PREV
3. Modularity

NEXT
5. Templates

4. Classes

Those types are not "abstract";
they are as real as int *and* float.

– Doug McIlroy

• Advice

4.1. INTRODUCTION

This chapter and the next aim to give you an idea of C++'s support for abstraction and resource management without going into a lot of detail:

• This chapter informally presents ways of defining and using new types (*user-defined types*). In particular, it presents the basic properties, implementation techniques, and language facilities used for *concrete classes*, *abstract classes*, and *class hierarchies*.

• The next chapter introduces templates as a mechanism for parameterizing types and algorithms with (other) types and algorithms. Computations on user-defined and built-in types are represented as functions, sometimes generalized to *template functions* and *function objects*.

These are the language facilities supporting the programming styles known as *object-oriented programming* and *generic programming*. Chapters 6-13 follow up by presenting examples of standard-library facilities and their use.

The central language feature of C++ is the *class*. A class is a user-defined type provided to represent a concept in the code of a program. Whenever our design for a program has a useful concept, idea, entity, etc., we try to represent it as a class in the program so that the idea is there in the code, rather than just in our head, in a design document, or in some comments. A program built out of a well chosen set of classes is far easier to understand and get right than one that builds everything directly in terms of the built-in types. In particular, classes are often what libraries offer.

Essentially all language facilities beyond the fundamental types, operators, and statements exist to help define better classes or to use them more conveniently. By "better," I mean more correct, easier to maintain, more efficient, more elegant, easier to use, easier to read, and easier to reason about. Most programming techniques rely on the design and implementation of specific kinds of classes. The needs and tastes of programmers vary immensely. Consequently, the support for classes is extensive. Here, we will just consider the basic support for three important kinds of classes:

• Concrete classes (§4.2)

• Abstract classes (§4.3)

• Classes in class hierarchies (§4.5)

An astounding number of useful classes turn out to be of these three kinds. Even more classes can be seen as simple variants of these kinds or are im-

plemented using combinations of the techniques used for these.

4.2. CONCRETE TYPES

The basic idea of *concrete classes* is that they behave "just like built-in types." For example, a complex number type and an infinite-precision integer are much like built-in int, except of course that they have their own semantics and sets of operations. Similarly, a vector and a string are much like built-in arrays, except that they are better behaved (§7.2, §8.3, §9.2).

The defining characteristic of a concrete type is that its representation is part of its definition. In many important cases, such as a vector, that representation is only one or more pointers to data stored elsewhere, but it is present in each object of a concrete class. That allows implementations to be optimally efficient in time and space. In particular, it allows us to

• place objects of concrete types on the stack, in statically allocated memory, and in other objects (§1.6);

• refer to objects directly (and not just through pointers or references);

• initialize objects immediately and completely (e.g., using constructors; §2.3); and

• copy objects (§4.6).

The representation can be private (as it is for Vector; §2.3) and accessible only through the member functions, but it is present. Therefore, if the representation changes in any significant way, a user must recompile. This is the price to pay for having concrete types behave exactly like built-in types. For types that don't change often, and where local variables provide much-needed clarity and efficiency, this is acceptable and often ideal. To increase flexibility, a concrete type can keep major parts of its representation on the free store (dynamic memory, heap) and access them through the part stored in the class object itself. That's the way vector and string are implemented; they can be considered resource handles with carefully crafted interfaces.

4.2.1. An Arithmetic Type

The "classical user-defined arithmetic type" is complex:

Click here to view code image

```
class complex {
        double re, im; //  representation: two doubles
public:
```

```
        complex(double r, double i) :re{r}, im{i} {}      // construct c
        complex(double r) :re{r}, im{0} {}                  // constru
        complex() :re{0}, im{0} {}                            // defau

        double real() const { return re; }
        void real(double d) { re=d; }
        double imag() const { return im; }
        void imag(double d) { im=d; }

        complex& operator+=(complex z) { re+=z.re, im+=z.im; retur

        complex& operator-=(complex z) { re-=z.re, im-=z.im; retur
        complex& operator*=(complex);    // defined out-of-class s
        complex& operator/=(complex);    // defined out-of-class s
};
```

This is a slightly simplified version of the standard-library complex (§12.4). The class definition itself contains only the operations requiring access to the representation. The representation is simple and conventional. For practical reasons, it has to be compatible with what Fortran provided 50 years ago, and we need a conventional set of operators. In addition to the logical demands, complex must be efficient or it will remain unused. This implies that simple operations must be inlined. That is, simple operations (such as constructors, +=, and imag()) must be implemented without function calls in the generated machine code. Functions defined in a class are inlined by default. It is possible to explicitly require inlining by preceeding a function declaration with the keyword inline. An industrial-strength complex (like the standard-library one) is carefully implemented to do appropriate inlining.

A constructor that can be invoked without an argument is called a *default constructor*. Thus, complex() is complex's default constructor. By defining a default constructor you eliminate the possibility of uninitialized variables of that type.

The const specifiers on the functions returning the real and imaginary parts indicate that these functions do not modify the object for which they are called.

Many useful operations do not require direct access to the representation of complex, so they can be defined separately from the class definition:

Click here to view code image

```
complex operator+(complex a, complex b) { return a+=b; }
complex operator–(complex a, complex b) { return a–=b; }
complex operator–(complex a){return {–a.real(), –a.imag()}; }
complex operator*(complex a, complex b) { return a*=b; }
complex operator/(complex a, complex b) { return a/=b; }
```

Here, I use the fact that an argument passed by value is copied, so that I can modify an argument without affecting the caller's copy, and use the result as the return value.

The definitions of `==` and `!=` are straightforward:

Click here to view code image

```
bool operator==(complex a, complex b)                // equal
{
        return a.real()==b.real() && a.imag()==b.imag();
}

bool operator!=(complex a, complex b)                // not equal
{
        return !(a==b);
}

complex sqrt(complex);           // the definition is elsewhere

// ...
```

Class `complex` can be used like this:

Click here to view code image

```
void f(complex z)
{
        complex a {2.3};             // construct {2.3,0.0} from 2.3
        complex b {1/a};
        complex c {a+z*complex {1,2.3}};
        // ...
        if (c != b)
```

```
        c = –(b/a)+2*b;
}
```

The compiler converts operators involving `complex` numbers into appropriate function calls. For example, `c!=b` means `operator!=(c,b)` and `1/a` means `operator/(complex{1},a)`.

User-defined operators ("overloaded operators") should be used cautiously and conventionally. The syntax is fixed by the language, so you can't define a unary `/`. Also, it is not possible to change the meaning of an operator for built-in types, so you can't redefine `+` to subtract `int`s.

4.2.2. A Container

A *container* is an object holding a collection of elements, so we call `Vector` a container because it is the type of objects that are containers. As defined in §2.3, `Vector` isn't an unreasonable container of `double`s: it is simple to understand, establishes a useful invariant (§3.4.2), provides range-checked access (§3.4.1), and provides `size()` to allow us to iterate over its elements. However, it does have a fatal flaw: it allocates elements using `new` but never deallocates them. That's not a good idea because although C++ defines an interface for a garbage collector (§4.6.4), it is not guaranteed that one is available to make unused memory available for new objects. In some environments you can't use a collector, and sometimes you prefer more precise control of destruction for logical or performance reasons. We need a mechanism to ensure that the memory allocated by the constructor is deallocated; that mechanism is a *destructor*:

Click here to view code image

```
class Vector {
private:
        double* elem;           // elem points to an array of sz doubl
        int sz;
public:
        Vector(int s) :elem{new double[s]}, sz{s}           // constru
        {
                for (int i=0; i!=s; ++i)          // initialize elements
                        elem[i]=0;
        }

        ~Vector() { delete[] elem; }                              // dest
```

```
        double& operator[](int i);
        int size() const;
};
```

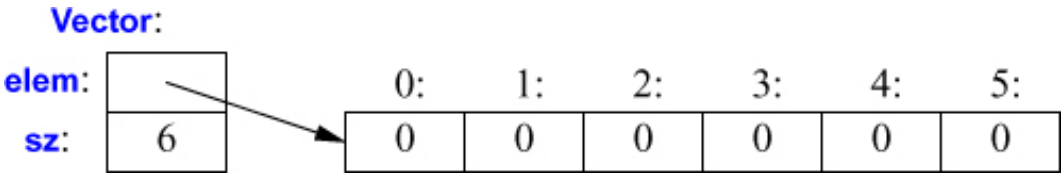
The name of a destructor is the complement operator, ~, followed by the name of the class; it is the complement of a constructor. Vector's constructor allocates some memory on the free store (also called the *heap* or *dynamic store*) using the new operator. The destructor cleans up by freeing that memory using the delete operator. This is all done without intervention by users of Vector. The users simply create and use Vectors much as they would variables of built-in types. For example:

Click here to view code image

```
void fct(int n)
{
        Vector v(n);

        // ... use v ...

        {
                Vector v2(2*n);
                // ... use v and v2 ...
        } // v2 is destroyed here

        // ... use v ..

} // v is destroyed here
```

Vector obeys the same rules for naming, scope, allocation, lifetime, etc. (§1.6), as does a built-in type, such as int and char. This Vector has been simplified by leaving out error handling; see §3.4.

The constructor/destructor combination is the basis of many elegant techniques. In particular, it is the basis for most C++ general resource management techniques (§11.2). Consider a graphical illustration of a Vector:



The constructor allocates the elements and initializes the Vector members appropriately. The destructor deallocates the elements. This *handle-to-data model* is very commonly used to manage data that can vary in size during the lifetime of an object. The technique of acquiring resources in a constructor and releasing them in a destructor, known as *Resource Acquisition Is Initialization* or *RAII*, allows us to eliminate "naked new operations," that is, to avoid allocations in general code and keep them buried inside the implementation of well-behaved abstractions. Similarly, "naked delete operations" should be avoided. Avoiding naked new and naked delete makes code far less error-prone and far easier to keep free of resource leaks (§11.2).

4.2.3. Initializing Containers

A container exists to hold elements, so obviously we need convenient ways of getting elements into a container. We can handle that by creating a Vector with an appropriate number of elements and then assigning to them, but typically other ways are more elegant. Here, I just mention two favorites:

• *Initializer-list constructor*: Initialize with a list of elements.

• push_back(): Add a new element at the end (at the back of) the sequence.

These can be declared like this:

Click here to view code image

```
class Vector {
public:
        Vector(std::initializer_list<double>);        // initialize with a
        // ...
        void push_back(double);                       // add ele
        // ...
};
```

The push_back() is useful for input of arbitrary numbers of elements. For example:

Click here to view code image

```
Vector read(istream& is)
{
        Vector v;
        for (double d; is>>d;)          // read floating-point values into
```

```
            v.push_back(d);          // add d to v
        return v;
    }
```

The input loop is terminated by an end-of-file or a formatting error. Until that happens, each number read is added to the `Vector` so that at the end, `v`'s size is the number of elements read. I used a `for`-statement rather than the more conventional `while`-statement to keep the scope of `d` limited to the loop. The way to provide `Vector` with a move constructor, so that returning a potentially huge amount of data from `read()` is cheap, is explained in §4.6.2.

The `std::initializer_list` used to define the initializer-list constructor is a standard-library type known to the compiler: when we use a `{}`-list, such as `{1,2,3,4}`, the compiler will create an object of type `initializer_list` to give to the program. So, we can write:

Click here to view code image

```
Vector v1 = {1,2,3,4,5};                  // v1 has 5 elements
Vector v2 = {1.23, 3.45, 6.7, 8};         // v2 has 4 elements
```

`Vector`'s initializer-list constructor might be defined like this:

Click here to view code image

```
Vector::Vector(std::initializer_list<double> lst)  // initialize with a li
    :elem{new double[lst.size()]}, sz{static_cast<int>(lst.size())}
{
    copy(lst.begin(),lst.end(),elem);              // copy from lst into 
}
```

I use the ugly `static_cast` (§14.2.3) to convert the size of the initializer list to an `int`. This is pedantic because the chance that the number of elements in a hand-written list is larger than the largest integer (32,767 for 16-bit integers and 2,147,483,647 for 32-bit integers) is rather low. However, it is worth remembering that the type system has no common sense. It knows about the possible values of variables, rater than actual values, so it might complain where there is no actual violation. However, sooner or later, such warnings will save the programmer from a bad error.

A static_cast is does not check the value it is converting; the programmer is trusted to use it correctly. This is not always a good assumption, so if in doubt, check the value. Explicit type conversions (often called *casts* to remind you that they are used to prop up something broken) are best avoided. Judicious use of the type system and well-designed libraries allow us to eliminate unchecked cast in higher-level software.

4.3. ABSTRACT TYPES

Types such as complex and Vector are called *concrete types* because their representation is part of their definition. In that, they resemble built-in types. In contrast, an *abstract type* is a type that completely insulates a user from implementation details. To do that, we decouple the interface from the representation and give up genuine local variables. Since we don't know anything about the representation of an abstract type (not even its size), we must allocate objects on the free store (§4.2.2) and access them through references or pointers (§1.8, §11.2.1).

First, we define the interface of a class Container which we will design as a more abstract version of our Vector:

Click here to view code image

```
class Container {
public:
        virtual double& operator[](int) = 0;      // pure virtual functi
        virtual int size() const = 0;              // const member fu
        virtual ~Container() {}                    // destructor (§4
};
```

This class is a pure interface to specific containers defined later. The word virtual means "may be redefined later in a class derived from this one." Unsurprisingly, a function declared virtual is called a *virtual function*. A class derived from Container provides an implementation for the Container interface. The curious =0 syntax says the function is *pure virtual*; that is, some class derived from Container *must* define the function. Thus, it is not possible to define an object that is just a Container; a Container can only serve as the interface to a class that implements its operator[]() and size() functions. A class with a pure virtual function is called an *abstract class*.

This Container can be used like this:

```
void use(Container& c)
{
        const int sz = c.size();

        for (int i=0; i!=sz; ++i)
                cout << c[i] << '\n';
}
```

Note how use() uses the Container interface in complete ignorance of implementation details. It uses size() and [ ] without any idea of exactly which type provides their implementation. A class that provides the interface to a variety of other classes is often called a *polymorphic type*.

As is common for abstract classes, Container does not have a constructor. After all, it does not have any data to initialize. On the other hand, Container does have a destructor and that destructor is virtual. Again, that is common for abstract classes because they tend to be manipulated through references or pointers, and someone destroying a Container through a pointer has no idea what resources are owned by its implementation; see also §4.5.

A container that implements the functions required by the interface defined by the abstract class Container could use the concrete class Vector:

Click here to view code image

```
class Vector_container : public Container {     // Vector_container
        Vector v;
public:
        Vector_container(int s) : v(s) { }  // Vector of s elements
        ~Vector_container() {}

        double& operator[](int i) { return v[i]; }
        int size() const { return v.size(); }
};
```

The :public can be read as “is derived from” or “is a subtype of.” Class Vector_container is said to be *derived* from class Container, and class Container is said to be a *base* of class Vector_container. An alternative terminology calls Vector_container and Container *subclass* and *superclass*, respectively. The derived class is said to inherit members from its base class,

so the use of base and derived classes is commonly referred to as *inheritance*.

The members operator[]() and size() are said to *override* the corresponding members in the base class Container. The destructor (~Vector_container()) overrides the base class destructor (~Container()). Note that the member destructor (~Vector()) is implicitly invoked by its class's destructor (~Vector_container()).

For a function like use(Container&) to use a Container in complete ignorance of implementation details, some other function will have to make an object on which it can operate. For example:

Click here to view code image

```
void g()
{
        Vector_container vc {10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
        use(vc);
}
```

Since use() doesn't know about Vector_containers but only knows the Container interface, it will work just as well for a different implementation of a Container. For example:

Click here to view code image

```
class List_container : public Container {          // List_containe
        std::list<double> ld;       // (standard-library) list of double
public:
        List_container() { }        // empty List
        List_container(initializer_list<double> il) : ld{il} { }
        ~List_container() {}

        double& operator[](int i);
        int size() const { return ld.size(); }

};

double& List_container::operator[](int i)
{
      for (auto& x : ld) {
```

```
            if (i==0) return x;
            --i;
        }
        throw out_of_range("List container");
    }
```

Here, the representation is a standard-library list<double>. Usually, I would not implement a container with a subscript operation using a list, because performance of list subscripting is atrocious compared to vector subscripting. However, here I just wanted to show an implementation that is radically different from the usual one.

A function can create a List_container and have use() use it:

Click here to view code image

```
void h()
{
        List_container lc = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
        use(lc);
}
```

The point is that use(Container&) has no idea if its argument is a Vector_container, a List_container, or some other kind of container; it doesn't need to know. It can use any kind of Container. It knows only the interface defined by Container. Consequently, use(Container&) needn't be recompiled if the implementation of List_container changes or a brand-new class derived from Container is used.

The flip side of this flexibility is that objects must be manipulated through pointers or references (§4.6, §11.2.1).

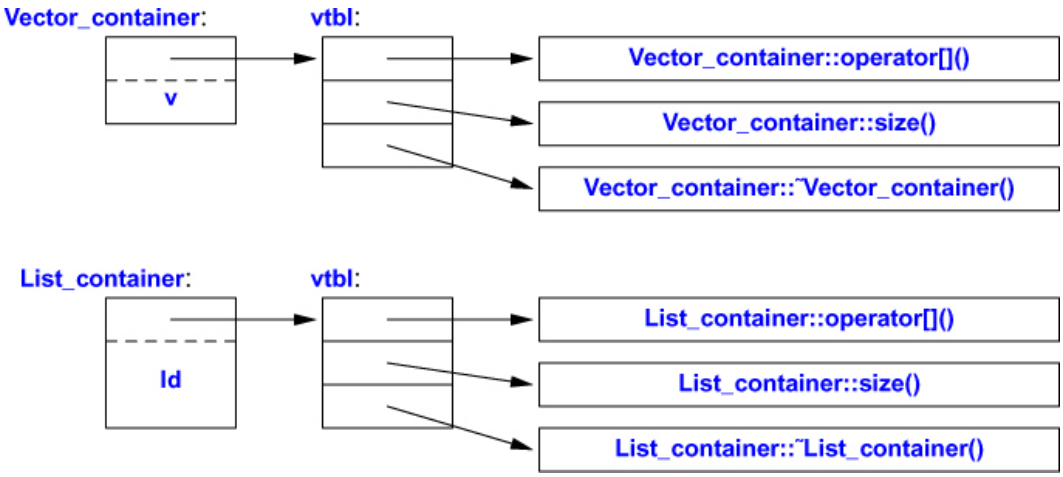
4.4. VIRTUAL FUNCTIONS

Consider again the use of Container:

```
void use(Container& c)
{
        const int sz = c.size();

        for (int i=0; i!=sz; ++i)
```

```
                cout << c[i] << '\n';
    }
```

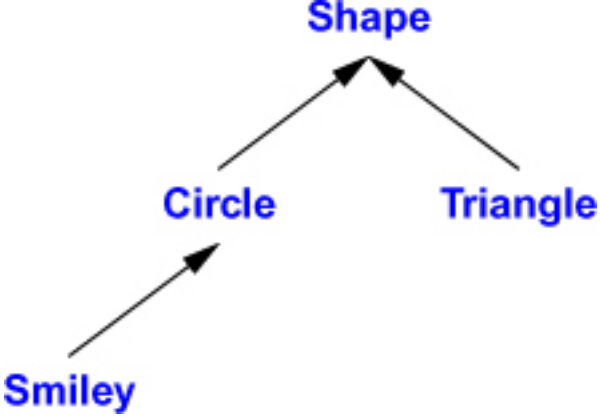
How is the call `c[i]` in `use()` resolved to the right `operator[]()`? When `h()` calls `use()`, `List_container`'s `operator[]()` must be called. When `g()` calls `use()`, `Vector_container`'s `operator[]()` must be called. To achieve this resolution, a `Container` object must contain information to allow it to select the right function to call at run time. The usual implementation technique is for the compiler to convert the name of a virtual function into an index into a table of pointers to functions. That table is usually called the *virtual function table* or simply the `vtbl`. Each class with virtual functions has its own `vtbl` identifying its virtual functions. This can be represented graphically like this:



The functions in the `vtbl` allow the object to be used correctly even when the size of the object and the layout of its data are unknown to the caller. The implementation of the caller needs only to know the location of the pointer to the `vtbl` in a `Container` and the index used for each virtual function. This virtual call mechanism can be made almost as efficient as the "normal function call" mechanism (within 25%). Its space overhead is one pointer in each object of a class with virtual functions plus one `vtbl` for each such class.

4.5. CLASS HIERARCHIES

The `Container` example is a very simple example of a class hierarchy. A *class hierarchy* is a set of classes ordered in a lattice created by derivation (e.g., `: public`). We use class hierarchies to represent concepts that have hierarchical relationships, such as "A fire engine is a kind of a truck which is a kind of a vehicle" and "A smiley face is a kind of a circle which is a kind of a shape." Huge hierarchies, with hundreds of classes, that are both deep and wide are common. As a semi-realistic classic example, let's consider shapes on a screen:



The arrows represent inheritance relationships. For example, class **Circle** is derived from class **Shape**. To represent that simple diagram in code, we must first specify a class that defines the general properties of all shapes:

Click here to view code image

```
class Shape {
public:
        virtual Point center() const =0;        // pure virtual
        virtual void move(Point to) =0;

        virtual void draw() const = 0;          // draw on current "Ca
        virtual void rotate(int angle) = 0;

        virtual ~Shape() {}                     // destructor
        // ...
};
```

Naturally, this interface is an abstract class: as far as representation is concerned, *nothing* (except the location of the pointer to the **vtbl**) is common for every **Shape**. Given this definition, we can write general functions manipulating vectors of pointers to shapes:

Click here to view code image

```
void rotate_all(vector<Shape*>& v, int angle) // rotate v's element
{
        for (auto p : v)
                p->rotate(angle);
}
```

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

Click here to view code image

```
class Circle : public Shape {
public:
        Circle(Point p, int rr);            // constructor

        Point center() const { return x; }
        void move(Point to) { x=to; }
        void draw() const;
        void rotate(int) {}                 // nice simple algorithm
private:
        Point x;     // center
        int r;       // radius
};
```

So far, the Shape and Circle example provides nothing new compared to the Container and Vector_container example, but we can build further:

Click here to view code image

```
class Smiley : public Circle {      // use the circle as the base for a
public:
        Smiley(Point p, int r) : Circle{p,r}, mouth{nullptr} { }

        ~Smiley()
        {
                delete mouth;
                for (auto p : eyes)
                        delete p;
        }

        void move(Point to);

        void draw() const;
        void rotate(int);

        void add_eye(Shape* s) { eyes.push_back(s); }
        void set_mouth(Shape* s);
```

```
        virtual void wink(int i);       // wink eye number i

        // ...

private:
        vector<Shape*> eyes;            // usually two eyes
        Shape* mouth;
};
```

The `push_back()` member function adds its argument to the `vector` (here, `eyes`), increasing that vector's size by one.

We can now define `Smiley::draw()` using calls to `Smiley`'s base and member `draw()`s:

```
void Smiley::draw()
{
        Circle::draw();
        for (auto p : eyes)
                p->draw();
        mouth->draw();
}
```

Note the way that `Smiley` keeps its eyes in a standard-library `vector` and deletes them in its destructor. `Shape`'s destructor is `virtual` and `Smiley`'s destructor overrides it. A virtual destructor is essential for an abstract class because an object of a derived class is usually manipulated through the interface provided by its abstract base class. In particular, it may be deleted through a pointer to a base class. Then, the virtual function call mechanism ensures that the proper destructor is called. That destructor then implicitly invokes the destructors of its bases and members.

In this simplified example, it is the programmer's task to place the eyes and mouth appropriately within the circle representing the face.

We can add data members, operations, or both as we define a new class by derivation. This gives great flexibility with corresponding opportunities for confusion and poor design.

4.5.1. Explicit Overriding

A function in a derived class overrides a virtual function in a base class if that function has exactly the same name and type. In large hierachies, it is not always obvious if overriding was intended. A function with a slightly different name or a slightly different type may be intended to override or it may be intended to be a separate function. To avoid confusion in such cases, a programmer can explicitly state that a function is meant to override. For example, I could (equivalently) have defined Smiley like this:

Click here to view code image

```
class Smiley : public Circle { //  use the circle as the base for a fac
public:
        Smiley(Point p, int r) : Circle{p,r}, mouth{nullptr} { }

        ~Smiley()
        {
                delete mouth;
                for (auto p : eyes)
                        delete p;
        }

        void move(Point to) override;

        void draw() const override;
        void rotate(int) override;

        void add_eye(Shape* s) { eyes.push_back(s); }
        void set_mouth(Shape* s);
        virtual void wink(int i);        //  wink eye number i

        //  ...

private:
        vector<Shape*> eyes;                //  usually two eyes
        Shape* mouth;
};
```

Now, had I mistyped move as mve, I would have gotten an error because no base of Smiley has a virtual function called mve. Similarly, had I added override to the declaration of wink(), I would have gotten an error message.

4.5.2. Benefits from Hierarchies

A class hierarchy offers two kinds of benefits:

• *Interface inheritance*: An object of a derived class can be used wherever an object of a base class is required. That is, the base class acts as an interface for the derived class. The Container and Shape classes are examples. Such classes are often abstract classes.

• *Implementation inheritance*: A base class provides functions or data that simplifies the implementation of derived classes. Smiley's uses of Circle's constructor and of Circle::draw() are examples. Such base classes often have data members and constructors.

Concrete classes – especially classes with small representations – are much like built-in types: we define them as local variables, access them using their names, copy them around, etc. Classes in class hierarchies are different: we tend to allocate them on the free store using new, and we access them through pointers or references. For example, consider a function that reads data describing shapes from an input stream and constructs the appropriate Shape objects:

Click here to view code image

```
enum class Kind { circle, triangle, smiley };

Shape* read_shape(istream& is)          // read shape descriptior
{
    // ... read shape header from is and find its Kind k ...

    switch (k) {
    case Kind::circle:
        // read circle data {Point,int} into p and r
        return new Circle{p,r};
    case Kind::triangle:
        // read triangle data {Point,Point,Point} into p1, p2, and
        return new Triangle{p1,p2,p3};
    case Kind::smiley:
        // read smiley data {Point,int,Shape,Shape,Shape} into
        Smiley* ps = new Smiley{p,r};
        ps->add_eye(e1);
        ps->add_eye(e2);
        ps->set_mouth(m);
        return ps;
```

```
    }
}
```

A program may use that shape reader like this:

Click here to view code image

```
void user()
{
        std::vector<Shape*> v;
        while (cin)
                v.push_back(read_shape(cin));
        draw_all(v);                    // call draw() for each elemen
        rotate_all(v,45);               // call rotate(45) for each eleme
        for (auto p : v)                // remember to delete elements
                delete p;
}
```

Obviously, the example is simplified – especially with respect to error handling – but it vividly illustrates that `user()` has absolutely no idea of which kinds of shapes it manipulates. The `user()` code can be compiled once and later used for new `Shape`s added to the program. Note that there are no pointers to the shapes outside `user()`, so `user()` is responsible for deallocating them. This is done with the `delete` operator and relies critically on `Shape`'s virtual destructor. Because that destructor is virtual, `delete` invokes the destructor for the most derived class. This is crucial because a derived class may have acquired all kinds of resources (such as file handles, locks, and output streams) that need to be released. In this case, a `Smiley` deletes its `eyes` and `mouth` objects.

4.5.3. Hierarchy Navigation

The `read_shape()` function returns `Shape *` so that we can treat all `Shapes` alike. However, what can we do if we want to use a member function that is only provided by a particular derived class, such as `Smiley`'s `wink()`? We can ask "is this `Shape` a kind of `Smiley`?" using the `dynamic_cast` operator:

Click here to view code image

```
Shape* ps {read_shape(cin)};
```

```
if (Smiley* p = dynamic_cast<Smiley*>(ps)) {
        // ... is the Smiley pointer to by p ...
}
else {
        // ... not a Smiley, try something else ...
}
```

If the object pointed to by the argument of dynamic_cast (here, ps) is not of the expected type (here, Smiley) or a class derived from the expected type, dynamic_cast returns nullptr.

We use dynamic_cast to a pointer type when a pointer to an object of a different derived class is a valid argument. We then test whether the result is nullptr. This test can often conveniently be placed in the initialization of a variable in a condition.

When a different type is unacceptable, we can simply dynamic_cast to a reference type. If the object is not of the expected type, bad_cast is thrown:

Click here to view code image

```
Shape* ps {read_shape(cin)};
Smiley& r {dynamic_cast<Smiley&>(*ps)};      // somewhere, catc
```

Code is cleaner when dynamic_cast is used with restraint. If we can avoid using type information, we can write simpler and more efficient code, but occasionally type information is lost and must be recovered. This typically happens when we pass an object to some system that accepts an interface specified by a base class. When that system later passes the object back to use, we might have to recover the original type. Operations similar to dynamic_cast are known as “is kind of” and “is instance of” operations.

4.5.4. Avoiding Resource Leaks

Experienced programmers will notice that I left open two obvious opportunities for mistakes:

• A user might fail to delete the pointer returned by read_shape().

• The owner of a container of Shape pointers might not delete the objects pointed to.

In that sense, functions returning a pointer to an object allocated on the free store are dangerous.

One solution to both problems is to return a standard-library unique_ptr (§11.2.1) rather than a "naked pointer" and store unique_ptrs in the container:

Click here to view code image

```
unique_ptr<Shape> read_shape(istream& is) // read shape descrip
{
        // read shape header from is and find its Kind k

        switch (k) {
        case Kind::circle:
                // read circle data {Point,int} into p and r
                return unique_ptr<Shape>{new Circle{p,r}};          //
        // ...
}

void user()
{
        vector<unique_ptr<Shape>> v;
        while (cin)
                v.push_back(read_shape(cin));
        draw_all(v);                    // call draw() for each elemen
        rotate_all(v,45);               // call rotate(45) for each eleme
} // all Shapes implicitly destroyed
```

Now the object is owned by the unique_ptr which will delete the object when it is no longer needed, that is, when its unique_ptr goes out of scope.

For the unique_ptr version of user() to work, we need versions of draw_all() and rotate_all() that accept vector<unique_ptr<Shape>>s. Writing many such _all() functions could become tedious, so §5.5 shows an alternative.

4.6. COPY AND MOVE

By default, objects can be copied. This is true for objects of user-defined types as well as for builtin types. The default meaning of copy is member-wise copy: copy each member. For example, using complex from §4.2.1:

Click here to view code image

```
void test(complex z1)
{
        complex z2 {z1};       // copy initialization
        complex z3;
        z3 = z2;                 // copy assignment
        // ...
}
```

Now z1, z2, and z3 have the same value because both the assignment and the initialization copied both members.

When we design a class, we must always consider if and how an object might be copied. For simple concrete types, memberwise copy is often exactly the right semantics for copy. For some sophisticated concrete types, such as Vector, memberwise copy is not the right semantics for copy, and for abstract types it almost never is.

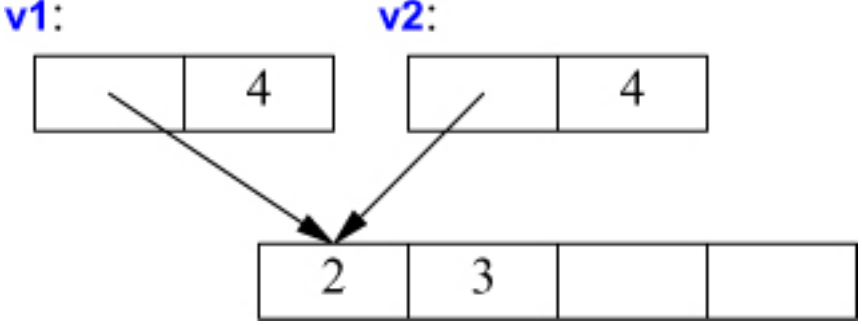
4.6.1. Copying Containers

When a class is a *resource handle* – that is, when the class is responsible for an object accessed through a pointer – the default memberwise copy is typically a disaster. Memberwise copy would violate the resource handle's inv ariant (§3.4.2). For example, the default copy would leave a copy of a Vector referring to the same elements as the original:

Click here to view code image

```
void bad_copy(Vector v1)
{
        Vector v2 = v1;       // copy v1's representation into v2
        v1[0] = 2;              // v2[0] is now also 2!
        v2[1] = 3;              // v1[1] is now also 3!
}
```

Assuming that v1 has four elements, the result can be represented graphically like this:



Fortunately, the fact that Vector has a destructor is a strong hint that the default (memberwise) copy semantics is wrong and the compiler should at least warn against this example. We need to define better copy semantics.

Copying of an object of a class is defined by two members: a *copy constructor* and a *copy assignment*:

Click here to view code image

```
class Vector {
private:
        double* elem; //  elem points to an array of sz doubles
        int sz;
public:
        Vector(int s);                                  //  construc
        ~Vector() { delete[] elem; }                    //  destructor: re

        Vector(const Vector& a);                        //  copy const
        Vector& operator=(const Vector& a);             //  copy assignr

        double& operator[](int i);
        const double& operator[](int i) const;

        int size() const;
};
```

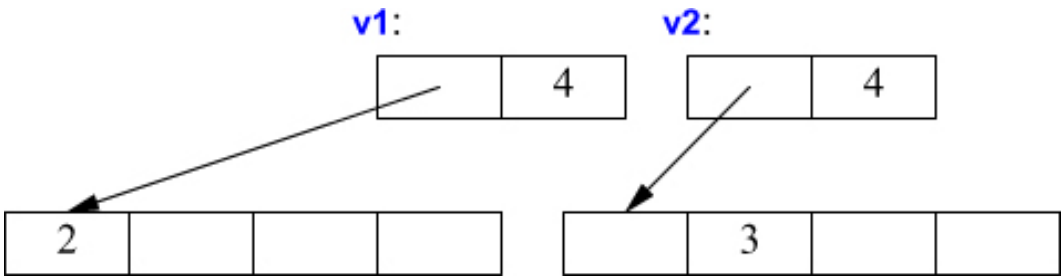
A suitable definition of a copy constructor for Vector allocates the space for the required number of elements and then copies the elements into it, so that after a copy each Vector has its own copy of the elements:

Click here to view code image

```
Vector::Vector(const Vector& a)         //  copy constructor
        :elem{new double[a.sz]},        //  allocate space for eleme
        sz{a.sz}
```

```
{
        for (int i=0; i!=sz; ++i)          // copy elements
                elem[i] = a.elem[i];
}
```

The result of the v2=v1 example can now be presented as:



Of course, we need a copy assignment in addition to the copy constructor:

Click here to view code image

```
Vector& Vector::operator=(const Vector& a)          // copy assig
{
        double* p = new double[a.sz];
        for (int i=0; i!=a.sz; ++i)
                p[i] = a.elem[i];
        delete[] elem;              // delete old elements
        elem = p;
        sz = a.sz;
        return *this;
}
```

The name this is predefined in a member function and points to the object for which the member function is called.

4.6.2. Moving Containers

We can control copying by defining a copy constructor and a copy assignment, but copying can be costly for large containers. We avoid the cost of copying when we pass objects to a function by using references, but we can't return a reference to a local object as the result (the local object would be destroyed by the time the caller got a chance to look at it). Consider:

Click here to view code image

```
Vector operator+(const Vector& a, const Vector& b)
{
        if (a.size()!=b.size())
                throw Vector_size_mismatch{};

        Vector res(a.size());
        for (int i=0; i!=a.size(); ++i)
                res[i]=a[i]+b[i];
        return res;
}
```

Returning from a + involves copying the result out of the local variable res and into some place where the caller can access it. We might use this + like this:

Click here to view code image

```
void f(const Vector& x, const Vector& y, const Vector& z)
{
        Vector r;
        // ...
        r = x+y+z;
        // ...
}
```

That would be copying a Vector at least twice (one for each use of the + operator). If a Vector is large, say, 10,000 doubles, that could be embarrassing. The most embarrassing part is that res in operator+() is never used again after the copy. We didn't really want a copy; we just wanted to get the result out of a function: we wanted to *move* a Vector rather than to *copy* it. Fortunately, we can state that intent:

Click here to view code image

```
class Vector {
        // ...

        Vector(const Vector& a);                    // copy constr
        Vector& operator=(const Vector& a);         // copy assignm
```

```
        Vector(Vector&& a);                 // move cons
        Vector& operator=(Vector&& a);      // move assign
};
```

Given that definition, the compiler will choose the *move constructor* to implement the transfer of the return value out of the function. This means that `r = x+y+z` will involve no copying of `Vector`s. Instead, `Vector`s are just moved.

As is typical, `Vector`'s move constructor is trivial to define:

Click here to view code image

```
Vector::Vector(Vector&& a)
        :elem{a.elem},              // "grab the elements" from a
        sz{a.sz}
{
        a.elem = nullptr;           // now a has no elements
        a.sz = 0;
}
```

The `&&` means "rvalue reference" and is a reference to which we can bind an rvalue. The word "rvalue" is intended to complement "lvalue," which roughly means "something that can appear on the left-hand side of an assignment." So an rvalue is – to a first approximation – a value that you can't assign to, such as an integer returned by a function call. Thus, an rvalue reference is a reference to something that *nobody else* can assign to, so that we can safely "steal" its value. The `res` local variable in `operator+()` for `Vector`s is an example.

A move constructor does *not* take a `const` argument: after all, a move constructor is supposed to remove the value from its argument. A *move assignment* is defined similarly.

A move operation is applied when an rvalue reference is used as an initializer or as the right-hand side of an assignment.

After a move, a moved-from object should be in a state that allows a destructor to be run. Typically, we should also allow assignment to a moved-from object.

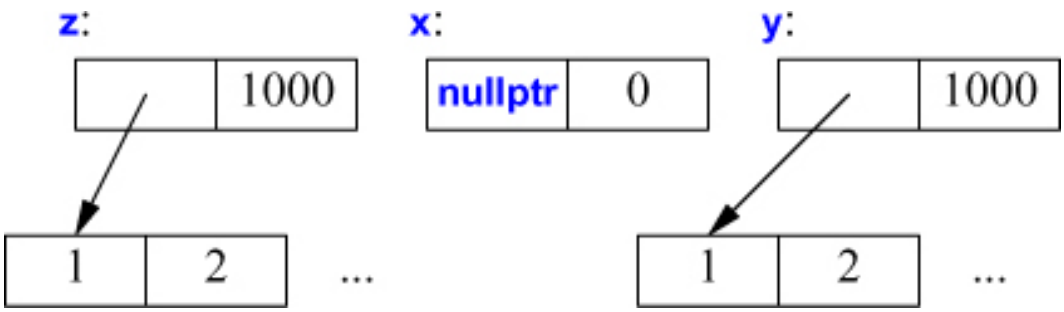
Where the programmer knows that a value will not be used again, but the compiler can't be expected to be smart enough to figure that out, the programmer can be specific:

Click here to view code image

```
Vector f()
{
        Vector x(1000);
        Vector y(1000);
        Vector z(1000);
        z = x;                          // we get a copy
        y = std::move(x);               // we get a move
        return z;                       // we get a move
};
```

The standard-library function `move()` returns doesn't actually move anything. Instead, it returns a reference to its argument from which we may move – an *rvalue reference*.

Just before the `return` we have:



When `z` is destroyed, it too has been moved from (by the `return`) so that, like `x`, it is empty (it holds no elements).

4.6.3. Essential Operations

Construction of objects plays a key role in many designs. This wide variety of uses is reflected in the range and flexibility of the language features supporting initialization.

Constructors, destructors, and copy and move operations for a type are not logically separate. We must define them as a matched set or suffer logical or performance problems. If a class `X` has a destructor that performs a nontrivial task, such as free-store deallocation or lock release, the class is likely to need the full complement of functions:

Click here to view code image

```
class X {
public:
        X(Sometype);                  // "ordinary constructor": create
        X();                          // default constructor
        X(const X&);                  // copy constructor
        X(X&&);                       // move constructor
        X& operator=(const X&); // copy assignment: clean up targe
        X& operator=(X&&);            // move assignment: clean up ta
        ~X();                         // destructor: clean up
        // ...
};
```

There are five situations in which an object is copied or moved:

• As the source of an assignment

• As an object initializer

• As a function argument

• As a function return value

• As an exception

In all cases, the copy or move constructor will be applied (unless it can be optimized away).

In addition to the initialization of named objects and objects on the free store, constructors are used to initialize temporary objects and to implement explicit type conversion.

Except for the "ordinary constructor," these special member functions will be generated by the compiler as needed. If you want to be explicit about generating default implementations, you can:

Click here to view code image

```
class Y {
Public:
        Y(Sometype);
        Y(const Y&) = default;    // I really do want the default copy
```

```
        Y(Y&&) = default;          // and the default copy construct
        // ...
};
```

If you are explicit about some defaults, other default definitions will not be generated.

When a class has a pointer or a reference member, it is usually a good idea to be explicit about copy of move operations. The reason is that a pointer or reference will point to something that the class needs to delete, in which case the default copy would be wrong, or it points to something that the class must not delete, in which case a reader of the code would like to know that.

A constructor taking a single argument defines a conversion from its argument type. For example, complex (§4.2.1) provides a constructor from a double:

Click here to view code image

```
complex z1 = 3.14; // z1 becomes {3.14,0.0}
complex z2 = z1*2; // z2 becomes {6.28,0.0}
```

Obviously, this is sometimes ideal, but not always. For example, Vector (§4.2.2) provides a constructor from an int:

Click here to view code image

```
Vector v1 = 7; // OK: v1 has 7 elements
```

This is typically considered unfortunate, and the standard-library vector does not allow this int-to-vector "conversion."

The way to avoid this problem is to say that only explicit "conversion" is allowed; that is, we can define the constructor like this:

Click here to view code image

```
class Vector {
public:
        explicit Vector(int s);          // no implicit conversion from ir
```

```
        // ...
};
```

That gives us:

Click here to view code image

```
Vector v1(7);    // OK: v1 has 7 elements
Vector v2 = 7;   // error: no implicit conversion from int to Vector
```

When it comes to conversions, more types are like Vector than are like complex, so use explicit for constructors that take a single argument unless there is a good reason not to.

4.6.4. Resource Management

By defining constructors, copy operations, move operations, and a destructor, a programmer can provide complete control of the lifetime of a contained resource (such as the elements of a container). Furthermore, a move constructor allows an object to move simply and cheaply from one scope to another. That way, objects that we cannot or would not want to copy out of a scope can be simply and cheaply moved out instead. Consider a standard-library thread representing a concurrent activity (§13.2) and a Vector of a million doubles. We can't copy the former and don't want to copy the latter.

Click here to view code image

```
std::vector<thread> my_threads;

Vector init(int n)
{
        thread t {heartbeat};                   // run heartbeat concurr
        my_threads.push_back(move(t));          // move t into my_threa
        // ... more initialization ...

        Vector vec(n);
        for (int i=0; i<vec.size(); ++i)
                vec[i] = 777;
        return vec;                                     // move res out of ini
}

auto v = init(10000);           // start heartbeat and initialize v
```

This makes resource handles, such as Vector and thread, an alternative to using pointers in many cases. In fact, the standard-library "smart pointers," such as unique_ptr, are themselves resource handles (§11.2.1).

I used the standard-library vector to hold the threads because we don't get to parameterize Vector with an element type until §5.2.

In very much the same way as new and delete disappear from application code, we can make pointers disappear into resource handles. In both cases, the result is simpler and more maintainable code, without added overhead. In particular, we can achieve *strong resource safety*; that is, we can eliminate resource leaks for a general notion of a resource. Examples are vectors holding memory, threads holding system threads, and fstreams holding file handles.

In many languages, resource management is primarily delegated to a garbage collector. C++ also offers a garbage collection interface so that you can plug in a garbage collector. However, I consider garbage collection the last alternative after cleaner, more general, and better localized alternatives to resource management have been exhausted.

Garbage collection is fundamentally a global memory management scheme. Clever implementations can compensate, but as systems are getting more distributed (think multicores, caches, and clusters), locality is more important than ever.

Also, memory is not the only resource. A resource is anything that has to be acquired and (explicitly or implicitly) released after use. Examples are memory, locks, sockets, file handles, and thread handles. A good resource management system handles all kinds of resources. Leaks must be avoided in any long-running systems, but excessive resource retention can be almost as bad as a leak. For example, if a system holds on to memory, locks, files, etc., for twice as long, the system needs to be provisioned with potentially twice as many resources.

Before resorting to garbage collection, systematically use resource handles: Let each resource have an owner in some scope and by default be released at the end of its owners scope. In C++, this is known as RAII (*Resource Acquisition Is Initialization*) and is integrated with error handling in the form of exceptions. Resources can be moved from scope to scope using move semantics or "smart pointers," and shared ownership can be represented by "shared pointers" (§11.2.1).

In the C++ standard library, RAII is pervasive: for example, memory (string, vector, map, unordered_map, etc.), files (ifstream, ofstream, etc.), threads (thread), locks (lock_guard, unique_lock, etc.), and general objects (through unique_ptr and shared_ptr). The result is implicit resource management that is invisible in common use and leads to low resource retention durations.

4.6.5. Suppressing Operations

Using the default copy or move for a class in a hierarchy is typically a disaster: given only a pointer to a base, we simply don't know what members the derived class has (§4.3), so we can't know how to copy them. So, the best thing to do is usually to *delete* the default copy and move operations, that is, to eliminate the default definitions of those two operations:

Click here to view code image

```
class Shape {
public:
        Shape(const Shape&) =delete;                    // no copy op
        Shape& operator=(const Shape&) =delete;

        Shape(Shape&&) =delete;                         // no move
        Shape& operator=(Shape&&) =delete;

        ~Shape();
        // ...
};
```

Now an attempt to copy a Shape will be caught by the compiler. If you need to copy an object in a class hierarchy, write a virtual clone function.

In this particular case, if you forgot to delete a copy or move operation, no harm is done. A move operation is *not* implicitly generated for a class where the user has explicitly declared a destructor, so you get a compiler error if you try to move a Shape. Furthermore, the generation of copy operations is deprecated in this case (§14.2.3), so you should expect the compiler to issue a warning if you try to copy a Shape.

A base class in a class hierarchy is just one example of an object we wouldn't want to copy. A resource handle generally cannot be copied just by copying its members (§4.6.1).

The =delete mechanism is general, that is, it can be used to suppress any operation.

4.7. ADVICE

[1] The material in this chapter roughly corresponds to what is described in much greater detail in Chapters 16-22 of [Stroustrup,2013].

[2] Express ideas directly in code; §4.1.

[3] A concrete type is the simplest kind of class. Where applicable, prefer a concrete type over more complicated classes and over plain data structures; §4.2.

[4] Use concrete classes to represent simple concepts and performance-critical components; §4.2.

[5] Define a constructor to handle initialization of objects; §4.2.1, §4.6.3.

[6] Make a function a member only if it needs direct access to the representation of a class; §4.2.1.

[7] Define operators primarily to mimic conventional usage; §4.2.1.

[8] Use nonmember functions for symmetric operators; §4.2.1.

[9] Declare a member function that does not modify the state of its object const; §4.2.1.

[10] If a constructor acquires a resource, its class needs a destructor to release the resource; §4.2.2.

[11] Avoid “naked” new and delete operations; §4.2.2.

[12] Use resource handles and RAII to manage resources; §4.2.2.

[13] If a class is a container, give it an initializer-list constructor; §4.2.3.

[14] Use abstract classes as interfaces when complete separation of interface and implementation is needed; §4.3.

[15] Access polymorphic objects through pointers and references; §4.3.

[16] An abstract class typically doesn’t need a constructor; §4.3.

[17] Use class hierarchies to represent concepts with inherent hierarchical structure; §4.5.

[18] A class with a virtual function should have a virtual destructor; §4.5.

[19] Use override to make overriding explicit in large class hierarchies; §4.5.1.

[20] When designing a class hierarchy, distinguish between implementation inheritance and interface inheritance; §4.5.2.

[21] Use dynamic_cast where class hierarchy navigation is unavoidable; §4.5.3.

[22] Use dynamic_cast to a reference type when failure to find the required class is considered a failure; §4.5.3.

[23] Use dynamic_cast to a pointer type when failure to find the required class is considered a valid alternative; §4.5.3.

[24] Use unique_ptr or shared_ptr to avoid forgetting to delete objects created using new; §4.5.4.

[25] Redefine or prohibit copying if the default is not appropriate for a type; §4.6.1, §4.6.5.

[26] Return containers by value (relying on move for efficiency); §4.6.2.

[27] For large operands, use const reference argument types; §4.6.2.

[28] If a class has a destructor, it probably needs user-defined or deleted copy and move operations; §4.6.5.

[29] Control construction, copy, move, and destruction of objects; §4.6.3.

[30] Design constructors, assignments, and the destructor as a matched set of operations; §4.6.3.

[31] If a default constructor, assignment, or destructor is appropriate, let the compiler generate it (don't rewrite it yourself); §4.6.3.

[32] By default, declare single-argument constructors explicit; §4.6.3.

[33] If a class has a pointer or reference member, it probably needs a destructor and non-default copy operations; §4.6.3.

[34] Provide strong resource safety; that is, never leak anything that you think of as a resource; §4.6.4.

[35] If a class is a resource handle, it needs a constructor, a destructor, and non-default copy operations; §4.6.4.

PREV
3. Modularity

NEXT
5. Templates