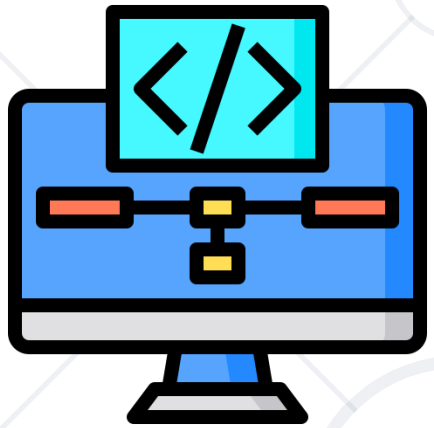
Rule of Three / Five / Zero



SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

Table of Contents

Have a Question?

Software University

sli.do

#cpp-oop

Resource Acquisition is Initialization

Associating Resources with Object Lifetime

Resource Acquisition is Initialization

- RAII – resource usage is tied to object lifetime
  - Objects acquire their resources on initialization
  - Objects release their resources on destruction
  - Effect: no resource leaks if no object leaks
- "Resources" – dynamic memory, streams, files, etc.
- Allocate in a constructor, deallocate in a destructor
  - Some cases might require allocation in methods
  - C++ guarantees destructor execution, even on error

RAII in the STL (1)

- C++ Streams are RAII
  - *E.g. file streams open file on construction & close on destruction*

```
void writeDataToFile(const std::string& data) {
  std::ifstream fileStream("log.txt", std::ios::in);
  //acquire resources

  istream << data << std::endl;

  istream.close(); //manually close the stream
} //destroy stream object. Even if the stream was not closed
  //the stream dtor would have closed it
```

RAII in the STL (2)

Software University

- All STL container classes are RAII
  - `vector<T>`, `list<T>`, `map<K, V>`, …
- `shared_ptr<T>` extends RAII to "multiple ownership"
  - Multiple objects own a resource
  - Release when a lifetime of last remaining owner ends

RAII

LIVE DEMO

Practice

Live Exercise in Class

Problem 1: SmartArray

- Implement a **SmartArray<T>** class that uses dynamic memory
  - Must be RAII, but STL containers/smart pointers are NOT allowed
  - Has size, has index access (with **operator[]**)
  - Can be resized
  - No support for copying/moving or assignment/move assignment
- *Bonus*: even more RAII
  - Don't use (directly) **new** methods
- *Bonus*: enable iteration (e.g. with range-based **for** loop)

Rule of Three / The Big Three

Destructors and Copies

- 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();
```

Copies Available -> Destructor Insufficient

- The example prints **-3** instead of **0** after all objects out of scope
- The problem is copy-construction/assignment
  - Counter not increased on copy
  - **3** locals -> **+3**
  - **3** copies into list -> **0** increments
  - Locals "destructed" -> **3-3=0**
  - List copies "destructed" -> **0-3=-3**

```
void example() {
  Lecturer a("Dandelion", 1)
  ...
  list<Lecturer> lecturers;
  all.push_back(a);
  ...
}
```

Copy that doesn't increment

Copies Available -> Destructor NOT Sufficient

LIVE DEMO

Destructor & Copies – Example (RAII issue)

Software University

- Let's use our `Array` from previous examples
  - Add destructor, auto-generated copy constructor/assignment
- Default copy constructor/assignment copies just the pointer
  - i.e. copy objects access and modify the same `data`
  - i.e. multiple `delete[]` at lifetime end on same data

```
void example() {
  Array arr(10);
  Array copyArr = arr;
  copyArr[3] = 42;
  cout << arr[3] // prints 42
}
```

arr does delete[] on data, then copyArr does delete[] on the same data

The Rule of Three

- 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() { ... }
```

Rule of Three – Copy Construct/Assign

- 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 & assign
  - ... or use the copy-and-swap idiom

Rule of Three

LIVE DEMO

Copy-and-Swap Idiom

Copy-and-Swap Idiom (1)

- 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;
} ;
```

Copy-and-Swap Idiom (2)

- The constructor / destructor are trivial:

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

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

Copy-and-Swap Idiom (3)

Software University

- 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 (4)

- Here comes the interesting part:
  - We provide a friend public method swap that can efficiently swap 2 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 (5)

- This way a new 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)

Practice

Live Exercise in Class

Problem 2: Rule of Three for SmartArray

- Implement the Rule of Three for the **`SmartArray<T>`** class
- Bonus: implement it using the copy-and-swap idiom

Quick Quiz TIME:

Software University

What the following program do?

a) print 42 and exit successfully

b) produce compilation error

c) print 42 and give a runtime error

d) undefined behaviour

```
int main() {
  SmartArray<int> arr(5);
  arr[2] = 42;

  SmartArray<int> arr2 = std::move(arr);
  std::cout << arr2[2] << std::endl;
  return 0;
}
```

```
template <typename T>
class SmartArray {
public:
  SmartArray(size_t size)
      : _size(size), _data(_size ? new
T[_size] { } : nullptr) {
  }

  SmartArray(SmartArray &&other)
      : _size(other._size),
_data(other._data) {
  }

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

C++ MOVE CONSTRUCTOR PITFALLS

When a custom implementation of move constructor is provided - the owned resources have to be actually **stolen**.
When the pointer is moved to the new objects - they have to be reset from the previous object.

Effectively assigning them a nullptr (or empty value).

The Rule of Five

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

- 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;
}
```

Copy and swap Idiom for the Rule of Five (1)

Software University

- 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 (2)

Software University

- If move assignment operator is added

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

- The compiler will be ambiguous, which assignment operator you want to call -> get a compilation error
- 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
- Only providing the **move constructor** should be implemented

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

The Rule of Four … and a half

LIVE DEMO

Single Responsibility

Software University

- 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
- So, need a resource? Wrap it in a class
  - Internal code deals with constructors / destructors / etc.
- Having such classes avoids the Rule of Three / Five

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

Software University

- 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 Zero

LIVE DEMO

Summary

- RAII – C++ pattern of initializing memory in the constructor
  - **Rule of Three** – implement or disable copy members
- **Rule of Zero** – delegate resource management to other classes

Questions?

SoftUni

Software University

SoftUni Svetlina

SoftUni Creative

SoftUni Digital

SoftUni Foundation

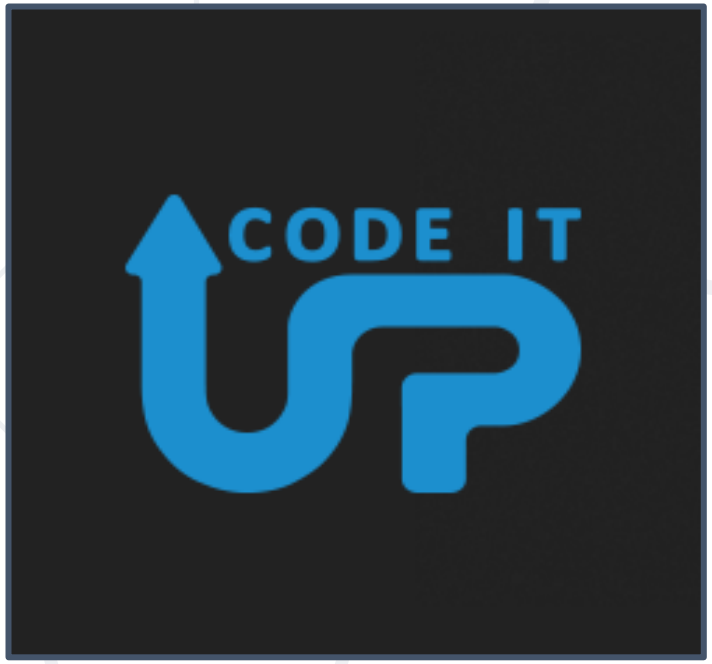
SoftUni Kids

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