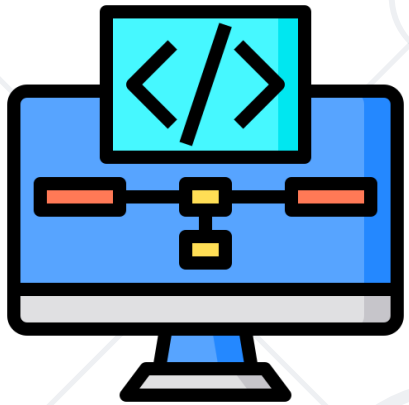
OOP Constructors

Constructors, Destructors, Copy Assignment



SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

Have a Question?

Software University

sli.do

#cpp-oop

Table of Contents

Special Class Members

Special Class Members

- Members called by C++ in special cases
  - **Default Constructor** – allocating objects
  - **Destructor** – deallocating objects (e.g. due to scope exit or delete)
  - **Copy Constructor** – creating objects from other objects of the same type
  - **Copy-assignment** – when `operator=` is used
  - **Move Constructor** – for move semantics
  - **Move-assignment** – when `operator=` is used

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
  {}
};
```

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

Software University

- 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;
}
...
```

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 – 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();
  }
}
```

Default and Deleted Members

Explicit Auto-gen and default

- Getting default special members with NO auto-generation
  - Class has a user-declared 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

- Sometimes auto-generated methods need to be disabled
  - `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;
...
};
```

Auto Generation of Special Members

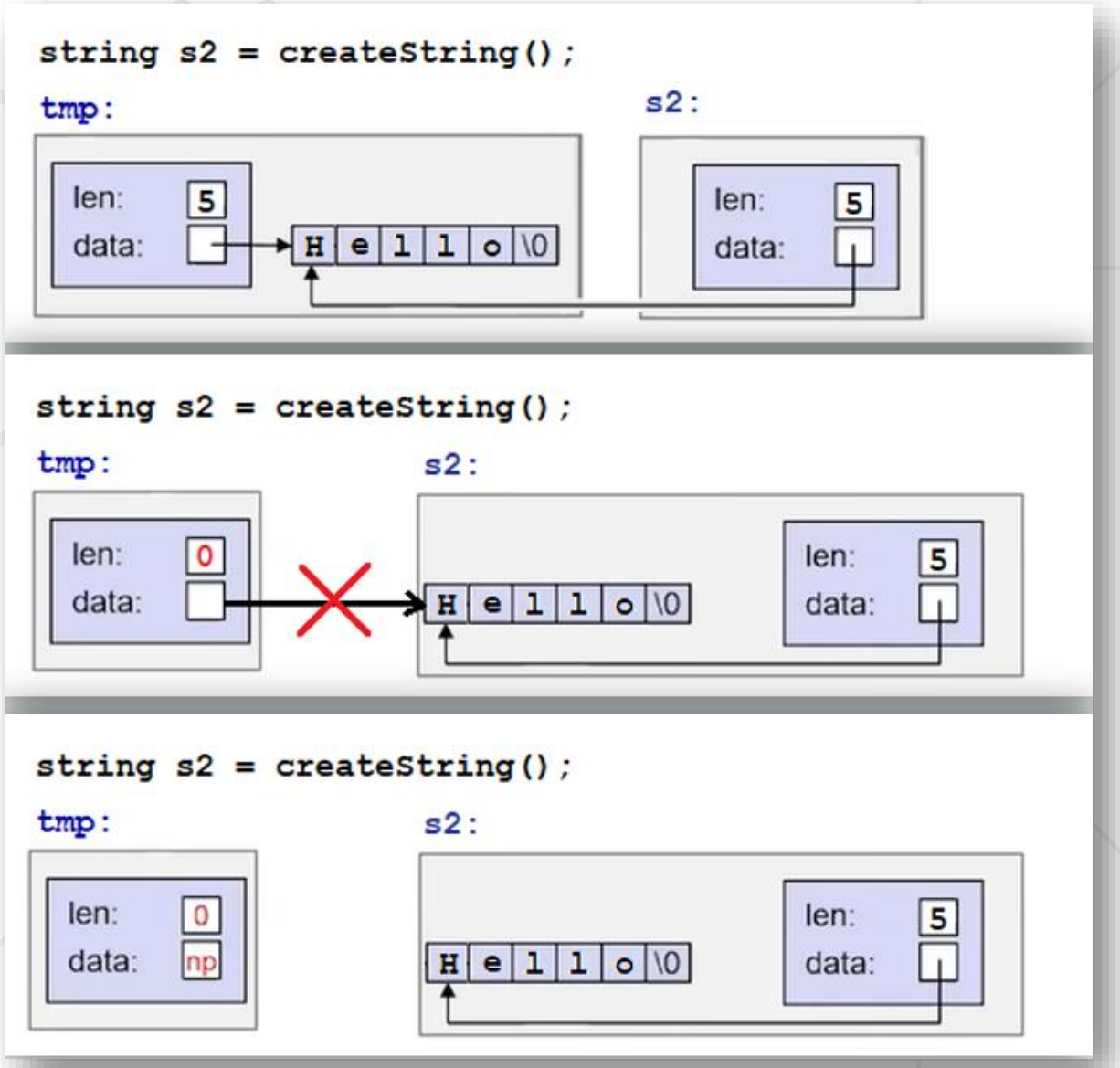
Software University

user declaration of \ forces	default constructor	copy constructor	copy assignment	move constructor	move assignment	destructor
nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
any constructor	undeclared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
copy constructor	undeclared	user declared	defaulted	undeclared (fallback enabled)	undeclared (fallback enabled)	defaulted
copy assignment	defaulted	defaulted	user declared	undeclared (fallback enabled)	undeclared (fallback enabled)	defaulted
move constructor	undeclared	deleted	deleted	user declared	undeclared (fallback disabled)	defaulted
move assignment	defaulted	deleted	deleted	undeclared (fallback disabled)	user declared	defaulted
destructor	defaulted	defaulted	defaulted	undeclared (fallback enabled)	undeclared (fallback enabled)	user declared

Move Constructor

Move Semantics

- The concept of moving is **transfer of content / ownership**, or "stealing" of the resources
- How?
  1. Make the pointer of the new object point to the data of the source object
  2. Null out the pointer of the source object (and other data of the object state)
- When? - when we no longer need the source object but want to use its data in the new object



Move Constructor

- Moves the resources **of a temporary object to the newly-constructed object**
  - **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 } // Make our pointer point to the source object data
{
 // Nulling out the pointer to the temporary data
  obj.data = nullptr;
}
```

Move Assignment Operator

Move Assignment Operator

- Is used for transferring the resources of a temporary object into 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 for creating a new object
- The parameter is an rvalue reference (T&&) to type *T*, where *T* is the object that defines the move assignment operator

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

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

- Is used to indicate that an object t may be "moved from"
- Doesn't move anything physically – **it only casts to rvalue reference**

- With **std::move** we tell the compiler **"I no longer need this value here, you can transfer its resources to another object"**
- Used on **named variables**

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";
          // Foo2 instance will continue to live,
          // despite p2 going out of scope
    }
        std::cout << "About to leave program...\n

}
```

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**

Example

Software University

```
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() {}
```

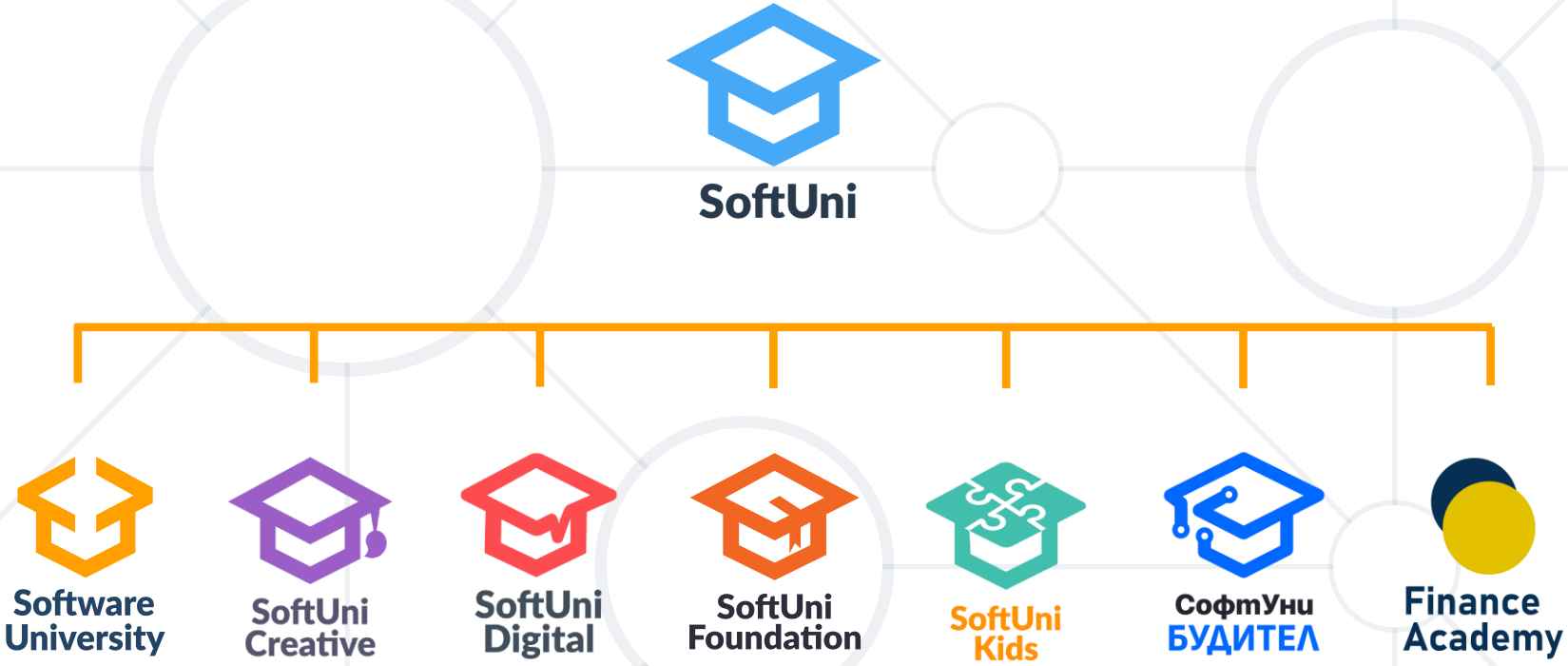
Example

```
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"
```

Summary

- 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 Diamond Partners

Software University

SmartIT

Coca-Cola HBC Bulgaria

SUPER HOSTING .BG

createX

Postbank

Решения за твоето утре

DRAFT KINGS

VIVACOM

AMBITIONED

INDEAVR

Serving the high achievers

SOFTWARE GROUP

PHAR VISION

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

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