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

6 More about dynamic memory¶

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

Contents

C++ Primer Chapter 13 (Copy control)

Handling dynamic memory correctly is surprisingly difficult, especially with C++. When an object is copied, assigned or moved around one must make sure that there are no **memory leaks**, i.e., pieces of memory that is unreleased after use, therefore unnecessarily using the computer memory capacity. In a larger program this can be difficult and memory leaks can quickly cause the memory run out and are very hard to track down. There can also be other kinds of resource leaks, for example regarding open files.

On the other hand resources must not be released prematurely, when there might still be references to an object from elsewhere in the program.

Particular cases of concern are when an object is copied or assigned. C++ synthesizes default implementations for these operations, but especially when dynamic resources are involved, special attention to these operations need to be paid, and usually own implementation needs to be provided.

6.1 Copy constructor and copy assignment¶

Copy constructor is a variant of constructor that is called when a new object is initialized as a direct copy of some other object, for example through assignment. In simple classes the compiler can implicitly construct such object just by doing a direct memory copy, but for example, when class is using dynamic resources (files, dynamically allocated memory, etc.), we will need to implement our own copy constructor that allocates the resources properly also for the new copied class. This also applies to our *Dungeon* class in earlier examples: when we copy a dungeon, we need to allocate new instances of the Creatures as well. Just copying the pointers is not enough: if old *Dungeon* is destroyed, the pointers would become invalid.

Likewise, for classes that manage dynamic resources, we need to implement our own **assignment operator** (*operator=*). The assignment operation needs to **first properly release the resources owned by the to-be-overwritten object before the assignment**, and then properly **allocate the new resources needed for the copied class**, similarly to copy constructor.

Below is a modified Dungeon with copy constructor and copy assignment:

```
 1#include <string>
 2#include <vector>
 3#include <iostream>
 4#include <list>
 5
 6class Creature {
 7public:
 8    Creature(const std::string& n, const std::string& t, int h)
 9        : name_(n), type_(t), hitpoints_(h) { }
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    virtual Creature* Clone() const = 0;
17
18private:
19    std::string name_;
20    const std::string type_;
21    int hitpoints_;
22};
23
24
25class Troll : public Creature {
26public:
27    Troll(const std::string& n) : Creature(n, "Troll", 10) { }
28
29    virtual std::string WarCry() const { return "ugazaga!"; }
30    virtual Creature* Clone() const { return new Troll(*this); }
31};
32
33
34class Dragon : public Creature {
35public:
36    Dragon(const std::string& n) : Creature(n, "Dragon", 50) { }
37
38    virtual std::string WarCry() const { return "whoosh!"; }
39    virtual Creature* Clone() const { return new Dragon(*this); }
40};
41
42
43class Dungeon {
44public:
45    Dungeon() { }
46
47    // Copy constructor. Note the use of reference (think about why)
48    Dungeon(const Dungeon& d);
49
50    ~Dungeon();
51
52    void Add(Creature* m) { inhabitants_.push_back(m); }
53
54    // Copy assignment. Note the use of reference in return value and argument.
55    Dungeon& operator= (const Dungeon& d);
56
57private:
58    std::list<Creature*> inhabitants_;
59};
60
61/** cpp file starts here **/
62
63// Copy constructor
64Dungeon::Dungeon(const Dungeon& d) {
65    std::cout << "Copy constructor called" << std::endl;
66
67    // Copy the contents of Dungeon <d> to a newly created Dungeon
68    for (auto i : d.inhabitants_) {
69        Creature* c = i->Clone(); // Creates a new creature by copying old
70        inhabitants_.push_back(c);
71    }
72}
73
74// Destructor
75Dungeon::~Dungeon() {
76    // Delete the creatures in the Dungeon
77    for(auto it : inhabitants_) {
78        std::cout << "Deleting " << it->GetName() << std::endl;
79        delete it;
80    }
81}
82
83// Copy assignment
84Dungeon& Dungeon::operator= (const Dungeon& d) {
85    std::cout << "Copy assignment called" << std::endl;
86
87    // 1) Delete the existing creatures in Dungeon
88    for (auto i : inhabitants_) {
89        std::cout << "Deleting " << i->GetName() << std::endl;
90        delete i;
91    }
92    inhabitants_.clear();  // empty the container
93
94    // Copy the new contents from Dungeon <d>
95    for (auto i : d.inhabitants_) {
96        Creature* c = i->Clone();
97        inhabitants_.push_back(c);
98    }
99
100    // Results in reference to current class (see return value type)
101    return *this;
102}
103
104int main() {
105    Dungeon dung;
106
107    Troll* tr = new Troll("Peikko");
108    dung.Add(tr);
109    dung.Add(new Dragon("Rhaegal"));
110
111    // Causes copy constructor to be called
112    std::cout << std::endl << "Copy constructor..." << std::endl;
113    Dungeon other = dung;
114
115    Dungeon third;  // Creates an empty Dungeon
116    third.Add(new Dragon("Puff"));
117
118    // Causes copy assignment to be called. Puff will die.
119    std::cout << std::endl << "Copy assignment..." << std::endl;
120    third = other;
121
122    std::cout << std::endl << "Program ending..." << std::endl;
123}
```

The copy constructor definition starts on line 22. This example presents an problem with abstract classes: because the list elements are pointers of *Creature* type, we cannot use *new* directly, because we would need to specify the actual type with it, which we do not know when writing the program. We solve this by adding a new virtual *Clone()* function to all *Creature* types, that creates a dynamically allocated copy of the object. For example, the *Clone()* implementation for *Dragon* looks like this:

```
virtual Creature* Clone() { return new Dragon(*this); }
```

In other words, it applies copy constructor for Dragon class. '**this**' is a C++ keyword that returns pointer to the current object ("my own address"), *this* returns a pointer, so we must use the dereference operator (*), to match copy constructor's reference type. Because the *Dragon* class is simple, and does not use dynamic resources, we can rely on its default synthesized copy constructor, and will not implement our own copy constructor.

The copy assignment definition starts from line 42. It first erases the existing list of inhabitants and the allocated memory, and then copies the new inhabitants from Dungeon *d* similarly to copy constructor.

It's worth observing that we can access private members of other objects, if they are of the same type.

The main function relies on copy constructor to create the *other* dungeon. It uses the copy assignment to create the *third* dungeon.

6.2 Preventing copy and assignment¶

Sometimes we might want to tell the compiler that a particular class cannot be copied or assigned, for example, if the program logic does not have any sensible way to implement these operations. In such cases we can tell the compiler to prevent these operations at the function declaration phase. Following code denies copy constructor and assignment.

```
class DontCopy {
    DontCopy() = default;  // explicitly tell to use default
    DontCopy(const DontCopy&) = delete; // disallow copying
    DontCopy& operator=(const DontCopy&) = delete; // disallow assign
    ~DontCopy() = default; // tell to use default
}
```

As the above example shows we can explicitly tell the compiler that certain functions can use synthesized (default) implementation. This can be done either at declaration (typically in header file), in which case the implementation will be inline function. Alternatively this could be done in the function definition (typically in the source file), in which case it will not be an inline function, but an actual compiled function in the binary.

Specifying default implementation explicitly also tells the other readers of the code that we haven't just accidentally omitted these implementations.

Point of interest iconRule-of-three
previous | next

"Rule of Three" was introduced in the 1990s to guide C++ programmers in handling classes with dynamic resources. It states that **whenever your object needs an explicit destructor** (for example, because of dynamically allocated resources), it also needs **an explicit copy constructor** and **an explicit copy assignment** to work properly.

Point of interest iconRule-of-five
previous | next

In C++11 and later there are additional **move semantics** for objects:

1. There are **move constructors** for moving an object's data from a reference. The move constructor should ensure that the original object does not have ownership for any of the dynamic resources in the object.
2. There is also **move assignment** for similar assignment operation. The move operations are used with temporary values, and in some cases they can improve the efficiency of assignments. With the move semantics, the rule of three becomes the "Rule of Five".

We will skip the details of move semantics here, but if you are interested, for example this tutorial explains move semantics, along with use of move constructor and move assignment.

6.3 Programming task¶

Points 25 / 25 My submissions 12 Deadline Friday, 8 October 2021, 19:59 To be submitted alone

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

Dragons

Objective: Practice abstract classes and inheritance. This time you will also need to implement the base class interface, and deal with the Rule of Three (RO3).

In this task you will need to implement various kinds of dragons, **Dragon** is an abstract base class for different kinds of dragons: **FantasyDragon** and **MagicDragon**. The classes are defined in the *.hpp* files, and the implementations should be in the *.cpp* files.

In addition, you will need to implement **DragonCave** (in *dragon_cave.hpp/cpp*) that stores the different kinds of dragons. DragonCave must not allow copying.

The header files give more detailed instructions about the functions you will need to implement.

dragon_cave.cpp
Choose File No file chosen

dragon_cave.hpp
Choose File No file chosen

dragon.cpp
Choose File No file chosen

dragon.hpp
Choose File No file chosen

fantasy_dragon.cpp
Choose File No file chosen

fantasy_dragon.hpp
Choose File No file chosen

magic_dragon.cpp
Choose File No file chosen

magic_dragon.hpp
Choose File No file chosen

Submit