunately, Microsoft Visual C++ does not handle the exception. See “Bug C2555: On Virtual Functions with Covariant Return Types” at
`http://support.microsoft.com/support/kb/articles/Q240/8/62.ASP?LN=EN-US&SD=gn&FR=0&qry=q240862&rnk=1&src=DHCS_MSPSS_gn_SRCH&SPR=VCC`

The base class `date` had a working `operator+=`, but we can write faster ones in the derived classes.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/pure/fastdate.h`

```
 1 #ifndef FASTDATEH
 2 #define FASTDATEH
 3 #include "date.h"
 4
 5 class fastdate: public date {
 6     int year;
 7     int month;   //date::january to date::december inclusive
 8     int day;     //1 to length[month] inclusive
 9
10     void print(ostream& ost) const {
11         ost << month << "/" << day << "/" << year;
12     }
13
14 public:
15     fastdate(int initial_month, int initial_day, int initial_year)
16         : date(initial_month, initial_day, initial_year),
17         year(initial_year), month(initial_month), day(initial_day) {}
18
```

```
    fastdate& operator++();
    fastdate& operator+=(int count);
};
#endif
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/pure/fastdate.C`

```
#include <cstdlib>   //for div
#include "fastdate.h"
using namespace std;

fastdate& fastdate::operator++()
{
    if (++day > length[month]) {
        day = 1;
        if (++month > december) {
            month = january;
            ++year;
        }
    }

    return *this;
}

fastdate& fastdate::operator+=(int count)
{
    div_t d = div(count, 365);
    if (d.rem < 0) {   //Make sure remainder is non-negative.
        d.rem += 365;
        --d.quot;
    }

    year += d.quot;

    for (day += d.rem; day > length[month];) {
        day -= length[month];
        if (++month > december) {
            month = january;
            ++year;
        }
    }

    return *this;
}
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/pure/smalldate.h`

```
#ifndef SMALLDATEH
#define SMALLDATEH
#include "date.h"

class smalldate: public date {
    static const int pre[];
    int day;   //number of days before or after January 1, 0 A.D.
```

```
    void print(ostream& ost) const;
public:
    smalldate(int initial_month, int initial_day, int initial_year)
        : date(initial_month, initial_day, initial_year),
        day(365 * initial_year + pre[initial_month] + initial_day - 1)
        {}

    smalldate& operator++() {++day; return *this;}
    smalldate& operator+=(int count) {day += count; return *this;}
};
#endif
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/pure/smalldate.C`

```
#include <cstdlib>
#include "smalldate.h"
using namespace std;

const int smalldate::pre[] = {
    0,                      //dummy element to give january subscript 0
    0,                      //january
    pre[ 1] + length[ 1],   //february
    pre[ 2] + length[ 2],   //march
    pre[ 3] + length[ 3],   //april
    pre[ 4] + length[ 4],   //may
    pre[ 5] + length[ 5],   //june
    pre[ 6] + length[ 6],   //july
    pre[ 7] + length[ 7],   //august
    pre[ 8] + length[ 8],   //september
    pre[ 9] + length[ 9],   //october
    pre[10] + length[10],   //november
    pre[11] + length[11]    //december
};

void smalldate::print(ostream& ost) const
{
    div_t d = div(day, 365);
    if (d.rem < 0) {   //Make sure remainder is non-negative.
        d.rem += 365;
        --d.quot;
    }

    int julian = d.rem + 1; //Julian date is in range 1 to 365, not 0 to 364.
    int month;              //uninitialized variable

    for (month = 1; julian > length[month]; ++month) {
        julian -= length[month];
    }

    ost << month << "/" << julian << "/" << d.quot;
}
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/pure/main2.C`

```
#include <iostream>
#include <cstdlib>
#include "date.h"
#include "fastdate.h"
#include "smalldate.h"
using namespace std;

int main()
{
    fastdate fd(date::january, 1, 2014);
    cout << fd << "\n";
    fd += 280;   //fd.operator+=(280);
    cout << fd << "\n\n";

    smalldate sd(date::january, 1, 2014);
    cout << sd << "\n";
    sd += 280;
    cout << sd << "\n\n";

    cout << "sizeof (date) == " << sizeof (date) << "\n"
        << "sizeof (fastdate) == " << sizeof (fastdate) << "\n"
        << "sizeof (smalldate) == " << sizeof (smalldate) << "\n";

    return EXIT_SUCCESS;
}
```

The above program consists of seven source code files:

(1) `date.C` and `term.h`

(2) `fastdate.h` and `fastdate.C`

(3) `smalldate.h` and `smalldate.C`

(4) `main2.C`

On my machine, a `date` object contains four bytes of overhead even though it has no data members. A `fastdate` has three `int` data members of four bytes each, plus the four bytes of overhead. A `smalldate` has one `int` data member, plus the four bytes of overhead. In each case, the overhead is the pointer to the virtual table (p. 498).

```
1/1/2014
10/8/2014

1/1/2014
10/8/2014

sizeof (date) == 4
sizeof (fastdate) == 16
sizeof (smalldate) == 8
```

Abstract base class and pure virtual functions

In the base class `date`, most of the other member functions would be like `operator++` and `print`: just an error message and an `exit`. They must all be declared in class `date`, however, because they must carry the keyword `virtual`. Fortunately, we have a notation to save us the trouble of defining a body for each one. Remove the definitions of `operator++` and `print` in lines 45–55 of the above `date.C`, and change their declarations in lines 7 and 29 of `date.h` to

```
    virtual void print(ostream& ost) const = 0;
    virtual date& operator++() = 0;
```

The `= 0`'s announce that class `date` is an incomplete class with two missing pieces named `print` and `operator++`. An incomplete class is called an *abstract* class, and the missing pieces are called *pure virtual functions.*

We're not allowed to construct an object whose most derived class is an abstract class. There are three ways of constructing an object, and all three will not compile:

```
    date d(date::january, 1, 2014);                    //declared
    date *const p = new date(date::january, 1, 2014); //dynamically allocated
    cout << date(date::january, 1, 2014) << "\n";     //anonymous temporary
```

Objects of an abstract class can still exist, but only when embedded in a derived object. (In the same way, a quark can exist only in a larger particle such as a proton). Even though we can no longer declare a `date` anywhere in a program, they can still exist. A function can therefore still receive a `date *` or a `date &` as an argument

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/pure/main3.C`

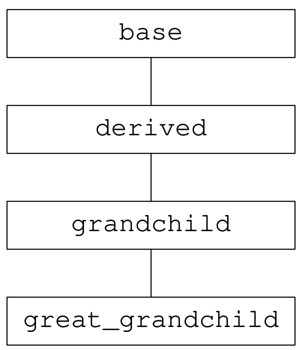
```
#include <iostream>
#include <cstdlib>
#include "fastdate.h"
#include "smalldate.h"
using namespace std;

void f(date *p);
void g(date& r);

int main()
{
    fastdate fd(date::january, 1, 2014);
    f(&fd);
    g(fd);

    smalldate sd(date::january, 1, 2014);
    f(&sd);
    g(sd);

    date *p = &fd;        //perfectly okay to have a date *
    return EXIT_SUCCESS;
}

void f(date *p)
{
    cout << *p << "\n";   //operator<<(cout, *p) << "\n";
    *p += 280;            //(*p).operator+=(280);
    cout << *p << "\n\n";
}

void g(date& r)   //same function, but with the reference notation
{
    cout << r << "\n";    //operator<<(cout, r) << "\n";
    r += 280;             //r.operator+=(280);
    cout << r << "\n\n";
}
```

Lines 26 and 33 call `operator<<`, which can call `fastdate::print` or `smalldate::print`. Lines 27 and 34 can call `fastdate::operator+=` or `smalldate::operator+=`.

```
1$ g++ main.C date.C fastdate.C smalldate.C
```

```
1/1/2014
10/8/2014

10/8/2014
7/15/2015

1/1/2014
10/8/2014

10/8/2014
7/15/2015
```

How long does a class stay abstract?



Class `base` is an abstract class because it has three pure virtual functions. Classes `derived` and `grandchild` are also abstract, because they still have two virtual functions. Only class `great_grandchild` is not abstract.

```
class base {
public:
    virtual void f() const = 0;
    virtual void g() const = 0;
    virtual void h() const = 0;
};

class derived: public base {
public:
    void f() const {}
};

class grandchild: public derived {
public:
};

class great_grandchild: public grandchild {
public:
```

```
    void g() const {}
    void h() const {}
};

int main()
{
    //base b;                   //won't compile: base has no f, g, or h
    //derived d;                //won't compile: derived has no g or h
    //grandchild g;             //won't compile: grandchild has no g or h
    great_grandchild gg;        //will compile: all present and accounted for
}
```

The influence travels in both directions

The behavior of a derived class is influenced by the behavior of its base class: the derived class inherits the code in the member functions of its base class. But the behavior of a base class may also be influenced by the behavior of its derived classes. How could this be? The base class inherits nothing from the derived class.

We construct two objects of class `base`: the `b` in line 24 and the anonymous `base` object inside the `d` in line 27. When line 25 calls the `g` member function of the first `base` object, `g` calls the `base::f` in line 10 and outputs a message. But when line 28 calls the `g` member function of the second `base` object, `g` will call the `derived::f` in line 19, outputting a different message. The behavior of the second base object has therefore been influenced by the code in line 19 of the derived class. "Insanity is hereditary: you get it from your children." (Erma Bombeck)

Warning: `base::f` is not overridden until we begin to construct a derived object around the base object. And the overriding ceases when we finish destructing the derived object. Lines 7 and 8 therefore always call `base::f`, not `derived::f`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/pure/override.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 class base {
 6 public:
 7     base() {f();}              //always calls base::f
 8     virtual ~base() {f();}     //always calls base::f
 9
10     virtual void f() const {cout << "base::f\n";}
11     void g() const {f();}      //doesn't necessarily call base::f
12 };
13
14 class derived: public base {
15 public:
16     derived(): base() {f();} //always calls derived::f
17     ~derived() {f();}          //always calls derived::f
18
19     void f() const {cout << "derived::f\n";}
20 };
21
22 int main()
23 {
24     base b;
```

```
25     b.g();
26
27     derived d;
28     d.g();
29
30     return EXIT_SUCCESS;
31 }
```

`base::f`	*Line 24 calls line 7, which calls line 10.*
`base::f`	*Line 25 calls line 11, which calls line 10.*
`base::f`	*Line 27 calls line 16, which calls line 7, which calls line 10.*
`derived::f`	*Line 27 executes the* {*body*} *of line 16, which now calls line 19.*
`derived::f`	*Line 28 calls line 11, which calls line 19.*
`derived::f`	*Destruct* `d`*: line 30 calls line 17, which still calls line 19.*
`base::f`	*Line 30 calls line 8, which calls line 10.*
`base::f`	*Destruct* `b`*: line 30 calls line 8, which calls line 10.*

Something you must never do

A program may blow up if it calls a pure virtual function that has not yet been overridden by a function in a derived class. For example, line 22 calls line 16, which calls line 7, which calls line 10, which blows up. Will line 22 even compile on your platform?

You can remove line 16 entirely. Even without it, class `derived` would still have a constructor which takes no arguments, and which would call the constructor for class `base` with no arguments.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/pure/blowup.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 class base {
 6 public:
 7     base() {f();}            //always calls base::f
 8     virtual ~base() {f();}  //would also call base::f if we ever got this far
 9
10     virtual void f() const = 0;
11     void g() const {f();}   //doesn't necessarily call base::f
12 };
13
14 class derived: public base {
15 public:
16     derived(): base() {}
17     void f() const {cout << "derived::f\n";}
18 };
19
20 int main()
21 {
22     derived d;
23     return EXIT_SUCCESS;
24 }
```

```
blowup.C: In constructor 'base::base()':
blowup.C:7:12: warning: abstract virtual 'virtual void base::f() const' called
from constructor
blowup.C: In destructor 'virtual base::~base()':
blowup.C:8:21: warning: abstract virtual 'virtual void base::f() const' called
from destructor
Undefined            first referenced
 symbol                  in file
base::f() const                      /var/tmp//ccE4aqFq.o
ld: fatal: symbol referencing errors. No output written to /dev/null
collect2: ld returned 1 exit status
```

5.8 Derive classes `wolf` and `rabbit` from `wabbit`

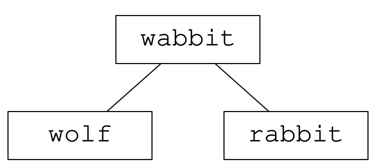
Inheritance in the real world

In an ideal world we would know in advance what classes we have to write. If they will be similar, we would begin by writing a base class for them. This would give us a head start for the classes derived from it.

In real life, your manager tells you what classes to write, one by one, in no particular order. After defining a few of them, you notice that they have features in common. They should have been derived from a common base class. But now it's too late: they've already been written.

Now that the above classes have been implemented, we notice too late that their member functions are largely the same and their data members are almost the same. In retrospect, it's obvious that they should have been derived from a common base class.

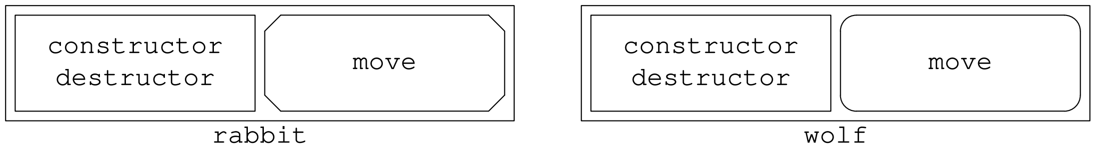
We will rewrite our classes `wolf` and `rabbit` the way they should have been written: by deriving them from a base class. The base class will be named `wabbit`, à la Bugs Bunny and Elmer Fudd. I'm sorry we didn't have the foresight to do this from the beginning, but that's the way it is in the real world. At least it will now be simpler to implement additional species of animals.



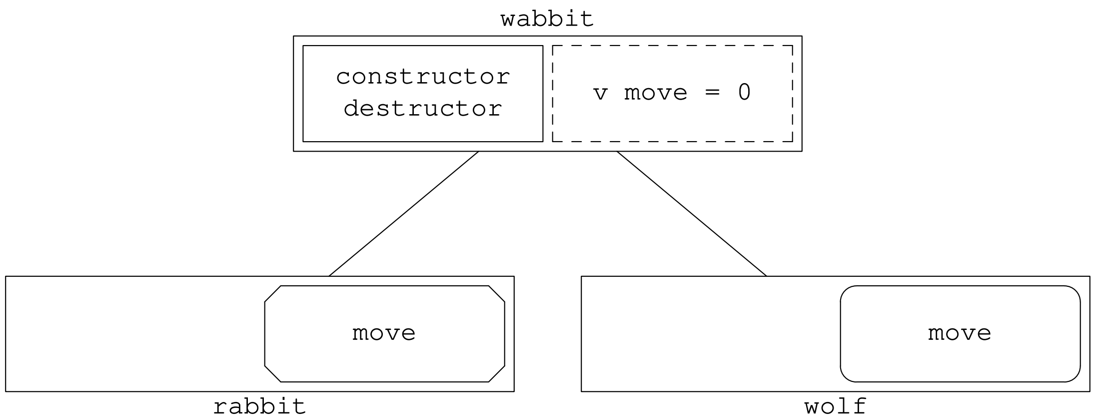
Consolidate the member functions of classes wolf and rabbit

Classes `wolf` and `rabbit` have a constructor and destructor. The rest of their code was lumped into a member function named `move`. This seemed reasonable, since `move`'ing is the only thing that an animal does besides birth and death. But we will now see that this was the wrong way to partition the code into member functions. When we consolidate these classes by deriving them from a common base, we will realize that they should have been modularized differently.

Let's draw a diagram of the modularization. The constructor and destructor for class `rabbit` are identical to those for `wolf`, so we draw them with the same shape: an unadorned rectangle. But the `move` function of each class is different, so we draw them with two shapes: beveled and rounded corners.



Since the constructors and destructors are the same in classes `rabbit` and `wolf`, we can easily consolidate them into a single copy up in the base class `wabbit`. But since the two `move` functions are different, it seems they will have to be left behind in the derived classes. The `move` up in class `wabbit` will be a pure virtual function: a missing piece to be filled in later. We draw it with a dashed box.



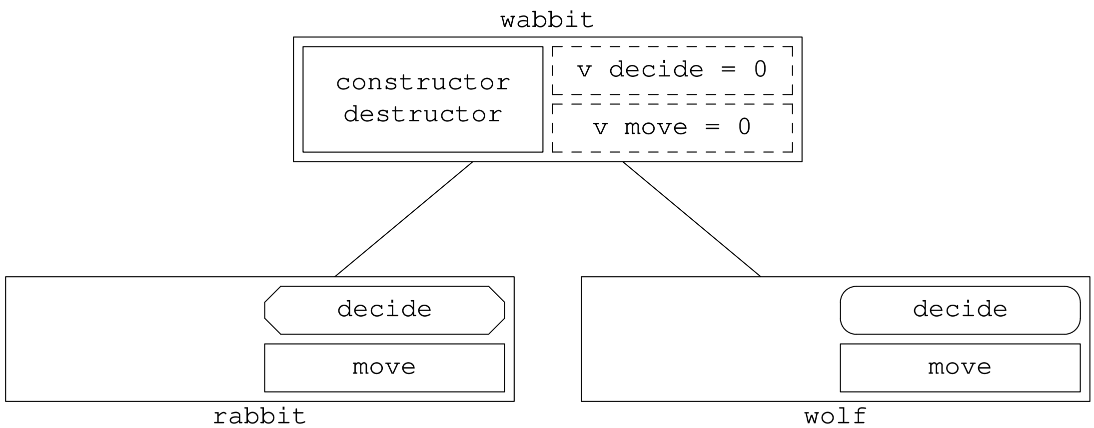
Pare down the code that gets stranded in the derived classes

Half the code is still stranded down in the derived classes. How can we minimize it? Here is where we discover that we should have modularized classes `rabbit` and `wolf` differently.

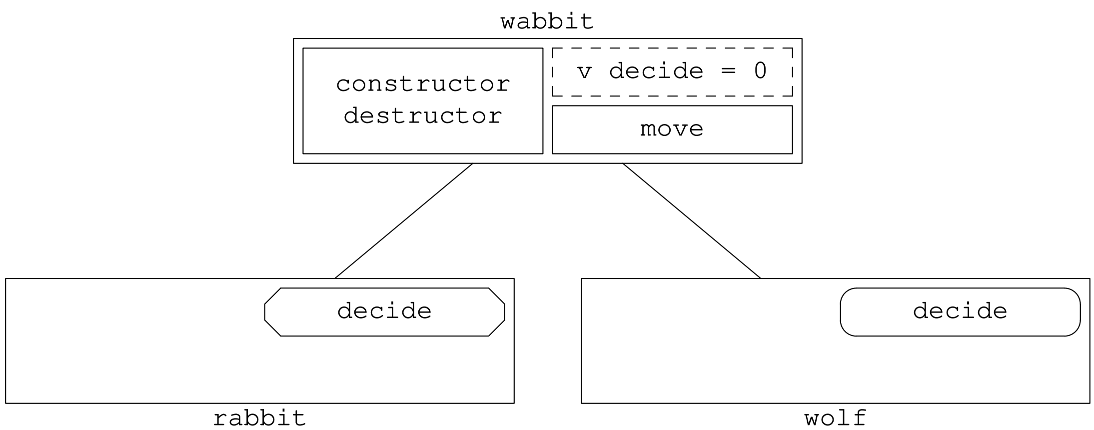
When `rabbit` was our only species of animal, no one suspected that `rabbit::move` should have been split into smaller functions.

But now we note that `rabbit::move` actually does two separate jobs: it decides which way to move by getting two random numbers, and then performs the move by updating the screen. Accordingly, we split it into two functions, named `rabbit::decide` and `rabbit::move`. Similarly, `wolf::move` does two jobs: it decides which way to move by getting a keystroke, and then performs the move by updating the screen. We split it the same way.

The resulting `rabbit::decide` and `wolf::decide` are very different: one gets two random numbers, the other gets a keystroke. We therefore draw them with different shapes and leave them down in the derived classes. On the other hand, the new `wabbit::move` and `wolf::move` are identical so we draw them with the same shape:



Since they are identical, the new `rabbit::move` and `wolf::move` can be consolidated into a single `wabbit::move`, leaving a smaller chunk of code behind in each derived class.
`wabbit::decide` will be a pure virtual function.

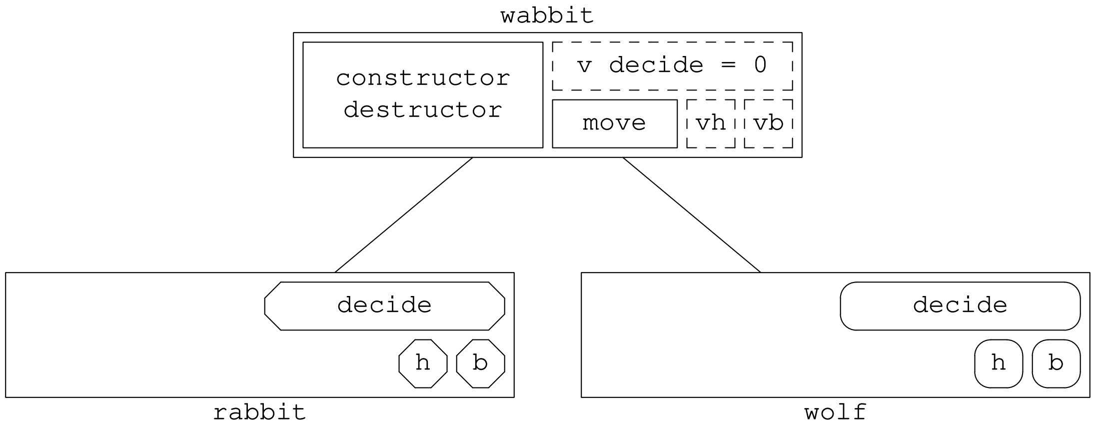


Hunger and bitterness

Unfortunately, too much code has been moved up to the base class. The original `rabbit::move` (pp. 196–197) was hardwired to give up the ghost when it met an animal of any other species. The original `wolf::move` (pp. 198–199) was hardwired to eat an animal of any species. Now that there is only a single `wabbit::move` function, how can it react correctly to another animal?

Recall that the member functions of a derived class can influence the behavior of the member functions of the base class (pp. 529–530). Every derived class (i.e., every species of animal) will have two new member functions telling how hungry it is and how bitter its flesh tastes. One animal will eat another if the first animal's level of hunger is greater than the second animal's level of bitterness.

Since each species may have a different level of hunger and bitterness, we have to implement these functions down in the derived classes. Up in the base class, they will be pure virtual functions:

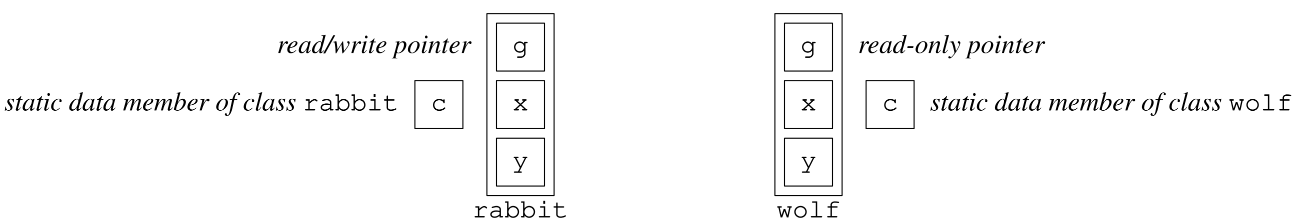


`wabbit::move` will now call the `hungry` and `bitter` member functions of the derived classes. It will use their return values to decide to eat, be eaten, or neither.

What we have just done to the base class `wabbit` is similar to what we did to the base class `date` whose derived classes had to know about leap years, the Year Zero, the Julian-to-Gregorian switchover, etc. See pp. 514 and 519. We identified the *smallest* chunks of code in the base class that would have to be written differently in one or more of the derived classes. Then we a declared a separate virtual member function of the base class for each chunk: `decide`, and `hungry` and `bitter`. (In the case of class `wabbit`, these functions are merely pure virtual.) Each derived class can now have its own style of motion and its own place in the food chain.

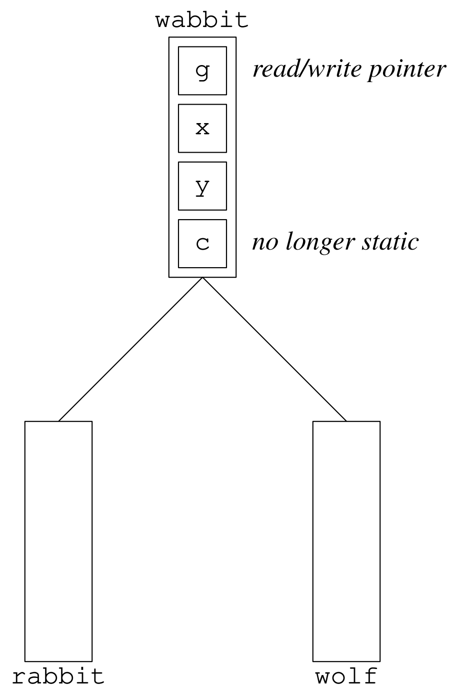
Consolidate the data members.

Classes `rabbit` and `wolf` have almost the same data members. The only difference is that the `g` member of class `rabbit` is a read/write pointer. (A newborn `rabbit` has to put its address on its `game`'s master list; a dying `rabbit` has to remove its address from the list.)



We will consolidate the data members into one copy in the base class `wabbit`. The derived classes `wolf` and `rabbit` will be left with no data members of their own.

Some of the derived animals will be `rabbit`'s, others will be `wolf`'s. This means that the `wabbit`'s will no longer all contain the same character, so the data member `wabbit::c` can no longer be static. It also means that the `wabbit::g` must be read/write pointer, since at least some of the derived animals will have to write into their `game`'s master list. (The `initial_g` argument of the constructor for class `wabbit` will therefore also be a read/write pointer.)



▼ Homework 5.8a:
Version 3.0 of the Rabbit Game: single inheritance: derive `wolf` and `rabbit` from `wabbit`

> KING CLAUDIUS. Now, Hamlet, where's Polonius?
> HAMLET. At supper.
> KING CLAUDIUS. At supper! Where?
> HAMLET. Not where he eats, but where he is eaten:
>
> —*Hamlet* IV, iii, 16–19

Derive class `wolf` and class `rabbit` from a base class named `wabbit`. Use public inheritance.

Class wabbit and its protected members

The `wabbit.h` header file will be included by the implementation file `wabbit.C`. It will also be included by the header files for the derived classes `wolf` and `rabbit`. But it will be included by no other file.

The `decide` member function in line 17 will have to return a pair of answers: the horizontal and vertical distances that the `wabbit` decided to move. But a C or C++ function can return only one value. One workaround would be to have `decide` return a structure with two fields. Another workaround would be to have `decide` deposit values into the two signed integers to which its arguments point. We'll choose the latter for the time being, but a better solution will appear on pp. 985–986 when we know more about containers, iterators, and `difference_type`.

The `x` and `y` data members in line 7 are unsigned integers, as are the arguments to the constructor in line 32. But the `dx` and `dy` arguments of `wabbit::decide` in line 17 are (pointers to) signed integers, as were the `dx` and `dy` structure members in `wolf::move` on pp. 198–199 and the other structure members on pp. 470–471. The unsigned vs. signed distinction appeared in the C Standard Library in `size_t` vs. `ptrdiff_t`, and will reappear on pp. 450–451 as `size_type` vs. `difference_type`.

Some of the member functions of the classes derived from `wabbit` will need to use the members of the `game` object to which the animals belong. For example, `wolf::decide` and `wolf::punish` will need to call the `key` and `beep` member functions of `g->term`. But `g` is a private member of class `wabbit`, so it cannot be mentioned by `wolf::decide` and `wolf::punish`. In addition, `term` is a private member of class `game`.

We therefore provide the `key` and `beep` member functions in lines 28–29, which give the derived classes access to the member functions of `g->term`. Since they are protected members of class `wabbit`, they can be called by `wolf::decide` and `wolf::punish`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/wabbit/wabbit.h`

```
1 #ifndef WABBITH
2 #define WABBITH
3 #include "game.h"
4
5 class wabbit {
6     game *const g;
7     unsigned x, y;
8     const char c;
9
10     //move calls these functions to decide who eats who.  wabbit w1 will eat
11     //wabbit w2 if w1.hungry() > w2.bitter(), i.e., if w1's hunger is
12     //stronger than w2's bitterness.
13     virtual int hungry() const = 0;
14     virtual int bitter() const = 0;
15
16     //move calls this function to decide which direction to move in.
17     virtual void decide(int *dx, int *dy) const = 0;
18
19     //move calls this function if this wabbit tries to move off the screen,
20     //or bumps into another wabbit that it can neither eat nor be eaten by.
21     //(Will also be called by manual::decide.)
22     virtual void punish() const {}
23
24     wabbit(const wabbit& another);             //deliberately undefined
25     wabbit& operator=(const wabbit& another);  //ditto
26
27 protected:
28     char key() const {return g->term.key();}   //called by wolf::decide
29     void beep() const {g->term.beep();}        //called by wolf::punish
30
31 public:
32     wabbit(game *initial_g, unsigned initial_x, unsigned initial_y,
33         char initial_c);
34     virtual ~wabbit();
35
36     bool move();
37
38     //A function that uses the x and y private data members of class wabbit.
39     friend wabbit *game::get(unsigned x, unsigned y) const;
40 };
41 #endif
```

Now that the data member `c` is no longer static, it must be initialized by the constructor for class `wabbit` just like the other data members `g`, `x`, and `y`. Other than that, the four-argument constructor for class `wabbit` will be just like the three-argument constructor for the original classes `rabbit` and `wolf`, except that the initial value of `c` will be passed in as an argument.

```
//Excerpt from the file wabbit.C.

```

```
wabbit::wabbit(game *initial_g, unsigned initial_x, unsigned initial_y,
    char initial_c)
    : g(initial_g), x(initial_x), y(initial_y), c(initial_c)
{
```

The body of the four-argument constructor for class wabbit will begin by checking if c is the same as the terminal's background character or if the initial x, y position is out of range. In each case, it will write an error message to cerr and exit. wabbit.C must therefore include <iostream> and <cstdlib>, as di the original rabbit.C and wolf.C.

If there was no error, the four-argument constructor for class wabbit will put the animal's character on the screen and the animal's address on the master list. The master list will therefore contain every wabbit, not just the rabbit's. It will now be a list of pointers to wabbit.

Class wabbit will also have a destructor, that beeps, pauses, removes the animal's address from the master list, and draws the terminal's background character on the screen at the animal's location.

The following move function will move a wolf or a rabbit, handling any encounter with another animal of any species. The this->'s in lines 26–27 are unnecessary. They are written only to rhetorically balance the other->'s.

A dynamically allocated object in C++ is not allowed to commit suicide—it might crash the program if an object said delete this. Instead, line 34 returns a value telling its caller that this wabbit should be destructed. Line 34 must come *after* line 30 because in the future we will have a species of animals that eat each other. We will have to execute both lines when encountering an animal that will eat and be eaten by this wabbit.

It is only fair to warn you that this is not the final version of wabbit::move. By the end of the course, every line will be rewritten.

—On the Web at
http://i5.nyu.edu/~mm64/book/src/wabbit/wabbit.C

```
 1 /*
 2 Delete any other wabbit that got eaten during the move (line 30), but do not
 3 delete this wabbit.  If this wabbit was eaten during the move, return false
 4 (line 34); otherwise return true.
 5 */
 6
 7 bool wabbit::move()
 8 {
 9     int dx;   //uninitialized variables
10     int dy;
11     decide(&dx, &dy);
12
13     if (dx and dy are both zero) {
14         return true;
15     }
16
17     const unsigned newx = x + dx;
18     const unsigned newy = y + dy;
19
20     if (!g->term.in_range(newx, newy)) {
21         punish();
22         return true;
23     }
24
25     if (wabbit *const other = g->get(newx, newy)) {
26         const bool I_ate_him =  this->hungry() > other->bitter();
```

```
        const bool he_ate_me = other->hungry() >  this->bitter();

        if (I_ate_him) {
            delete other;
        }

        if (he_ate_me) {
            return false;   //not allowed to delete myself
        }

        if (!I_ate_him) {
            //I bumped into a wabbit that I could neither eat nor be
            //eaten by.
            punish();
            return true;
        }
    }

    g->term.put(x, y);      //Erase this wabbit from its old location.
    x = newx;
    y = newy;
    g->term.put(x, y, c);   //Redraw this wabbit at its new location.

    return true;
}
```

Why do we need separate functions for hunger and bitterness?

Why didn't we make a single member function named `rank`, and have the animal with the higher `rank` eat the other one? Let's say we want to have two animals of species `a` eat each other, while two animals of species `b` bounce off each other without either being eaten. A single function would not let us do this. But with two functions, we can get any of the four possible outcomes.

```
//a and b eat each other.
int a::hungry() const {return 30;}
int b::bitter() const {return 20;}
int b::hungry() const {return 10;}
int a::bitter() const {return  0;}

//b and a bounce off each other.
int b::bitter() const {return 30;}
int a::hungry() const {return 20;}
int a::bitter() const {return 10;}
int b::hungry() const {return  0;}

//b eats a, but a doesn't eat b.
int b::bitter() const {return 30;}
int a::hungry() const {return 20;}
int b::hungry() const {return 10;}
int a::bitter() const {return  0;}

//a eats b, but b doesn't eat a.
int a::hungry() const {return 30;}
int b::bitter() const {return 20;}
int a::bitter() const {return 10;}
```

```
int b::hungry() const {return  0;}
```

Class rabbit

The three-argument constructor for the new class `rabbit` does nothing more than call the four-argument constructor for the base class `wabbit`. The copy constructor, `operator=`, and destructor for class `rabbit` are inherited from class `wabbit`. Ditto for class `wolf`.

The `INT_MIN` in lines 9–10, and the corresponding `INT_MAX`, are macros from the standard library header `<climits>` for the smallest and largest `int` values.

The return values of `hungry` and `bitter` are constant values. Why, then, are they functions rather than simple data members? Well, in a later version of the game they might have to do some computation. For example, an animal's level of hunger might depend on how many times it has `move`'d since its last meal.

It looks like `hungry` and `bitter` can be static member functions, since they use no non-static members. (In fact, they use no members at all.) But `hungry` and `bitter` must be virtual member functions, and a static member function cannot be virtual.

Since all the member functions of class `rabbit` are now inline, there is no longer any `rabbit.C` file.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/rabbit3/rabbit.h`

```
 1 #ifndef RABBITH
 2 #define RABBITH
 3 #include <cstdlib>   //for rand
 4 #include <climits>   //for INT_MIN
 5 #include "wabbit.h"
 6 using namespace std;
 7
 8 class rabbit: public wabbit {
 9     int hungry() const {return INT_MIN;}
10     int bitter() const {return INT_MIN;}
11
12     void decide(int *dx, int *dy) const {
13         *dx = rand() % 3 - 1;
14         *dy = rand() % 3 - 1;
15     }
16
17 public:
18     rabbit(game *initial_g, unsigned initial_x, unsigned initial_y)
19         : wabbit(initial_g, initial_x, initial_y, 'r') {}
20 };
21 #endif
```

Class wolf

The new `wolf.h` file will be the same as the new `rabbit.h`, with four differences:

(1) A `wolf`'s character is uppercase `'W'`; a `rabbit`'s is lowercase `'r'`. All deadly animals will be uppercase.

(2) A `wolf` is at the top of the food chain. It has `INT_MAX` hunger and `INT_MAX` bitterness.

(3) If a `rabbit` tries to move off the screen, or bumps into another animal that it can neither eat nor be eaten by, it's no one's fault. We are therefore content to let class `rabbit` inherit the empty `punish` member function from class `wabbit`. But if a `wolf` tries to move off the screen or bumps into another animal that it can neither eat nor be eaten by, there is a human being who requires chastisement. Ideally we

would administer a series of gradually increasing electrical shocks, but for the present we simply give class wolf the following inline private member function. It calls the beep member function inherited from class wabbit:

```
    void punish() const {beep();}
```

(4) The wolf::decide function is too long to be inline. Define it in the file wolf.C. Since wolf.h does not call rand, it does not need to include <cstdlib>.

Like rabbit::decide, wolf::decide will merely decide which direction to move in, and then return its decision to wabbit::move. Transplant the decision-making code from the original wolf::move on pp. 198–199 into wolf::decide. Like rabbit::decide, wolf::decide should not check for falling off the screen or colliding with a rabbit: these checks are already performed by wabbit::move.

Now that class wolf no longer has a data member named g, wolf::decide can no longer say g->term.key() and wolf::punish can not say g->term.beep(). They will have to call the key and beep member functions inherited from class wabbit.

wolf.C will no longer include rabbit.h, since it no longer mentions rabbit's. And wolf.C will not include iostream and cstdlib, since it no longer uses anything declared in these header files.

Here is the end of the wolf::decide function, picking up from line 35 of wolf.C on p. 198.

—On the Web at
http://i5.nyu.edu/~mm64/book/src/wabbit/wolf.C

```
    if (the key member function inherited from class wabbit says that
        the user pressed a key k) {

        for (search the array of structures using a pointer p) {
            if (k == p->c) {
                *dx = p->dx;
                *dy = p->dy;
                return;
            }
        }

        punish();   //Punish user who pressed an illegal key.
    }

    //Arrive here if the user pressed no key, or pressed an illegal key.
    *dx = *dy = 0;
}
```

Changes to class game

All the animals, not just the rabbit's, will be on the same master list. game::master will therefore be a list<wabbit *>, and game::get will return a wabbit *. game.h will need a forward declaration for class wabbit, not rabbit.

Want to make sure we never again have to change the return type of game::get? Declare its return value, and its local variable p, to be of data type game::master_t::value_type. That's what value_type is for. Within the {curly braces} of the declaration for class game, and within the body of game::get, you don't have to write the game:: at the start of game::master_t::value_type.

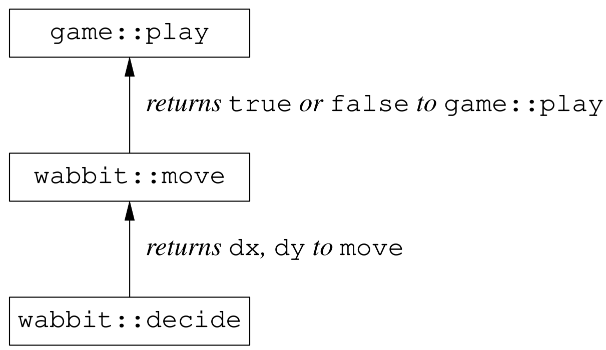
Only class wabbit will now be a friend of class game; classes rabbit and wolf will no longer be.

game.C will still include rabbit.h and wolf.h. I'm not happy about this, however. It means we have to modify game.C whenever we create a new species of animal.

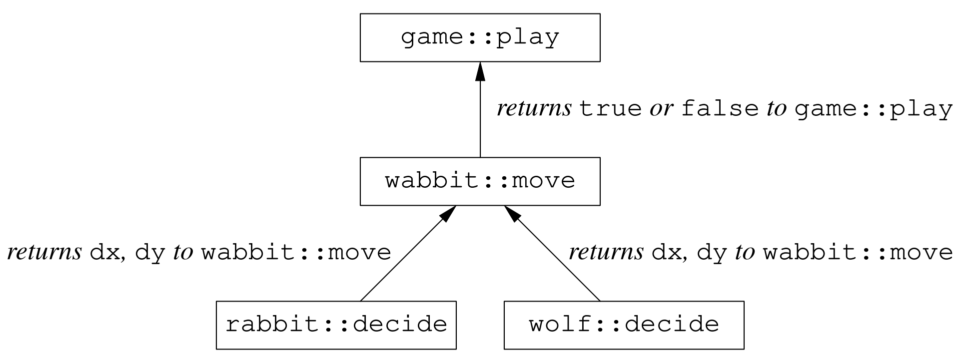
The `wolf` will now be dynamically allocated (constructed with `new`) with all the other animals in `game::game`, instead of automatically allocated (constructed with a declaration) in `game::play`. But don't construct the `wolf` in the loop in `game::game`—construct it with a `new` outside the loop.

The main loop in game::play

The `wolf` will now be on the master list. The call to `empty` in line 71 on p. 470 will therefore become the `master.size() > 1` in the following line 3, and the loop in the following lines 4–16 will move *all* the animals, not just the `rabbit`'s. The `wolf` no longer requires any special handling in the main loop, so the calls to the two `move`'s in lines 72 and 80 on p. 470 can be consolidated into the following line 8. The main loop will still call a `move` function, and `move` will call `decide`.



More precisely, the `wabbit::move` function will call either `rabbit::decide` or `wolf::decide` thanks to the magic of virtual functions.



The `move` in line 8 and the `delete` in line 14 will remove elements from the master list. But a list iterator cannot be incremented after the element to which it refers has been removed; see the "increment of death" on pp. 444–445. To avoid this misdeed, the `++it` must come between these two lines, at line 9.

Let's look at the previous version of this loop. On p. 470, the `move` in line 80 was applied only to `rabbit`'s. A `rabbit` not being carnivorous, this call to `move` destructed no other animal on the master list. It was therefore safe to increment the iterator in line 78 before calling `move`.

But the `move` in the following line 8 will be applied to every animal, `wolf` and `rabbit`. When applied to a `wolf`, it may destruct another animal on the list. The iterator must therefore be incremented *after* we call `move`, in line 9. Had we incremented it before the `move`, the iterator might have landed on an element that would then be destructed and removed by the `move`.

The `delete` in line 14 destructs the element to which the iterator in line 7 refers. The increment in line 9 must therefore come before the `delete` in line 14. Similarly, the increment in line 25 must be executed before the `delete` in line 26. For the same reason, the increment on p. 470 in line 78 had to come before the `delete` in line 81.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/game4/game.C`

```
 1 //Excerpt from game.C, showing the body of game::play.
 2
 3     for (; master.size() > 1; term.wait(250)) {
 4         for (master_t::const_iterator it = master.begin();
 5             it != master.end();) {
 6
 7             wabbit *const p = *it;
 8             const bool alive = p->move();
 9             ++it;
10
11             if (!alive) {
12                 //The wabbit that moved in line 8 blundered
13                 //into another wabbit and was eaten.
14                 delete p;
15             }
16         }
17     }
18
19 //The following lines go at the end of the destructor for class game.
20
21     //Delete any remaining wabbit's.
22
23     for (master_t::const_iterator it = master.begin(); it != master.end();) {
24         wabbit *const p = *it;
25         ++it;
26         delete p;
27     }
28 }
```

List of the 12 source files that constitute the game

(1) `term.h` and `term.c` (pp. 85–89). These are the only two written in C; the rest are C++.

(2) `terminal.h` and `terminal.C` (pp. 157–163)

(3) `game.h` and `game.C` (pp. 540–542)

(4) `wabbit.h` and `wabbit.C` (pp. 535–538)

(5) `wolf.h` and `wolf.C` (pp. 539–540)

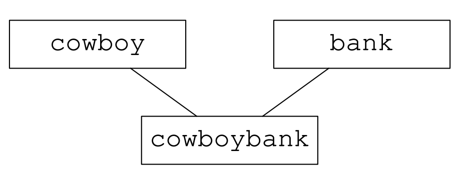
(6) `rabbit.h` (p. 539). There no longer is any `rabbit.C` file.

(7) `main.C` (pp. 193–194)

▲

5.9 Multiple Inheritance

5.9.1 A Simple Example



A C++ class can be derived from more than one base class. This is called *multiple inheritance.* Java has only single inheritance.

Our first example will be a silly one, just to illustrate the syntax and scoping rules. We start with two base classes to model the behavior of a cowboy and a bank.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/multiple/cowboy.h`

```
#ifndef COWBOYH
#define COWBOYH
#include <iostream>
using namespace std;

class cowboy {
    int i;
public:
    cowboy(int initial_i): i(initial_i) {}

    void chew() const {cout << this << " Gimme a chaw 'a 'baccy.\n";}
    void draw() const {cout << this << " Put 'em up, pardner!\n";}
};
#endif
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/multiple/bank.h`

```
#ifndef BANKH
#define BANKH
#include <iostream>
using namespace std;

class bank {
    int j;
public:
    bank(int initial_j): j(initial_j) {}

    void deposit() const {cout << this << " Please take a deposit slip.\n";}
    void draw() const {cout << this << " Your account is overdrawn.\n";}
};
#endif
```

Before the establishment of law and finance in the Wild West, many of the functions of banks were performed by itinerant cowboys. We will use multiple inheritance to model the behavior of typical "cowboy bank". He can do everything that a cowboy can do, as well as everything that a bank can do.

As usual, the constructor for the derived class begins by calling the constructor for the base class. But now there are two base classes and two constructors. Because of line 8, the constructor for `cowboy` will be called before the constructor for `bank`. (The order has nothing to do with the fact that line 12 lists the arguments for `cowboy` before those for `bank`.) Then the constructors will be called for the data members introduced in class `cowboybank` (the `k` in line 9).

When the `cowboybank` dies, the destructors for the data members introduced in class `cowboybank` will be called first. Then we will destruct the `bank`, and finally the `cowboy`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/multiple/cowboybank.h`

```
 1 #ifndef COWBOYBANKH
 2 #define COWBOYBANKH
```

```
#include <iostream>
#include "cowboy.h"
#include "bank.h"
using namespace std;

class cowboybank: public cowboy, public bank {   //say "public" twice
    int k;
public:
    cowboybank(int initial_i, int initial_j, int initial_k)
        : cowboy(initial_i), bank(initial_j), k(initial_k) {}

    void run() const {cout << this << " Time to clear out of town.\n";}
};
#endif
```

There's no problem with the function calls in lines 10–12. But the call to `draw` in line 14 is ambiguous and will not compile. Lines 15 and 16 disambiguate it in two directions. See the binary scope operator `::` in line 10 on p. 123; line 25 of `eclipse.C` on p. 246; line 42 of `derived.C` on p. 477.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/multiple/main.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "cowboybank.h"
 4 using namespace std;
 5
 6 int main()
 7 {
 8     cowboybank cbb(10, 20, 30);
 9
10     cbb.chew();          //inherited from cowboy
11     cbb.deposit();       //inherited from bank
12     cbb.run();           //introduced in cowboybank
13
14     //cbb.draw();        //won't compile: ambiguous
15     cbb.cowboy::draw(); //the draw inherited from cowboy
16     cbb.bank::draw();   //the draw inherited from bank
17
18     cout << "\n"
19         << &cbb << " == &cbb\n"
20         << static_cast<cowboy *>(&cbb) << " == addr of cowboy in cbb\n"
21         << static_cast<bank *>(&cbb) << " == addr of bank in cbb\n"
22         << reinterpret_cast<bank *>(&cbb) << "\n";
23
24     return EXIT_SUCCESS;
25 }
```

An *upcast* is a conversion from "pointer to derived" to "pointer to base". When the above lines 20 and 21 upcast the address of `cbb`, we get the address of the `cowboy` object and the `bank` object within the `cowboybank`. On my platform, the address of the `bank` object is `sizeof (cowboy)` bytes from the start of the `cowboybank`. This is our first example of a cast that changes the value of a pointer.

An upcast must always be done with a `static_cast`. Line 22 shows what goes wrong when we try to do it with a `reinterpret_cast`. For a "downcast", see p. 718.

Now that we have seen the addresses of the base objects inside the derived object, let's look at the values of `this` in the lines 10–11 and 15–16. A call to a member function of classes `cowboy` or `bank` will always receive the address of an object whose most derived class is `cowboy` or `bank`.

`0xffbff1e4 Gimme a chaw 'a 'baccy.`	*address of* `cowboy` *object within* `cowboybank`
`0xffbff1e8 Please take a deposit slip.`	*address of* `bank` *object within* `cowboybank`
`0xffbff1e4 Time to clear out of town.`	*address of* `cowboybank` *object*
`0xffbff1e4 Put 'em up, pardner!`	*address of* `cowboy` *object within* `cowboybank`
`0xffbff1e8 Your account is overdrawn.`	*address of* `bank` *object within* `cowboybank`
`0xffbff1e4 == &cbb`	
`0xffbff1e4 == addr of cowboy in cbb`	
`0xffbff1e8 == addr of bank in cbb`	`== &cbb + sizeof (cowboy)`
`0xffbff1e4`	*the address of the* `cowboybank`

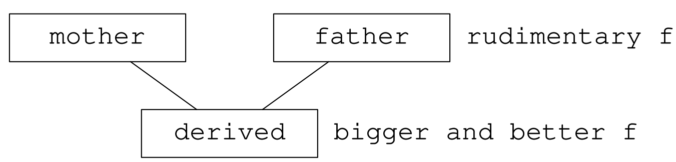
At this point, multiple inheritance still looks simple, doesn't it? The only problem was a name collision and some pointer adjustment.

▼ Homework 5.9.1a:

Will a `static_cast` from a "pointer to a `cowboybank`" to a "pointer to a `bank`" always change the value of the pointer? What if the pointer to a `cowboybank` is zero?
▲

5.9.2 Hidden Pointers II: a Thunk

Now let's add virtual functions to multiple inheritance and look at a possible implementation. The following program has three classes, two of which have a member function named `f`.

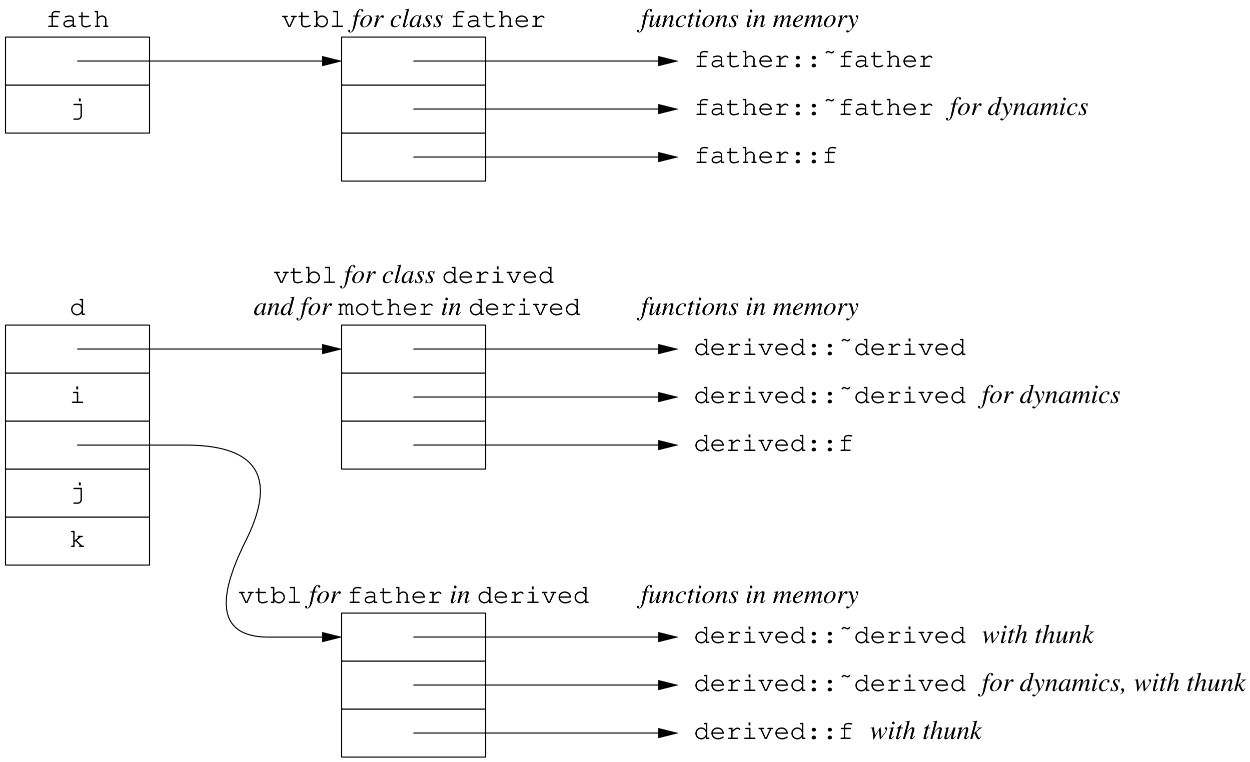


We would expect the program to have exactly two member functions named `f`, but the output on my platform shows that it has three. The stand-alone `father` object in lines 9–19 of `main.C` has one; the `derived` object in lines 21–31 of `main.C` has another; and the `father` object inside the `derived` object has a third (line 33–42).

We usually think of the last two `f`'s as being the same function. After all, this `father`'s `f` has been overridden by the `derived`'s `f`, hasn't it? We can even see the name `derived::f` in the output of each call to this function.

But if we look at the arguments, we can see that the `f` of class `derived` and the `f` of the `father` in the `derived` must be slightly different. The `f` of class `derived` prints out its implicit argument unchanged. The `f` of the `father` in the `derived` begins by subtracting `sizeof (mother)` from its implicit argument. The extra code that performs the subtraction is called a *thunk*. The thunk is necessary because `derived::f` must always have an implicit argument which is the address of a `derived` object, not the address of the `father` object in the `derived`.

I wouldn't be surprised if the `f` of the `father` in the `derived` is merely the thunk, followed by a "jump" to the start of the `f` of class `derived`.



—On the Web at
http://i5.nyu.edu/~mm64/book/src/thunk/mother.h

```
#ifndef MOTHERH
#define MOTHERH
using namespace std;

class mother {
    int i;
public:
    mother(int initial_i): i(initial_i) {}
    virtual ˜mother() {}   //this example simpler if every class has a vtbl
};
#endif
```

—On the Web at
http://i5.nyu.edu/~mm64/book/src/thunk/father.h

```
#ifndef FATHERH
#define FATHERH
#include <iostream>
using namespace std;

class father {
    int j;
public:
    father(int initial_j): j(initial_j) {}
    virtual ˜father() {}
    virtual void f() const {cout << "father::f, this == " << this << "\n";}

    struct vtbl {
```

```
        void (*ptr_to_destructor)(father *);
        void (*ptr_to_dynamic_destructor)(father *);
        void (*ptr_to_f)(const father *);
    };

    struct layout {
        const vtbl *ptr_to_vtbl;
        int j;
    };
};
#endif
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/thunk/derived.h`

```
#ifndef DERIVEDH
#define DERIVEDH
#include <iostream>
#include "mother.h"
#include "father.h"
using namespace std;

class derived: public mother, public father {
    int k;
public:
    derived(int initial_i, int initial_j, int initial_k)
        : mother(initial_i), father(initial_j), k(initial_k) {}
    void f() const {cout << "derived::f, this == " << this << "\n";}

    struct vtbl {
        void (*ptr_to_destructor)(derived *);
        void (*ptr_to_dynamic_destructor)(derived *);
        void (*ptr_to_f)(const derived *);
    };

    struct layout {
        const vtbl *ptr_to_vtbl;  //vtbl for derived & mother in derived
        int i;
        const father::vtbl *ptr_to_fvtbl;   //vtbl for father in derived
        int j;
        int k;
    };
};
#endif
```

The repetition in `main.C` will be consolidated in two stages, on pp. 676–677 when we have templates, and on 1017 when we have Runtime Type Identification.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/thunk/main.C`

```
#include <iostream>
#include <cstdlib>
#include "father.h"
#include "derived.h"
using namespace std;

```

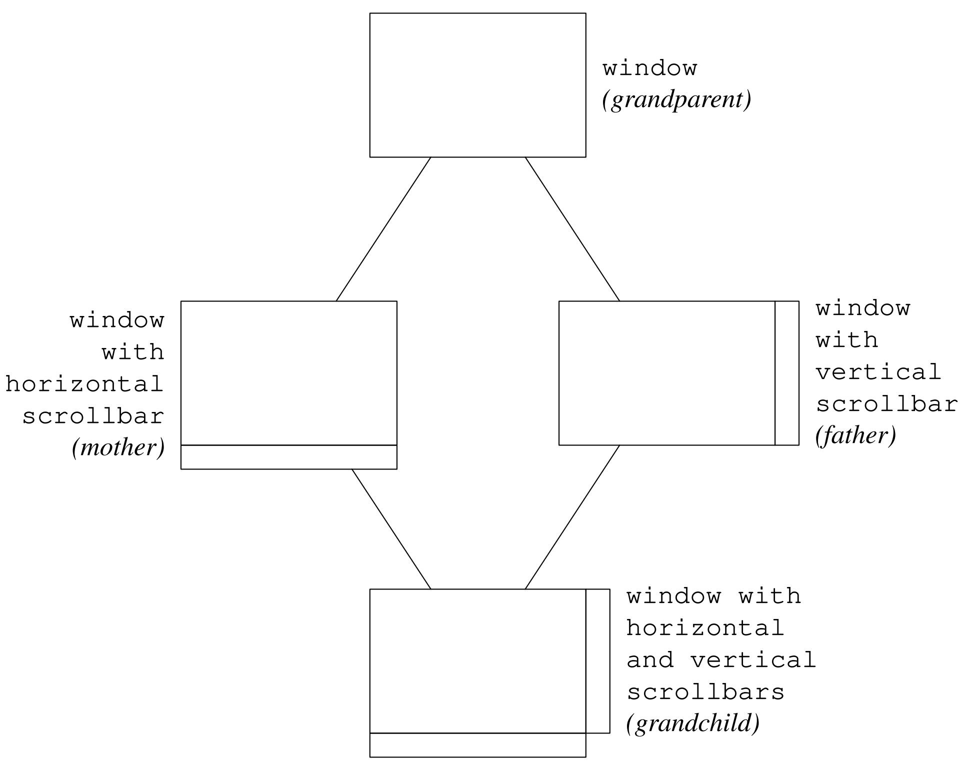
```
int main()
{
    father fath(10);
    const father::layout& flay =
        reinterpret_cast<const father::layout &>(fath);

    cout << "father at address " << &fath << " has an f whose address is "<<
        reinterpret_cast<const void *>(reinterpret_cast<size_t>(
        flay.ptr_to_vtbl->ptr_to_f)) << ".\n"
        "Let’s call this function twice, passing it " << &fath << ".\n";
    fath.f();
    flay.ptr_to_vtbl->ptr_to_f(&fath);  //low-level way to do the same thing
    cout << "\n";

    derived d(20, 30, 40);
    const derived::layout& dlay =
        reinterpret_cast<const derived::layout &>(d);

    cout << "derived at address " << &d << " has an f whose address is " <<
        reinterpret_cast<const void *>(reinterpret_cast<size_t>(
        dlay.ptr_to_vtbl->ptr_to_f)) << ".\n"
        "Let’s call this function twice, passing it " << &d << ".\n";
    d.f();
    dlay.ptr_to_fvtbl->ptr_to_f(&d);   //low-level way to do the same thing
    cout << "\n";

    const father *const p = &d;
    const father::layout& flay2 =
        reinterpret_cast<const father::layout &>(*p);

    cout << "father at address " << p << " has an f whose address is " <<
        reinterpret_cast<const void *>(reinterpret_cast<size_t>(
        flay2.ptr_to_vtbl->ptr_to_f)) << ".\n"
        "Let’s call this function twice, passing it " << p << ".\n";
    p->f();
    flay2.ptr_to_vtbl->ptr_to_f(p);   //low-level way to do the same thing

    return EXIT_SUCCESS;
}
```

```
father at address 0xffbff0e8 has an f whose address is 0x11800.
Let's call this function twice, passing it 0xffbff0e8.
father::f, this == 0xffbff0e8
father::f, this == 0xffbff0e8

derived at address 0xffbff0d4 has an f whose address is 0x119a8.
Let's call this function twice, passing it 0xffbff0d4.
derived::f, this == 0xffbff0d4
derived::f, this == 0xffbff0d4

father at address 0xffbff0dc has an f whose address is 0x119fc.
Let's call this function twice, passing it 0xffbff0dc.
derived::f, this == 0xffbff0d4
derived::f, this == 0xffbff0d4
```

5.9.3 Virtual Base Classes

A virtual base class

Now that we have multiple inheritance, a class can inherit DNA from the same ancestor along two different bloodlines. Let's start with a class represinting a window in a GUI. Using single inheritance, we augment it with a horizontal and vertical scrollbars. Then we use multiple inheritance to gather the two branches together to make a window with both scrollbars. A diagram with this shape is called *diamond inheritance.*



Let's give anthropomorphic names to the classes: the *grandparent, mother, father,* and *grandchild.* The grandchild should inherit everything that its mother has: a window and a horizontal scrollbar. It should also inherit everything that its father has: a window and a vertical scrollbar. But the grandchild must inherit only *one* window. In other words, the two windows that it inherits must be the same window.

To make them the same window, write the keyword `virtual` in lines 15 and 30. The `virtual` in line 15 tells the mother to be prepared to share its window with another object; the one in line 30 tells the father the same thing. Here the word has nothing to do with virtual functions. The designer of the language just wanted to get as much mileage as possible out of the smallest number of keywords.

The `virtual`'s also cause the grandparent (i.e., the `window`) in the grandchild to be constructed and destructed only once. (How bad would it be if the same object was constructed or destructed twice? Let's hope we never find out.) To accomplish this, however, we will have to make an exception to one of the principle rules of inheritance.

Until now, a constructor for a derived class has always begun by calling a constructor for the base class, or the constructor for every base class if there is more than one (`cowboybank` had two.) But now, for the first time, we don't want to do this. The grandchild's constructor will indeed call the constructors for its two base classes, the mother and father. But if the constructors for the mother and father then both called the constructor for *their* base class, the grandparent, we'd end up constructing the grandparent twice.

So which parent will have the privilege of constructing the grandparent? To avoid favoritism, neither one. The parents will be relieved of their customary duty of constructing the grandparent. It will be the constructor for the grandchild that calls the constructor for the grandparent. Similarly, the two parents will be relieved of the duty of destructing the grandparent. It will be the destructor for the grandchild that calls the destructor for the grandparent.

All of this is arranged by writing the keyword `virtual` in lines 15 and 30. The `virtual` in line 15, for example, makes the constructor for the mother skip line 19 when the mother is part of a grandchild. In this case, the grandparent in the mother has already been constructed by the grandchild, in line 54. On the other hand, when the mother is not part of a grandchild, the constructor for the mother will execute line 19 in the normal way.

Normally a constructor can call the constructors only for its immediate parent(s). But the constructor for our grandchild can make a direct call to the constructor for a remote ancestor in line 54 because the ancestor is virtual.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/multiple/virtual_base.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 class window {                                              //grandparent
 6     int i;
 7 public:
 8     window(int initial_i): i(initial_i) {
 9         cout << "construct window " << i << "\n";
10     }
11
12     ~window() {cout << "destruct window " << i << "\n";}
13 };
14
15 class window_with_horizontal: public virtual window {   //mother
16     int j;
17 public:
18     window_with_horizontal(int initial_i, int initial_j)
19         : window(initial_i),
20         j(initial_j) {
21         cout << "construct window_with_horizontal "
22             << initial_i << " " << initial_j << "\n";
23     }
24
```

```
    ˜window_with_horizontal() {
        cout << "destruct window_with_horizontal " << j << "\n";
    }
};

class window_with_vertical: public virtual window {     //father
    int k;
public:
    window_with_vertical(int initial_i, int initial_k)
        : window(initial_i),
        k(initial_k) {
        cout << "construct window_with_vertical "
            << initial_i << " " << initial_k << "\n";
    }

    ˜window_with_vertical() {
        cout << "destruct window_with_vertical " << k << "\n";
    }
};

class window_with_horizontal_and_vertical:              //grandchild
    public window_with_horizontal,
    public window_with_vertical {

    int l;
public:
    window_with_horizontal_and_vertical(int initial_i, int initial_j,
        int initial_k, int initial_l)

        : window(initial_i),
        window_with_horizontal(initial_i, initial_j),
        window_with_vertical(initial_i, initial_k),
        l(initial_l) {
        cout << "construct window_with_horizontal_and_vertical "
            << initial_i << " "
            << initial_j << " "
            << initial_k << " "
            << initial_l << "\n";
    }

    ˜window_with_horizontal_and_vertical() {
        cout << "destruct window_with_horizontal_and_vertical "
            << l << "\n";
    }
};

int main()
{
    window_with_horizontal_and_vertical w(10, 20, 30, 40);
    cout << "\n";
    return EXIT_SUCCESS;
}
```

The one copy of the grandparent is now shared by the mother, father, and grandchild:

```
construct window 10                                          Line 54 calls line 8.
construct window_with_horizontal 10 20                       55 calls 18, which skips 19.
construct window_with_vertical 10 30                         56 calls 33, which skips 34.
construct window_with_horizontal_and_vertical 10 20 30 40    lines 36–37

destruct window_with_horizontal_and_vertical 40              lines 66–67
destruct window_with_vertical 30                             line 65 calls line 40, which skips line 12
destruct window_with_horizontal 20                           line 65 calls line 25, which skips line 12
destruct window 10                                           line 65 calls line 12
```

What happens if we remove one or both of the virtual's

To cause the grandchild to inherit only one `window`, the keyword `virtual` is needed on both of the above lines 15 and 30. If we remove one or both of them, the grandchild will inherit two copies of the grandparent.

We'll probably never want to remove one `virtual`, but we'll show what happens anyway. If we remove the one in line 15, we get two grandparents in the grandchild. As above, we begin by constructing the grandparent that the grandchild inherits virtually (in this case, the `window` in the father). Then we construct the mother, including its grandparent. The (rest of the) father comes last, because of the order of the above lines 46–47.

```
construct window 10                                          construct window shared by father and grandchild
construct window 10                                          construct mother's window
construct window_with_horizontal 10 20                       construct rest of mother
construct window_with_vertical 10 30                         construct rest of father
construct window_with_horizontal_and_vertical 10 20 30 40    construct rest of grandchild

destruct window_with_horizontal_and_vertical 40              destruct grandchild, 'cept for its moth, fath, wind
destruct window_with_vertical 30                             destruct father, except for his window
destruct window_with_horizontal 20                           destruct mother, except for her window
destruct window 10                                           destruct mother's window
destruct window 10                                           destruct window shared by father and grandchild
```

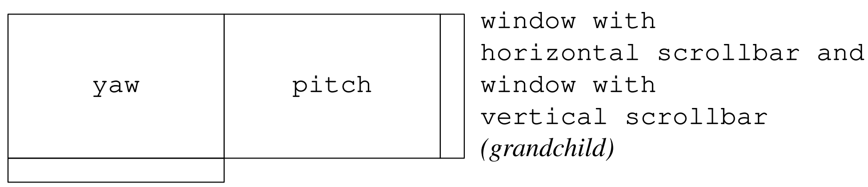
If we restore the `virtual` in line 15 remove the one in line 30 the output changes to the following. Again, we construct the grandparent that the grandchild inherits virtually (in this case, the `window` in the mother). Then we construct the (rest of the) mother. The father comes last, because of the above lines 46–47.

```
construct window 10                                          construct window shared by mother and grandchild
construct window_with_horizontal 10 20                       construct rest of mother
construct window 10                                          construct father's window
construct window_with_vertical 10 30                         construct rest of father
construct window_with_horizontal_and_vertical 10 20 30 40    construct rest of grandchild

destruct window_with_horizontal_and_vertical 40              destruct grandchild, 'cept for its moth, fath, wind
destruct window_with_vertical 30                             destruct father, except for his window
destruct window 10                                           destruct father's window
destruct window_with_horizontal 20                           destruct mother, except for her window
destruct window 10                                           destruct window shared by mother and grandchild
```

Finally, here is the output with both `virtual`'s removed. The constructor for a grandchild can make a direct call to the constructor for a grandparent only when the grandparent is inherited virtually along at least one bloodline to the grandchild. We therefore also had to remove the `window(initial_i),` in line 54.

```
construct window 10                                         construct mother's window
construct window_with_horizontal 10 20                      construct rest of mother
construct window 10                                         construct father's window
construct window_with_vertical 10 30                        construct rest of father
construct window_with_horizontal_and_vertical 10 20 30 40   construct rest of grandchild

destruct window_with_horizontal_and_vertical 40  destruct grandchild, 'cept for its moth, fath, wind
destruct window_with_vertical 30                 destruct father, except for his window
destruct window 10                               destruct father's window
destruct window_with_horizontal 20               destruct mother, except for her window
destruct window 10                               destruct mother's window
```

Now that each parent in the grandchild has its own grandparent, the grandchild could be a flight simulator. The two windows could display yaw and pitch, and the two scrollbars could control them:

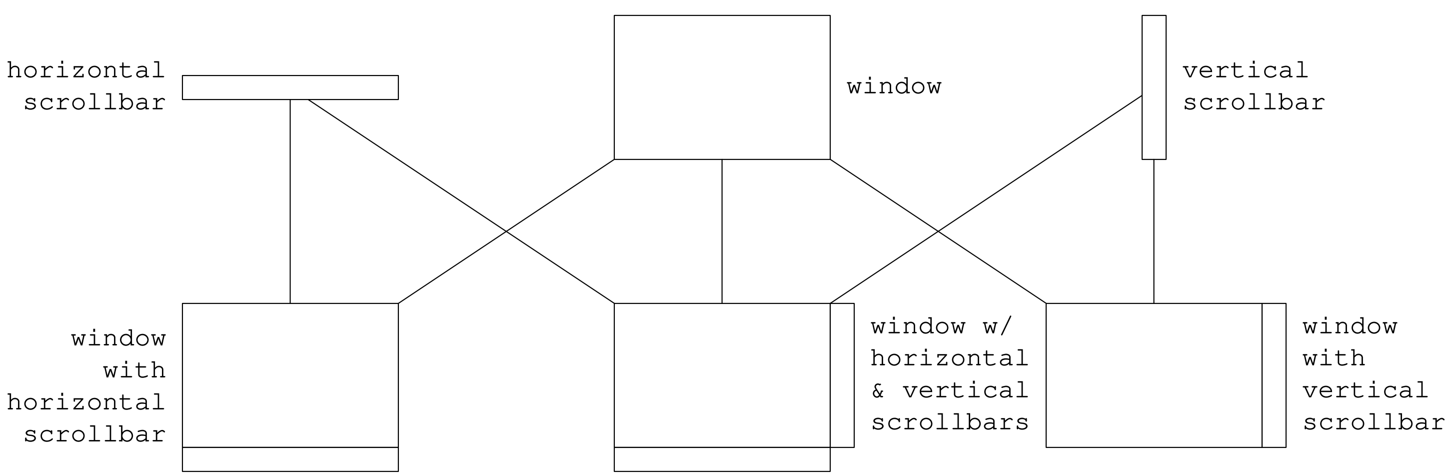


If we forget to remove the `window(initial_i),` from line 54, the error message on my platform is

```
virtual_base.C: In constructor
int, int, int)':
virtual_base.C:54:5: error: type 'window' is not a direct base of
```

An alternative dag

Why not make three separate classes, `window`, `horizontal_scrollbar`, and `vertical_scrollbar`, and derive the other classes from them? This would get rid of the diamond inheritance, so there would be no more trouble with virtual base classes:



I didn't do this because the connection between a window and its scrollbars is so intimate. Since every member function of a scrollbar would need to access the private members of its window, it would be awkward for the scrollbars to be separate classes.

Multiple inheritance with and without virtual base classes

A class `nurse` provides another example of multiple inheritance. Here in New York State, a `nurse_practitioner` can do everything that a `nurse` can do, plus more: he or she can prescribe additional drugs. And a `nurse_midwife` can do everything that a `nurse` can do, plus more: he or she can deliver babies. A `nurse_practitioner_midwife` can do everything that a `nurse_practitioner` and a `nurse_midwife` can do, plus more. But a `nurse_practitioner_midwife` should inherit only one `nurse`.

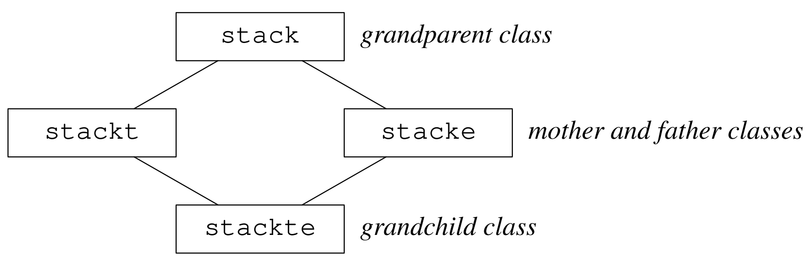
On the other hand, consider a small medical partnership comprising two nurses: a `nurse_practitioner` and a `nurse_midwife`. (In hip medical circles, this kind of partnership is known as "a stamp and a clamp".) In this case the partnership should inherit two separate nurses.

Another example: class `iostream` is derived from classes `istream` and `ostream`. But these two classes have a common parent, `ios_base`. The grandchild class `iostream` has only one copy of its grandparent `ios_base`. See the diamond diagram on pp. 383−385.

One last example: A bacon cheeseburger has only one hamburger.

Derive class stackte from classes stackt and stacke

We derived classes `stackt` and `stacke` from class `stack` on pp. 503−505. Now let's derive a grandchild that will inherit the features of both derived classes. We can keep the original grandparent class `stack` unchanged.



Before we had multiple inheritance, it seemed reasonable to divide the code in the parent classes `stackt` and `stacke` into two major member functions, `push` and `pop`. But if we kept this division, there would be no way to write the member functions of the grandchild class correctly. For example, the following `stackte::push` would accidentally call `stack::push` twice. (The binary scope operator `::` in lines 3 and 4 was last seen in lines 15−16 of `main.C` on p. 544.)

```
1 void stackte::push(int i)
2 {
3     stacke::push(i);
4     stackt::push(i);
5 }
```

before they can share a child, we will have to change the way the code in classes `stackt` and `stacke` is partitioned into member functions.

In the new implementation of class `stackt`, the `pop` function calls its `_pop` *after* it calls `stack::pop`: it must extract the number from the stack before it can print it. But in the new classs `stacke`, the pop calls its `_pop` *before* it calls `stack::pop`: it must check for underflow before extracting a number from the stack.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/stack2/stackt.h`

```
#ifndef STACKTH
#define STACKTH
#include <iostream>
#include "stack.h"
```

```
using namespace std;

class stackt: public virtual ::stack {
public:
    stackt() {cout << "stackt()\n";}
    ~stackt() {cout << "~stackt()\n";}

    void _push(int i) const {cout << "push(" << i << ")\n";}
    void  _pop(int i) const {cout <<  "pop(" << i << ")\n";}

    void push(int i) {::stack::push(i); _push(i);}
    int pop() {const int i = ::stack::pop(); _pop(i); return i;}
};
#endif
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/stack2/stacke.h`

```
#ifndef STACKEH
#define STACKEH
#include "stack.h"

class stacke: public virtual ::stack {
public:
    ~stacke();

    void _push() const;   //no explicit argument
    void  _pop() const;

    void push(int i) {_push(); ::stack::push(i);}
    int pop() {_pop(); return ::stack::pop();}
};
#endif
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/stack2/stacke.C`

```
#include <iostream>
#include <cstdlib>
#include "stacke.h"
using namespace std;

stacke::~stacke()
{
    if (size() != 0) {
        cerr << "stack destructed with nonzero size " << size() << "\n";
    }
}

void stacke::_push() const
{
    if (size() >= capacity()) {
        cerr << "size == " << size() << ", capacity == " << capacity() << "\n";
        exit(EXIT_FAILURE);
    }
}
```

```

void stacke::_pop() const
{
    if (size() <= 0) {
        cerr << "can’t pop stack with size " << size() << "\n";
        exit(EXIT_FAILURE);
    }
}
```

—On the Web at
http://i5.nyu.edu/~mm64/book/src/stack2/stackte.h

```
#ifndef STACKTEH
#define STACKTEH
#include "stackt.h"
#include "stacke.h"

class stackte: public stacke, public stackt {
public:
    void push(int i);
    int pop();
};
#endif
```

Now we can write the member functions of the grandchild class.

—On the Web at
http://i5.nyu.edu/~mm64/book/src/stack2/stackte.C

```
#include "stackte.h"

void stackte::push(int i)
{
    stacke::_push();            //must come before the call to ::stack::push
    ::stack::push(i);
    stackt::_push(i);
}

int stackte::pop()
{
    stacke::_pop();             //must come before the call to ::stack::pop
    const int i = ::stack::pop();
    stackt::_pop(i);            //must come after the call to ::stack::pop
    return i;
}
```

—On the Web at
http://i5.nyu.edu/~mm64/book/src/stack2/main.C

```
#include <iostream>
#include <cstdlib>
#include "stackte.h"
using namespace std;

int main()
{
    stackte s;
```

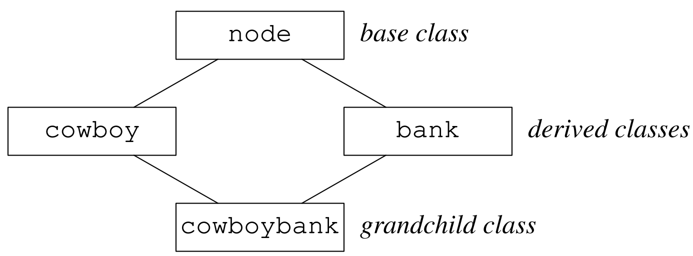
```

    s.push(10);
    cout << s.pop() << "\n";
    cout << s.pop() << "\n";
    return EXIT_SUCCESS;
}
```

```
stackt()
push(10)
pop(10)
10
can't pop stack with size 0
```

Multiple inheritance without virtual base classes

To keep all the `cowboy`'s on a linked list, each `cowboy` must have a `next` data member. To keep all the `bank`'s on another list, each `bank` must also have a `next` data member. The two classes can inherit this data member from a common base class named `node`. We saw how to provide an `operator<<` for a base class on pp. 496–497.

A `cowboybank` would have to be on two separate lists, the list of `cowboy`'s and the list of `bank`'s. It must therefore have two different `next` data members, so its parents must not be virtual.



The static data member `cowboy::begin` in line 21 contains the address of the first `cowboy` on the list, or zero if the list is empty. The constructor for `cowboy` (line 22) places the newborn `cowboy` at the beginning of the cowboy list, in front of any other `cowboy`'s that may already be on the list. Similarly, the constructor for `bank` (line 33) places the newborn `bank` at the beginning of the bank list.

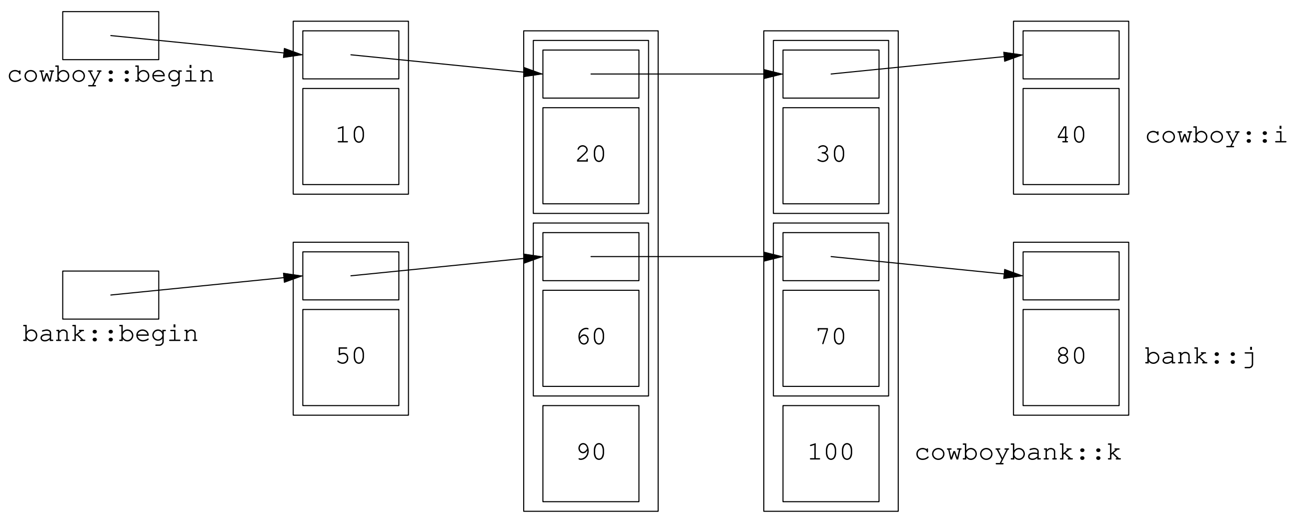
The `next` data member of class `node` should be private, and the user should be able to loop along the lists without writing the arrows in lines 68–70 and 75–77. We'll fix these problems when we do iterators. The `print` member functions of classes `cowboy` and `bank` are protected so that they can be called by the `print` member function of the grandchild class `cowboybank`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/multiple/cowboybank.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 class node {
 6     virtual void print(ostream& ost) const = 0;
 7 public:
 8     node *next;
 9     node(node *initial_next): next(initial_next) {}
10     virtual ~node() {}
11
12     friend ostream& operator<<(ostream& ost, const node& n) {
```

```
        n.print(ost);
        return ost;
    }
};

class cowboy: public node {
    int i;
public:
    static cowboy *begin;
    cowboy(int initial_i): node(begin), i(initial_i) {begin = this;}
protected:
    void print(ostream& ost) const {ost << "cowboy " << i;}
};

cowboy *cowboy::begin = 0;

class bank: public node {
    int j;
public:
    static bank *begin;
    bank(int initial_j): node(begin), j(initial_j) {begin = this;}
protected:
    void print(ostream& ost) const {ost << "bank " << j;}
};

bank *bank::begin = 0;

class cowboybank: public cowboy, public bank {
    int k;

    void print(ostream& ost) const {
        ost << "cowboybank ";
        cowboy::print(ost);
        ost << ", ";
        bank::print(ost);
        ost << ", " << k;
    }

public:
    cowboybank(int initial_i, int initial_j, int initial_k)
        : cowboy(initial_i), bank(initial_j), k(initial_k) {}
};

int main()
{
    cowboy c1 = 40;
    bank b1 = 80;

    cowboybank cb1(30, 70, 100);
    cowboybank cb2(20, 60, 90);

    cowboy c2 = 10;
    bank b2 = 50;

```

```
    cout << "Here are the cowboys:\n";
    for (const node *p = cowboy::begin; p != 0; p = p->next) {
        cout << *p << "\n";
    }

    cout << "\n";

    cout << "Here are the banks:\n";
    for (const node *p = bank::begin; p != 0; p = p->next) {
        cout << *p << "\n";
    }

    return EXIT_SUCCESS;
}
```

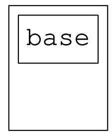



```
Here are the cowboys:
cowboy 10
cowboybank cowboy 20, bank 60, 90
cowboybank cowboy 30, bank 70, 100
cowboy 40

Here are the banks:
bank 50
cowboybank cowboy 20, bank 60, 90
cowboybank cowboy 30, bank 70, 100
bank 80
```

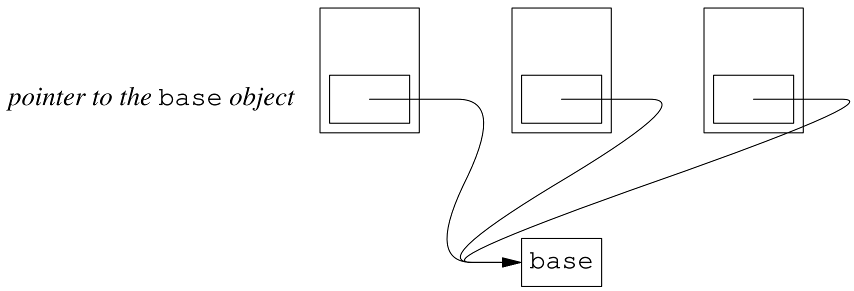
5.9.4 Hidden Pointers III: a Virtual Base Class Creates a Discontinuous Object

The simplest implementation of a virtual base class has one strange consequence. It may result in the creation of a spacially discontinuous object.

Before we had virtual base classes, a base object belonged to only one derived object, or at least to only one derived object that was not in turn part of an even larger one. The simplest way to give the derived object access to the base object was to put the latter physically inside of the former.

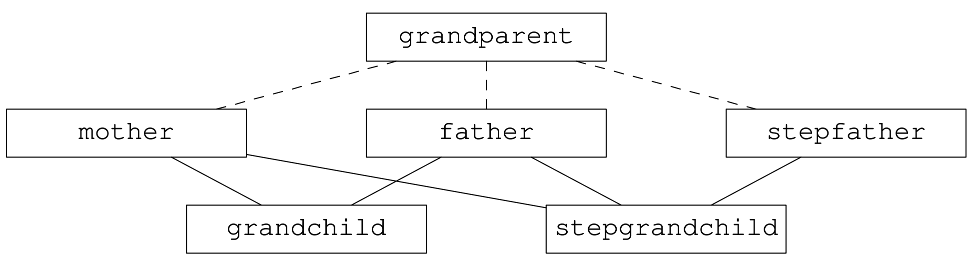


But when a base class is virtual, a base object may belong to more than one derived object. The base object can no longer always be inside of the derived object to which it belongs. Instead, each of the derived objects contains a pointer to the shared base object.



Each derived object, together with its base object, is considered to be one big object. For example, the `sizeof` a derived object will include the `sizeof` the base object, even though the latter may be some distance away in memory. The derived object is spacially discontinuous.

The following family with three ‘‘parent’’ classes will demonstrate that two discontinuous objects of the same class may have different distances between their parts. Derivation from a virtual base class is dashed.



In line 68 of `main.C`, the `mother` and the `father` inside of `g` will share the same `grandparent` object. And in line 72, the `mother`, `father`, and the `stepfather` inside of `sg` will share the same `grandparent` object.

On my platform, a derived object does not have an actual pointer to a base object of a virtual base class. It has a pointer to a table of data, whose first field is the offset in bytes from the end of the derived object to the start of its base object. This offset is of the data type `ptrdiff_t` (line 20), which should always be used for a distance that could be positive or negative.

The `mother` and `father` in line 68 share the same `grandparent`. On my platform, the `mother` is separated from the `grandparent` by a total of 12 bytes. The `father` occupies 8 bytes (a pointer and the data member `j`), and the `l` data member of the `grandchild` occupies 4 bytes.

The `mother`, `father`, and `stepfather` in line 72 share the same `grandparent`. On my platform, this `mother` is separated from its `grandparent` by 20 bytes. The `father` and `stepfather` each occupy bytes; the `l` occupies 4 bytes.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/discontinuous/main.C`

```
1 #include <iostream>
```

```
#include <iomanip>
#include <cstdlib>
using namespace std;

class grandparent {
    int i;
public:
    grandparent(int initial_i): i(initial_i) {}
    int f() const {return i;}
};

class mother: public virtual grandparent {
    int j;
public:
    mother(int initial_i, int initial_j)
        : grandparent(initial_i), j(initial_j) {}

    struct table {
        ptrdiff_t diff;   //offset from end of mother to its grandparent
    };

    struct layout {
        const table *p;
        int j;
    };
};

struct father: public virtual grandparent {
    int k;
    father(int initial_i, int initial_k)
        : grandparent(initial_i), k(initial_k) {}
};

struct stepfather: public virtual grandparent {
    int k2;
    stepfather(int initial_i, int initial_k2)
        : grandparent(initial_i), k2(initial_k2) {}
};

struct grandchild: public mother, public father {
    int l;
    grandchild(int initial_i, int initial_j, int initial_k, int initial_l)
        : grandparent(initial_i),
        mother(initial_i, initial_j),
        father(initial_i, initial_k),
        l(initial_l) {}
};

struct stepgrandchild: public mother, public father, public stepfather {
    int l;
    stepgrandchild(int initial_i, int initial_j, int initial_k,
        int initial_k2, int initial_l)
        : grandparent(initial_i),
        mother(initial_i, initial_j),
```

```
        father(initial_i, initial_k),
        stepfather(initial_i, initial_k2),
        l(initial_l) {}
};

void print(const mother *m);

int main()
{
    cout << "sizeof mother, not counting its grandparent, is "
        << sizeof (mother) - sizeof(grandparent) << ".\n\n";

    grandchild g(10, 20, 30, 40);
    cout << "mother in grandchild:\n";
    print(&g);

    stepgrandchild sg(50, 60, 70, 80, 90);
    cout << "mother in stepgrandchild:\n";
    print(&sg);

    return EXIT_SUCCESS;
}

void print(const mother *p)
{
    const mother::layout& lay =
        reinterpret_cast<const mother::layout &>(*p);
    const ptrdiff_t diff = lay.p->diff;
    const char *const cp = reinterpret_cast<const char *>(p)
        + sizeof (mother) - sizeof (grandparent);
    const grandparent *const gp =
        reinterpret_cast<const grandparent *>(cp + diff);

    cout
        << p << " == address of mother\n"
        << static_cast<const void *>(cp)
            << " == address of first byte after mother"
            " (not counting its grandparent)\n"
        << hex << setw(10) << diff << dec
            << " == offset to mother's grandparent (in hex)\n"
        << gp << " == address of mother's grandparent\n"
        << static_cast<const grandparent *>(p)
            << " == static_cast<const grandparent *>(p)\n"
        << "grandparent's f returns " << gp->f() << ".\n\n";
}
```

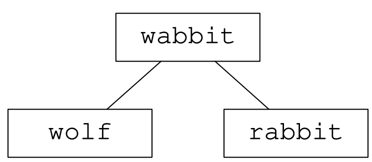
```
sizeof mother, not counting its grandparent, is 8.

mother in grandchild:
0xffbff078 == address of mother
0xffbff080 == address of first byte after mother (not counting its grandparent)
         c == offset to mother's grandparent (in hex)
0xffbff08c == address of mother's grandparent
0xffbff08c == static_cast<const grandparent *>(p)
grandparent's f returns 10.

mother in stepgrandchild:
0xffbff058 == address of mother
0xffbff060 == address of first byte after mother (not counting its grandparent)
        14 == offset to mother's grandparent (in hex)
0xffbff074 == address of mother's grandparent
0xffbff074 == static_cast<const grandparent *>(p)
grandparent's f returns 50.
```

5.9.5 Mix and Match the Ancestor Classes

▼ Homework 5.9.5a:
Version 3.1 of the Rabbit Game: multiple inheritance: mix and match the ancestor classes

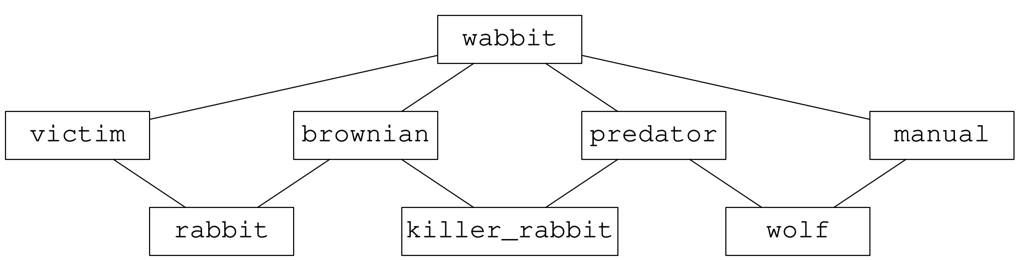
Class `wabbit` has two groups of missing pieces. It does not know what its place in the food chain is (`hungry`, `bitter`), and it does not know how to decide which way to move (`decide`, with an empty `punish` function that a derived class might want to override).

It seemed reasonable to derive class `wolf` directly from class `wabbit`, filling in the two missing groups.



But this bundled the `wolf`'s rank in the food chain (`wolf::hungry` and `wolf::bitter`) with its style of motion (`wolf::decide` and `wolf::punish`). Any derived class that inherits the `wolf`'s rank would also be forced to inherit its motion, and vice versa. Inheritance gives you *all* the members of the base class, whether you want them or not.

To inherit one without the other, we will use multiple inheritance:



Keep class `wabbit` the same. Derive two new classes, `manual` and `predator`, from it. Move the two member functions for the `wolf`'s style of motion and punishment from class `wolf` to class `manual`. Move the two member functions for the `wolf`'s rank in the food chain from class `wolf` to class predator. Then derive class wolf from classes manual and predator.

Similarly, derive two more classes named brownian and victim from class wabbit. (In physics, “brownian motion” is random motion.) Move the member function for the rabbit’s style of motion from class rabbit to class brownian. (rabbit::punish happens to be the same function as wabbit::punish, so don’t move it anywhere.) Move the two member functions for the rabbit’s rank in the food chain from class rabbit to class victim. Then derive class rabbit from classes brownian and victim.

The extra layer of classes will let us mix and match any style of motion with any rank in the food chain. For example, we can derive a class killer_rabbit that inherits the same motion as a rabbit and the same rank as a wolf.

In fact, we will derive sixteen classes from wabbit. The rows are styles of motion; the columns are ranks in the food chain. Deadly species (hungry==INT_MAX) have uppercase names.

	inert hungry==INT_MIN bitter==INT_MAX	victim hungry==INT_MIN bitter==INT_MIN	predator hungry==INT_MAX bitter==INT_MAX	halogen hungry==INT_MAX bitter==INT_MIN
immobile	'b' boulder	's' sitting_duck	'B' black_hole	'L' land_mine
brownian	'g' gnat	'r' rabbit	'R' killer_rabbit†	'S' strangelove
manual	'h' horse*	'f' fugitive	'W' wolf	'K' kamikaze
visionary	'p' pest	'd' deer	'A' alien	'P' positron

*This horse is the wooden crowd barrier.

†For Monty Python’s killer rabbit (1975), see

http://us.imdb.com/Title?0071853

For the killer rabbit that attacked President Carter on April 20, 1979, see

http://en.wikipedia.org/wiki/Jimmy_Carter_rabbit_incident

The four rank classes

Derive four classes from wabbit: inert, predator, victim, and halogen. They will override wabbit::hungry and wabbit::bitter as follows.

(1) An inert has no appetite and is unpleasant to eat. inert::hungry should return INT_MIN and inert::bitter should return INT_MAX as in the following lines 7–8.

(2) A predator has a hearty appetite and is unpleasant to eat. predator::hungry and predator::bitter should both return INT_MAX.

(3) A victim has no appetite and is tasty. victim::hungry and victim::bitter should both return INT_MIN.

(4) A halogen has a hearty appetite and is tasty. halogen::hungry should return INT_MAX and halogen::bitter should return INT_MIN.

Here is class inert. The other three rank classes will be the same except for their levels of hunger and bitterness.

—On the Web at
http://i5.nyu.edu/~mm64/book/src/inert.h

```
1 #ifndef INERTH
2 #define INERTH
3 #include <climits>   //for INT_MIN and INT_MAX
4 #include "wabbit.h"
5
6 class inert: public virtual wabbit {
7     int hungry() const {return INT_MIN;}
```

```
    int bitter() const {return INT_MAX;}
public:
    inert(game *initial_g, unsigned initial_x, unsigned initial_y,
        char initial_c)
        : wabbit(initial_g, initial_x, initial_y, initial_c) {}
};
#endif
```

The four rank classes will not override `wabbit::decide`, and so will remain abstract classes.

The motion classes

Then derive three more classes from `wabbit`: `immobile`, `brownian`, and `manual`. (We will do class `visionary` later.) They will override `wabbit::decide` and `wabbit::punish` as follows.

(1) An `immobile` never moves. `immobile::decide` always returns `0, 0` to `wabbit::move`, as in line 6 of the following `immobile.h`. Do not override `wabbit::punish`. There will be no `immobile.C` file.

(2) A `brownian` moves randomly around the screen. `brownian::decide` returns two random values to `wabbit::move`, as our old `rabbit::decide` did. `brownian::decide` will be inline in the file `brownian.h`, which will have to include `<cstdlib>` and `use namespace std` for the `rand` function. Do not override `wabbit::punish`. There will be no `brownian.C` file.

(3) A `manual` moves when we press a legal key (and beeps when we press an illegal one). Like our old `wolf::decide`, `manual::decide` looks up the keystroke in a table and finds the corresponding pair of `int`'s. It then returns these two `int`'s to `wabbit::move`. `manual::decide` is too big to be inline, so define it in a `manual.C` file. This file will mention nothing that belongs to namespace `std`, so it will not need to say `using namespace std;`. For the time being, do not construct more than one `manual`. Think about the machinery necessary to have several of them; it will appear on pp. 799–802.

Class `manual` will also need a `punish` function that beeps. Move the `punish` from class `wolf` to class `manual`, keeping it private.

For example, here is class `immobile`. The other motion classes will be similar.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/immobile.h`

```
1 #ifndef IMMOBILEH
2 #define IMMOBILEH
3 #include "wabbit.h"
4
5 class immobile: public virtual wabbit {
6     void decide(int *dx, int *dy) const {*dx = *dy = 0;}
7
8 public:
9     immobile(game *initial_g, unsigned initial_x, unsigned initial_y,
10         char initial_c)
11         : wabbit(initial_g, initial_x, initial_y, initial_c) {}
12 };
13 #endif
```

The motion classes will not override `wabbit::hungry` and `wabbit::bitter`, and so will remain abstract classes. Update the "called by" comments in lines 28–29 of `wabbit.h` on p. 536.

The grandchildren

Finally, use multiple inheritance to create three or four of the sixteen possible "grandchild" classes. For example, derive `rabbit` from `brownian` and `victim`, `wolf` from `manual` and `predator`, and `boulder` from `immobile` and `inert`. To make both parents public, the following line 6 will have to say `public` twice. There was no compelling reason for line 6 to construct the `immobile` before the `inert`. I adopted this order only because the name of each motion class is an adjective, while most of the rank classes are nouns.

Each grandchild class will inherit its `decide`, `punish`, `hungry`, and `bitter` member functions from its two parents. In fact, other than the constructor, a grandchild will have no member functions of its own. The declarations for the grandchild classes will therefore be almost identical. On pp. 695–696, this repetition will be consolidated with a “template”.

Here is class `boulder`. The `immobile` and the `inert` inside the `boulder` each contain a `wabbit`. But it’s the *same* `wabbit`, thanks to the magic of virtual base classes.

Line 9 calls the constructor for class `wabbit`, which initializes the `boulder`’s `wabbit`. Then line 10 calls the constructor for `immobile`, which would normally initialize the entire `immobile`. But class `immobile` is derived virtually from class `wabbit`, so the call in line 10 initializes only the part of the `immobile` that is not contained in the `wabbit`. Similarly, call to the constructor for class `inert` in line 11 initializes only the part of the `inert` that is not contained in the `wabbit`. The `wabbit` in the `boulder` is initialized only once.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/boulder.h`

```
 1 #ifndef BOULDERH
 2 #define BOULDERH
 3 #include "immobile.h"
 4 #include "inert.h"
 5
 6 class boulder: public immobile, public inert {
 7 public:
 8     boulder(game *initial_g, unsigned initial_x, unsigned initial_y)
 9          : wabbit(initial_g, initial_x, initial_y, 'b'),
10          immobile(initial_g, initial_x, initial_y, 'b'),
11          inert   (initial_g, initial_x, initial_y, 'b')
12          {}
13 };
14 #endif
```

Dominance

Class `boulder` inherits two different versions of `decide`: the flesh-and-blood `decide` inherited from class `immobile` and the ghostly, pure virtual `decide` inherited from class `wabbit` via class `inert`. Fortunately, when a `boulder` says `decide`, it gets `immobile::decide` rather than `wabbit::decide`; the deciding factor is that `immobile` is the derived class and `wabbit` is the base class. We therefore say that `immobile::decide` *dominates,* or hides, `wabbit::decide`.

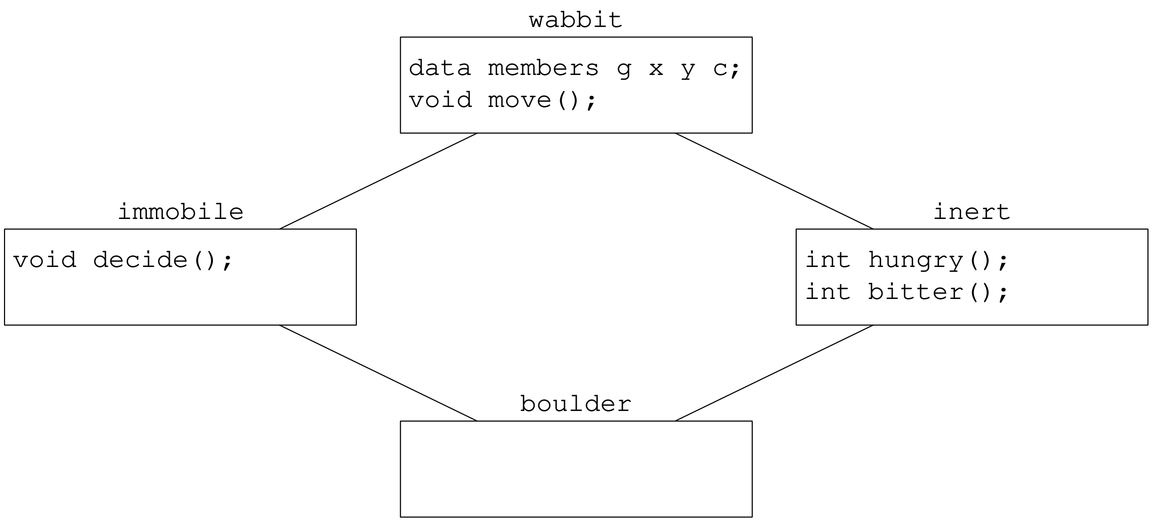
You can ignore the Microsoft Visual C++ warning about dominance; it is only a warning, not an error. If you find it annoying, disable it by saying

```
#pragma warning (disable: 4250)
```

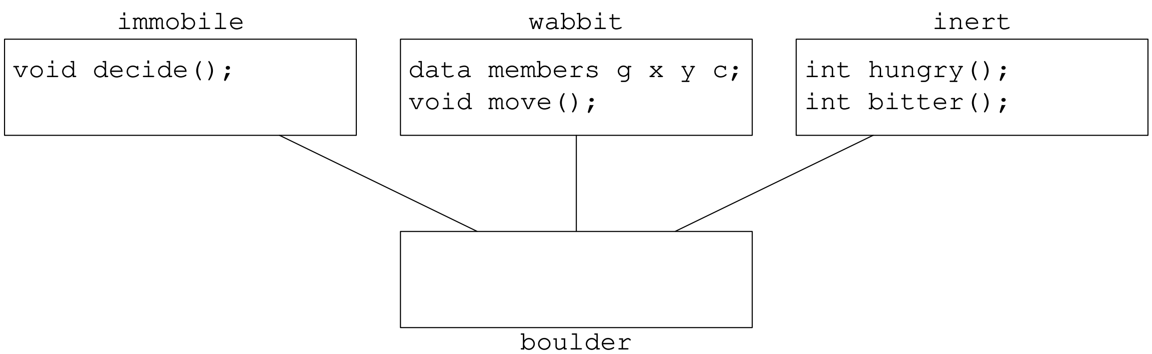
See

```
http://msdn.microsoft.com/
    library/default.asp?url=/library/en-us/vccore98/HTML/c4250.asp
```

Why not eliminate the dominance by eliminating the multiple inheritance? We currently have a diamond.



Why not change it to a pitchfork, eliminating the dominance?



This would be okay for now, but might inhibit future growth. For example, an animal's level of hunger might depend on how many times it has moved since its last meal, and class `wabbit` will have a new member function returning this number. Or the direction in which an animal decides to move might depend on its distance to the nearest animal, and class `wabbit` will have a new member function returning this distance. In either case, future versions of the `decide`, `hungry`, and `bitter` in the derived classes will have to call these new member functions of class `wabbit`.

In fact, `manual::decide` already calls the `key` and `punish` member functions of `wabbit`. We have seen how easy this is when `manual` is derived from `wabbit`. But if `manual` were not so derived, each `manual` object would need to contain a pointer to the `wabbit` object whose `key` and `punish` functions it should call. We will therefore keep the motion and rank classes derived from class `wabbit`.

A counting function

We will need to keep a count of the animals of each species to know when to terminate the game. Add the following private, non-inline, non-static member function to class `game`.

```
	master_t::size_type count(char c) const;
```

It will return the number of animals in the master list whose data member `c` has the given value. For example, `count('r')` will return the number of `rabbit`'s in the game. The function is named after the `count` algorithm in the C++ Standard Library. But there is no name conflict, because our `count` has the last name `game`.

`game::count` will contain a loop similar to the one in `game::get`. `game::get` accessed two data members of the `wabbit` pointed to by `*it`. Your `game::get` probably did not dereference the iterator three times:

```
        if ((*it)->x == x && (*it)->y == y) {
            return *it;
        }
```

It was easier to store `*it` into a pointer `p` once and for all.

```
        wabbit *const p = *it;  //or const master_t::value_type p = *it;
        if (p->x == x && p->y == y) {
            return p;
        }
```

But `game::count` will access only one data member of the `wabbit` pointed to by `*it`, so don't bother with a pointer. Just say `(*it)->c`.

To access the `c` data member of each `wabbit`, `game::count`, like `game::get`, will have to be a friend of class `wabbit`. Make sure that the friend declaration in `wabbit.h` correctly specifies the function's name, arguments, return value, and whether it is a `const` or non-`const` member function. Give it a comment like the one in line 38 of `wabbit.h` on p. 536. The comment on `game::get` in `wabbit.h` should now refer to `game::get` and to `game::count`.

To count the elements in a `game::master_t`, the return type of `count` must be `game::master_t::size_type`. Inside the {curly braces} of the class definition for class `game`, and inside the body of a member function of that class, we can refer to type `game::master_t` simply by saying `master_t`. But outside these two places, we must refer to it by its full name `game::master_t`. See lines 3 and 5 of `clinton2.C` on p. 423.

Instead of recounting the animals on demand, it would be faster to hold the count of each species in a separate data member of class `game`. But I don't want to have to add a new member to class `game` whenever a new class is derived:

```
    //hypothetical private data members of class game:
    //a maintenance nightmare

    master_t::size_type count_of_rabbits;
    master_t::size_type count_of_wolves;
    master_t::size_type count_of_boulders;
    //etc.
```

Could each count be a data member (static or otherwise) of the corresponding grandchild class? No, because we may want the program to run multiple games. Each `game` object will needs its own count of `rabbit`'s, its own count of `wolf`'s, etc. If a count were a data member of `game`, it would have to be non-static.

We'll get the speed of data members, but without their proliferation, when we do `map`'s on pp. 795–796. Our counting code has yet to reach its final form.

The constructor for class game

Create enough `boulder`'s to give the screen some texture, and throw in some `mine`'s. Or make a maze whose walls are made of `boulder`'s, with a `wolf` and a `sitting_duck`.

The most programmer-friendly way to create many animals of many species at many places is to draw the rectangular picture in lines 8–14. The data type of an array subscript should always be `size_t` (lines 6, 16, 18, 19); see p. 66. Get rid of the `struct location` and the array of `location`'s on pp. 470–471.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/game5/game.C`

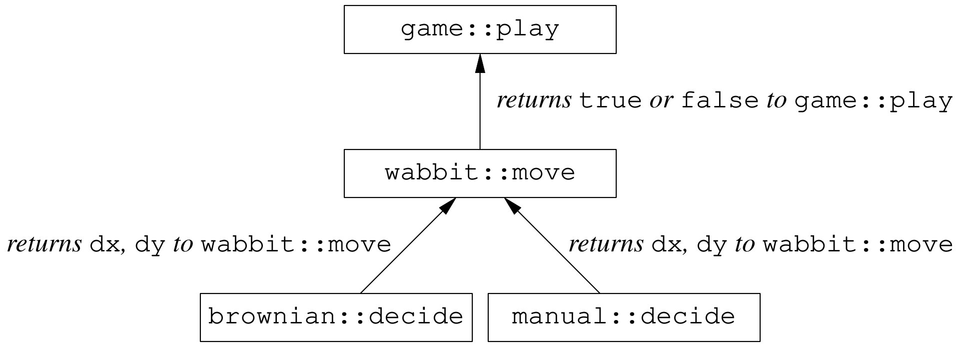
```
1 //Excerpt from game.C
2
3 game::game(char initial_c)
```

```
    : term(initial_c)
{
    static const size_t xmax = 8;         //number of columns in the picture
    static const char a[][xmax + 1] = {   //plus 1 for terminating ’\0’
        "bbbbbbbb",   //a maze of boulders
        "b......b",
        "b.bbbb.b",
        "b..s.b.b",   //The ’s’ is a sitting duck.
        "bbbbbb.b",
        "W......b",   //The ’W’ is a wolf.
        "bbbbbbbb"
    };
    static const size_t ymax = sizeof a / sizeof a[0];

    for (size_t y = 0; y < ymax; ++y) {
        for (size_t x = 0; x < xmax; ++x) {
            if (term.in_range(x, y)) {
                switch (a[y][x]) { //sorry the y comes before the x
                case ’.’:
                    break;

                case ’b’:
                    new boulder(this, x, y);
                    break;

                case ’s’:
                    new sitting_duck(this, x, y);
                    break;

                case ’W’:
                    new wolf(this, x, y);
                    break;

                default:
                    cerr << "bad character ’" << a[y][x]
                        << "’ at (" << x << ", " << y << ")\n";
                    exit(EXIT_FAILURE);
                }
            }
        }
    }
}
```

The `switch` statement will be replaced with a “map” on pp. 797–798.

The game::play function

Compare the diagrams on p. 541.



You decide when the game should be over. A reasonable choice would be to end the game when there are no more animals of any edible species (i.e., those derived from classes `victim` or `halogen`). If you have a `manual` animal and another animal hungry enough to eat it, you might also want to end the game when the `manual` animal is gone. At that point, the user would have nothing left to do.

There may still be many surviving `wabbit`'s when the game is over, so move the test from the outer loop (line 3 of `game.C` on p. 542) to the inner loop (lines 19–27 below). Remove the message from `game::~game` and replace it with messages like those in lines 19–27.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/game5a/game.C`

```
 1 //Excerpt from game.C, showing the body of the function game::play.
 2
 3     for (;; term.wait(250)) {
 4         for (master_t::const_iterator it = master.begin();
 5             it != master.end();) {
 6
 7             wabbit *const p = *it;
 8             const bool alive = p->move();
 9             ++it;
10
11             if (!alive) {
12                 //The wabbit that moved in line 8 blundered into
13                 //another wabbit and was eaten.
14                 delete p;
15             }
16
17             //Change lines 19-27 to fit your game.
18
19             if (count('r') <= 0) {
20                 term.put(0, 0, "No more rabbits.");
21                 return;
22             }
23
24             if (count('W') <= 0) {
25                 term.put(0, 0, "No more wolves.");
26                 return;
27             }
28         }
29     }
```

▲

▼ Homework 5.9.5b:
Version 3.2 of the Rabbit Game: mass extinction

The destructor for class `game` now destructs and deallocates all the surviving `wabbit`'s. If every one of these animals performed a beep and pause, it would drive you crazy.

Remove the beep and pause from the destructor for class `wabbit`. A `wabbit` will now beep and pause only when it is killed in an encounter with another `wabbit`.

Make these three changes:

(1) Remove the `g->term.beep();` and `g->term.wait(1000);` from the destructor for class `wabbit`.

(2) To make a `wabbit` beep and pause when another animal runs into it and eats it, insert `other->beep();` and `other->g->term.wait(1000);` at line 29½ of `wabbit.C` on p. 538.

(3) To make a `wabbit` beep and pause when it runs into an animal that eats it, insert `beep();` and `g->term.wait(1000);` at line 33½ of `wabbit.C` on p. 538.

We inserted two different beeps, the `other->beep()` and the plain `beep()` (i.e., `this->beep()`), to make each sound issue from the correct source. Our audio is currently monophonic, but it might become stereo.

▲

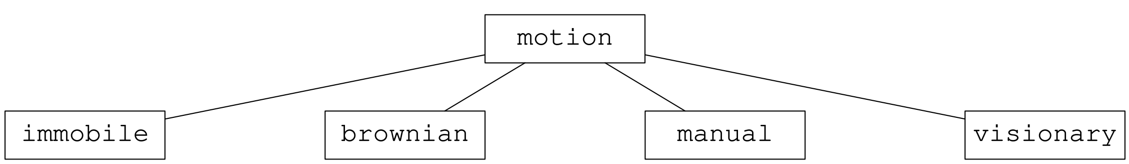
5.10 An Alternative to Inheritance

The above animals cannot change from one species to another. But the following scheme would allow a `rabbit` to turn into a `killer_rabbit` and back again. Or an animal could become sluggish after a big meal by temporarily turning into an `inert`.

The alternative scheme would no longer have any class `rabbit` or class `killer_rabbit`. In fact, there would no longer be any classes derived from `wabbit`. Instead, each `wabbit` will have pointers to two other objects that know how to move and eat: a `motion` object and a `rank` object.

This new scheme is an example of a *design pattern*. In particular, it's the "strategy" design pattern in the well-known Erich Gamma *Design Patterns* book, pp. 315–323.

These lines represent inheritance:



All four `motion` objects are static data members of classes derived from class `motion`. An object can be a static data member of its own class; our first example was the `origin` member of class `point` on p. 239. (Of course, an object cannot be a non-static data member of its own class; the object would blow up to infinite size.) Letting an object be a static data member of its own class ensures that at least one object of that class will be constructed. In our case, no additional ones should be constructed. This is the "singleton" design pattern in Gamma pp. 127–134.

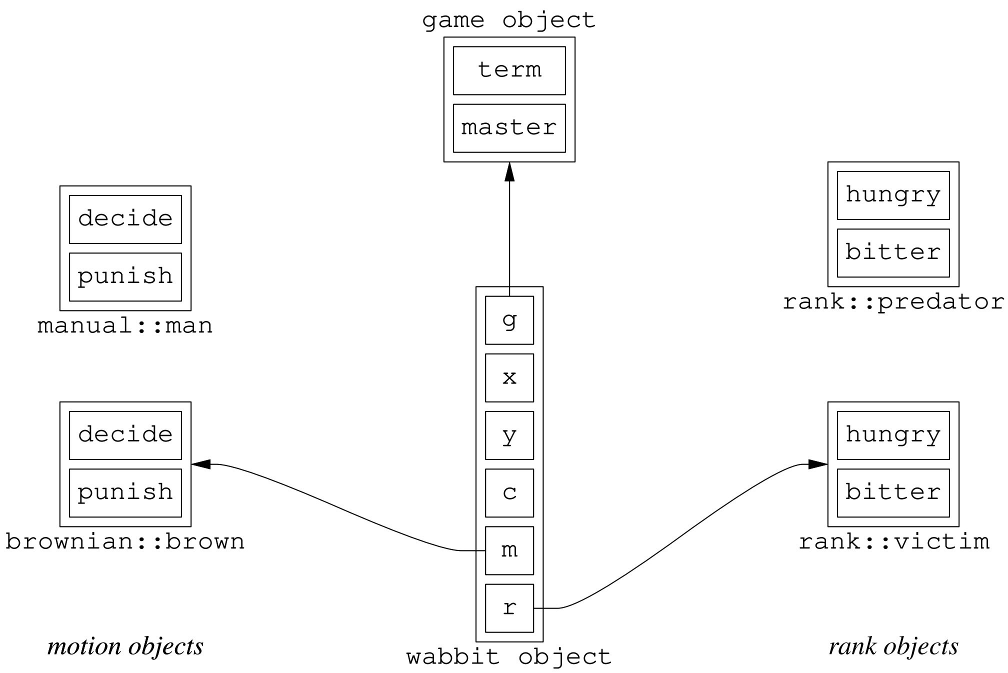
```
class motion {
public:
    virtual void decide(int *dx, int *dy) const = 0;
    virtual void punish() const = 0;
};

class immobile: public motion {
    void decide(int *dx, int *dy) const {*dx = *dy = 0;}
```

```
 9 public:
10     static const immobile imm;
11 };
12
13 class brownian: public motion {
14     void decide(int *dx, int *dy) const {
15         *dx = rand() % 3 - 1;
16         *dy = rand() % 3 - 1;
17     }
18 public:
19     static const brownian brown;
20 };
21
22 class manual: public motion {
23     void decide(int *dx, int *dy) const;
24     void punish() const;
25 public:
26     static const manual man;
27 };
28
29 class visionary: public motion {
30     void decide(int *dx, int *dy) const;
31 public:
32     static const visionary vis;
33 };
```

All four `rank` objects are static data members of class `rank`.

```
34 class rank {
35     const int h;
36     const int b;
37 public:
38     rank(int initial_h, int initial_b): h(initial_h), b(initial_b) {}
39
40     static const rank    inert(INT_MIN, INT_MAX);
41     static const rank   victim(INT_MIN, INT_MIN);
42     static const rank predator(INT_MAX, INT_MAX);
43     static const rank  halogen(INT_MAX, INT_MIN);
44
45     int hungry() const {return h;}
46     int bitter() const {return b;}
47 };
```

These arrows represent pointers. We saw on p. 253 that a normal pointer (lines 52–53) can point to a static data member (lines 80–84).




```
class wabbit {
    game *const g;
    unsigned x, y;
    char c;             //no longer const

    const rank *r;
    const motion *m;
public:
    bool move();
    //etc.
};

bool wabbit::move()
{
    int dx;                             //uninitialized variables
    int dy;
    m->decide(&dx, &dy);

    //etc.
    if (there is another wabbit) {
        const bool I_ate_him =  this->r->hungry() > other->r->bitter();
        const bool he_ate_me = other->r->hungry() >  this->r->bitter();

        //etc.

        if (neither ate the other) {
            m->punish();

    //etc.

    //One percent of the time, change the species of the object.
    if (rand() % 100 == 0) {
```

```
        r = either &rank::inert or &rank::victim or &rank::predator
            or &rank::halogen;

        m = either &immobile::imm or &brownian::brown or &manual::man
            or &visionary::vis;

        c = the character for the species we just turned into;
    }
}
```

5.11 Class `visionary`

Class visionary

Class `visionary` will be another motion class, like `immobile`, `brownian`, and `manual`. A `visionary`'s range of vision extends three units in every direction. The following diagram has a heavy line around the squares within visual range of a `visionary` in the center location. Each square is labeled with the distance from its center to the center of the square that holds the `visionary`.

$4\sqrt{2}$	5	$2\sqrt{5}$	$\sqrt{17}$	4	$\sqrt{17}$	$2\sqrt{5}$	5	$4\sqrt{2}$
5	$3\sqrt{2}$	$\sqrt{13}$	$\sqrt{10}$	3	$\sqrt{10}$	$\sqrt{13}$	$3\sqrt{2}$	5
$2\sqrt{5}$	$\sqrt{13}$	$2\sqrt{2}$	$\sqrt{5}$	2	$\sqrt{5}$	$2\sqrt{2}$	$\sqrt{13}$	$2\sqrt{5}$
$\sqrt{17}$	$\sqrt{10}$	$\sqrt{5}$	$\sqrt{2}$	1	$\sqrt{2}$	$\sqrt{5}$	$\sqrt{10}$	$\sqrt{17}$
4	3	2	1	0	1	2	3	4
$\sqrt{17}$	$\sqrt{10}$	$\sqrt{5}$	$\sqrt{2}$	1	$\sqrt{2}$	$\sqrt{5}$	$\sqrt{10}$	$\sqrt{17}$
$2\sqrt{5}$	$\sqrt{13}$	$2\sqrt{2}$	$\sqrt{5}$	2	$\sqrt{5}$	$2\sqrt{2}$	$\sqrt{13}$	$2\sqrt{5}$
5	$3\sqrt{2}$	$\sqrt{13}$	$\sqrt{10}$	3	$\sqrt{10}$	$\sqrt{13}$	$3\sqrt{2}$	5
$4\sqrt{2}$	5	$2\sqrt{5}$	$\sqrt{17}$	4	$\sqrt{17}$	$2\sqrt{5}$	5	$4\sqrt{2}$

With each move, a `visionary` animal will take one step away from an enemy within visual range. If there are several enemies, it will arbitrarily pick one. If there are no enemies close enough to see, the `visionary` will have the luxury of taking one step towards food. If there is no food either, the `visionary` will be lethargic and not move.

To test class `visionary`, we can derive `visionary victim`, known as a `deer`, and trap it between a pair of `immobile predator`'s, known as `black_hole`'s. A `deer` is a lowercase `d`, a `black_hole` is an uppercase `B`; the latter are subscripted for ytour convenience in the following diagrams.

If two or more enemies are in visual range, it would be hard to predict what our simple `visionary` will do. It will arbitrarily pick one enemy and recoil from it, ignoring the other. Similarly, if there are no enemies and two or more pieces of food, our `visionary` will arbitrarily pick one and head toward it, ignoring the other. To make the following `deer`'s behave predictably, we have only one `black_hole` in visual range of the `deer` at any given time.

The `deer` in our first two examples are driven back and forth between two `black_hole`'s. It bounces to and from the location marked with a `1`. A smarter `visionary` would escape at right angles instead of vibrating forever; an even smarter one would know that a `black_hole` is `immobile` and can be approached safely as long as we don't touch it.

	B_0			d	1			B_1	

	B_0						
			d				
				1			
						B_1	

These `deer`'s will be driven around and around the numbered paths:

				B_1					
				1	d			B_0	
	B_2			2	3				
					B_3				

						B_1		
	B_2							
				1				
			2		d			
				3				
							B_0	
		B_3						

We can use a carrot as well as a stick. Our carrot will be an `immobile` `victim`, known as a `sitting_duck` with a lowercase `s`. The next two examples assume a genetically-engineered `deer` whose hunger has been increased so that it can eat a `sitting_duck`, but whose bitterness is unchanged so it can still be eaten by a `black_hole`. Implement this by giving class `deer` the following public inline member function, overriding the `hungry` function that `deer` inherits from `victim`:

```
int hungry() const {return INT_MIN + 1;} //hungry enough to eat a victim
```

If an enemy and a meal are within visual range at the same time, the `visionary` will flee from the former and ignore the latter. For example, the `deer` at the starting position in the left diagram will flee from B_0 and ignore s_1.

							B_0		
	B_1				d		s_1		
				1		5			
			2		4				
		s_0		3				B_3	
		B_2							

					s_0		B_1		
	B_2								
					2				
	s_1		4	3	1				
				5	7	d		s_3	
				6					
								B_0	
		B_3		s_2					

A smart player would have the `wolf` corner the `deer` in a corner of the screen. Could an `alien` (a `visionary predator`) chase a `deer` across the board? Could a `deer` lure an `alien` into a `black_hole`?

▼ Homework 5.11a:
Version 3.3 of the Rabbit Game: a friend of class `wabbit`

It would seem natural to write the visual logic in `visionary::decide`. But we can't. The code will need to use the animal's `x` and `y` data members, and these are private members of class `wabbit`.

One possibility would be to expose the values of `x` and `y` to the derived classes. We can do this by making them public or protected, or giving class `wabbit` the following public or protected member functions.

```
	unsigned get_x() const {return x;}
	unsigned get_y() const {return y;}
```

But exposing the values is a dangerous narcotic. If a derived class becomes addicted to x, y coördinates, it will be hard to change the base class to polar coördinates.

Another possibility would be to write all the visionary logic in class `wabbit`. (Code follows the data members, p. 467.) But the logic doesn't belong there. It belongs in class `visionary`.

What would be the *smallest* piece of code we could add to class `wabbit` that would allow `visionary::decide` to do what it has to do? All we need is the `difference` friend of class `wabbit` in line 9. It will return the offset that would move us from the location of `wabbit w1` to the location of `wabbit w2`. For example, if `w1` was at (10, 10) and `w2` was to the upper right of `w1` at (13, 6), the return value would be (3, –4): three units to the right and four units up.

`difference` needs to use the private members `x` and `y` of class `wabbit`, so it must be a member function or a friend of that class. I made it a friend because it deals evenhandedly with two `wabbit`'s. Had it dealt with only one, or had one of them played a starring rôle, I would have made it a member function.

It doesn't matter whether a friend function is declared in the public, private, or protected section of its class. But as documentation, please declare it with the protected members of `wabbit` since `difference` is intended for use by a derived class.

`difference` begins by verifying that the two animals belong to the same game. It makes no sense to measure the distance and direction between animals in different games.

The subtractions in lines 15 and 16 must be able to yield positive, negative, or zero results. To get these signed results, both operands must be signed. The data members `x` and `y` are unsigned, so we cast them to `int` before the subtraction.

The cast would yield "implementation defined" results if `x` or `y` were greater than the maximum integer value `INT_MAX`. But a check for this would have been grim professionalism.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/visionary/wabbit.C`

```
//Excerpt from wabbit.C.

/*
Return the offset that would move us from the location of w1 to the
location of w2.  For example, if w1 was at (10, 10) and w2 was at (13, 6),
the return value would be (3, -4), i.e., 3 units to the right and 4 units up.
*/

void difference(const wabbit *w1, const wabbit *w2, int *dx, int *dy)
{
    if (w1->g != w2->g) {
        cerr and exit with EXIT_FAILURE;
    }

    *dx = static_cast<int>(w2->x) - static_cast<int>(w1->x);
    *dy = ditto: the vertical offset between the two wabbit's.
}
```

I'm sorry that `difference`, like `wabbit::decide`, returns its pair of answers through a pair of read/write pointer arguments. See the two workarounds, neither of them satisfactory, on p. 535. Eventually, however, `difference` will have a single return value, called a `difference_type`, which is why this function is named `difference`.

▲

▼ Homework 5.11b:
Version 3.4 of the Rabbit Game: three functions that are neither members nor friends

`visionary::decide` will need three more functions that deal with distances and directions: `signum`, `step`, and `dist`. They do not need to be member functions or friends of any class. Since they will be called only by each other and by `visionary::decide`, define them in the file `visionary.C` and let them be static to ensure that they can be called from no other file. Do not declare them in any header file. Here are the first lines of their definitions:

```
//Excerpts from visionary.C.

/*
Return 1 if the argument is positive, -1 if the argument is negative, 0 if 0.
*/
static int signum(int i)
{

/*
Return the offset that would take one step from the location of w1 to the
location of w2.  For example, if w1 was at (10, 10) and w2 was at (13, 6), the
return value would be (1, -1), i.e., one step diagonally to the upper right.
*/
static void step(const wabbit *w1, const wabbit *w2, int *dx, int *dy)
{

/*
Return the distance between w1 and w2.  For example, if w1 was at (10, 10) and
```

```
w2 was at (13, 6), the return value would be 5 == sqrt(3*3 + 4*4).
*/
static double dist(const wabbit *w1, const wabbit *w2)
{
```

`signum` means "sign" in Latin. If the function is small enough, let it be inline. In that case, you won't need the keyword `static`: an inline non-member function is static by default.

`step` will begin by calling `difference`. It will then call `signum` twice, to reduce the horizontal and vertical components of the offset to integers in the range −1 to 1 inclusive.

The `dist` function is so named to avoid conflict and confusion with the `distance` function in the C++ Standard Library. Like `step`, `dist` will begin by calling `difference`. It will then use the Pythagorean theorem $\sqrt{x^2 + y^2}$ to discover the length of the offset. The multiplication and addition should be `int`, not `double`, because `int` arithmetic is faster. Square each number by multiplying it by itself; this is faster than calling the `pow` function in the C++ Standard Library.

The C Standard Library has only one square root function:

```
/* Excerpt from <math.h> */

double sqrt(double);
```

The C++ Standard Library has three, not counting the one that takes a `valarray`.

```
//Excerpt from <cmath>

float sqrt(float);
double sqrt(double);
long double sqrt(long double);
```

You will therefore have to say which `sqrt` function you want. Do this by casting the sum $x^2 + y^2$ to `double` before passing it to `sqrt`. (Write a C++ `static_cast`, not a C `(double)` cast.) `visionary.C` will include `cmath` and say `using namespace std;` for the `sqrt` function.

▲

▼ Homework 5.11c:
Version 3.5 of the Rabbit Game: allow the derived classes to loop through the master list

`visionary::decide` will have to loop through the master list, searching for enemies and food. But this is currently impossible, since the master list is a private data member of another class. To give `visionary::decide` read-only access to the master list, add the following three protected members to class `wabbit`. The name of the data type `const_iterator` in lines 3–4 is created by the `typedef` in line 2.

```
1     //used by visionary::decide
2     typedef game::master_t::const_iterator const_iterator;
3     const_iterator begin() const {return g->master.begin();}
4     const_iterator   end() const {return g->master.end();}
```

▲

▼ Homework 5.11d:
Version 3.6 of the Rabbit Game: class `visionary`: step away from enemies and towards food

Derive class `visionary` from class `wabbit`, overriding `wabbit::decide`. Give class `visionary` no member functions except the constructor and `decide`. `visionary` will not override `wabbit::hungry` and `wabbit::bitter`, and so it will remain an abstract class. `visionary` will not override `wabbit::punish` either.

Since every `visionary` animal has the same radius of vision, and since the radius is used only in one function, we can make it a local static variable in line 5. But if the radius of each animal were different, it would have to be a non-static data member of class `visionary`.

The `const_iterator`, `begin`, and `end` in line 9 are the three new members of class `wabbit` in the previous Homework. They allow `visionary::decide` to loop through the master list without knowing that its name is `master` or even that it is a `list`.

No animal should be afraid of itself, and no animal should contemplate eating its own flesh. Accordingly, line 12 verifies that the `other` animal is not the same one as `this` one. To verify that two objects are not the same object, we compare their addresses. But `this` and `other` are pointers to different data types: `this` is a pointer to a `visionary`, while `other` is merely a pointer to a basic `wabbit`. To avoid any warning about comparing pointes to different types, we cast `this` to the greatest common denominator. Since we're inside a `const` member function, we must cast `this` to a read-only pointer.

The `this->` in line 14 is merely for rhetorical symmetry; it balances the `other->` in the same line. To get the `this->bitter()` to compile, `bitter` could be a protected or public member of class `wabbit`. But to get the `other->hungry()` to compile, `hungry` must be a *public* member of class `wabbit`. In a member function of class `visionary`, protected isn't good enough when the `other` object is not a visionary; see p. 495. By the time you have also coded the opposite relation, in lines 27–30, both functions will have to be public members of class `wabbit`. Update the comments in `wabbit.h` to explain why `hungry` and `bitter` must now be public.

In the rank classes derived from `wabbit` (`inert`, `victim`, etc.), the `hungry` and `bitter` functions can remain private.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/visionary/visionary.C`

```
1 //Excerpt from visionary.C.
2
3 void visionary::decide(int *dx, int *dy) const
4 {
5     static const unsigned radius = 3;   //of vision
6
7     //Move one step away from a wabbit that could eat me.
8
9     for (const_iterator it = begin(); it != end(); ++it) {
10         const wabbit *const other = *it;
11
12         if (other != static_cast<const wabbit *>(this) &&
13             dist(this, other) <= radius &&
14             other->hungry() > this->bitter()) {
15
16             step(other, this, dx, dy);
17             return;
18         }
19     }
20
21     /*
22     Arrive here if there were no enemies within the visual radius.
23     Now see if there's any food I could eat within the visual radius.
24     If so, take one step towards it.
25     */
26
27     for (const_iterator it =
28         do almost the same loop, ending with a step in the opposite
29         direction: step(this, other, ...
30     }
31
```

```
    //Arrive here if there were neither enemies nor food nearby:
    //lethargic (or random, if you wish) in the absence of stimulation.
    *dx = *dy = 0;
}
```

5.12 Private Inheritance and its Variants

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/cricket/cricket.h`

```
#ifndef CRICKETH
#define CRICKETH

class cricket {
    unsigned chirps;   //per 15 seconds
public:
    cricket(unsigned initial_chirps): chirps(initial_chirps) {}
    double fahrenheit() const {return chirps + 39;}
};
#endif
```

A `metric_cricket` can do everything that a `cricket` can do, plus more.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/cricket/metric_cricket.h`

```
#ifndef METRIC_CRICKETH
#define METRIC_CRICKETH
#include "cricket.h"

class metric_cricket: public cricket {
public:
    metric_cricket(unsigned initial_chirps): cricket(initial_chirps) {}
    double celsius() const {return (fahrenheit() - 32) * 5 / 9;}
};
#endif
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/cricket/main2.C`

```
#include <iostream>
#include <cstdlib>
#include "metric_cricket.h"
using namespace std;

int main()
{
    metric_cricket mc(33);
    cout << "celsius == " << mc.celsius() << "\n";
    cout << "fahrenheit == " << mc.fahrenheit() << "\n";

    cricket *p = &mc;
    cout << "fahrenheit == " << p->fahrenheit() << "\n";

    cricket& r = mc;
```

```
16     cout << "fahrenheit == " << r.fahrenheit() << "\n";
17
18     return EXIT_SUCCESS;
19 }
```

```
celsius == 22.2222
fahrenheit == 72
fahrenheit == 72
fahrenheit == 72
```

More precisely, the keyword `public` in line 5 of `metric_cricket.h` does two things:

(1) It lets the public members of `cricket` become public members of `metric_cricket`. For example, that's why line 10 of `main2.C` can use the `fahrenheit` member of `mc`.

(2) It lets a pointer to a `cricket` point to a `metric_cricket` (line 12 of `main2.C`) and lets a reference to a `cricket` refer to a `metric_cricket` (line 15 of `main2.C`) without needing a cast. A pointer to a base class can always point to an object of a (publicly) derived class.

But if we changed the `public` to `private` in line 5 of `metric_cricket.h`, the above two things would change:

(1′) The public members of `cricket` would now be *private* members of `metric_cricket`. Thus the `fahrenheit` member of class `cricket` could no longer be called for the object `mc` in line 10 of `main2.C`, although it still could be called by the `celsius` member function in line 8 of `metric_cricket.h`.

(2′) A pointer to a `cricket` could no longer point to a `metric_cricket` (line 12 of `main2.C`), and a reference to a `cricket` could no longer refer to a `metric_cricket` (line 15 of `main2.C`). It would be a secret that class `metric_cricket` is derived from class `cricket`.

Interface inheritance vs. implementation inheritance

A class's public members are called its *user interface.* With this definition we can state the two reasons we build a derived class from a base class:

(1) We want to endow the derived class with the same user interface as the base class, plus more. In this case, we use `public` inheritance, also called *interface inheritance* or *type inheritance.*

(2) We want to endow the derived class with all of the functionality of the base class (e.g., the ability to compute the temperature from the chirping speed), but we want to force the user to use a totally different interface. In this case, we use `private` inheritance, also called *implementation inheritance.* (Note that public derivation actually gives us implementation inheritance as well as interface inheritance.)

Protected inheritance

There is also *protected inheritance,* in which the public members of the base class become protected members of the derived class. The following table shows how accessible a member of a base class would be in each kind of derived class. For example, in public inheritance, the public members of the base class become public members of the derived class. And in every kind of inheritance, the private members of the base class are mentionable only by the base class.

	member of base class is		
	`public`	`protected`	`private`
base class is `public`	`public`	`protected`	*unmentionable*
base class is `protected`	`protected`	`protected`	*unmentionable*
base class is `private`	`private`	`private`	*unmentionable*

▼ Homework 5.12a: Version 3.7 of the Rabbit Game: private inheritance

There is no reason to derive each grandchild class publicly from its two parents. Derive them privately by changing the two `public` keywords to `private` in each grandchild class.

We would also like to derive the motion and rank classes privately, e.g., deriving `brownian` and `victim` privately from class `wabbit`. But if we did this, no other class would know that `brownian` and `victim` are derived from `wabbit`. In particular, a grandchild class such as `rabbit` would be unaware of its own `wabbit` ancestry, and the constructor for `rabbit` would be unable to make the direct call the constructor for its grandparent `wabbit`.

To permit the constructor for a grandchild to call the constructor for `wabbit`, we must give every grandchild at least one parent that is derived publicly or protectedly from class `wabbit`. We arbitrarily decide to derive the motion classes (`immobile`, `brownian`, `manual`, `visionary`) protectedly from `wabbit`, and the rank classes (`inert`, `victim`, `predator`, `halogen`) privately from `wabbit`. (Alternatively, we could have derived the motion classes privately and the rank classes protectedly.) As long as the grandchild knows that at least one parent is derived from `wabbit`, the grandchild will be able to mention `wabbit`.

Now that the inheritance is no longer public, we are guaranteed that the member functions of class `game` (other than `get` and `count`, which are friends of class `wabbit`) will never be able to make direct calls to `decide` and the other non-public member functions of `wabbit`.

▲

Partial inheritance

A derived class can inherit all of the implementation but only part of the interface of a base class. To do this, use private inheritance and the *using declaration* in line 13. By writing the declaration in the public section of class `derived`, we have made `base::f` a public member of `derived`. `base::g` is also present in class `derived`, but only as a private member.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/cricket/using.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 class base {
 6 public:
 7     void f() const {cout << "base::f\n";}
 8     void g() const {cout << "base::g\n";}
 9 };
10
11 class derived: private base {
12 public:
13     using base::f;   //using declaration
14 };
15
16 int main()
17 {
18     derived d;
19     d.f();      //will compile
20     //d.g();    //won't compile
21     return EXIT_SUCCESS;
22 }
```

```
base::f
```

A using declaration is also used in a “namespace”; see p. 1023.