Software engineering

v2024

Lab 3:

Programming with Modern C++ – Part I

Contents

FESB in cooperation with ZORAJA Consulting d.o.o.

Exercise 1: Semantics

Beside the default, ordinary and conversion constructors, there are two more types of constructors. Those are:

Copy constructor (CC) and

Move constructor (MC).

By default, the compiler will generate an overridable CC for classes if we don't provide our own implementation. But it will also generate an implementation of a *Copy assignment operator* (CAO). While CC creates objects by making copies of its argument's class fields, CAO copies class fields from the object on the right side of the operator.

CC → T(const T&)

CAO → T& operator=(const T&)

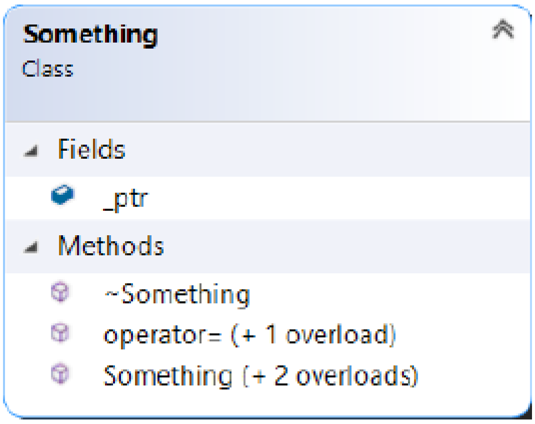
Beside copy semantics, C++11 introduced the concept of *move semantics*, which enables avoidance of unnecessary copies when working with temporary objects.

If the user doesn't provide a destructor and MC/CC/CAO/MAO, the compiler will implicitly declare a MC.

MC → T(T&&)

MAO→ T& operator=(T&&)

The UML in the figure below represents the class to be created.



Create a Project

1. Create an Empty Project named Semantics.
2. Add file Program.cpp to the project

Add a Header File to the Project

1. Add a file Something.h to the project.
2. Protect the file from multiple inclusions.
3. Add a namespace abc.
4. Inside the namespace add a class named Something with a pointer to int named _ptr as a member variable. Assign nullptr to it.
5. Declare a constructor with a parameter type int and set its default value to -1.
6. Declare a copy constructor.
7. Declare a move constructor (MC and MAO should not throw, so use noexcept).
8. Declare a copy assignment operator.
9. Declare a move assignment operator.
10. Declare a virtual destructor.

Add a File to the Project

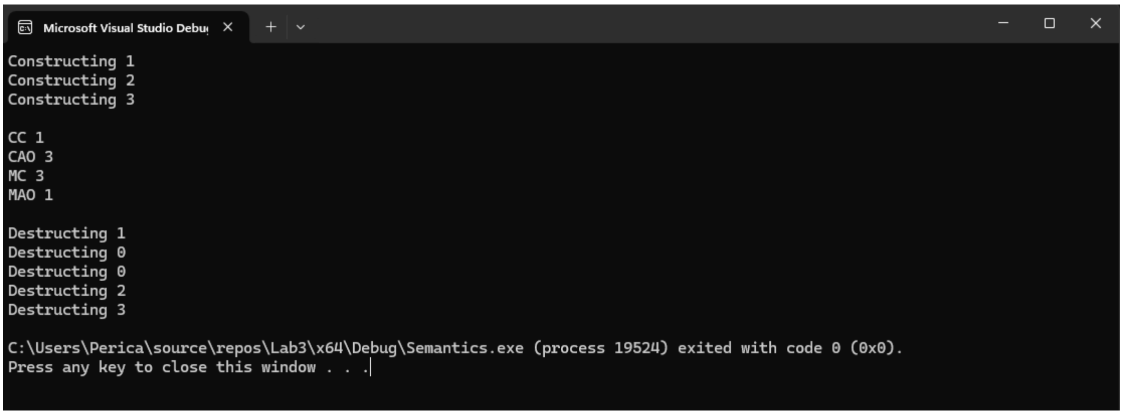
1. Add a file Something.cpp to the project.
2. Include iostream and Something.h header files.
3. Use the using directive for namespaces abc and std.
4. Implement the constructor so that you initialize _ptr and output the value of its pointee.
5. Implement a copy constructor.
6. Implement a move constructor. Don't forget to set the _ptr of the argument to nullptr to properly complete the theft!
7. Implement the CAO. Perform necessary checks to avoid self-assignment.
8. Implement the MAO. Perform the necessary checks to avoid self-assignment.
9. Implement the destructor so that it outputs the value _ptr points to and releases the resources.

Test Semantics

1. Inside Program.cpp include iostream and Something.h header files.
2. Use the using directive for namespaces abc and std.
3. In the main function add three local objects of type Something and initialize them with values 1, 2 and 3, respectively.
4. Add a local object d of type Something and initialize it with a.
5. Assign c to a.
6. Add a local object e of type Something and initialize it by casting c to an r-value using std::move.

7. Cast d to an r-value and assign it to e.

Run the App



```
Constructing 1
Constructing 2
Constructing 3

CC 1
CAO 3
MC 3
MAO 1

Destructing 1
Destructing 0
Destructing 0
Destructing 2
Destructing 3

C:\Users\Perica\source\repos\Lab3\x64\Debug\Semantics.exe (process 19524) exited with code 0 (0x0).
Press any key to close this window . . .
```

Exercise 2: Memory

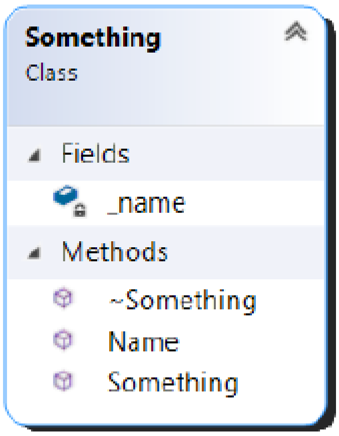
Memory, file handles, thread handles, etc. are examples of *resources*. Resources get acquired but need to be released. Failing to do so results in *leaks*.

Standard library provides *smart pointers* to help manage object on free store. They are defined in the std namespace, in <memory> header file. Using smart pointers is an application of the ***RAII*** technique where the main goal is to give ownership of heap-allocated resources to a stack-allocated resource, which becomes responsible for the 'cleanup'.

Types of smart pointers:

1. unique_ptr → used to represent unique ownership (allows only one owner of the underlying pointer). Only move semantics supported.
2. shared_ptr → used to represent shared ownership. Uses reference counting. Not deleted until all owners are out of scope.
3. weak_ptr → used with shared_ptr. Does not participate in reference counting.

The UML in the figure below represents the class to be created:



Create a Project

1. Create an Empty Project named Memory.
2. Add file Program.cpp to the project.

Add a File to the Project

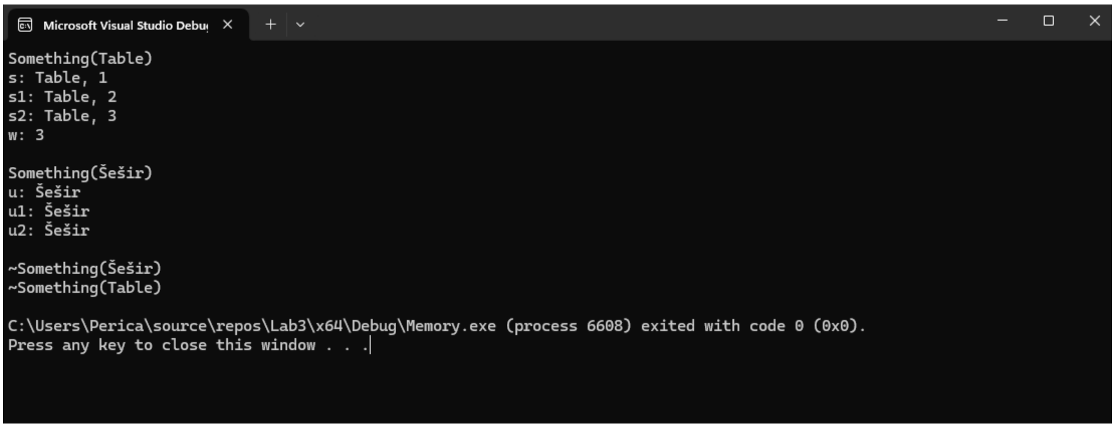
1. Add file Something.h to the project.

2. Protect the file from multiple inclusions.
3. Add a namespace abc.
4. Inside the namespace add a class named `Something` with a private member variable `_name` of type `wstring`.
5. Implement a constructor with a parameter of type `wstring`.
6. Implement a constant function `Name` which returns the value of `_name` variable. Use `auto` as return type.
7. Implement a virtual destructor.

Test Smart Pointers

1. In the Program.cpp include iostream, string and memory header files.
2. Include Helpers.h (which can be found in course materials section) and Something.h.
3. Use the using directive for namespaces std and abc.
4. In the main function enable UTF8 on the wcout stream with the SetUTF8 helper function.
5. Use make_shared function to create a shared pointer to Something and initialize it with a wstring value. Assign the created pointer to variable s. Use auto for type deduction.
6. Call the functions Name and use_count on s and output their values.
7. Add another auto variable named s1 and use s to initialize it.
8. Output the return values of functions Name and use_count for s1.
9. Add another auto variable named s2 and use s1 to initialize it.
10. Output the return values of functions Name and use_count for s2.
11. Add a weak pointer w to Something and initialize it with s2.
    - What would happen if you were to call function Name on w?
12. Call the function use_count on w and output the result.
13. Create a unique pointer to Something by using the make_unique function. Assign the result to variable u.
14. Use the dereferencing operator on u and call the function Name. Output the result.
15. Cast u to an r-value and use it to initialize u1.
    - Can you use u to initialize u1 without using std::move?
16. Call the Name function on u1 and output the result.
17. Cast u1 to an r-value and assign it to u2.
18. Call the function Name on u2 and output the result.
    - Can you call function Name on u1?

Run the App



```
Something(Table)
s: Table, 1
s1: Table, 2
s2: Table, 3
w: 3

Something(Šešir)
u: Šešir
u1: Šešir
u2: Šešir

~Something(Šešir)
~Something(Table)

C:\Users\Perica\source\repos\Lab3\x64\Debug\Memory.exe (process 6608) exited with code 0 (0x0).
Press any key to close this window . . .
```