This course has already ended.
The latest instance of the course can be found at: Object oriented programming with C++: 2023 Autumn

3 Object relationships in C++¶

- **You can download the template for the programming tasks of the module as a `zip` file from `this link`.**

Contents

3.1 Name visibility between classes¶

Until now we have seen two categories of variables and functions in classes: **public** variables or functions are visible to all other parts of the program that know about the class definition. **private** variables or functions are only available to functions inside the class.

In a third category are **protected** variables (or functions) that are visible to the class itself, and all those classes that are inherited from it. For example, if we modify the *Creature* class definition into the following, variables *name_type* and *hitpoints* are protected variables so *Troll* and *Dragon* classes, but not for other classes that do not inherit from *Creature*.

```
3class Creature {
2public:
1    Creature(const std::string& name, const std::string& type, int hp)
4        : name_(name), type_(type), hitpoints_(hp) { }
5
6    const std::string& GetName() const { return name_; }
7    const std::string& GetType() const { return type_; }
8    int GetHitPoints() const { return hitpoints_; }
9
10protected:
11    std::string name_;
12    const std::string type_;
13    int hitpoints_;
14};
```

C++ Primer Chapter 7.2.1 (Friends)

A class can declare another class to be a **friend**. Such other class has direct access to all protected or private members of the class declaring the friendship. This should be used sparingly and with careful consideration, because it bypasses the principles of information hiding.

A class can also declare an independent **function to be a friend**. Such function can access the "private parts" of the class. As with friend classes, this mechanism should be used sparingly. A common example of this is when we add I/O stream support by overloading the << and >> operators to handle the class I/O properly. We will see an example of this later.

Below example shows how class *SomeClass* is declared a friend of *Creature*, and how function *Print(Creature& c)* is declared a friend. *Print* is a function in global namespace (not part of any class or in any explicit namespace), that (presumably) prints something about the class.

```
class Creature {
public:
    Creature(const std::string& name, const std::string& type, int hp)
        : name_(name), type_(type), hitpoints_(hp) { }

    const std::string& GetName() const { return name_; }
    const std::string& GetType() const { return type_; }
    int GetHitpoints() const { return hitpoints_; }

    friend class SomeClass;
    friend void Print(Creature& c);

private:
    std::string name_;
    const std::string type_;
    int hitpoints_;
};
```

Inherited classes can also have an accessor modifier in front of the class name, as we saw before:

```
class Troll : public Creature { ... };
```

That modifier states the **loosest** access mode for the members of the inherited class. In this case, the loosest mode is *public* and, as public is the loosest modifier in existence, the members of Creature preserve their access modes. If, instead, one were to use *protected*, the loosest mode is *protected* meaning that all members that are looser than *protected* will be changed to *protected*. In this case that means that the *public* member functions of *Creature* would be treated as being protected members of *Troll*. The modifier is *private* by default for classes, which means that if no access modifier is present, it is treated as being *private*.

3.2 Static members and functions¶

C++ Primer Chapter 7.6 (Static class members)

Static members are members that are shared by all instances of the class. They are declared by prepending the keyword **static** in front of an usual declaration of the member, as in the example below.

For **static member variables**, this means that its value is shared and when it is altered by one object (or by a static function), the change will be visible to others. For example, one could use a static variable to keep track on how many instances have been created of a given class, as shown in the example below. One should be careful with static variables, however, and only apply them in limited cases.

Static member functions are functions that are defined within the class' namespace. As calling them isn't associated with any specific object, they **cannot access the object-specific members**. However, they can access them through an object (i.e. *if name_* is a non-static member of the class, they cannot do *name_ = "John"* but they can do *obj.name_ = "John"* if *obj* is an object of the class. Static member functions can be thought of as normal functions that are defined in the class' namespace and which are friends of the class (i.e. they have access to the private members).

Because a static member variables and functions are not associated with an object, they can be accessed using the class name directly, as done in the main function in the example below with the *CreateOne* function and *count_* variable.

```
1#include <stdlib.h>
2#include <iostream>
3#include <string>
4
5class Creature {
6public:
7    Creature(const std::string& name, const std::string& type, int hp)
8        : name_(name), type_(type), hitpoints_(hp) { }
9
10    const std::string& GetName() const { return name_; }
11    const std::string& GetType() const { return type_; }
12    int GetHitPoints() const { return hitpoints_; }
13
14private:
15    std::string name_;
16    const std::string type_;
17    int hitpoints_;
18};
19
20class Troll : public Creature {
21public:
22    // Create a new Troll object.
23    static Troll CreateOne() {
24        std::string name = "Troll-";
25        name += 'A' + (rand() & 0xf);
26        count_++;
27        std::cout << "We now have " << count_ << " Troll instances." << std::endl;
28        return Troll(name, rand() & 0x3f);
29    }
30
31private:
32    Troll(const std::string& name, int hp) : Creature(name, "Troll", hp) { }
33
34    static int count_;  // common to all Troll instances
35};
36
37// initialize the static variable
38int Troll::count_ = 0;
39
40int main(void) {
41    // no object reference needed for calling static function
42    Troll t = Troll::CreateOne();
43    std::cout << "Troll named " << t.GetName() << " has "
44        << t.GetHitPoints() << " hitpoints" << std::endl;
45    Troll t2 = Troll::CreateOne();
46    std::cout << "Troll named " << t2.GetName() << " has "
47        << t2.GetHitPoints() << " hitpoints" << std::endl;
48}
```

The example calls static function **CreateOne** that returns a Troll object with some random attributes. The function can be called before we have any Troll-typed object, but it provides us a new object. One common use of static functions is to create objects of the particular type.

3.3 Programming task¶

Points 10 / 10 My submissions 2 Deadline Friday, 8 October 2021, 19:59 To be submitted alone

⚠ This course has been archived (Saturday, 17 December 2022, 19:59).

Mammals

Objective: Practice defining inherited classes that contain static members

The template contains a definition of a **Mammal** class, and a main function that uses this class along with a **Dog** class and a **Cat** class. However, definitions of these classes are missing. Implement these classes according to the following:

- Both classes should inherit the Mammal class.
- Both classes should have a constructor with two arguments. The first argument is a string representing the name of the creature, and the second argument is the weight. Remember that the weight attribute is common to all Mammals, and is stored in the base class.
- Implement accessor function **GetName** that returns the name of the creature.
- Implement static function **MakeSound** that returns a string that contains the sound that the creature makes. A dog should say *Wuff!* and a cat should say *Meow*
- Investigate how the main function is implemented and how it uses these classes.
- Define the classes so that they compile together with the main function without producing errors or warnings when compiled.

dog.hpp

Choose File No file chosen

dog.cpp

Choose File No file chosen

cat.hpp

Choose File No file chosen

cat.cpp

Choose File No file chosen

Submit

3.4 Polymorphism¶

C++ Primer Chapter 15.3 (Virtual functions)

Polymorphism is one of the core concepts in object-oriented programming, related to class inheritance. Through polymorphism, the same function interface can be used to access different types of objects, resulting in behavior specific for that particular object.

Concretely, subclasses that inherit the base class can have alternative implementations of same function. The actual object type then determines which version of the function is used. This determination can be made dynamically at run time

C++ allows member functions of a base class to be declared **virtual** that tells the compiler that the actual function implementation to be used should be determined at run time using a virtual method table (vtable). If we have a vector consisting of Creature pointers, the actual objects can still be Dragons or Trolls, causing an appropriate version of the virtual function to be called.

For example, we could have a function *virtual std::string WarCry()* const; defined for *Creature* class, and have specific implementations for the function for both *Troll* and *Dragon* classes. During programming phase we only need to know that we will use this function for some *Creature*, without knowing its actual subclass. Using a vtable, the program will identify the actual type of the object and use the correct version of the function at run time, for example when we have a container that contains all sorts of *Creatures*. See the following example:

```
1#include <string>
2#include <vector>
3#include <iostream>
4
5class Creature {
6public:
7    Creature(const std::string& name, const std::string& type, int hp)
8        : name_(name), type_(type), hitpoints_(hp) { }
9
10    const std::string& GetName() const { return name_; }
11    const std::string& GetType() const { return type_; }
12    int GetHitPoints() const { return hitpoints_; }
13    virtual std::string WarCry() const { return "(nothing)"; }
14
15private:
16    std::string name_;
17    const std::string type_;
18    int hitpoints_;
19};
20
21
22class Troll : public Creature {
23public:
24    Troll(const std::string& name) : Creature(name, "Troll", 10) { }
25
26    virtual std::string WarCry() const { return "Ugaaga!"; }
27};
28
29
30class Dragon : public Creature {
31public:
32    Dragon(const std::string& name) : Creature(name, "Dragon", 50) { }
33
34    virtual std::string WarCry() const { return "Whoosh!"; }
35};
36
37
38int main() {
39    Troll tr("Ollka");
40    Dragon dr("Rhuagal");
41    Dragon dr2("Viserion");
42
43    std::vector<Creature*> monsters;
44    monsters.push_back(&tr);
45    monsters.push_back(&dr);
46    monsters.push_back(&dr2);
47
48    for (auto it : monsters) {
49        std::cout << it->GetName() << " says: " << it->WarCry()
50                  << std::endl;
51    }
52}
```

In the above example, the *WarCry()* function is declared as **virtual**. This causes the actual function-implementation to be determined dynamically at run time. The *main* function has a *monsters* vector consisting of *Creature*-type objects, and the behaviour of the *WarCry* function depends on the actual type of the creature. Try removing the *virtual* keyword from *WarCry* function declarations above, to see how behavior changes.

By declaring the function virtual, the compiler adds necessary code to allow the implementation determine at runtime which is the correct version of the function to be called (this is called **dynamic binding**). In C++ *dynamic* binding **works only for reference or pointer types**, and therefore we use *Creature* pointers in the vector in the *main* function. Actually, the pointers point to *Troll* or *Dragon* objects, and the vtable will help determine the correct version of function to be called.

On the other hand, if the *virtual* specifier was not given for the *WarCry* function, there would be no runtime information needed for the dynamic binding of the function. This would result in applying only the base class version of the function (you can try it).

Related link: Tutorial on polymorphism

Related Stackoverflow question: "C++ polymorphism without pointers"

3.5 Pure virtual functions and abstract classes¶

C++ Primer Chapter 15.4 (Abstract base classes)

A class member function can be declared **pure virtual**, meaning that there is no implementation for the function. **No objects can be created for a class that has pure virtual functions**, but then the class only works as base class for the classes that inherit from it, and provide an implementation for the functions that are pure virtual in the base class. A class with pure virtual functions is called an **abstract class**. It works as an interface to other classes, but cannot be used for creating objects. **A class is also abstract if it inherits from an abstract class but does not implement all the pure virtual functions of that class.**

Below is a modification to the *Creature* class such that the *WarCry* function becomes pure virtual, and the derived *Creature* classes becomes therefore an abstract class. This is indicated by *= 0* after the function declaration. We cannot directly create variables of type *Creature* after this, but will have to use *Troll* or *Dragon* types for new objects. The main function shown above works as before, because we did not create plain *Creature* objects, but only use it in the container (as a pointer) to indicate that any Creature can be stored in it.

Apart from the new *WarCry* function declaration, other parts of the program are as in the previous example.

```
3class Creature {
2public:
1    Creature(const std::string& name, const std::string& type, int hp)
4        : name_(name), type_(type), hitpoints_(hp) { }
5
6    const std::string& GetName() const { return name_; }
7    const std::string& GetType() const { return type_; }
8    int GetHitPoints() const { return hitpoints_; }
9    virtual std::string WarCry() const = 0;  // pure virtual function
10
11private:
12    std::string name_;
13    const std::string type_;
14    int hitpoints_;
15};
```