Lesson 9 - Classes
Objects and Instantiation

Summary

In this lesson we explain the concept of classes, objects and instantiation.

New Concepts

Classes, objects, fields, methods, constructors, destructors and instantiation.

Classes

In this lesson we introduce a particularly important aspect of C++, namely classes. Classes allow us to design our projects in terms of "objects", a technique referred to as Object Orientated Programming (OOP). We define a class providing a specification or blueprint for how an object should behave. Then we instantiate objects (creating instances) from the specification.

Take a look at the `classes.cpp` listing. On line 1 we begin the declaration of our first class using the keyword `class` followed by the name, in this case `Enemy`. We now proceed to define the common behaviour and state that our `Enemy` objects will possess. We define the behaviour via class "methods" and state via class "fields". Methods are simply functions that belong to a particular class, whereas fields are the variables.

First note the use of the keywords `public` and `private` on lines 2 and 9 respectively. These are access-modifiers. When declaring a new class, you should specify which data and behaviours you wish to remain visible to any program that uses your class using the `public` modifier. You can also specify which data should be private and hence inaccessible, using `private`.

The class declaration below provides "forward declarations" of a number of class methods for accessing and retrieving fields of the class; these are public. In addition there are two data fields `hit_points` and `score` which are private. You must always close a class declaration with a closing bracket and colon (};), see line 12.

All classes you declare require a "constructor" and a "destructor", and you will generally implement at least one constructor and destructor for you class definitions (if you don't however, the compiler will generate defaults for you). The constructor and destructor for the Enemy class are located on lines 3 and 4 respectively. They share the same name as the class you are defining; a destructor always begins with a tilde (∼). As you can see we've provided one constructor which takes an integer variable as a parameter.

```
1  class Enemy {
2  public:
3     Enemy(int hps);
4     ~Enemy();
5     int get_hit_points() const;
6     int get_score() const;
7     void set_hit_points(int new_hit_points);
8     void set_score(int new_score);
9  private:
10    int hit_points;
11    int* score;
12 };
```

classes.cpp

Once you have declared your class, you must provide implementations of the class methods. Usually we put the declaration into a header file (`Enemy.h`) and the implementations into a corresponding cpp file (`Enemy.cpp`).

First we provide the implementation of the class constructor (see line 13). Notice the use of the scope operator (`::`) once again. We use the scope operator to specify that the methods we wish to declare belong to a particular class. On line 14 we initialise the `hit_points` field of the `Enemy` class and on line 15 we allocate some memory for the `score` field. Now when an Enemy class is instantiated the `hit_points` field will be set to the `hps` parameter passed to the constructor. In addition, memory will be allocated and set to 0 for the `score` field. It is good practise to allocate any heap memory in the class constructor, and subsequently release that memory in the destructor, which we do on line 19.

```
13 Enemy::Enemy(int hps):
14 hit_points(hps) {
15    score = new int(0);
16 }
17
18 Enemy::~Enemy() {
19    delete score;
20 }
```

classes.cpp

Now we define the rest of the class methods. These consist of so-called "getters" and "setters" for the class fields. On lines 21 and 25 we define methods to "get" the `hit_points` and `score` fields respectively; on lines 29 and 33 we define methods to "set" these fields to a parameter value. It would have been possible to make the class fields `public`, and therefore grant access without the need for methods, but you should avoid this as much as possible. It's better to control access to class fields via methods, where you can control what operations are possible. For example, you can decide whether you want to return copies or original data members via references or pointers. If you do return original data members you can specify whether they can be modified via the `const` keyword.

In our example we return-by-value, hence we grant access to copies of the data members in the "getter" methods. Note the use of `const` on lines 21 and 25 which stipulates to the compiler that these methods will not change the data members of the Enemy class.

```
21 int Enemy::get_hit_points() const {
22    return hit_points;
23 }
24
25 int Enemy::get_score() const {
26    return *score;
27 }
28
29 void Enemy::set_hit_points(int new_hit_points) {
30    hit_points = new_hit_points;
31 }
32
33 void Enemy::set_score(int new_score) {
34    *score = new_score;
35 }
```

classes.cpp

Now let's instantiate some "enemies" in our main method. First take a look at line 38 where we initialise an object of the `Enemy` class on the Stack. We call this object `e1`, and we pass the value 2 as the required parameter to the constructor. We can verify that the `hit_points` field does in fact hold the value 2 by calling the corresponding "getter" method `get_hit_points` see line 39. Running the program should indeed show the `hit_points` field to be 2.

Next we declare another Enemy object called `e2`. This time however we declare our object on the heap. On line 42 we call the "setter" method `set_hit_points` to set `e2`'s `hit_points` field to 3. We verify this happened on line 43. Next we also set the score of the object `e2` to 2 by calling the `set_score` method and verify its value on line 46. Notice the user of the `->` operator to call a method on a pointer to an object. We could accomplish the same operation like so: `(e2).set_score(2)` but `->` simply provides a neat short-cut.

```
36 int main(void) {
37
38    Enemy e1(2);
39    cout << "hit points = " << e1.get_hit_points() << "\n";
40
41    Enemy* e2 = new Enemy(2);
42    e2->set_hit_points(3);
43    cout << "hit points = " << e2->get_hit_points() << "\n";
44
45    e2->set_score(2);
```

```
46     cout << "score = " << e2->get_score() << "\n";
47
48     delete e2;
49     e2 = NULL;
50     return 0;
51 }
```

classes.cpp

Finally on line 48 we use `delete` on `e2` to release the heap memory allocated and set the pointer to `NULL`. This concludes our simple introduction to classes.

C++ Style

In C++ it's typical to write your class definition in a header file (.h or hpp) and implement each class method in a corresponding source file (.cpp). As the number of new classes you create grows, you will want your objects to make use of each other's functionality. This can be achieved by 'including' another class' header file. For example, `cache.h` and `element.h` show two class declarations. As the `cache` class uses the `element` class, the `element.h` file has been included in `cache.h`, (see line 2 below).

In C++, when including header files you must avoid the creation of circular dependencies where two classes require each other's functionality (either directly or indirectly). For example, suppose at some point in the project you decide you also want to include the `cache` header file in the `element` header file. The compiler will complain because you have created a circular dependency between the `cache` and the `element` classes. While circular dependencies signify the presence of bad design, you can solve the problem of the dependency between the `cache` and `element` classes with the use of 'forward declarations'. To write a forward declaration, write the name of the class you wish to forward declare before the class declaration (see `element.h` line 2). You can then declare either pointers or references to the forwardly declared class. In order to use the methods of a forwardly declared class, include the class' header file into the source file of the using class (see `element.cpp`, lines 1 and 4).

```
1 #pragma once
2 #include "element.h" // cache contains element objects
3
4 class cache {
5     element mElements[N]; // including element.h lets us create elements by 'value'
6 public:
7     void cacheMethod();
8 };
```

cache.h

```
1 #pragma once
2 class cache; // forward declaration of cache class
3
4 class element {
5     cache *mPtrCache; // must be a pointer or reference type!
6 public:
7     void elementMethod();
8 };
```

element.h

```
1 #include "cache.h"
2
3 void element::elementMethod() {
4     mPtrCache->cacheMethod(); // can now invoke methods of cache class
5 }
```

element.cpp

Leaving aside circular dependences, forward declarations may also speed up the compilation time of your project because they produce less work for the compiler than including other source files. You should therefore favour using forward declarations whenever possible. Note, however, you can only declare a pointer or reference to a forwardly declared class in the header file!

Exercises

1. Modify the `Enemy` constructor so that the `score` field is set to an initial supplied value like `hit_points`.

2. Create a new class called `Player` which possesses the same fields and methods as `Enemy`. In addition, give the `Player` a string field called "name", use the `std::string` class. Create "name" on the heap and provide methods to "get" and "set" the name of player.

3. Re-write the Binary Search Tree code from Lesson 5 to create a Binary Search Tree class. Implement a Constructor, Destructor in addition to methods which allow an integer to be inserted and the tree to be printed to screen. Implement a search method which returns `true` if a supplied value is present in the tree, `false` otherwise.

4. Modify the "battleship" game in the previous exercise to use classes. Create a battleship class which possess a "hit-points" and "score" field. When the user lands a shot, the score field should be increased until hit-points is reached. This should denote that your battleship has been sunk. How you represent the battleground is your choice.