3

A Tour of C++: Abstraction Mechanisms

Don´t Panic!
– Douglas Adams

3.1 Introduction [tour2.intro]

This chapter aims 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*. Templates are introduced 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*. The next two chapters follow up by presenting examples of standard-library facilities and their use.

The assumption is that you have programmed before. If not, please consider reading a textbook, such as *Programming: Principles and Practice using C++* [Stroustrup, 2009], before continuing here. Even if you have programmed before, the language you used or the applications you wrote may be very different from the style of C++ presented here. If you find this ‘‘lightning tour’’ confusing, skip to the more systematic presentation starting in Chapter 6.

3.2 Classes [tour2.class]

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 (§3.2.1)
- abstract classes (§3.2.3)
- classes in class hierarchies (§3.2.5)

An astounding number of useful classes turn out to be of these three kinds. Even more can be seen as simple variants of these are implemented using combinations of the techniques used for these.

3.2.1 Concrete Types [tour2.concrete]

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 (§4.2, §4.3.3, §4.4.1).

The defining characteristic of a concrete type is that its representation is part of its definition. 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.
- refer to objects directly (and not just through pointers or references).

- initialize objects immediately and completely (e.g., by using constructors; §2.3.2).
- copy objects (§3.3).

The representation can be private (as it is for **Vector**; §2.3.2) 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 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.

3.2.1.1 An Arithmetic Type [tour2.complex]

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

```
class complex {
        double re, im;   // representation: two doubles
public:
        complex(double r, double i) :re{r}, im{i} {}   // construct complex from two scalars
        complex(double r) :re{r}, im{0} {}             // construct complex from one scalar
        complex() :re{0}, im{0} {}                     // default complex: {0,0}

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

        complex operator+=(complex z) { return {re+=z.re, im+=z.im}; }   // add to re and im
                                                                         // and return the result
        complex operator−=(complex z) { return {re−=z.re, im−=z.im}; }
        complex operator*=(complex); // defined out-of-class somewhere
        complex operator/=(complex); // defined out-of-class somewhere
};
```

This is a slightly simplified version of the standard library **complex** (§3.2.1.1, §39.4). The class definition itself contains only the operations requiring access to the representation. Complex has a simple and conventional representation (which for practical reasons has to be compatible with what Fortran provided 50 years ago) and a lot of conventional operators. In addition to the logical demands, **complex** must also 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. An industrial strength **complex** (like the standard library one) would be 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 unintialized variables of that type.

Note the const specifier on the functions returning the real and imaginary parts. There, const is used to indicate that a function may not modify the object for which it was invoked.

Other useful operations can be defined separately from the class definition:

```
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()}; }  // unary minus
complex operator∗(complex a, complex b) { return a∗=b; }
complex operator/(complex a, complex b) { return a/=b; }

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);

// ...
```

Class complex can be used like this:

```
void f(complex z)
{
        complex a {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 re-define + to subtract ints.

3.2.1.2 A Container [tour2.container]

A *container* is an object holding a collection of elements, so we call a type like Vector a container because it is the type of container objects. As defined in §2.3.2, Vector isn’t an unreasonable container of doubles: it is simple to understand, establishes a useful invariant (§2.4.3.2), provides range-checked access (§2.4.3.1), 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 (§34.8), it is not guaranteed that one is available or will run to make unused memory available for new objects. In some environments you can't use a collector and sometimes you prefer more detailed control of construction and destruction (§13.6.4) 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*:

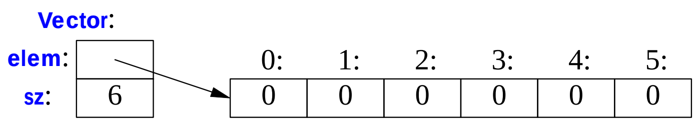
```
class Vector {
private:
        double* elem;  // elem points to an array of sz doubles
        int sz;
public:
        Vector(int s) :elem{new double[s]}, sz{s}      // constructor: acquire resources
        {
                for (int i=0;i<s; ++i) elem[i]=0;      // initialize elements
        }
        ~Vector() { delete[] elem; }                   // destructor: release resources

        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. The 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:

```
void fct(int x)
{
        Vector v(x);
        // use v
        {
                Vector v2(2*x);
                // use v and v2
        } // v2 is destroyed here
        // use v
} // v is destroyed here
```

The Vector obeys the same rules for naming, scope, allocation, lifetime, etc., as does a built-in type, such as int and char. For details on how to control the lifetime of an object, see §6.4.

The constructor/destructor combination is the basis of many elegant techniques and is in particular the basis for most C++ general resource management techniques (§5.2, §13.3). 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 the destructor technique, 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 (§5.2).

3.2.2 Initializing Containers [tour2.initializer_list]

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 assign to them, but typically other ways are more elegant. Here, I 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:

```
class Vector {
        // ...
        Vector(std::initializer_list<double>);     // initialize with a list
        // ...
        void push_back(double);                    // add element at end increasing the size by one
        // ...
};
```

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

```
Vector read(istream& is)
{
        Vector v;
        for (double d; is>>d;) v.push_back(d);
        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 implementation of **push_back()** is discussed in §13.6.4.3. 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 §3.3.2.

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:

```
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:

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

3.2.3 Abstract Types [tour2.abstract]

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 must 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 (§3.2.1.2, §11.2) and access them through references or pointers (§2.2.5, §7.2, §7.7).

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

```
class Container {
public:
     virtual double& operator[](int) = 0;    // pure virtual function
     virtual int size() const = 0;           // const member function (§3.2.1.1)
     virtual ˜Container() {}                 // destructor (§3.2.1.2)
};
```

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." 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* (§20.3.2).

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 §3.2.5.

Not surprisingly, the implementation could consist of everything from the concrete class `Vector`:

```
class Vector_container : public Container { // Vector_container implements 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` (§20.3.2). The destructor (`~Vector_container()`) overrides the base class destructor (`~Container()`). Note that the member destructor (`~Vector()`) is implicitly invoked by its class' 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:

```
void g()
{
        Vector_container vc(200);
        // fill vc
        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:

```
class List_container : public Container {      // List_container implements Container
        std::list<double> ld;          // (standard library) list of doubles (§4.4.2)
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:

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

The key 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.

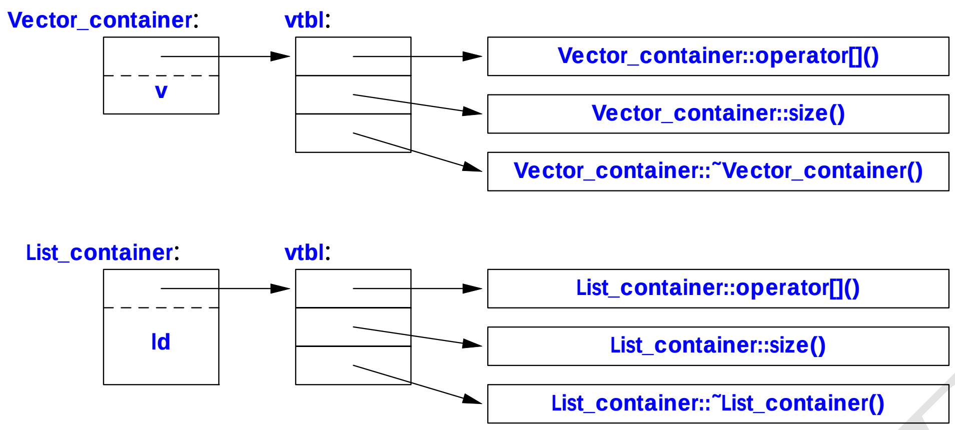
The flip side of this flexibility is that objects must be manipulated through pointers or references (§3.3).

3.2.4 Virtual Functions [tour2.virtual]

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';
}
```

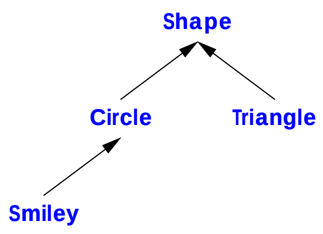
How is the call c[i] in use() resolved to the right operator[]()? When use() is called from h(), List_container::operator[]() must be called. When use() is called from g(), Vector_container::operator[]() must be called. To achieve this resolution, a Container object must contain information to allow 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:


Vector_container:
vtbl:
v
Vector_container::operator[]()
Vector_container::size()
Vector_container::~Vector_container()
List_container:
vtbl:
ld
List_container::operator[]()
List_container::size()
List_container::~List_container()

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.

3.2.5 Class Hierarchies [tour2.hier]

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:

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

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

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

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:

```
void rotate_all(vector<Shape*>& v, int angle) // rotate v's elements by angle degrees
{
        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):

```
class Circle : public Shape {
private:
        Point x;      // center
        int r;        // radius
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
};
```

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

```
class Smiley : public Circle {        // use the circle as base for a face
private:
        vector<Shape*> eyes;              // usually two eyes
        Shape* mouth;
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
};
```

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 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. See Chapter 21. 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 use 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:

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

Shape∗ read_shape(istream& is)        // read shape descriptions from input stream is
{
        // 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 p3
                return new Triangle{p1,p2,p3};
        case Kind::smiley:
                // read smiley data {Point,int,Shape,Shape,Shape} into p, r, e1 ,e2, and m
                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:

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

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.

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

- A user might fail to place the pointer returned by `read_shape` into a container and also forget to `delete` it.
- The owner of the container of `Shape` pointers might forget to `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` (§5.2.1) rather than a ‘‘naked pointer’’ and store `unique_ptr`s in the container:

```
unique_ptr<Shape> read_shape(istream& is)      // read shape descriptions from input stream is
{
        // 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}}; // §5.2.1
        // ...
}

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

Now the object is owned by the **unique_ptr** which will **delete** it when needed.

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 §3.4.3 shows an alternative.

3.3 Copy and Move [tour2.copy]

By default, objects can be copied. This is true for objects of user-defined types as well as for built-in types. The default meaning of copy is memberwise copy: copy each member. For example, using **complex** from (§3.2.1.1):

```
complex z1 {1,2};
complex z2 {z1};        // copy initialization
complex z3;
z3 = z2;                // copy assignment
```

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

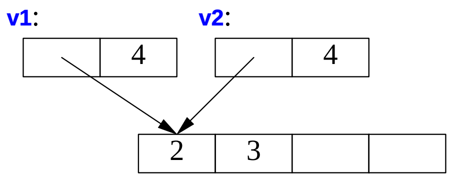
When we design a class, we must always consider if and how an object might be copied. For simple concrete types, memberwise copy is often exactly the right semantics for copy. For some sophisticated concrete types, such as **Vector**, memberwise copy is not the right semantics for copy and for abstract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

3.3.1 Copying Containers [tour2.copy.container]

When a class is a *resource handle*; that is, it 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 invariant (§2.4.3.2). For example, the default copy would leave a copy of a Vector referring to the same elements as the original:

```
Vector v1(4);
Vector v2 = v1;
v1[0] = 2;          // v2[0] is now also 2!
v2[1] = 3;          // v1[1] is now also 3!
```

Graphically:



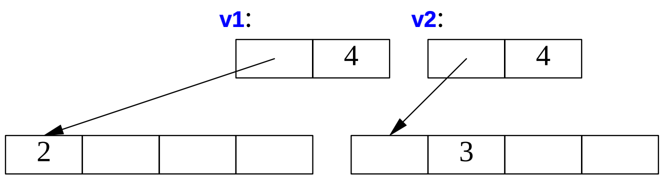
Fortunately, the fact that Vector has a destructor is a strong hint that the default (memberwise) copy semantics is wrong and the compiler should at least warn against this example (§17.6). We need to define a better copy semantics. Copying is defined by two functions: a *copy constructor* and a *copy assignment*:

```
class Vector {
private:
        // elem points to an array of sz doubles
        double* elem;
        int sz;
public:
        Vector(int s);                    // constructor: establish invariant, acquire resources
        ˜Vector() { delete[] elem; }      // destructor: release resources

        Vector(const Vector& a);              // copy constructor
        Vector& operator=(const Vector& a);  // copy assignment

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

        int size() const;
};
```

A suitable definition of Vector copy for a container simply copies the elements:

```
Vector::Vector(const Vector& a) // copy constructor
        :sz(a.sz)
{
        elem = new double[sz];
        for (int i=0; i<sz; ++i)
                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:

```
Vector& Vector::operator=(const Vector& a)          // copy assignment
{
        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 every member function and points to the object for which the member function is called.

A copy constructor and a copy assignment for a class **X** are typically declared to take an argument of type **const X&**.

3.3.2 Moving Containers [tour2.copy.move]

We can control copying by defining a copy constructor and a copy assignment, but copying can be costly for large containers. Consider:

```
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:

```
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 10000 `double`s, that could be embarrassing. The most embarrassing part is that `res` 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:

```
class Vector {
private:
        // elem points to an array of sz doubles
        double* elem;
        int sz;
public:
        // ...

        Vector(const Vector& a);              // copy constructor
        Vector& operator=(const Vector& a);   // copy assignment

        Vector(Vector&& a);                   // move constructor
        Vector& operator=(Vector&& a);        // move assignment

        // ...
};
```

Given that, the compiler will choose the *move constructor* to implement the transfer of the return value out of the function. This means that the `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:

```
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 (§6.4.1). The word "rvalue" is intended to complement "lvalue," which roughly means "something that can appear on the left hand 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, and an rvalue reference is a reference to something that nobody else can assign to. Note that 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 rght-hand side of an assignment.

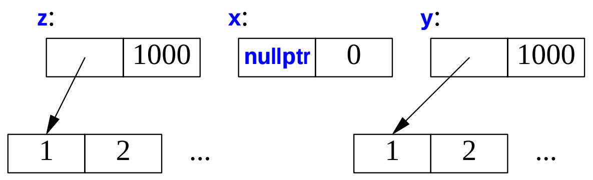
After a move, an object should be in a state that allows a destructor to be run. Typically, we should also allow assignment to a moved-from object (§17.5, §17.6.2).

In cases 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:

```
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 an rvalue reference to its argument.

Just before the return we have:



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

3.3.3 Resource Management [tour2.copy.resource]

By defining constructors, copy operations, move operations, and a destructors, a programmer can provide complete control of the lifetime of a contained resource (such as the elements of a container). In particular, a move constructor allows an object to move simply and cheaply from one scope to another. That way, we can move objects that we cannot or would not want to copy out of a scope. Consider a standard-library thread representing a concurrent activity (§5.3.1) and a Vector of a million doubles. We can't copy the former and don't want to copy the latter.

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

Vector init()
{
        thread t {heartbeat};                       // run heartbeat concurrently (on its own thread)
        my_threads.push_back(move(t));              // move t into my_threads

        Vector<double> vec;
        // ... fill vec ...
        return vec;                                 // move res out of run()
}

auto v = init();    // 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 such resource handles (§5.2.1).

I used the standard-library `vector` because we don’t get to parameterize `Vector` with an element type until §3.4.1.

3.3.4 Preventing Copy and Move [tour2.copy.hier]

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 (§3.3.3), 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 to default definitions of those two operations:

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

        Shape(Shape&&) =delete;                     // no move operations
        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 some kind of clone function (§22.2.4).

In case 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. Furthermore, the generation of copy operations are deprecated in this case (§42.2.3). This can be a good reason to explicitly define a destructor even where the compiler would have implicitly provided one (§17.2.3).

A base class in a class hierarchy is just one example of an object we wouldn't want to copy. A resource handle generally can't be copied just by copying its members (§5.2, §17.2.2).

3.4 Templates [tour2.generic]

Someone who wants a vector is unlikely always to want a vector of `double`s. A vector is a general concept, independent of the notion of a floating-point number. Consequently, the element type of a vector ought to be represented independently. A *template* is a class or a function that we parameterize with a set of types or values. We use templates to represent concepts that are best understood as something very general from which we can generate specific types and functions by specifying arguments, such as the element type `double`.

3.4.1 Parameterized Types [tour2.containers]

We can generalize our vector-of-doubles type to a vector-of-anything type by making it a `template` and replacing the specific type `double` with a parameter. For example:

```
template<typename T>
class Vector {
private:
        T* elem;    // elem points to an array of sz elements of type T
        int sz;
public:
        Vector(int s);                   // constructor: establish invariant, acquire resources
        ~Vector() { delete[] elem; }     // destructor: release resources

        // copy and move operations

        T& operator[](int i);
        const T& operator[](int i) const;
        int size() const { return sz; }
};
```

The `template<typename T>` prefix makes `T` a parameter of the declaration it prefixes. It is C++'s version of the mathematical ''for all T'' or more precisely ''for all types T.''

The member functions might be defined similarly:

```
template<typename T>
Vector<T>::Vector(int s)
{
        if (s<0) throw Negative_size{};
        elem = new T[s];
        sz = s;
}
```

```
template<typename T>
const T& Vector<T>::operator[](int i) const
{
        if (i<0 || size()<=i) throw out_of_range{"Vector::operator[]"};
        return elem[i];
}
```

Given these definitions, we can define Vectors like this:

```
Vector<char> vc(200);           // vector of 200 characters
Vector<string> vs(17);          // vector of 17 integers
Vector<list<int>> vli(45);      // vector of 45 lists of integers
```

The >> in Vector<list>> terminates the nested template arguments; it is not a misplaced input operator. It is not (as in C++98) necessary to place a space between the two >s.

We can use Vectors like this:

```
void f(const Vector<string>& vs)        // Vector of some strings
{
        for (int i = 0; i<vs.size(); ++i))
                cout << vs[i] << '\n';
}
```

If we also want to use the range-for loop for our Vector, we must define suitable begin() and end():

```
template<typename T>
T* begin(Vector<T>& x)
{
        return &x[0];           // pointer to first element
}

template<typename T>
T* end(Vector<T>& x)
{
        return x.begin()+x.size();    // pointer to one-past-last element
}
```

Given those, we can write:

```
void f2(const Vector<string>& vs)       // Vector of some strings
{
        for (auto s : vs)
                cout << s << '\n';
}
```

Similarly, we can define lists, vectors, maps (that is, associative arrays), etc., as templates (§4.4, §23.2, Chapter 31).

Templates are a compile-time mechanism, so their use incurs no run-time overhead compared to ‘‘hand-written code’’ (§23.2.2).

3.4.2 Function Templates [tour2.algorithms]

Templates have many more uses than simply parameterizing a container with an element type. In particular, they are extensively used for parameterization of both types and algorithms in the standard library (§3.4.1, §3.4.2). For example, we can write a function that calculates the sum of the element values of any container like this:

```
template<typename Container, typename Value>
Value sum(const Container& c, Value v)
{
        for (auto x : c) v+=x;
        return v;
}
```

The **Value** template argument and the function argument **v** are there to allow the caller to specify the type and initial value of the accumulator (the variable in which to accumulate the sum):

```
void user(Vector<int>& vi, std::list<double>& ld, std::vector<complex<double>>& vc)
{
        int x = sum(vi,0);                          // the sum of a vector (add ints)
        double d = sum(vi,0.0);                     // the sum of a vector (add doubles)
        double dd = sum(ld,0.0);                    // the sum of a list of doubles
        auto z = sum(vc,complex<double>{});         // the sum of a vector of complex<double>
}
```

The point of adding **int**s in a **double** would be to gracefully handle a number larger than the largest **int**. Note how the types of the template arguments for **sum<T,V>** are deduced from the function arguments.

This **sum()** is a simplified version of the standard-library **accumulate()** (§39.6).

3.4.3 Function Objects [tour2.functionobjects]

One particularly useful kind of template is the *function object* (sometimes called a *functor*), which is used to define objects that can be called like functions. For example:

```
template<typename T>
class Less_than {
        const T& val;      // value to compare against
public:
        Less_than(const T& v) :val(v) { }
        bool operator()(const T& x) const { return x<val; }        // call operator
};
```

The function called **operator()** implements the ''function call,'' ''call,'' or ''application'' operator **()**.

We can define named variables of type **Less_than** for some argument type:

```
Less_than<int> lti {42};                 // will compare to 42 (using <)
Less_than<string> lts {"Backus"};        // will compare to "Backus" (using <)
```

We can call such an object, just as we call a function:

```
void fct(int n, const string & s)
{
        bool b1 = lti(n);          // true if n<42
        bool b2 = lts(s);          // true if s<"Backus"
        // ...
}
```

Such function objects are widely used as arguments to algorithms. For example, we can count the occurrences of values for which a predicate returns **true**:

```
template<typename C, typename P>
int count(const C& c, P pred)
{
        int cnt = 0;
        for (const auto& x : c)
                if (pred(x)) ++cnt;
        return cnt;
}
```

A *predicate* is something that we can invoke to return **true** or **false**. For example:

```
void f(const Vector<int>& vec, const list<string>& lst, int x, const string& s)
{
        cout << "number of values less than " << x
                << ": " << count(vec,Less_than<int>{x})
                << '\n';
        cout << "number of values less than " << s
                << ": " << count(lst,Less_than<string>{s})
                << '\n';
}
```

Here, **Less_than<int>{x}** constructs an object for which the call operator compares to the **int** called **x**; **Less_than<string>{s}** constructs an object that compares to the **string** called **s**. The beauty of these function objects is that they carry the value to be compared against with them. We don't have to write a separate function for each value (and each type) and we don't have to introduce nasty global variables to hold values. Also, for a simple function object like **Less_than** inlining is simple so that a call of **Less_than** is far more efficient than an indirect function call. The ability to carry data plus their efficiency makes function objects particularly useful as arguments to algorithms.

Function objects that is used to specify the meaning of key operations of a an general algorithm (such as **Less_than** for **count()**) are often referred to a *policy objects*.

We have to define **Less_than** separately from its use. That could be seen as inconvenient. Consequently, there is a notation for implicitly generating function objects:

```
void f(const Vector<int>& vec, const list<string>& lst, int x, const string& s)
{
        cout << "number of values less than " << x
                << ": " << count(vec,[&](int a){ return a<x; })
                << '\n';
        cout << "number of values less than " << s
                << ": " << count(lst,[&](const string& a){ return a<s; })
                << '\n';
}
```

The notation `[&](int a){ return a<x; }` is called a *lambda expression* (§11.4). It generates a function object exactly like `Less_than<int>`. The `[&]` is a *capture list* specifying that local names used (such as `x`) will be passed by reference. Had we wanted to ‘‘capture’’ only `x`, we could have said so: `[&x]`. Had we wanted to give the generated object a copy of `x`, we could have said so: `[=x]`. Capture nothing is `[]`, capture all local names used by references is `[&]`, and capture all local names used by value is `[=]`.

Using lambdas can be convenient and terse, but also obscure. For non-trivial actions (say, more than a simple expression), I prefer to name the operation so as to more clearly state its purpose and to make it available for use in several places in a program.

In §3.2.5, we noticed the annoyance of having to write many functions to perform operations on elements of `vector`s of pointers and `unique_ptr`s, such as `draw_all()` and `rotate_all()`. Function objects (in particular, lambdas) can help by allowing us to separate the traversal of the container from the specification of what is to be done with each element.

First we need a function that applies an operation to each object pointed to by the elements of a container of pointers:

```
template<class C, class Oper>
void for_all(C& c, Oper op)              // assume that C is a container of pointers
{
        for (auto& x : c) op(*x);      // pass op() a reference to each element pointed to
}
```

Now we can write a version of `user()` from §3.2.5 without writing a set of *_all* functions:

```
void user()
{
        vector<unique_ptr<Shape>> v;
        while (cin)
                v.push_back(read_shape(cin));
        for_all(v,[](Shape& s){ s.draw(); });          // draw_all()
        for_all(v,[](Shape& s){ s.rotate(45); });      // rotate_all(45)
}
```

I pass a reference to `Shape` to a lambda so that the lambdas don’t have to care exactly how the objects are stored in the container. In particular, those `for_all()` calls would still work if I changed `v` to a `vector<Shape*>`.

3.4.4 Variadic Templates [tour2.variadic]

A template can be defined to accept an arbitrary number of arguments of arbitrary types. Such a template is called a *variadic template*. For example:

```
template<typename T, typename... Tail>
void f(T head, Tail... tail)
{
        g(head);    // do someting to head
        f(tail...);    // try again with tail
}

void f() { } // do nothing
```

The key to implementing a variadic template is to note that when you pass a list of arguments to it, you can separate the first argument from the rest. Here, we do something to the first argument (the head) and then recursively call f() with the rest of the arguments (the tail). The ellipses, ..., is used to indicate ‘‘the rest’’ of a list. Eventually, of course, the tail will become empty and we need a separate function to deal with that.

We can call this f() like this:

```
int main()
{
        cout << "first: ";
        f(1,2.2,"hello");

        cout << "\nsecond: "
        f(0.2,'c',"yuck!",0,1,2);
        cout << "\n";
}
```

This would call f(1,2.2,"hello"), which will call f(2.2,"hello"), which will call f("hello"), which will call f(). What might g() do? Obviously, in a real program it will do whatever we wanted done to each argument. For example, we could make it write its argument to output:

```
template<typename T>
void g(T x)
{
        cout << x << " ";
}
```

Given that, the output will be:

```
first: 1 2.2 hello
second: 0.2 c yuck! 0 1 2
```

It seems that f() is a simple variant of printf() printing arbitrary lists or values – implemented in three lines of code plus their surrounding declarations.

The strength of variadic templates (sometimes just called variadics) is that they can accept any arguments you care to give them. The weakness is that the type checking of the interface is a possibly elaborate template program. For details, see §28.6. For examples, see §34.2.4.2 (N-tuples) and Chapter 29 (N-dimensional matrices).

3.4.5 Aliases [tour2.alias]

Surprisingly often, it is useful to introduce a synonym for a type or a template (§6.5). For example, the standard header <cstddef> contains a definition of the alias size_t, maybe:

```
using size_t = unsigned int;
```

The actual type named size_t is implementation dependent, so in another implementation size_t may be an unsigned long. Having the alias size_t allows the programmer to write portable code.

It is very common for a parameterized type to provide an alias for types related to their template arguments. For example:

```
template<typename T>
class Vector {
public:
        using value_type = T;
        // ...
};
```

In fact, every standard library container provides value_type as the name of their value type (§31.3.1). This allows us to write code that will work for every container that follows this convention. For example:

```
template<typename C>
using Element_type = typename C::value_type;

template<typename Container>
void algo(Container& c)
{
        Vector<Element_type<Container>> vec;     // keep results here
        // ...
}
```

The aliasing mechanism can be used to define a new template by binding some or all templates arguments. For example:

```
template<typename Key, typename Value>
class Map {
        // ...
};

template<typename Value>
using String_map = Map<string,Value>;

String_map<int> m;   // m is a Map<string,int>
```

See §23.6.

3.5 Advice [tour2.advice]

[1] Express ideas directly in code; §3.2.
[2] Define classes to represent application concepts directly in code; §3.2.
[3] Use concrete classes for simple concepts and performance critical components; §3.2.1.
[4] Avoid ‘‘naked’’ `new` and `delete` operations; §3.2.1.2.
[5] Use resource handles and RAII to manage resources; §3.2.1.2.
[6] Use abstract classes as interfaces when complete separation of interface and implementation is needed; §3.2.3.
[7] Use class hierarchies to represent concepts with an inherent hierarchical structure; §3.2.5.
[8] When designing a class hierarchy, distinguish between implementation inheritance and interface inheritance; §3.2.5.
[9] Control construction, copy, move, and destruction of objects; §3.3.
[10] Use containers, defined as resource handle templates, to hold collections of values of the same type; §3.4.1.
[11] Use function templates to represent general algorithms; §3.4.2.
[12] Use function objects, including lambdas, to represent policies and actions; §3.4.3.
[13] Use type and template aliases to provide a uniform notation for types that may vary among similar types or among implementations; §3.4.5.