Special Class Members

Constructors, Destructors, Copy Assignment

SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

Table of Contents

Have a Question?

Software University

sli.do

#cpp-oop

Special Class Members

Special Class Members

- Members called by C++ in special cases
  - **Default Constructor** – allocating objects & arrays
  - **Destructor** – when lifetime ends (e.g. due to scope or delete)
  - **Copy Constructor** – passing non-reference parameters/returning values
  - **Copy-assignment** – when **`operator=`** is used
  - **Move Constructor** – for move semantics

Default Constructor

Default Constructor Callers

- Automatic local/global non-primitive objects
- Arrays with default values
- Fields missing from the initializer list
  - Called in declaration order
  - Before the owner's constructor body

```
class Lecturer {
 double rating; string name;
public: Lecturer(string name)
  // rating() default ctor call
  : name(name) {}
};
```

```
string s; // default ctor call
Lecturer steve; // default ctor call
Lecturer cpp[2]{ Lecturer("GG") }; // default ctor for cpp[1]
```

Auto-gen Default Constructor

- Initializes each **object field** – calls default ctors in initializer list
- Auto-generated if no constructor declared explicitly
  - All fields have a default constructor

```
class Lecturer {
  double rating;
  string name;
};
```

```
class Lecturer {
  double rating;
  string name;
public:
  Lecturer() : name() // set to ""
  // NOTE: rating not set
  {}
};
```

Default Constructor

LIVE DEMO

Copy Constructor & Assignment

Copy Constructor and Copy Assignment

- `ClassName(const ClassName& other)`
  - `return` statements and non-reference parameters
- `ClassName& operator=(const ClassName& other)`
  - Assigning a value to an object with `=`
- Copy-elision: compilers optimize to avoid copies
  - Inlining functions & merging initialization and assignment
  - Can be disabled (e.g. **`-fno-elide-constructors`** in g++/gcc)

Auto-gen Copy Constructor/Assignment

- Copy-construct/assign each field with matching from parameter
- Auto-generated if no move constructor/assignment
  - Each field supports copy-construction/assignment

```
Lecturer(const Lecturer& other) : rating(other.rating), name(other.name) {}
...
Lecturer& operator=(const Lecturer& other) {
  this->rating = other.rating; this->name = other.name;
  return *this;
}
...
```

Copy Constructor & Assignment

LIVE DEMO

Quick Quiz

- What will this code do?

```
Array arr(10);
arr[0] = 42; arr = arr;
cout << arr[0] <<endl;
```

a) Print **"42"**

b) Behavior is undefined

c) Cause a compilation

d) Cause a runtime error

```
class Array {
  ...
  Array& operator=(const Array& o) {
    int* copyData = new int[o.size];
    delete[] this->data;

    for (int i = 0; i < o.size; i++) {
      copyData[i] = o.data[i];
    }

    this->data = copyData;
    this->size = o.size;
  }
};
```

C++ PITFALL: MISSING SELF-ASSIGNMENT CHECK AND DELETE BEFORE COPY

Two issues here – no self-assignment check and value copying done after deletion.

Hence we read data that has been removed from memory (**`this == &other`**).

NOTE: if the copy was done before **`delete`**, the code would work correctly.

Destructor

Destructors

- `~ClassName() ...` – called at the end of an object lifetime
  - e.g. `delete` or automatic storage scope end
- Common usage: free used resources
  - e.g. `delete` memory allocated by `new`

```
class IntArray {
int* data; int size;
public:
  IntArray(int size) : data(new int[size]), size(size) {}

  ~IntArray() {
    delete[] this->data;
  }
```

Auto-gen Destructor

- "Destructs" each object field – i.e. calls each field's destructor
- Auto-generated if no destructor is declared
  - NOTE: inheritance can change this behavior

```
class NamedArray {
  int* data; int size;
  string name;
}
```

```
class NamedArray {
int* data;
int size;
string name;
public:
...
  ~NamedArray() {
    // NOTE: no call for primitives
    name.~basic_string();
  }
}
```

Destructor

LIVE DEMO

Default and Deleted Members

Explicit Auto-gen and default

- Getting default special members with NO auto-generation
  - E.g. class has constructor, no default constructor auto-generated
  - Hard way – write implementation matching auto-generated
  - Easy way (C++11) – use `= default` after member signature

```
Lecturer() : name() {}

Lecturer(const Lecturer& other) : rating(other.rating), name(other.name) {}
```

```
Lecturer() = default

Lecturer(const Lecturer& other) = default
```

Disabling Special Members with `delete`

Software University

- Sometimes auto-generated methods need to be disabled
  - E.g. `unique_ptr<T>` disables copying
  - Hard way – declare the members as private
  - Easy way – use `= delete` after member signature

```
class Array {
...
private:
Array(const Array& other) { ... }
...
};`
```

```
class Array {
...
Array(const Array& other) = delete;
...
};
```

Default and Deleted Members

LIVE DEMO

Quick Quiz

- What will this code do?

```
Lecturer a("Bill", 4.2);
Lecturer other(a);
cout << other.name << endl;
```

a) Print **"Bill"**

b) Print **""**

c) Print an undefined string

d) Cause a runtime error

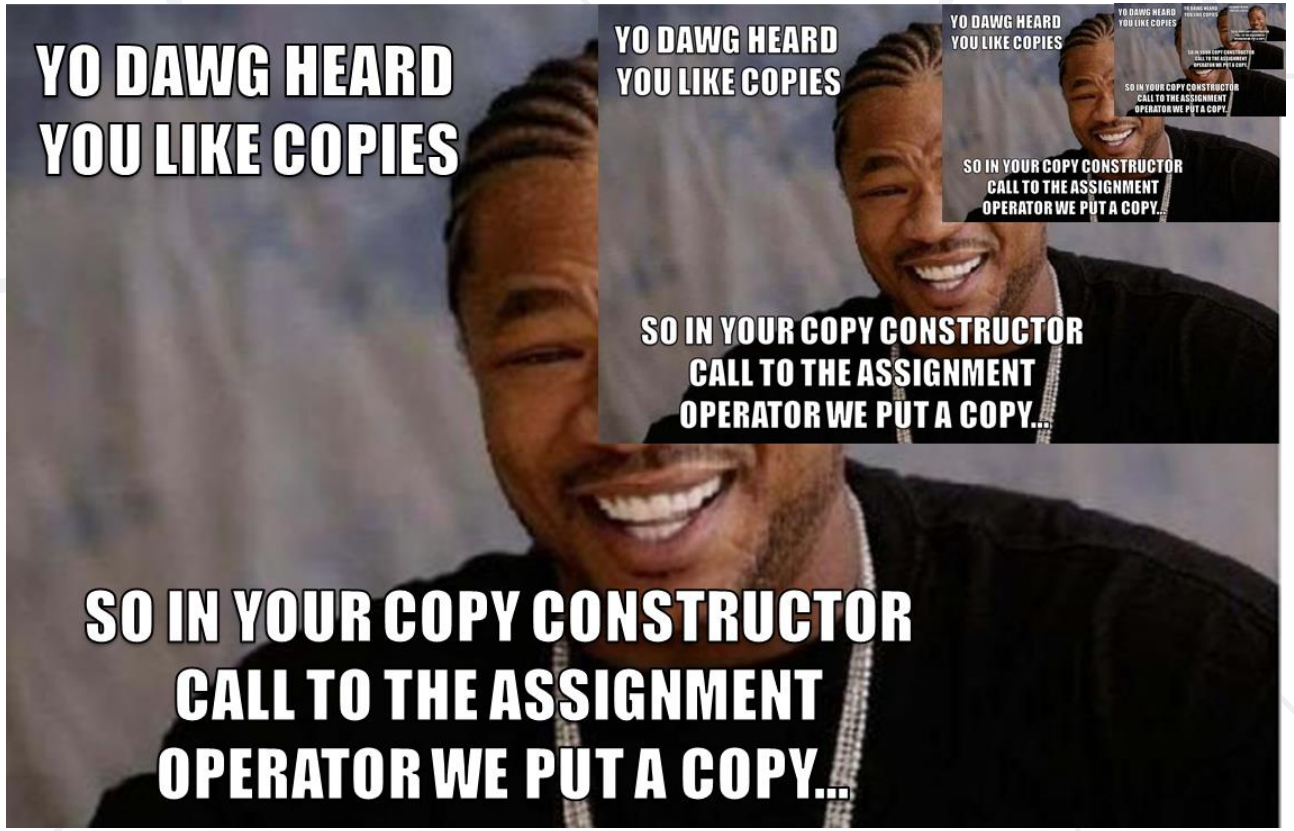
```
struct Lecturer {
  double rating; string name;
  Lecturer(string name, double rating)
  : name(name), rating(rating) {}

  Lecturer(const Lecturer& other) {
    *this = other;
  }
  Lecturer& operator=(Lecturer other) {
    this->name = other.name;
    this->rating = other.rating;
    return *this;
  }
};
```

C++ PITFALL: DOING A COPY IN A COPY CONSTRUCTOR

The **`operator=`** used by the copy constructor here accepts a copy.

compile this (e.g. Visual C++ 2017) That's an infinite indirect recursion – copy constructor calls itself to create the copy for the parameter of **`operator=`** it calls

Some compilers will refuse to



Move Constructor

Move constructor

- Moves the resources in the heap/stack (a.k.a. "stealing resources")
- Makes the pointer of the declared object point to the data of a temporary object
- Nulls out the pointer of the temporary objects
- Prevents unnecessarily copying data in the memory

```
Object_name(Object_name&& obj)
 : data{ obj.data }
{
 // Nulling out the pointer to the temporary data
  obj.data = nullptr;
}
```

Move Assignment Operator

Move Assignment Operator

- Is used for transferring a temporary object to an existing object
- Is a special member function and can be overloaded
- Is different than a move constructor
  - It is called on an existing object, while a move constructor is called on an object created by the operation
- The parameter is an rvalue reference (T&&) to type *T*, where *T* is the object that defines the move assignment operator

```
Object_name& operator =(Object_name&&){}
```

Move Assignment Operator

LIVE DEMO

std::move

std::move

Software University

- Is a helper function to force *move semantics* on values
- Is used to *indicate* that an object t may be "moved from"
- Allowing the efficient transfer of resources from t to another object
- Obtains an rvalue reference to its argument and converts it to an xvalue
- Produces an xvalue expression that identifies its argument

std::move on a unique_ptr

Software University

```
struct Foo {
        int id;
        Foo(int id) : id(id) { std::cout << "Foo " << id << '\n'; }
        ~Foo() { std::cout << "~Foo " << id << '\n'; }

};

int main(){

    std::unique_ptr<Foo> p1( std::make_unique<Foo>(1) );
    {
          std::cout << "Creating new Foo...\n";
          std::unique_ptr<Foo> p2( std::make_unique<Foo>(2) );
          // p1 = p2; // Error ! can't copy unique_ptr
          p1 = std::move(p2);
          std::cout << "About to leave inner block...\n";
          // Foo instance will continue to live,
          // despite p2 going out of scope
          }
        std::cout << "About to leave program...\n";
        }
```

std::move

LIVE DEMO

Meyer's Singleton Design Pattern

What is a Singleton?

- Singleton is a Design Pattern that enforces the creation of **only** a single object of a specific type
  - Imagine this scenario:

```
Class Application {
…
};

Application mainApplication; //we create a single object
//…

//Nothing is stopping us from creating an additional object of that type
Application anotherApplication;
```

What is a Singleton?

- Singletons are used when they control a unique resource or have unique control over some piece of code
- Keep in mind that this design pattern is both a blessing and a curse
- It is the sole reason for a "spaghetti code" if used improperly

Meyer's Singleton Design Pattern

Software University

- Exploits three important properties:
  - Static function objects are initialized when control flow hits the function for the first time
  - The lifetime of function static variables begins the first time the program flow encounters the declaration and ends at program termination
  - If control enters the declaration concurrently while the variable is being initialized, the concurrent execution shall wait for the completion of the initialization

```
File “Application.h”
#ifndef APPLICATION_H_
#define APPLICATION_H_

class Application {
public:
  /* This function creates an instance of singleton Application.
     It is lazy and thread safe. */
  static Application& getInstance() {
    static Application app;
    return app;
  }
  //Copy/Move constructor is disallowed.
  Application(const Application& other) = delete;
  Application(Application&& other) = delete;

  // Disallowing copy/move assignment operator:
  Application& operator= (const Application& other) = delete;
  Application& operator= (Application&& other) = delete;

  void foo() {}
```

```
private:
  // We can't independently instantiate this object.
  Application() { /* ... */ }

  // Also we can't independently destruct this object.
  ~Application() { /* ... */ }
};

#endif /* APPLICATION_H_ */

File main.cpp
#include "Application.h"
```

Meyer's Singleton Design Pattern

LIVE DEMO

Summary

- C++ calls Special Members in certain situations
  - Each can be auto-generated under some conditions
- Destructors free allocated resources
- Copy constructors/assignments copy object resources
- Move constructors/assignments
  - std::move
- Meyer's Singleton Design Pattern

Questions?

SoftUni

Software University

SoftUni Svetlina

SoftUni Creative

SoftUni Digital

SoftUni Foundation

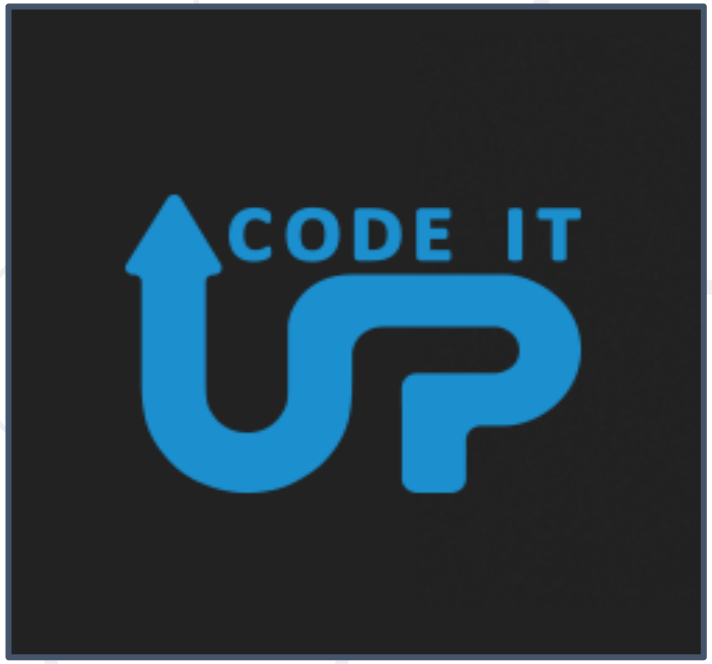
SoftUni Kids

SoftUni Diamond Partners

Software University

Educational Partners

Software University



VIRTUAL RACING SCHOOL

License

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – https://about.softuni.bg/
- © Software University – https://softuni.bg

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, about.softuni.bg
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg