A review of Bjarne's small book

Leanne Dong

October 20, 2020

Contents

1 The Basics **1**

2 User-defined type **2**
2.1 Structures . . . 2
2.2 Classes . . . 3
2.3 Unions . . . 5
2.4 Enumerations . . . 5

3 Modularity **5**

4 Classes **5**

5 Essential Operations **5**
5.1 Copy and Move . . . 5
5.2 Resource Management . . . 5

6 Templates **6**
6.1 Lambda . . . 6

7 Concepts and Generic Programming **6**

8 Library Overview **6**

9 String and Regular Expression **6**

10 Input and Output **6**

11 Containers **6**

12 Algorithms **6**

13 Utilities **6**
13.1 Resource Management . . . 6
13.1.1 unique_ptr and shared_ptr . . . 6

14 Numerics **6**

15 Concurrency **6**

16 History and Compatibility **6**

1 The Basics

- Programs
- Functions and control flows
- Types, Variables and Arithmetic

- Scope and Lifetime
- Constants
- Pointers, arrays and References
- Tests
- Mapping to Hardware

Iteration

2 User-defined type

The set of C++ build-in types[1] and operations is rich, but deliberately low-level. The directly and efficiently reflect the capabilities of conventional computer hardware. However, they don't provide programmers with high-level facilities to write advanced applications easily. To overcome this, 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.

2.1 Structures

The first step is building a new type is to putting elements we need into a data structure, a struct:

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

The first version of Vector consists of an int and a double*. A variable of type Vector can be defined as

```
Vector v;
```

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

```
void vector_init(Vector& v, int s)
{
    v.elem = new double[s];
    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 ref, so that vector_init() can modify the vector passed to it.

The new operator allocates memory from *free store* (also known as *dynamic memory, or heap*). Object allocated on the free store are independent of the scopt from which they are created and live until they are destroyed using the delete operator.

A simple implementation of Vector could be

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

[1] types that can be built from the fundamental types, the const modifier, and the declarator operator

```
    for (int i = 0; i!=s; ++i)
    {
        sum+=v.elem[i];
    }
    std::cout << sum << " \n";
    return sum;
}

int main()
{
    read_and_sum(10);
}
```

There was a long to go from above to the elegant std::vector.

Compiler Explorer

We use .(dot) to access struct members through a name (and through a reference) and -> to access struct members through a pointer. For instance

```
void f(Vector v, Vector& rv, Vector* pv)
{
    int i1 = v.sz; // access through name
    int i2 = rv.sz; // access through reference
    int i3 = pv->sz; // access through pointer
}
```

Now another example from Herb's talk.

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

struct User {
    std::string name;
    int age;
};

std::vector<User> users = { {"Cat", 3}, {"Fish", 5} };

int main()
{
    auto sort_by_age = [] (auto& lhs, auto& rhs)
    {
        return lhs.age < rhs.age;
    };

    std::sort(users.begin(), users.end(), sort_by_age);
}
```

Compiler Explorer

2.2 Classes

From the struct example we see that having the data specified separately from the operations on it has advantages. Now we will establish a tighter connection between the representation and the operations for a user-defined type to have all the properties of a 'real type'. Specifically, we should keep representation inaccessible to users so as to ease use, ensure consistent use of the data, and allow us to later improve the representation. To this end, we need to distinguish

1. The interface

2. A type (to be used by all)

3. The implementation of the type (which has access to the otherwise inaccessible data)

The is the so-called class. Class provides greater level of abstractions compared to struct.The interface is defined by the public member of a class, and private members are accessible only through that interface. For instance,

```
class Vector {
public:
Vector(int s) :elem{new double[s]}, sz(s) { } //construct a vector
double& operator[] (int i) { return elem[i]; } //element access via subscripting
int size() { return sz; }

private:
double* elem; // the pointers to the elements
int sz;       // the number of elements
}
```

Let us now define a variable of our new type Vector:

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

Intuitively, the Vector object is a placeholder containing a pointer to the elements (elem) and the number of elements (sz). The number of elements can vary from Vector object to Vector object, and one particular Vector object can have a different number of elements at different times which we shall see soon. However, the Vector object itself is always the same size.[2] This is the fundamental building blocks for handling varying amounts of information in C++. Later on, we will discuss the design and usage of such objects.

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

```
double read_and_sum(int s)
// read s integers from cin and return their sum, s is assumed to be positive
{
    Vector v(6); // make a vector of s elements
    for (int i = 0; i!=v.size(); ++i)
    {
        std::cin >> v[i]; // read into elements
    }
    double sum = 0;
    for (int i = 0; i!=s; ++i)
    {
        sum+=v[i];
    }
    std::cout << sum << " \n";
    return sum;
}
```

A member "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() earlier. Unlike an ordinary function, a constructor is guaranteed to be used to initialize objects of its class. Hence, defining a constructor eliminates the problem of uninitialized variables for a class.

Vector(int) defines how objects of type Vector are constructed by explicitly stating that it needs an integer to do so. The 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)
```

[2]There are two different meaning of 'size' here, the first one is the size indicated by the sz member (v.sz), which can vary; The second one is the size of the Vector type (sizeof(v)), i.e. the size of pointer and an integer.example 1 and example 2. Note in the 2nd example that, sizeof(V3) is different from what one might expect, due to the padding introduced for the purpose of the alignment. Note also that, do not assume that the size of a type is just the sum of the sizes of its members.

Here it says, we first initialize elem with a pointer to s elements of type double obtained from the free store. Then we initialize sz to s. The subscript function operator[] allows us to access elements. It returns a reference to the appropriate element (a double& allowing both reading and writing).

The size() function provides the users with the number of elements.

Clearly, several aspects are missing.

1. Error handling
2. A lack of mechanism to return the array of doubles acquired by new. Shortly, we will show how to use destructor does this.

There is no fundamental difference between a struct and a class; a struct is simply a class with members public by default. For instance, one can define constructors and member functions inside struct. The implementation of the discussed example is at Compiler Explorer

2.3 Unions

2.4 Enumerations

3 Modularity

4 Classes

5 Essential Operations

5.1 Copy and Move

5.2 Resource Management

Garbage collection is fundamentally a global memory management scheme.

Before resorting to garbage collection, one should systematically use resource handles, that is, we let each resource have an owner in some scope and by default be released at the end of its owners scope. This is the so-called RAII in C++. RAII stands for Resource Acquisition Is Initialization and is integrated with error handling with error handling in the form of exception. Under RAII, we can move resource from scope to scope via move semantics or smart pointers, and shared ownership can be done with "shared pointers".

Here is an example,

```
//Without RAII
  try
  {
    auto x = make_unique<X>(Stuff);
    // risky stuff
  }
  catch (exception& e)
  {
          //react
  }
```

```
// RAII
  try
  {
    auto x = new X(Stuff);
    // risky stuff
        delete x;
  }
  catch (exception& e)
  {
          //react
  }
```

RAII is used a lot in C++ standard library. For instance, memory (string, vector, map, unordered_mapetc., files(ifstream, ofstream, etc.), threads (thread), locks (lock_guard, unique_lock, etc), and general objects (through unique_ptr and shared_ptr). The results is implicit resource management that is invisible in common use and leads to low resource retention durations.

6 Templates

6.1 Lambda

Lambdas come up when we would like to write programs that utilize unnamed function objects. The following snippet shows how a lambda is defined in C++ source code. The syntax for a lambda is as follows:

```
[](){};
```

The braces represent the capture block. A lambda uses a block to capture existing variables to be used in the lambda.

7 Concepts and Generic Programming

8 Library Overview

9 String and Regular Expression

10 Input and Output

11 Containers

12 Algorithms

13 Utilities

13.1 Resource Management

13.1.1 unique_ptr and shared_ptr

In <memory>, the standard library provides two smart pointers to help manage objects on the free store. Namely,

1. unique_ptr to represent unique ownership
2. shared_ptr to represent shared ownership

Example of unique_ptr

14 Numerics

15 Concurrency

16 History and Compatibility