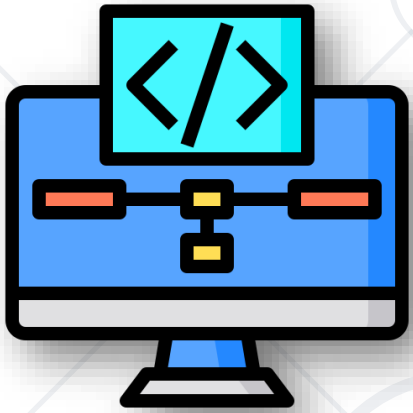


Rule of Three / Five / Zero



SoftUni Team
Technical Trainers



SoftUni



Software University

<https://softuni.bg>

sli.do

#cpp-oop

Table of Contents

1. Rule of Three / Five
 - Copy and Swap Idiom
2. Rule of Zero

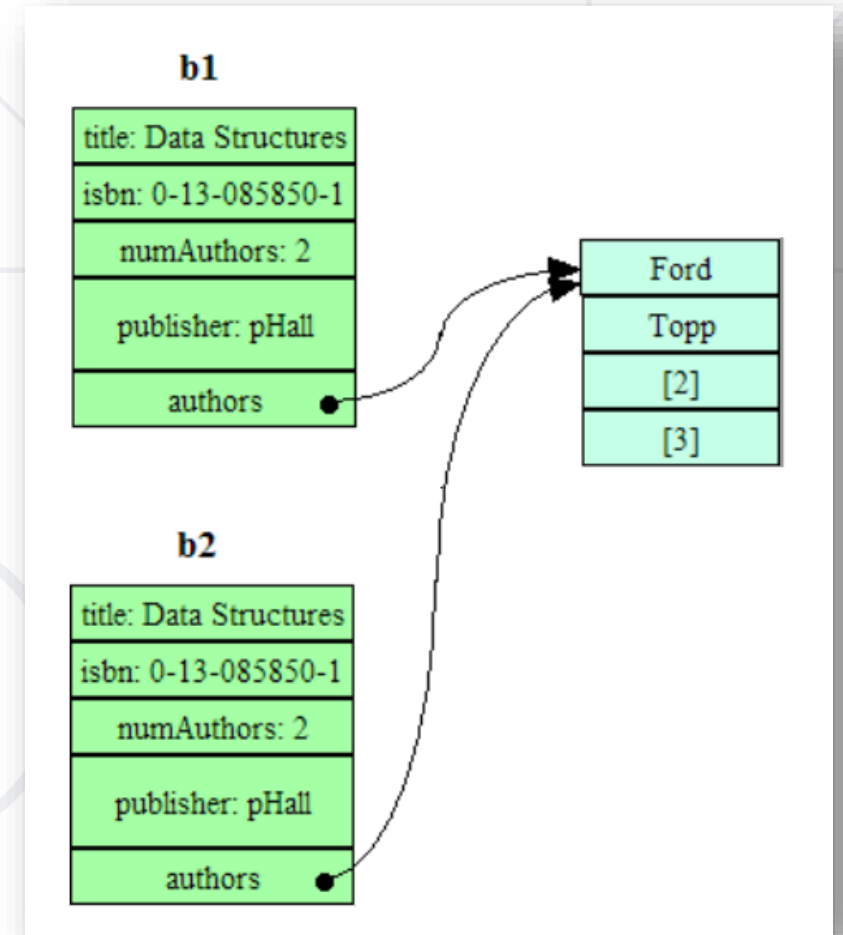




Rule of Three / The Big Three

Ownership

- The concept of **ownership**: the object responsible for cleaning up a resource owns that resource
- **Copy-by-value semantics**: Known as "**shallow copy**," where a new object is created, but the values of the resources are not duplicated and are shared.



- Constructor increases a static value, destructor decreases

```
void example()
{
    Lecturer a("Dandelion", 1),
    b("Geralt", 1.3),
    c("Yen", 4.2);


    vector<Lecturer> lecturers;
    lecturers.push_back(a);
    lecturers.push_back(b);
    lecturers.push_back(c);
}
```

```
class Lecturer
{
    static int Total;
    ...
public:
    Lecturer(...) ... { Total++; }
    ~Lecturer() { Total--; }
    ...
};

int Lecturer::Total= 0;
```

```
example(); cout << Lecturer::getTotal();
```

The Rule of Three

- 
- Allows programmers to define **copy constructors** and **copy assignment** operators
 - If a class **needs one of the following**:
 - **Copy Constructor**
 - **Copy Assignment operator=**
 - **Destructor**
 - Then it probably needs ALL of them:

```
IntArray(const IntArray& other) { ... }
```

```
IntArray& operator=(const IntArray& other) { ... }
```

```
~IntArray() { ... }
```

- General guidelines:
 - **new** can cause errors – make sure object state valid in that case
 - Free any **current object resources**
- Patterns:
 - Copy other object data into local variable, then set **this** fields
 - Extract a function to reuse code for copy construct and assign or use the **copy-and-swap idiom**



Copy-and-Swap Idiom

- Copy and swap idiom is used for simpler handling of dynamic resource (preventing **new** / **delete** / **delete[]** errors)
- Image a simple SmartArray implementation:

```
template <typename T>
class SmartArray
{
    // ...
private:
    size_t _size;
    T *_data;
};
```

- The constructor / destructor are trivial:

```
SmartArray(size_t size) : _size(size), _data(_size ? new T[_size] { } : nullptr)
{
}
}
```

```
~SmartArray()
{
    if (_data)
    {
        delete[] _data;
    }
}
```

- The copy constructor is also trivial:

```
SmartArray(const SmartArray &other)
: _size(other._size), _data(_size ? new T[_size] { } : nullptr)
{
    std::copy(other._data, other._data + _size, _data);
}
```

Copy-and-Swap Idiom

- Here comes the interesting part:
 - We provide a friend public method swap that can efficiently swap two objects:

```
friend void swap(SmartArray &first, SmartArray &second)
{
    std::swap(first._size, second._size);
    std::swap(first._data, second._data);
}
```

- Then the copy assignment operator is actually making a copy of the object:

```
SmartArray& operator=(SmartArray other)
{
    swap(*this, other);
    return *this;
}
```



Copy-and-Swap Idiom

- This way a **new temporary object** is created, it is being populated by the copy constructor
- Then swapped with **the real object**
- The destructor of the previous object (previous this) takes care of deleting **dynamic allocated resources** (if any)





The Rule of Five

- If a class needs one of the following:

- **Copy Constructor**
- **Copy Assignment operator=**
- **Destructor**
- **Move Constructor**
- **Move Assignment operator=**

Then it probably needs
ALL of them:

```
IntArray(const IntArray& other) { ... }  
IntArray& operator=(const IntArray& other) { ... }  
IntArray(IntArray&& other) { ... }  
IntArray& operator=(IntArray&& other) { ... }  
~IntArray() { ... }
```


- Rule of Five = Rule of Three
- Move construct / assign
 - Custom implementation could also be provided for **Move Constructor** and **Move Assignment Operator**

```
SmartArray(SmartArray &&other) : _data(other._data), _size(other._size)
{
    other._size = 0;
    other._data = nullptr;
}
```

Rule of Five = Rule of Three

```
SmartArray& operator=(SmartArray &&other)
{
    if (this != &other)
    {
        _data = other._data;
        _size = other._size;

        other._size = 0;
        other._data = nullptr;
    }
    return *this;
}
```

- In the implementation of **copy and swap for the Rule of Three** the copy assignment operator was implemented as such:

```
SmartArray& operator=(SmartArray other)
{
    swap(*this, other);
    return *this;
}
```

Copy and Swap Idiom for the Rule of Five

- If **move assignment operator** is added:

```
SmartArray& operator=(SmartArray &&other)
{
    //...
}
```

- The compiler will be ambiguous, which assignment operator you want to call
- This would mean that we have to **keep the current implementation** of the copy assignment operator, which now calls only assignment operator



The Rule of Four ... and a half

The Rule of Four and a Half

- In order to enable the **Copy and Swap Idiom** for the **move** methods as well
- Given that the assignment operator takes its parameter **by non-const value**
- Only the **move constructor** should be implemented

```
// initialize using the default constructor first  
SmartArray(SmartArray &&other) : SmartArray(0)  
{  
    swap(*this, other);  
}
```

- If a class has one of **The Three / Five**, then:
 - It manages a **resource** (memory or something else)
 - It should manage a **single** resource
 - It should not do anything other than **manage the resource**
- Internal code deals **with constructors / destructors**
- Having such classes avoids implementing the **Rule of Three / Five ourselves**



Rule of Zero

Delegating Resource Management

Rule of Zero

- STL has **containers, smart pointers**, etc.
 - Wrap other resources with classes implementing Rule of 3 (or 5)
- All remaining classes use the above, so:
 - No need for **explicit destructor**
 - No need for **explicit copy-constructor**
 - No need for **explicit copy-assignment operator**
- If you can – **avoid resource management**



Rule of Zero for Array Class

- Avoid memory management – `shared_ptr<int> data;`
- Tell `shared_ptr<T>` to release using array `delete[]`:
 - Second parameter accepts code to execute for deletion
 - `data(..., default_delete<int[]>())`
 - or `data(..., [](int* p) { delete[] p; })`
- No destructors, No copy construction, No copy assignment
- Or just use a `vector<T>`

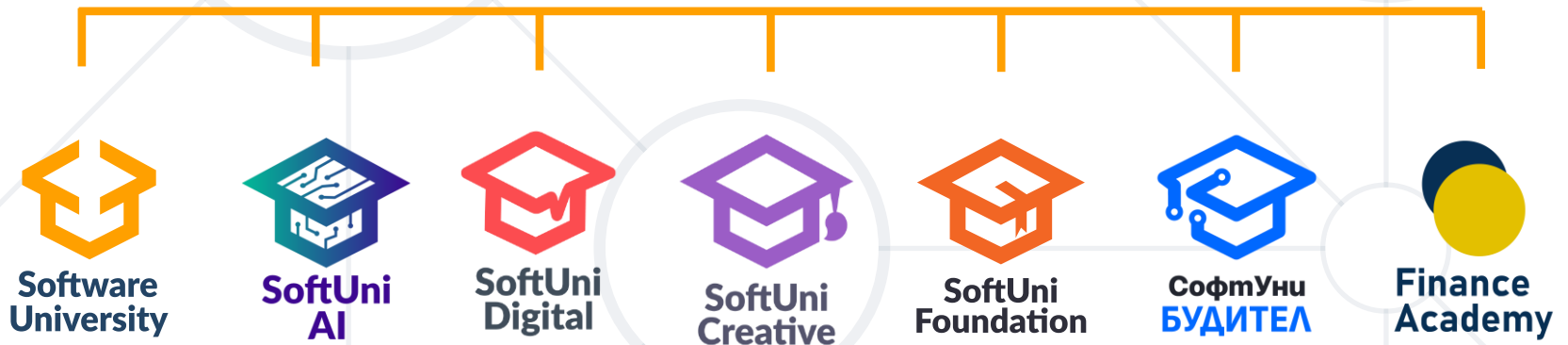
- **Rule of Three** – implement or disable copy members
- **Rule of Zero** – delegate resource management to other classes



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, about.softuni.bg
- Software University Foundation
 - softuni.foundation
- Software University @ Facebook
 - 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>

