Lesson 13 - Vectors
Dynamic Data Storage

Summary

In this lesson we introduce the Standard Template Library by demonstrating the use of Vectors to provide dynamic storage of data elements.

New Concepts

Vectors, the STL, default and copy constructors, APIs.

Vectors

In order to aid C++ Programmers, the Standard Template Library (STL) is provided with a suite of containers, algorithms and iterators, suitable for addressing a range of common programming tasks. The tools of the STL are expressed as templates to allow code reuse. Now that you are familiar with the basics of templates, this is a good time to explore a particular data structure that is part of the STL, namely the vector.

We demonstrate the vector because this tends to be one of the easier containers to understand while being one of the most useful. Once you've got to grips with it, you should have enough understanding to explore many other containers within the STL, like the stack, queue, map and set.

Let's begin with a situation where we would require the functionality of a vector. For instance suppose we have a number of `Enemy` objects (defined in the Lesson on Classes). Ordinarily we might allocate them in an array, but in this instance we do not know beforehand how many `Enemy` objects we need. We could allocate them dynamically on the heap, but that becomes cumbersome when we have extra `Enemy` objects to add. What we need is a data structure where we can conveniently add, access and remove elements without worrying about the underlying implementation; the vector fits the bill nicely for this task.

To begin with we need to make some additions to the `Enemy` Class. Note the amended declaration beginning on line 1. Two lines in particular are important in this lesson, the 'default constructor' on line 4 and the 'copy constructor' on line 6. We'll soon see that these methods are necessary to create a vector of `Enemy` objects. Note too that we have overridden the assignment operator on line 7, this is necessary if we want to reassign `Enemy` objects. As operator overloading is covered in lesson 12 we do not explain the technique here, although we provide the method definition on line 21 for completeness.

```
 1  class Enemy {
 2  public:
 3     static const int default_hps = 4;
 4     Enemy():hit_points(default_hps){ score = new int(0); }
 5     Enemy(int hps):hit_points(hps){ score = new int(0); }
 6     Enemy(const Enemy& src);
 7     Enemy& operator=(const Enemy& rhs);
 8     virtual ~Enemy(){ delete score; }
 9     virtual int get_hit_points() const { return hit_points; }
10     virtual int get_score() const { return *score; }
11     virtual void set_hit_points(int new_hit_points){ hit_points = new_hit_points; }
12     virtual  void set_score(int new_score){ *score = new_score; }
13  protected:
14     int hit_points;
15     int* score;
16  };
```

vectors.cpp

First let's talk about the default constructor (line 4). A default constructor is a constructor that takes no parameters. If you provide the declaration of a class but omit a constructor, then the compiler will automatically provide a 'nil-argument' default constructor for you. When we declared the `Enemy` class however, we provided a constructor which expects an integer parameter (line 5). Due to the existence of this constructor our class overrides the default behaviour for the compiler, meaning a default constructor won't be provided. We have to provide a nil-argument constructor for ourselves which we do on line 4.

On line 6 we provide the declaration of a copy constructor. A copy constructor is called whenever the program is required to create a copy of an object. This too is generated automatically by the compiler if we don't provide one. If the class allocates any memory on the Heap however, (like the `Enemy` class does in its constructors), then we need to provide a copy constructor. If we don't, then the program will not allocate the new memory when the copy is created. The program will simply copy the address of any pointers in the old class. This situation results in two objects containing pointers to the same chunk of memory. Sooner or later when one of the objects is deleted, its allocated memory will be released, leaving the remaining object with a pointer to released memory. When you subsequently use this pointer, your program will most likely crash.

Lines 17 to 19 provide the definition for the copy constructor, which accepts a constant reference to the original object. Note on line 19 the `score` field is allocated new memory and initialised to the value of the original object (`src`). This is followed by the definition of the overloaded assignment operator on line 21.

```
17 Enemy::Enemy(const Enemy& src):
18 hit_points(src.hit_points),
19 score(new int(*src.score)) { }
20
21 Enemy& Enemy::operator=(const Enemy& rhs) {
22    if(this == &rhs) {
23       return (*this);
24    }
25    //free old memory
26    delete score;
27
28    //copy new memory
29    hit_points = rhs.hit_points;
30    score = new int(*rhs.score);
31
32    return *this;
33 }
```

vectors.cpp

Now let's practise using some vectors. First we need to include the appropriate header file (`<vector>`) which we take care of on line 36. On line 39 we also inform the compiler that we are 'using' `std::vector`. We include this line to demonstrate an alternate (more specific) application of the `using` statement, but in fact this line is unnecessary because line 38 serves the same purpose, albeit in a more general way. Note on line 35 we also include the 'header' file for our `Enemy` class.

```
34 #include <iostream>
35 #include "Enemy.h"
36 #include <vector>
37
38 using namespace std;
39 using std::vector; //unnecessary here
```

vectors.cpp

We begin the `main` function by creating a vector of integers on the stack. We pass the vector constructor two integer arguments (line 42). The first states how many elements we wish to create in our vector (5) and the second states what their values should be (0). Hence, when the vector called `scores` is created it will contain 5 integer variables with the value 0.

On line 44 we use the array index operator (`[ ]`) to assign values to elements of the vector. Note this is the same syntax as we use with arrays. This is possible because the vector class overloads the array index operator, providing convenient syntax for 'random access' of our vector.

On line 46 we use the `size` method provided by the vector class, to iterate through the `scores` vector. You'll discover the `size` method is very useful in this capacity, because vectors can change size dynamically and `size` provides a convenient way for determining how many elements are contained within any vector. Within the `for` loop we use the array index operator again to display each element of the vector.

```
40 int main() {
41
42    vector<int> scores(5, 0);
43
44    scores[0] = 1; scores[1] = 2; scores[2] = 3;
45
46    for(int i = 0; i < scores.size(); ++i) {
47       cout << "scores " << i << " = " << scores[i] << "\n";
48    }
```

vectors.cpp

Now let's create a vector of `Enemy` objects (see line 49). As you can see we can create vectors on the heap too by using the `new` keyword. We pass a single argument to the constructor this time, the integer value 3. This value simply tells the program to construct a vector holding 3 `Enemy` objects. The vector will call the default constructors of the `Enemy` objects it creates here, hence the need to provide a default constructor.

On lines 53 and 54 we demonstrate the `push_back` method. The `push_back` method, as its name suggests, adds objects of the appropriate type to the 'back' of the vector. When we run the program, line 55 should display that the size of the vector is now 5 (3 initial `Enemy` objects plus 2 pushed onto the back of the vector). The `push_back` method accepts a constant reference to the object being 'pushed' onto the vector. A copy of this object will be created in the process. Creating this copy will result in the object's copy constructor being called, hence the need to provide a copy constructor.

```
49    vector<Enemy>* enemies = new vector<Enemy>(3);
50
51    Enemy fourth, fifth;
52
53    enemies->push_back(fourth);
54    enemies->push_back(fifth);
55    cout << "size of enemies = " << enemies->size() << "\n";
```

vectors.cpp

Now take a look at line 57. In this `for` loop we use an `iterator` to 'iterate' through the elements of the `enemies` vector. Every vector class provides the built-in methods `begin` and `end` to create iterators at the beginning and the end of any vector container. An iterator can be used as a convenient pointer to elements within the vector, hence on line 59 and 60 we call `set_score` and `get_score` on the `Enemy` object `it` points to.

```
56    int i = 0;
57    for(vector<Enemy>::iterator it = enemies->begin(); it != enemies->end(); ++it)
58    {
59       it->set_score(scores[i]);
60       cout << "score = " << it->get_score() << "\n";
61       ++i;
62    }
```

vectors.cpp

So far you have seen two methods of accessing elements within a vector: using the array index operator and iterators. Iterators have several benefits over array indexing, the first being that you can use an iterator to insert an item into a vector at a specified position. This is accomplished with the `insert` method which requires an iterator as the first argument and a reference to the item we wish to insert as the second (line 64). When an item is inserted into a vector the items that come after the insertion point are shuffled down to make way for the new addition. Similarly you can use the method `erase` to remove items from a vector (line 65). Inserting and erasing require the re-assignment of elements within a vector so you must ensure you provide a method to overload the assignment operator (`=`) in your element class.

```
63    Enemy newthird;
64    enemies->insert(enemies->begin() + 2, newthird);
65    enemies->erase(enemies->begin() + 2);
```

vectors.cpp

We can also remove elements from the end of a vector using the `pop_back` method (see line 69). The `pop_back` method doesn't return a value. If you wish to keep the last element of the vector, call `back` and assign it to a reference before you remove it (see line 66).

```
66    Enemy& back = enemies->back();
67    cout << "hit points last element = " << back.get_hit_points() << "\n";
```

```
68
69     enemies->pop_back();
70     cout << "new size of enemies = " << enemies->size() << "\n";
```

vectors.cpp

Finally we can completely clear a vector vector of elements with the `clear` method. We do this on line 71 before freeing the heap memory containing our `enemies` vector.

```
71     enemies->clear();
72
73     delete enemies;
74
75     return 0;
76  }
```

vectors.cpp

The vector class contains many more methods than we have explained in this lesson, although we have provided a demonstration of the most frequently used. When you're using an STL template class you can find a complete Application Programming Interface (API) at:

`http://www.cplusplus.com/reference/stl`.

An API provides standard descriptions on classes and their methods, created to aid programmers in their use. The type of information available includes: what methods are available, what parameters they methods require and what types methods return. You can typically also find information on performance characteristics of methods and algorithms to help you make informed choices about which method is appropriate for which situation.

New Feature

C++11 has added a new feature through the resurrection of the keyword `auto` (for many years `auto` hung around as a deprecated keyword in both C and C++). By declaring a variable to be of 'type' `auto` you inform the compiler that it must discover the precise type of your variable at run-time. Consequently, this bit of 'syntactic sugar' means less code for you to write. A handy example is the somewhat long-winded `for` loop declared earlier in this lesson (see lines 56-62) which we can now redefine using `auto` on lines 77-82. (Declaring the variable `i` on line 77 as `auto` would be pointless!)

```
77     int i = 0;
78     for(auto it = enemies->begin(); it != enemies->end(); ++it) { // auto used here!
79        it->set_score(scores[i]);
80        cout << "score = " << it->get_score() << "\n";
81        ++i;
82     }
```

vectors.cpp

Note that you cannot use `auto` as the return type in a function or method declaration without first providing some extra help for the compiler (see lines 83-86). To declare a return type as `auto` you must use the new trailing return type syntax in combination with `decltype` (see line 89).

```
83  template <typename X, typename Y>
84  auto addWrong(X x, Y y) { // error! expects trailing return type
85     return x + y;
86  }
87
88  template <typename X, typename Y>
89  auto addRight(X x, Y y) -> decltype(x + y) { // that's better!
90     return x + y;
91  }
92
93  // code omitted for brevity
94  auto y = addRight(10, 20.134);
```

vectors.cpp

Exercises

1. How does storage differ between a `vector` container and a `list` container? Given the way Caches reduce costly memory access, which container do you think might sometimes be better in terms of improving "cache hit rate"?

2. Create a vector of `ArmedEnemy` types by implementing the necessary code in `ArmedEnemy` so that it can be used with the vector template class.

3. Explore some of the other container classes in the STL. Create a "Stack" and a "Queue" of `ArmedEnemy` types, practise adding, removing and iterating through the elements as shown with the vector.

4. Create a vector of 10 `ArmedEnemy` types. Give each `ArmedEnemy` class in the vector a different value for its `hit_point` field. Use the appropriate "Algorithms" functionality in the STL to first randomly shuffle the `ArmedEnemy` members of the vector then sort them according to the value in their `hit_point` field.