Course
ELEC-A7151
Course materials
Your points
Code
Code Vault
Course Pages
MyCourses
Teams Channel

« 1 Introduction Course materials 3 Object relationships in C++ »
ELEC-A7151 / Module 3: Classes and Object-oriented Programming / 2 Object-oriented programming

2 Object-oriented programming¶

Contents

Module 1 already briefly discussed the concept of a class, with member variables and functions that are used to access instantiations of a class. An object-oriented C++ program can be entirely build on classes, and objects that represent instances of a particular class, apart from the *main* function that starts the program. Class is an abstract data type that hides the internal implementation of handling the data, and the other parts of the program only consider the public interface of the class. Usually, as a good design principle, all data of an object is accessed through functions of the class, and the variables are rarely operated directly. The public part of the class is the **interface** to the class.

Through **inheritance**, general class definitions can be refined into more specific ones. For example, assuming a basic RPG game, we might have a base class *Creature* that specifies properties that are common to all kinds of creatures in the game, such as *size*, *hitpoints* and so on. Further on, we might have class *Troll* that inherits from *Creature*. Troll has all the properties (i.e., data and functions) defined for Creature, but may add more Troll-specific features that are not common to other Creatures. Inheritance represents an "is-a" relationship: a *Troll* is a *Creature* (but *Creature* is not necessarily a *Troll*).

Below is an example of C++ class definitions of two such creatures.

```
1#include <string>
2#include <vector>
3#include <iostream>
4
5class Creature {
6public:
7    // Creature constructor has three arguments.
8    // We initialize the member variables based on the arguments using
9    // initialization list
10    Creature(const std::string& name, const std::string& type, int hp)
11        : name_(name), type_(type), hitpoints_(hp) { }
12
13    // For querying name, type or HP. Will not change object state.
14    const std::string& GetName() const { return name_; }
15    const std::string& GetType() const { return type_; }
16    int GetHitPoints() const { return hitpoints_; }
17
18private:
19    // Naming convention: member variables have a trailing underscore
20    std::string name_;
21    const std::string type_;
22    int hitpoints_;
23};
24
25
26class Troll : public Creature {
27public:
28    // Construct Troll, use following arguments for Creature initialization
29    Troll(const std::string& name) : Creature(name, "Troll", 10) { }
30};
31
32
33class Dragon : public Creature {
34public:
35    // Construct Dragon, use following arguments for Creature initialization
36    Dragon(const std::string& name) : Creature(name, "Dragon", 50) { }
37};
38
39
40int main() {
41    Troll troll("Diiba");
42    Dragon dragon("Rhaegal");
43
44    // Assignment from Troll to Creature is possible in this case
45    Creature cr = troll;
46
47    std::cout << troll.GetName() << " has " << troll.GetHitPoints()
48              << " hit points." << std::endl;
49
50    // This vector can consist of all Creature subclasses
51    std::vector<Creature> monsters;
52    monsters.push_back(troll);
53    monsters.push_back(dragon);
54    monsters.push_back(Dragon("Viserion"));
55}
```

First, on line 5 starts the definition of base class **Creature** that defines variables and functions common to all creatures. The initial values for the three variables are set in the **initializer list** of the constructor. The initialization list sets the initial value of member variable *name_* to constructor argument *n*, and so on. Use of initialization list is recommended whenever possible: it ensure that the object has its members properly initialized at all times.

The class has three functions to query the values of the member variables. Note the use of *const* in the function interface: for example, *GetName* returns a **reference to const string**, i.e. the string inside the object cannot be modified through the return value. On the other hand, *const* after the function name tells that the function call is not going to change the state (any of the variables) of the object. The variables themselves are declared with *private* visibility, so they cannot be directly accessed, according to the information hiding principles. Such functions that are intended for querying the state of the object in secure way are called **accessors**.

We add underline at the end of member variables. This is a **naming convention** that separates names that are class member variables from other names, such as local variables or function attributes. This naming convention is recommended, e.g., in the Google C++ Style Guide, which is one of the style guides for C++. There also exists other guides and naming conventions. Following a consistent and agreed upon style is important, especially in larger software with multiple developers, to make the code easier to understand.

Definition of class **Troll** starts from line 26. The first line says that *Troll* is **inherited** from class *Creature*, i.e., it inherits the functions and member variables defined in base class. Therefore the class definitions looks pretty short, even though Troll has all the same capabilities as any other Creature. There is no need to rewrite the code that is common to all creatures. Don't worry about the *public* in front of Creature yet. We will get to it soon.

The *Troll* constructor also has an **initializer list** that sets initial values for the members of the new object, but it looks a bit different than the earlier initialization lists we have seen. The *Troll* initialization calls the constructor of the base class with given parameters (for name, type and initial hit points). Otherwise the constructor does not include any functionality, as indicated by the empty code block inside the brackets.

The *Dragon* class definition is very similar to the *Troll* class at this point, except for different initial values for member variables.

The main function starting from line 40 first creates two variables: **troll** of the *Troll* type, and **dragon** of the *Dragon* type. In other words, *troll* and *dragon* are objects of these two classes. Because of the inheritation relationship, object *troll* also matches the *Creature* type. Therefore the assignment on line 45 is ok.

The output on line 47 demostrates that we can use a *Troll* type object to access functions defined in the *Creature* class (e.g. *GetName*) because of the inheritance relationship. Likewise, if we create a vector for *Creature* type objects, we can add any types inherited from the Creature class to the vector.

2.1 Constructors and destructors¶

As we have seen in past examples, the lifecycle of a new object begins with calling a **constructor** of a class. Constructor initializes the object's state, and if needed, allocates resources needed by the class. In the class definition, a constructor looks like function that has the same name as the class and no return type:

```
Creature(const std::string& name, const std::string& type,
        int hit_points) : name_(name),type_(type), hitpoints_(hit_points)
```

The constructor may initialize the class members using an **initializer list** that follows after the colon. Use of an initializer list is recommended whenever non-default initialization is needed, because it ensures that the class is properly initialized from the beginning.

If a constructor is implemented outside the class definition, it looks like this:

```
Creature::Creature(const std::string& name,
        const std::string& type, int hit_points)
```

followed by a block of code inside curly braces, as with any function. The constructor **does not have a return value** and, thus, does not return anything.

Sometimes it is necessary to define an explicit **destructor** for a class. The destructor is called as the last thing before an object is deleted. In examples so far we haven't defined the destructor, and the compiler just releases the memory automatically allocated for the direct class members. If the class implementation has allocated some additional resources (files, additional dynamic memory, etc.), those need to be explicitly released in destructor.

Destructor for the Creature class would be defined inside the class definition along with other functions, and it looks almost like any other function:

```
~Creature() { /* code */ }
```

However, destructor does not have a return value, and it does not have parameters, because it is usually automatically called when an object is about to be deleted.

The destructor can also be defined outside the class definition:

```
Creature::~Creature() { /* code */ }
```

« 1 Introduction Course materials 3 Object relationships in C++ »