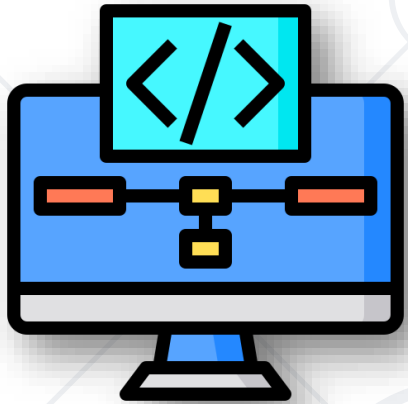


# OOP Constructors

Constructors, Destructors, Copy Assignment



SoftUni Team  
Technical Trainers



**SoftUni**



Software University

<https://softuni.bg>

sli.do

**#cpp-oop**

1. Default Constructor
2. Copy Construction and Copy Assignment
3. Meyer's Singleton Design Pattern





# **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





**Default Constructor**

- 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]
```

- 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**

- **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)

- 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;
    }
}
```

- Destroys 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

- 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

		forces					
		default constructor	copy constructor	copy assignment	move constructor	move assignment	destructor
user declaration of	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



# 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**

- 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"
```



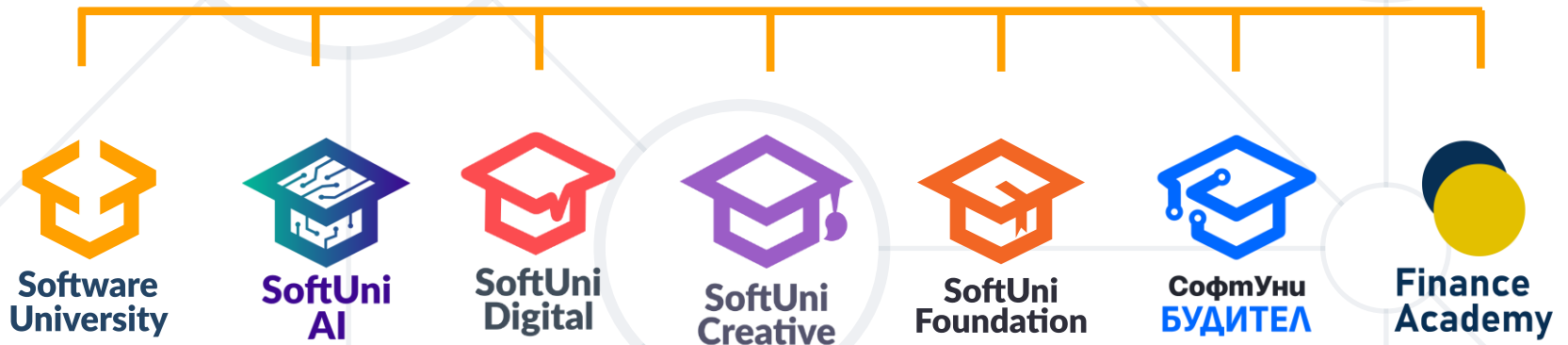
- Calls special members in certain situations
- Each can be **auto-generated** under some conditions
- **Destructors** free allocated resources
- Copy **constructors** / **assignments** copy object resources
- Meyer's Singleton Design Pattern



# Questions?



SoftUni



# SoftUni Diamond Partners



**SUPER  
HOSTING  
.BG**



**INDEAVR**  
Serving the high achievers



**THE CROWN IS YOURS**

**VIVACOM**

- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)



- 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>

