Destructors, Constructors, Copy-Assignment

Special/Default Class Members, Destructors, Rule of 3/5/0

SoftUni Foundation

Georgi Georgiev

A guy that knows C++

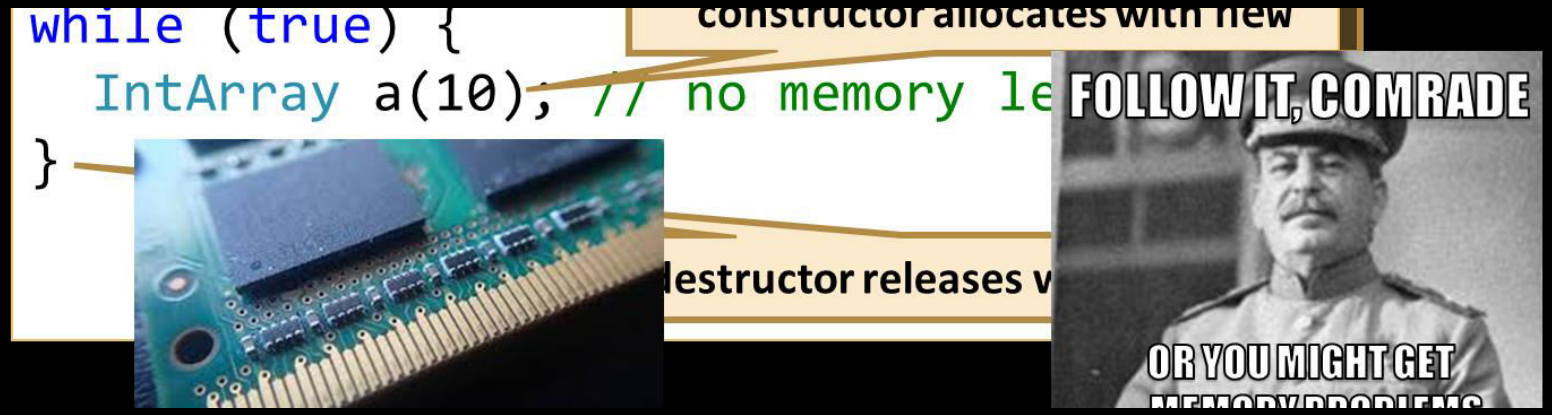


Table of Contents

Questions

SoftUni Foundation

sli.do

#cpp-softuni

SoftUni Foundation

Special Class Members

Destructors, Copy-Assignment/Construction

Special Class Members

SoftUni Foundation

- Members called by C++ in special cases
- Default Constructor – allocating objects & arrays
- Destructor – when lifetime ends (e.g. due to scope or `delete`)
- Copy-ctor – passing non-reference parameters/returning values
- Copy-assignment – when `operator=` is used
- Move ctor & assignment – for C++11 move semantics

Default Constructor Callers

SoftUni Foundation

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

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

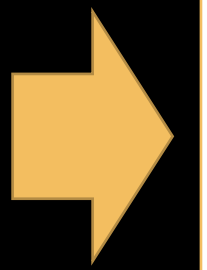
```
string s; // default ctor call
Lecturer steve; // default ctor call
Lecturer cpp[2]{ Lecturer("GG") }; // default ctor for cpp[1]
```

Auto-gen Default Constructor

SoftUni Foundation

- Initializes each object field – calls default ctors in initializer list
- Auto-generated if NO constructor declared explicitly
  - ... and 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
  {}
};
```


SoftUni
Foundation
Default Constructor
LIVE DEMO

Copy Construction/Assignment

SoftUni Foundation

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

Auto-gen Copy Constructor/Assignment

SoftUni Foundation

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

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


SoftUni
Foundation
Copy Construction & Assignment
LIVE DEMO

Destructors

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

```
while (true) {
  IntArray a(10); // no memory leak
}
```

constructor allocates with new

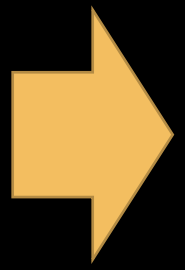
destructor releases with delete

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

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

Auto-gen Destructor

- "Destructs" each object field – i.e. calls each field's destructor
- Auto-generated if no destructor 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();
  }
}
```


SoftUni
Foundation
Destructor
LIVE DEMO

Explicit Auto-gen and default

SoftUni Foundation

- 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
```

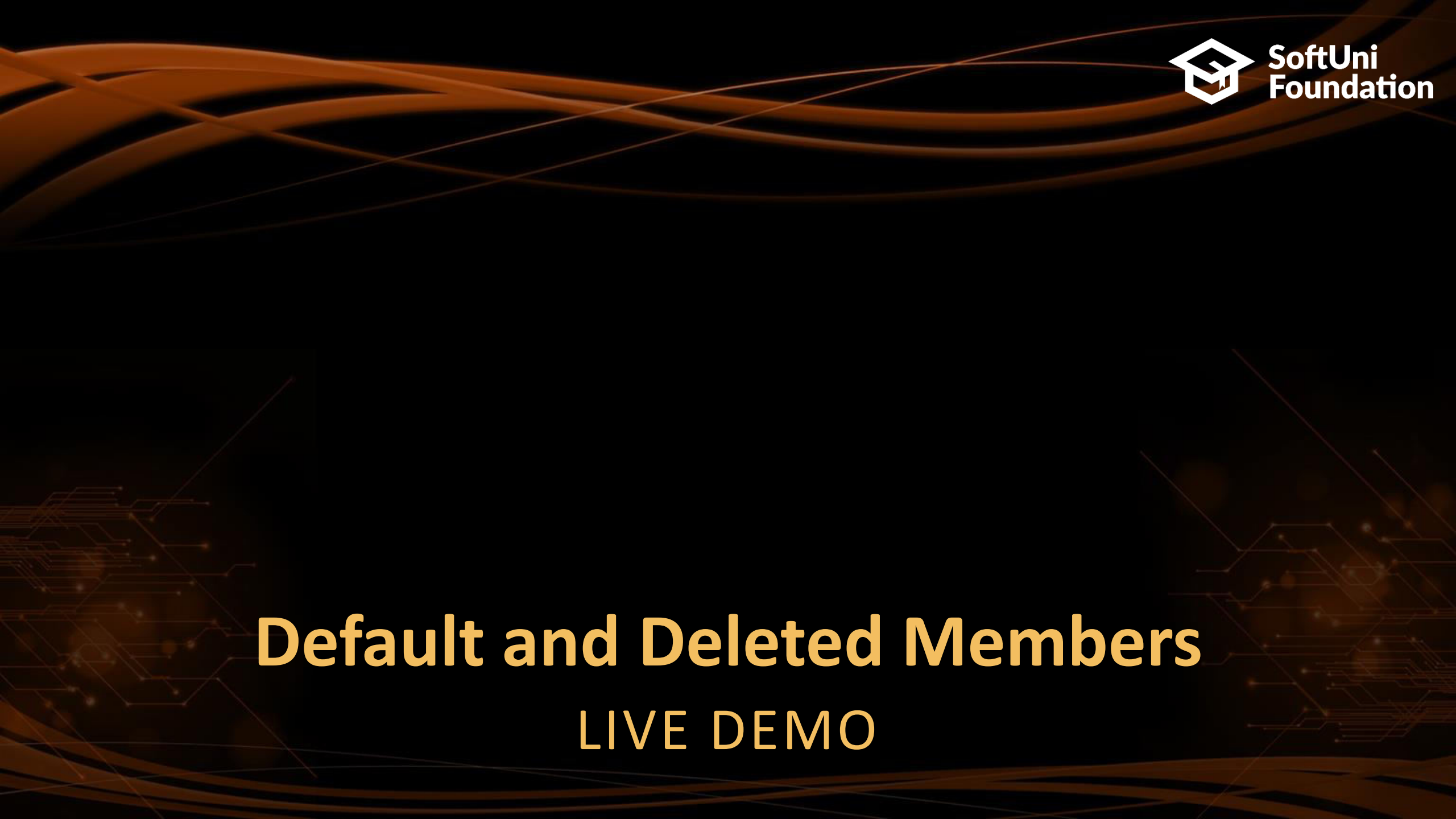
Disabling Special Members with delete

- Sometimes auto-generated methods need to be disabled
  - E.g. `unique_ptr<T>` disables copying
  - *Hacky*[1] Hard way – declare the members as private
  - Easy way (C++11) – use `= delete` after member signature

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

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

- [1] this only prevents outside access, which isn't always enough


SoftUni
Foundation
Default and Deleted Members
LIVE DEMO

Quick Quiz TIME:

SoftUni Foundation

- 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

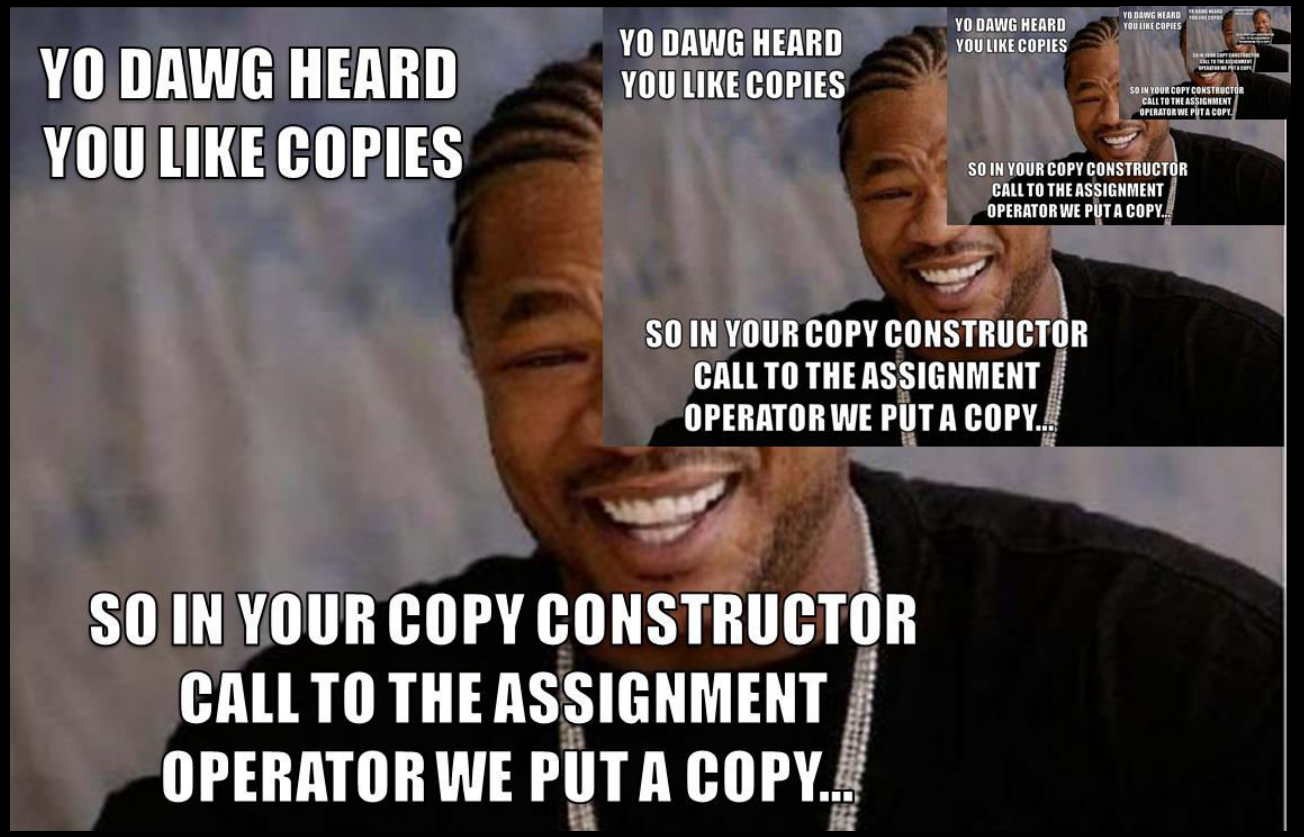
```
struct Lecturer {
  double rating; string name;
  Lecturer(string name, double rating)
  : name(name), rating(rating) {}

  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.

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 compile this (e.g. Visual C++ 2017)



SoftUni Foundation

Resource Acquisition is Initialization

Associating Resources with Object Lifetime

RAII

SoftUni Foundation

- 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, locks, etc.
- Allocate in constructor, deallocate in destructor
  - Some cases might require allocation in methods
  - C++ guarantees destructor execution, even on error

RAII in the STL

SoftUni Foundation

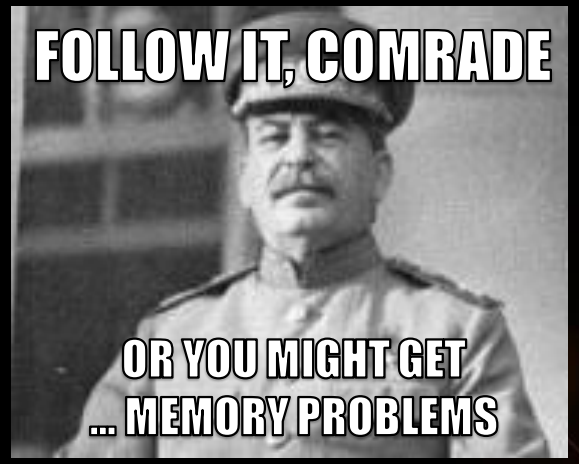
- All STL container classes are RAII
  - `vector<T>`, `list<T>`, `map<K, V>`, …
- C++ Streams are RAII
  - *E.g. file streams open file on construction & close on destruction*
- `shared_ptr<T>` extends RAII to "multiple ownership"
  - Multiple objects own a resource
  - Release when lifetime of last remaining owner ends

Exercise 1: SmartArray

SoftUni Foundation

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

SoftUni Foundation



Rule of Three / The Big Three

Why Constructor + Destructor isn't Enough

Destructors and Copies

SoftUni Foundation

- 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

SoftUni Foundation

- 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


SoftUni
Foundation
Copies Available ->
Destructor NOT Sufficient
LIVE DEMO

Destructor & Copies – Example RAII issue

SoftUni Foundation

- *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

SoftUni Foundation

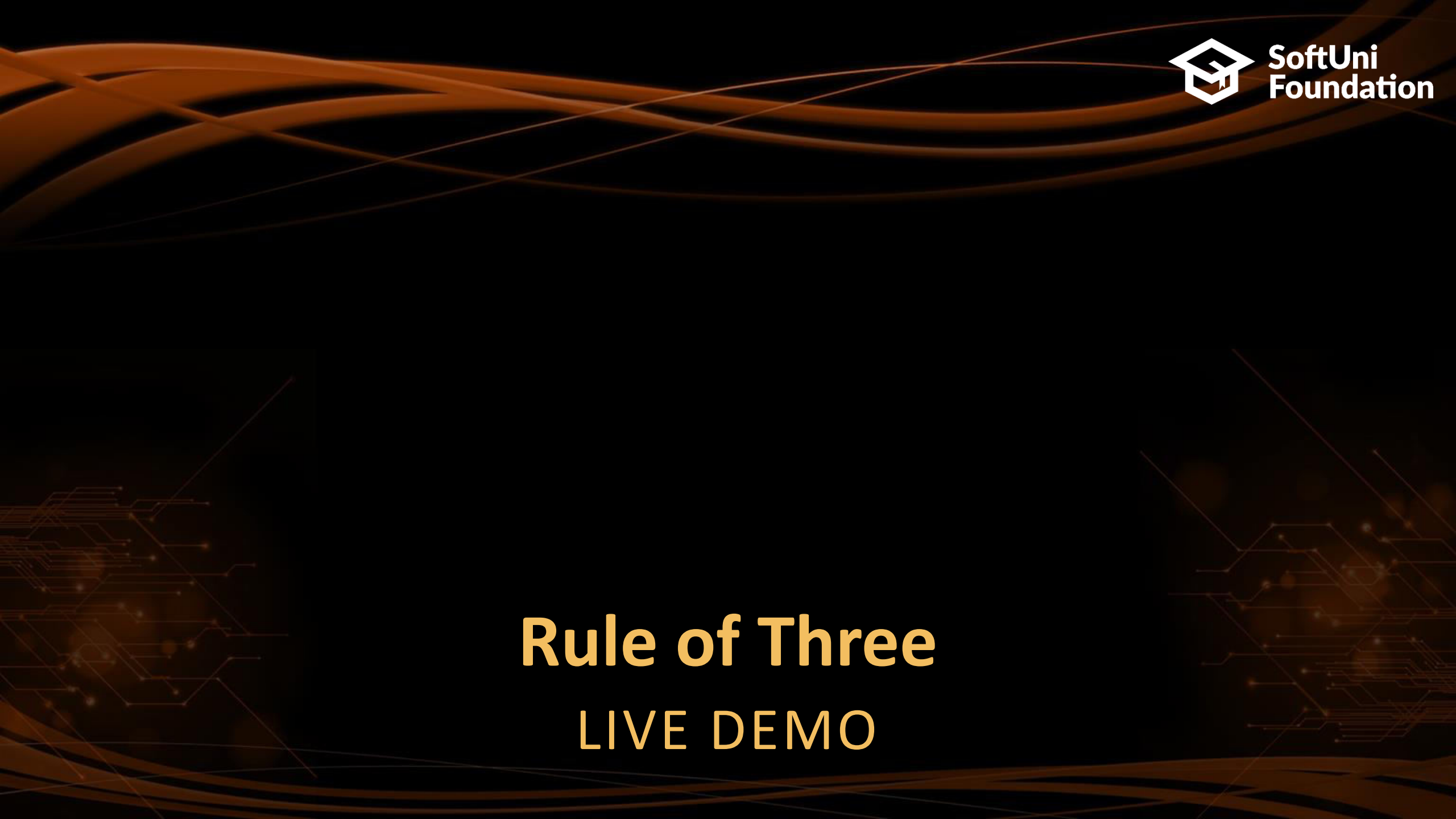
- If a class needs ONE of the following:
  - Destructor
  - Copy Constructor
  - Copy Assignment operator=
- Then it probably needs ALL of them:

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

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

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

Rule of Three – Copy Construct/Assign

SoftUni Foundation

- 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


SoftUni
Foundation
Rule of Three
LIVE DEMO

Exercise 2: Rule of Three for `SmartArray`

SoftUni Foundation

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

Quick Quiz TIME:

SoftUni Foundation

- 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.

Single Responsibility

SoftUni Foundation

- If a class has on of The Three, then:
  - It manages a resource (memory or something else)
  - It should manage a SINGLE resource
  - It should NOT do anything other than managing 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

Rule of Zero

Delegating Resource Management

Rule of Zero

SoftUni Foundation

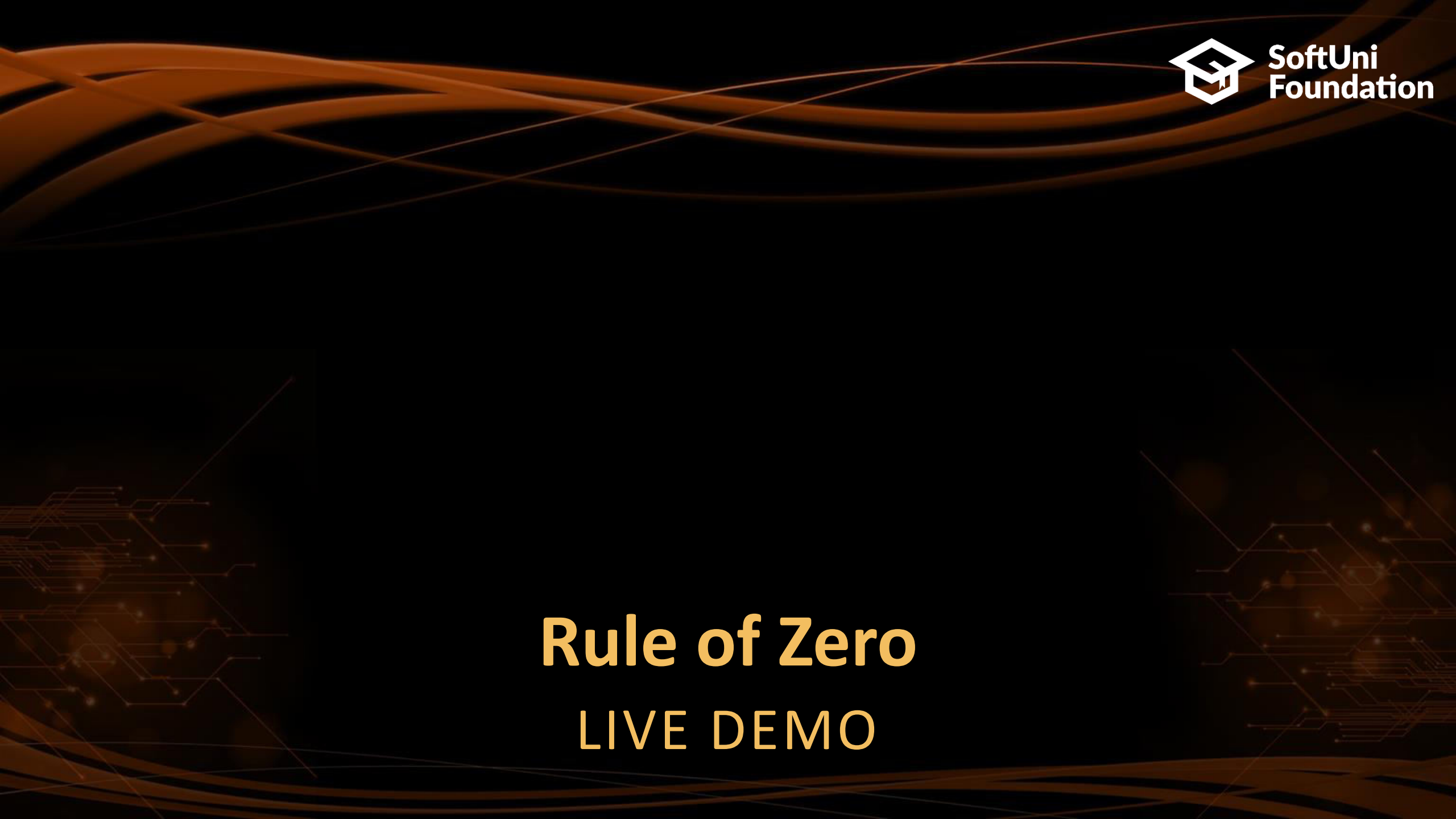
- STL has containers, smart pointers, etc.
  - Wrap other resources with classes implementing Rule of 3 (or 5)
- All remaining classes use the above, so:
  - No need for explicit destructor
  - No need for explicit copy-constructor
  - No need for explicit copy-assignment operator
- *In short: if you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

Rule of Zero for Array Class

SoftUni Foundation

- 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>`


SoftUni
Foundation
Rule of Zero
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
- RAII – C++ pattern of initializing memory in constructor
  - And freeing it in destructor
  - Rule of Three – implement or disable copy members
- Rule of Zero – delegate resource management to other classes

Destructors, Constructors, Copy-Assignment

SoftUni Foundation

Questions?

https://softuni.bg/courses/