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

5 Dynamic Memory in C++¶

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

Contents

C++ Primer Chapter 12.1.2 (Managing memory directly)

Even though the C++ standard library and its containers readily implement some of the basic cases of dynamic memory management such as dynamically resizing vectors or linked lists, often we need to allocate some memory by ourselves.

C++ has built-in mechanisms for allocating memory and releasing it. The language reserves keywords **new** for allocation, and **delete** for releasing the memory. *malloc* or *free* functions should never be used, except, maybe, for the *very* advanced use case of implementing your own memory management system. In addition to the above, *new* calls the constructor of a class and *delete* calls the destructor of a class. We already know that constructor implements the proper initialization of an object. Respectively, **destructor** does the necessary operations to properly release the resources used by an object. It is the last bit of code that is executed before the object is gone.

Below we extend the previous examples with creatures with a new class called *Dungeon*, that holds a varying number of Creatures. We will use a *vector* to store the creatures, but will have to use pointers, to allow dynamic binding for correct type.

```
1#include <string>
2#include <vector>
3#include <iostream>
4#include <list>
5
6class Creature {
7public:
8    Creature(const std::string& name, const std::string& type, int hp)
9        : name_(name), type_(type), hitpoints_(hp) { }
10   virtual ~Creature() { }
11
12   const std::string& GetName() const { return name_; }
13   const std::string& GetType() const { return type_; }
14   int GetHitPoints() const { return hitpoints_; }
15   virtual std::string WarCry() const { return "(nothing)"; }
16
17private:
18   std::string name_;
19   const std::string type_;
20   int hitpoints_;
21};
22
23
24class Troll : public Creature {
25public:
26   Troll(const std::string& name) : Creature(name, "Troll", 10) { }
27
28   virtual std::string WarCry() const { return "Ugazaga!"; }
29};
30
31
32class Dragon : public Creature {
33public:
34   Dragon(const std::string& name) : Creature(name, "Dragon", 50) { }
35
36   virtual std::string WarCry() const { return "Whoosh!"; }
37};
38
39class Dungeon {
40public:
41   Dungeon() { }
42   ~Dungeon();
43   void AddCreature(Creature* m) { inhabitants_.push_back(m); }
44
45private:
46   std::list<Creature*> inhabitants_;
47};
48
49Dungeon::~Dungeon() {
50   for(auto it : inhabitants_) {
51       std::cout << "Deleting " << it->GetName() << std::endl;
52       delete it;
53   }
54}
55
56
57int main() {
58   Dungeon dung;
59
60   Troll* tr = new Troll("Peikko");
61   dung.AddCreature(tr);
62   dung.AddCreature(new Dragon("Rhaegal"));
63}
```

The constructor of *Dungeon* class does not do anything special, and is defined inline as an empty function. Line 4 declares a **destructor** for the *Dungeon* class. The function definition is provided later, outside the class, starting on line 11. A destructor is almost like any other function, except that it does not have return value or any parameters, and it is called as the last action when the object going to be released. Therefore, it is the proper place to release any memory allocations that the class is responsible of, or release any other resources that have been used by the class. Destructor is called automatically, when an object is going to disappear, and the program should not call it explicitly. Destructor should not do anything else, but clean up the resources used by the object, but here we print out something, just to demonstrate that the destructor is indeed called at the end of the program.

Note

In C++ there are smarter and easier ways of handling pointers than using raw pointers, as done here. Those will be discussed later, but in these examples the use of raw pointers serves as to demonstrate the idea of destructor, copy constructor, and copy assignment.

In the *main* function we create a new Troll object, by dynamically allocating it using the **new** operator. Note that the result will be a **pointer** to a new *Troll* object. We can also use the *new* operator directly as part of another expression, as done for the *Dragon* object. Then we reach the end of the main function. At this point the destructor of *dung* object is called automatically, and following the destructor implementation, each item in the *inhabitants_* vector is deleted (because they were dynamically allocated), after printing its name.

Because *Creature* is a virtual class, we need to also create a virtual destructor for *Creature*, so that when a Creature pointer is deleted the destructors of the actual types are called. When lacking an explicit destructor, compiler will normally perform standard actions for object cleanup, but for virtual classes we need to specify the destructor ourselves. Trolls and Dragons may need specific destructors for cleaning up the class-specific things, although in this case there isn't anything that needs special action. Nevertheless, we will need the following addition to *Creature* class:

```
virtual ~Creature() { }
```

This shows that a destructor can (and should) be virtual as any other functions involving virtual classes, and, in fact, it must be virtual, if there are any other virtual functions in a class.

If you need to know the pointer of the currently handled object from within the class, you can use **this**. It is an automatically defined pointer that points to "myself", the object currently executing a function. However this not often needed, as discussed in this Stack overflow article.

5.1 Programming task¶

Points 25 / 25 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).

Birds

Objective: Practice abstract classes, inheritance and little bit of operator overloading.

You will need to implement different kinds of birds that all can fly (except *const* birds). Birds also speak, but in different languages specific to the bird type. **Bird** (in *bird.hpp*) is the abstract base class for birds. You will need to implement **Duck** and **Owl** (and define them in *bird.hpp*). To save some effort, you implement the needed functions inline in the bird-specific header files together with the the class definitions.

The bird language is simple: duck says `QUACK` and owl says `WHUU`. The speak function should output the name of the bird, colon, space, followed by one of the above and ended with a newline, e.g.:

```
Donald: QUACK\n
```

In addition, you will need to implement class **Aviary**, which stores the birds, in the *aviary.hpp* file and its member functions in the *aviary.cpp* file.

Note

- The copy constructor is a constructor that takes in a const reference to the same type as the class e.g. for Aviary: `Aviary(const Aviary&)`. The copy assignment operator is the `=` operator that takes in a reference to a const object of the same type and returns a reference to itself i.e. `*this`, e.g. for Aviary: `Aviary& operator=(const Aviary&)`. Copying can be disallowed, for example, by making these two private, then copies cannot be made from outside the class itself.
- You may ignore throwing `std::out_of_range` in the `[]` operators, unlike the .hpp file says. Alternatively, if you wish to do it, use `throw <exception object>` to throw an exception.

Hint

You will find more instructions in the header files.

aviary.cpp
Choose File No file chosen

aviary.hpp
Choose File No file chosen

duck.hpp
Choose File No file chosen

owl.hpp
Choose File No file chosen

Submit