PREV
1. The Basics

NEXT
3. Modularity

2. User-Defined Types

Don't Panic!

– Douglas Adams

- Introduction
- Structures
- Classes
- Unions
- Enumerations
- Advice

2.1. INTRODUCTION

We call the types that can be built from the fundamental types (§1.5), the const modifier (§1.7), and the declarator operators (§1.8) *built-in types*. C++'s set of built-in types and operations is rich, but deliberately low-level. They directly and efficiently reflect the capabilities of conventional computer hardware. However, they don't provide the programmer with high-level facilities to conveniently write advanced applications. Instead, C++ augments the built-in types and operations with a sophisticated set of *abstraction mechanisms* out of which programmers can build such high-level facilities. The C++ abstraction mechanisms are primarily designed to let programmers design and implement their own types, with suitable representations and operations, and for programmers to simply and elegantly use such types. Types built out of the built-in types using C++'s abstraction mechanisms are called *user-defined types*. They are referred to as classes and enumerations. Most of this book is devoted to the design, implementation, and use of user-defined types. The rest of this chapter presents the simplest and most fundamental facilities for that. Chapters 4- 5 are a more complete description of the abstraction mechanisms and the programming styles they support. Chapters 6-13 present an overview of the standard library, and since the standard library mainly consists of user-defined types, they provide examples of what can be built using the language facilities and programming techniques presented in Chapters 1- 5 .

2.2. STRUCTURES

The first step in building a new type is often to organize the elements it needs into a data structure, a struct:

Click here to view code image

```
struct Vector {
        int sz;           // number of elements
        double* elem; // pointer to elements
};
```

This first version of Vector consists of an int and a double*.

A variable of type Vector can be defined like this:

```
Vector v;
```

However, by itself that is not of much use because v's elem pointer doesn't point to anything. To be useful, we must give v some elements to point to. For example, we can construct a Vector like this:

Click here to view code image

```
void vector_init(Vector& v, int s)
{
        v.elem = new double[s];     // allocate an array of s doubles
        v.sz = s;
}
```

That is, v's elem member gets a pointer produced by the new operator and v's sz member gets the number of elements. The & in Vector& indicates that we pass v by non-const reference (§1.8); that way, vector_init() can modify the vector passed to it.

The new operator allocates memory from an area called *the free store* (also known as *dynamic memory* and *heap*). Objects allocated on the free store are independent of the scope from which they are created and "live" until they are destroyed using the delete operator (§4.2.2).

A simple use of Vector looks like this:

Click here to view code image

```
double read_and_sum(int s)
        // read s integers from cin and return their sum; s is assumed
{
        Vector v;
        vector_init(v,s);                       // allocate s elements fo
        for (int i=0; i!=s; ++i)
                cin>>v.elem[i];                 // read into elements

        double sum = 0;
        for (int i=0; i!=s; ++i)
                sum+=v.elem[i];                 // take the sum of the
        return sum;
}
```

There is a long way to go before our Vector is as elegant and flexible as the standard-library vector. In particular, a user of Vector has to know every detail of Vector's representation. The rest of this chapter and the next two gradually improve Vector as an example of language features and techniques. Chapter 9 presents the standard-library vector, which contains many nice improvements.

I use vector and other standard-library components as examples

• to illustrate language features and design techniques, and

• to help you learn and use the standard-library components.

Don't reinvent standard-library components, such as vector and string; use them.

We use . (dot) to access struct members through a name (and through a reference) and –> to access struct members through a pointer. For example:

Click here to view code image

```
void f(Vector v, Vector& rv, Vector* pv)
{
        int i1 = v.sz;          // access through name
        int i2 = rv.sz;         // access through reference
        int i4 = pv–>sz;        // access through pointer
}
```

2.3. CLASSES

Having the data specified separately from the operations on it has advantages, such as the ability to use the data in arbitrary ways. However, a tighter connection between the representation and the operations is needed for a user-defined type to have all the properties expected of a "real type." In particular, we often want to keep the representation inaccessible to users, so as to ease use, guarantee consistent use of the data, and allow us to later improve the representation. To do that we have to distinguish between the interface to a type (to be used by all) and its implementation (which has access to the otherwise inaccessible data). The language mechanism for that is called a *class*. A class is defined to have a set of *members*, which can be data, function, or type members. The interface is defined by the public members of a class, and private members are accessible only through that interface. For example:

Click here to view code image

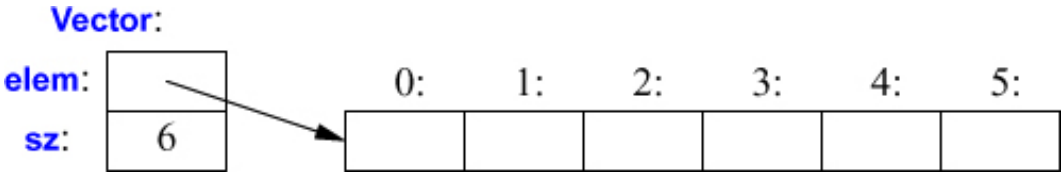
```
class Vector {
public:
        Vector(int s) :elem{new double[s]}, sz{s} { }     // construct a
        double& operator[](int i) { return elem[i]; }     // element acc
        int size() { return sz; }
private:
        double* elem;    // pointer to the elements
        int sz;          // the number of elements
};
```

Given that, we can define a variable of our new type **Vector**:

Click here to view code image

```
Vector v(6);   // a Vector with 6 elements
```

We can illustrate a **Vector** object graphically:



Basically, the **Vector** object is a "handle" containing a pointer to the elements (**elem**) plus the number of elements (**sz**). The number of elements (6 in the example) can vary from **Vector** object to **Vector** object, and a **Vector** object can have a different number of elements at different times (§4.2.3). However, the **Vector** object itself is always the same size. This is the basic technique for handling varying amounts of information in C++: a fixed-size handle referring to a variable amount of data "elsewhere" (e.g., on the free store allocated by **new**; §4.2.2). How to design and use such objects is the main topic of Chapter 4.

Here, the representation of a **Vector** (the members **elem** and **sz**) is accessible only through the interface provided by the **public** members: **Vector()**, **operator[]()**, and **size()**. The **read_and_sum()** example from §2.2 simplifies to:

Click here to view code image

```
double read_and_sum(int s)
{
        Vector v(s);                                    // make a vector of s
        for (int i=0; i!=v.size(); ++i)
                cin>>v[i];                              // read into element

        double sum = 0;
        for (int i=0; i!=v.size(); ++i)
                sum+=v[i];                               // take the sum of
        return sum;
}
```

A "function" with the same name as its class is called a *constructor*, that is, a function used to construct objects of a class. So, the constructor, `Vector()`, replaces `vector_init()` from §2.2. Unlike an ordinary function, a constructor is guaranteed to be used to initialize objects of its class. Thus, defining a constructor eliminates the problem of uninitialized variables for a class.

`Vector(int)` defines how objects of type `Vector` are constructed. In particular, it states that it needs an integer to do that. That integer is used as the number of elements. The constructor initializes the `Vector` members using a member initializer list:

```
:elem{new double[s]}, sz{s}
```

That is, we first initialize `elem` with a pointer to `s` elements of type `double` obtained from the free store. Then, we initialize `sz` to `s`.

Access to elements is provided by a subscript function, called `operator[]`. It returns a reference to the appropriate element (a `double&`).

The `size()` function is supplied to give users the number of elements.

Obviously, error handling is completely missing, but we'll return to that in §3.4. Similarly, we did not provide a mechanism to "give back" the array of `double`s acquired by `new`; §4.2.2 shows how to use a destructor to elegantly do that.

There is no fundamental difference between a `struct` and a `class`; a `struct` is simply a `class` with members `public` by default. For example, you can define constructors and other member functions for a `struct`.

2.4. UNIONS

A union is a struct in which all members are allocated at the same address so that the union occupies only as much space as its largest member. Naturally, a union can hold a value for only one member at a time. For example, consider a symbol table entry that holds a name and a value:

Click here to view code image

```
enum Type { str, num };

struct Entry {
        char* name;
        Type t;
        char* s; // use s if t==str
        int i;   // use i if t==num
};

void f(Entry* p)
{
        if (p->t == str)
                cout << p->s;
        // ...
}
```

The members s and i can never be used at the same time, so space is wasted. It can be easily recovered by specifying that both should be members of a union, like this:

```
union Value {
        char* s;
        int i;
};
```

The language doesn't keep track of which kind of value is held by a union, so the programmer must do that:

Click here to view code image

```
struct Entry {
        char* name;
```

```
        Type t;
        Value v; //  use v.s if t==str; use v.i if t==num
};

void f(Entry* p)
{
        if (p->t == str)
                cout << p->v.s;
        //  ...
}
```

Maintaining the correspondence between a *type field* (here, `t`) and the type held in a `union` is error-prone. To avoid errors, one can encapsulate a `union` so that the correspondence between a type field and access to the `union` members is guaranteed. At the application level, abstractions relying on such *tagged unions* are common and useful, but use of "naked" `union`s is best minimized.

2.5. ENUMERATIONS

In addition to classes, C++ supports a simple form of user-defined type for which we can enumerate the values:

Click here to view code image

```
enum class Color { red, blue, green };
enum class Traffic_light { green, yellow, red };

Color col = Color::red;
Traffic_light light = Traffic_light::red;
```

Note that enumerators (e.g., `red`) are in the scope of their `enum class`, so that they can be used repeatedly in different `enum class`es without confusion. For example, `Color::red` is `Color`'s `red` which is different from `Traffic_light::red`.

Enumerations are used to represent small sets of integer values. They are used to make code more readable and less error-prone than it would have been had the symbolic (and mnemonic) enumerator names not been used.

The `class` after the `enum` specifies that an enumeration is strongly typed and that its enumerators are scoped. Being separate types, `enum class`es

help prevent accidental misuses of constants. In particular, we cannot mix Traffic_light and Color values:

Click here to view code image

```
Color x = red;                          // error: which red?
Color y = Traffic_light::red;   // error: that red is not a Color
Color z = Color::red;           // OK
```

Similarly, we cannot implicitly mix Color and integer values:

Click here to view code image

```
int i = Color::red;         // error: Color::red is not an int
Color c = 2;                // error: 2 is not a Color
```

By default, an enum class has only assignment, initialization, and comparisons (e.g., == and <; §1.5) defined. However, an enumeration is a user-defined type so we can define operators for it:

Click here to view code image

```
Traffic_light& operator++(Traffic_light& t)
        // prefix increment: ++
{
        switch (t) {
        case Traffic_light::green:    return t=Traffic_light::yellow;
        case Traffic_light::yellow:   return t=Traffic_light::red;
        case Traffic_light::red:      return t=Traffic_light::green;
        }
}

Traffic_light next = ++light;           // next becomes Traffic_light::
```

If you don't want to explicitly qualify enumerator names and want enumerator values to be ints (without the need for an explicit conversion), you can remove the class from enum class to get a "plain" enum. The enumerators from a "plain" enum are entered into the same scope as the name of their enum and implicitly converts to their integer value. For example:

Click here to view code image

```
enum Color { red, green, blue };
int col = green;
```

Here col gets the value 1. By default, the integer values of enumerators starts with 0 and increases by one for each additional enumerator. The "plain" enums have been in C++ (and C) from the earliest days, so even though they are less well behaved, they are common in current code.

2.6. ADVICE

[1] The material in this chapter roughly corresponds to what is described in much greater detail in Chapter 8 of [Stroustrup,2013].

[2] Organize related data into structures (structs or classes); §2.2.

[3] Represent the distinction between an interface and an implementation using a class; §2.3.

[4] A struct is simply a class with its members public by default; §2.3.

[5] Define constructors to guarantee and simplify initialization of classes; §2.3.

[6] Avoid "naked" unions; wrap them in a class together with a type field; §2.4.

[7] Use enumerations to represent sets of named constants; §2.5.

[8] Prefer enum classes over "plain" enums to minimize surprises; §2.5.

[9] Define operations on enumerations for safe and simple use; §2.5.