Objects and Classes Advanced

Advanced Class Members

SoftUni Team

Technical Trainers

SoftUni

Software University

https://softuni.bg

Software University

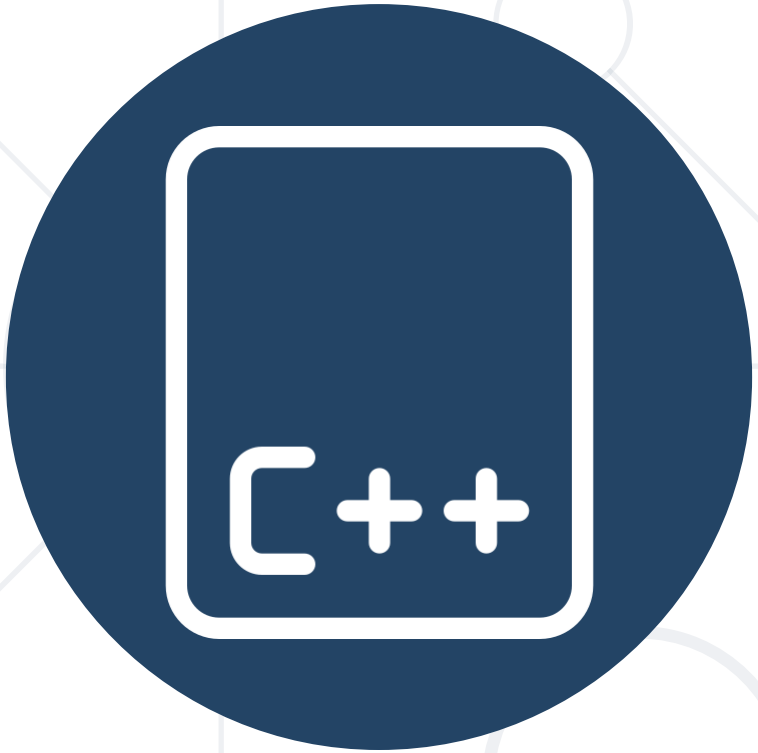
Table of Contents

Have a Question?

Software University

sli.do

#cpp-oop



Namespaces

Organizing Code into Named Groups

Namespaces

- Named groups of variables, functions, classes, etc.

```
namespace GroupName { ... /*members*/ ... }
```

  - Members access each other normally

```
namespace SoftUni {
  namespace CppFundamentals {
    const int numLectures = 6
    std::string lectures[numLectures]{ "Basic Syntax", ... };
  }
  namespace CppAdvanced {
    using namespace std;
    vector<string> lectures{ "Pointers and References", ... };
  }
}
```

Namespaces

- Outside code uses group name followed by `operator::`

```
int main() {
  for (std::string s : SoftUni::CppFundamentals::lectures)
    std::cout << s << std::endl;
}
```

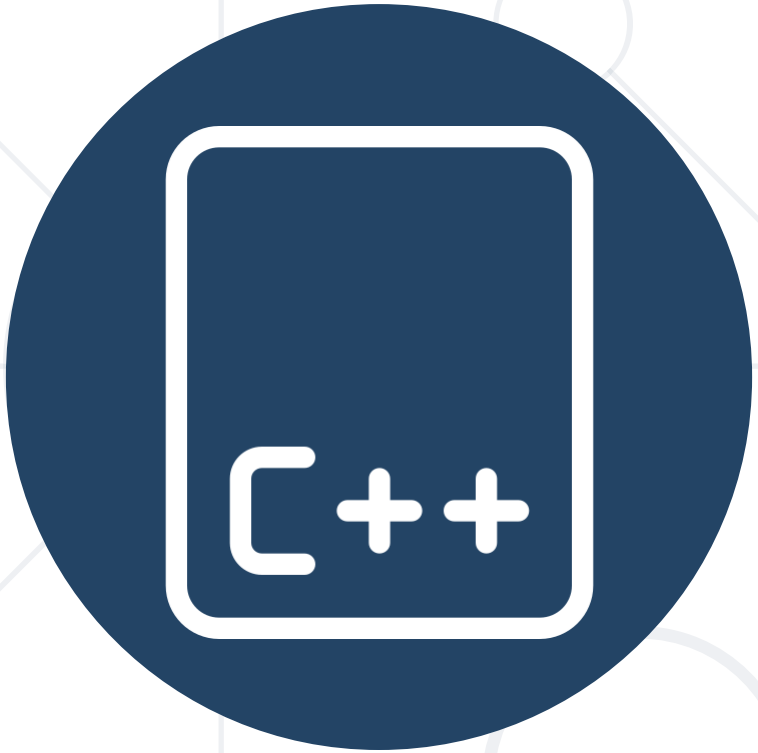
- `using` declarations tell compiler where to look "by default"
  - `using namespace std;`

```
int main() {
  using namespace SoftUni::CppFundamentals;
  for (std::string s : lectures)
    std::cout << s << std::endl;
}
```

Namespaces Application

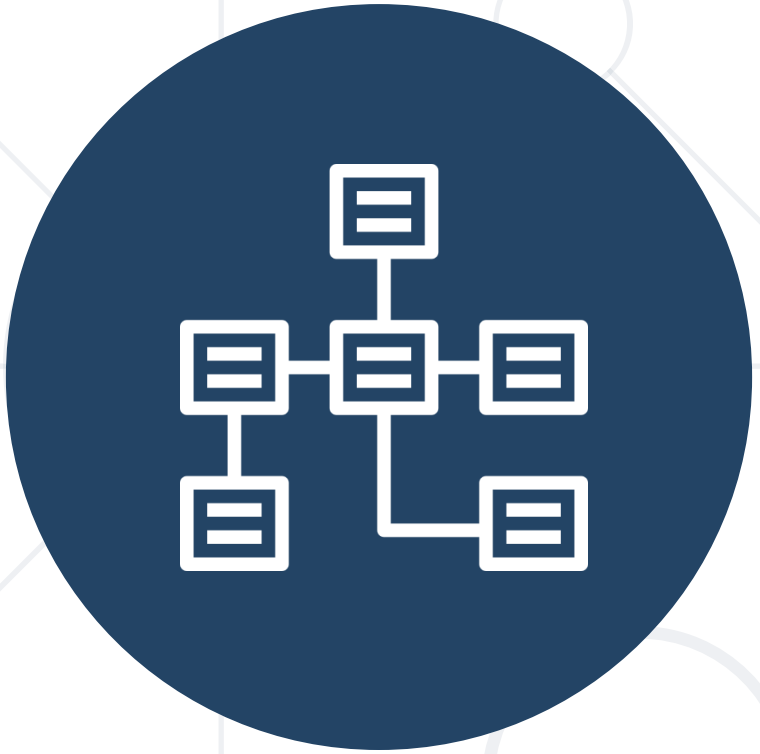
- Main purpose of namespaces – avoid name conflicts
- Example: a 2D Geometry library vs. C++ `std` library
  - `std::vector` – dynamic linear container
  - `geometry2d::vector` – a vector in 2D space (with `x`, `y`)
  - Namespaces prevent `vector` name conflict
- Avoid `using` declarations

```
using namespace std; using namespace Geometry2D;
vector v; // compilation error
```



Namespaces

LIVE DEMO



Members

`static`, `const`, `mutable`

Static Members in OOP

- Members NOT related to any specific object
  - Used without an object
- Access similar to identifiers in namespaces
  - class name and `operator::`

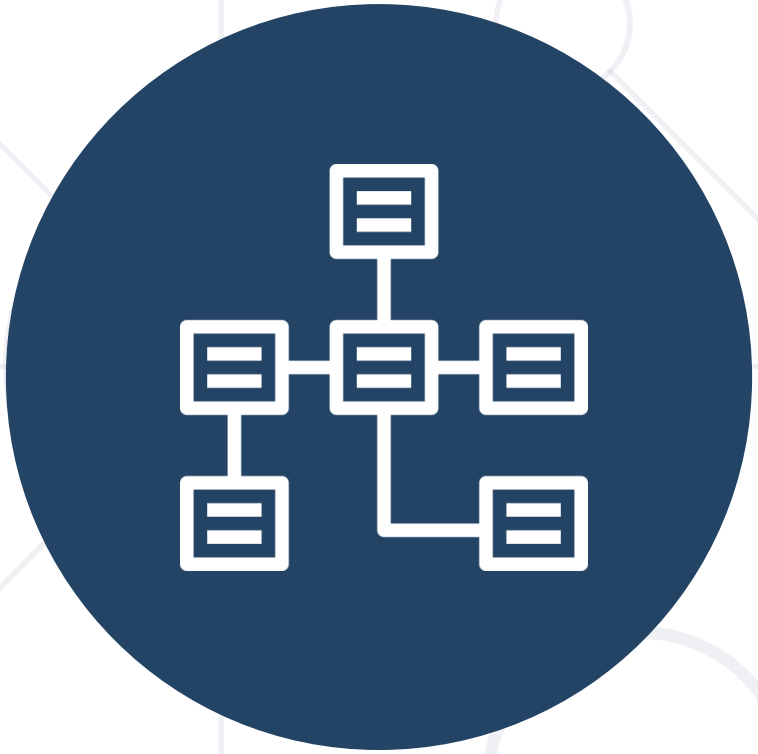
Static Members in OOP

```
class Company {
public:
  static const int ID_LENGTH = 8;
  string id;
  long long capitalDollars;
  ...
  static string generateId() {
    string id(ID_LENGTH, ' ');
    for (int i = 0; i < ID_LENGTH; i++)
      id[i] = 'A'+rand()%(1+'Z'-'A');
    return id;
  }
}
...
int main() {
  Company randomIdCompany{ Company::generateId(), 100 };
  Company z{ string(Company::ID_LENGTH, 'Z'), 1000 };
  ...
```

C++ static Fields

- Exist in the class, not in each object
- Defined and initialized outer class, in a `.cpp` file

```
class Company {
public:
static int CREATED_COMPANIES;
  ...
  Company(...) { CREATED_COMPANIES++; }
};
int Company::CREATED_COMPANIES = 0;
int main() {
  Company a{ ... }; Company b{ ... }; Company c{ ... };
  cout << Company::CREATED_COMPANIES; // prints 3
...
```



Static Members

LIVE DEMO

C++ const Fields

- Fields can be **const** – same as **const** variables
  - If non-static, initialized in constructor initializer list

```
class Company {
public:
  const std::string id;
  Company(std::string id, ...) : id(id), ... {}
}
```

```
const Company* c = new Company{ "GOOGINC.", ... };
cout << c.id << endl; // prints GOOGINC.
c.id = "thiswontcompile"; // compilation error
```

C++ const Methods

Software University

```
class Company {
  ...
  long long dollars; string id;
  void addCapital(long long dollars) {
    this->dollars += dollars;
  }

  void print() const {
    cout << this->id << " " << this->dollars;
  }
};
```

Method name

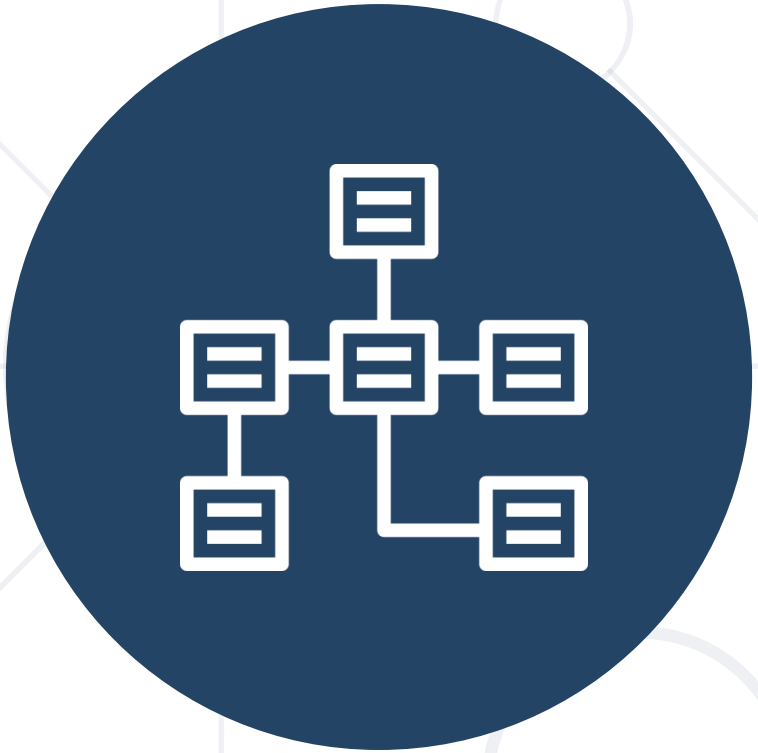
const

Return type

const methods can NOT change fields

const object/reference/pointer can only call **const** methods

```
Company c{ "GOOGINC.", 999 };
const Company& constRef = c;
constRef.print(); // GOOGINC. 999
c.addCapital(999999);
constRef.addCapital(999999); // compilation error
```



Constant Members

LIVE DEMO

Quick Quiz

- Which of the parts of code here will have compilation errors?

a) The **`printOlder`** method and the **`Person`** ctor

b) The **`Person`** ctor

c) The **`printOlder`** method

d) None, the code is valid

```
class Person {
public:
  int age; const string name;
  Person(string name, int age) {
    this->name = name; this->age = age;
  }
int getAge() { return this->age; }
};
```

```
void printOlder(const Person& a, const Person& b) {
  if (a.getAge() >= b.getAge()) {
    cout << a.name;
} else cout << b.name;
```

```
Person a{ "joro", 26 };
Person b{ "ben dover", 46 };
printOlder(a, b);
```

C++ PITFALL: MISSING CONST ON GETTERS AND NOT SETTING CONST FIELDS IN INITIALIZER LIST

`const` fields can only be initialized in constructor initializer list. They can't be assigned in constructor body.

Getters should usually be marked **`const`** – they don't change the object, and outside code calling them may be doing so from const references/pointers.

The mutable Keyword

- Fields marked **mutable** can be changed by **const** methods
  - External code accesses **const**
  - Internal code changes state
  - Typically used for caching, logs, mutexes and other metadata

```
class Person {
  int age; const string name;
  mutable int ageChecks = 0;
public:
  Person(string name, int age)
  : name(name), age(age) {}

  int getAge() const {
    this->ageChecks++;
    return this->age;
  }
  int getAgeChecks() const {
    return this->ageChecks;
  }
};
```

```
const Person a{ "george", 26 };

a.getAge(); a.getAge(); a.getAge();

cout << a.getAgeChecks() << endl; // prints 3
```

Practice

Live Exercise in Class

Problem 1: Rolling Sticks

- You are given code that animates sticks
  - Represented on a line on the console
  - "roll" by changing their symbol and position on the line
  - Symbols: start from **_**, then ****, then **|**, then **/** and back to **_**
  - Position starts from **0**. When symbol becomes **|** – move to next
- The code already does the animation, you need to implement a **`Stick`** class that keeps and updates the state of a **`Stick`**
  - Implement the code in a **`Stick.h`** file **`include`**d by the **`RollingSticksMain.cpp`** file

Friend Functions and Classes

Sharing Access to Private Members

The `friend` Keyword

- Allows access to private members
  - Declared inside the "shared" class
  - The friend can access the "shared" class
- Can be **function** or **class**:

```
friend Type functionName();

friend classNAme;
```

Defining a friend function

Defining a friend class

- "Sharing" is one-way – from declaring a class to a friend

The C++ friend Usage

- Friend functions are often used for directly reading fields of a class
- Friends can usually be changed to members

```
class Company {
  private: string id; long long dollars;
  ...
  friend void getCompany(istream& in, Company& c);
};

void getCompany(istream& in, Company& c) {
  in >> c.id >> c.dollars;
}
```

```
Company c;
getCompany(std::cin, c);
```

Friend Functions and Classes

LIVE DEMO

Operator Overloading

Operator Overloading

- Redefining operators for user-defined classes
  - Almost all operators can be redefined (except `operator::`)
  - `+`, `-`, `*`, `/`, `++`, `--`, `<<`, `>>`, `<`, `>`, `=`, `operator bool`, …
- Operators are just specially-named functions/methods

```
Type operator+(...)
bool operator<(...)
...
```

- As members – first operand `this`, others are parameters
- As non-members – all operands are parameters

Member Operator Overload

- Syntax (replace **T** with the operator, e.g. **+**, **-**, **<**, ...)

```
ResultT operatorT(RighthandT r)    // binary
```

```
ResultT operatorT()                // unary
```

```
class Price {
  int cents; string currency;
  ...
  Price operator+(const Price& other) const {
    string resultCurrency = ...;
    return Price{ this->cents + other.cents,
resultCurrency };
  }
};
```

```
Price a{ 499, "usd" };
Price b{ 1000, "usd" };


Price sum = a + b;
// sum is { 1499, "usd" }
```

Member Operator Overload

LIVE DEMO

Non-Member Operator Overload

- Syntax (replace **T** with the operator, e.g. **+**, **-**, **<**, ...)

```
ResultT operatorT(LefthandT l, RighthandT r)    // binary
```

```
ResultT operatorT(T operand)                    // unary
```

```
Price operator+(const Price& a, const Price& b) {
  string currency = ...;
  return Price(a.getCents() + b.getCents(), currency);
}
```

```
Price a{ 499, "usd" };
Price b{ 1000, "usd" };

Price sum = a + b; // sum is { 1499, "usd" }
```

Specifics of Non-Member Overload

Software University

- Non-member overloads allow any **left-hand** class
- Can be used to define operators for other types

```
string operator+(const string& s, const Price& p) {
  ostringstream out;
  out << s << p.getCents() << " " << p.getCurrency();
  return out.str();
}
```

```
Price a{ 499, "usd" };
Price b{ 1000, "usd" };
Price sum = a + b;
cout << std::string("Sum is ") + sum << endl;
```

Overloading Stream Read/Write

- `ostream` and `istream` use operators for output/input
  - `operator<<` and `operator>>` respectively
  - Defined for primitive types and `string`
  - Our classes contain primitives/`string`
- Overloading read/write for our classes
  - Read/write each field from/to the stream
  - Return the stream to enable chaining
  - Left operand stream, a right operand user object

Overloading Stream Read/Write

- Overriding read from `istream` – `friend` if fields private

```
class Price {... friend istream& operator>>(istream& in, Price& p); ... };
```

```
istream& operator>>(istream& in, Price& p) {
  return in >> p.cents >> p.currency;
}
```

```
Price a, b; cin >> a >> b;
```

- Overriding write to `ostream`

```
ostream& operator<<(ostream& out, const Price& p) {
  return out << p.getCents() << " " << p.getCurrency();
}
```

```
std::cout << a + b << std::endl;
```

Non-Member Operator Overload

LIVE DEMO

Quick Quiz

- What will the following code do?

```
istream& operator>>(istream& in, Price& p) {
  in >> p.cents >> " " >> p.currency;
}
ostream& operator<<(ostream& out, const Price& p) {
  out << p.getCents() << " " << p.getCurrency();
}
```

```
Price a, b; cin >> a >> b;
std::cout << a + b << std::endl;
```

a) Print the sum of two prices read from the console

b) Give a compilation error

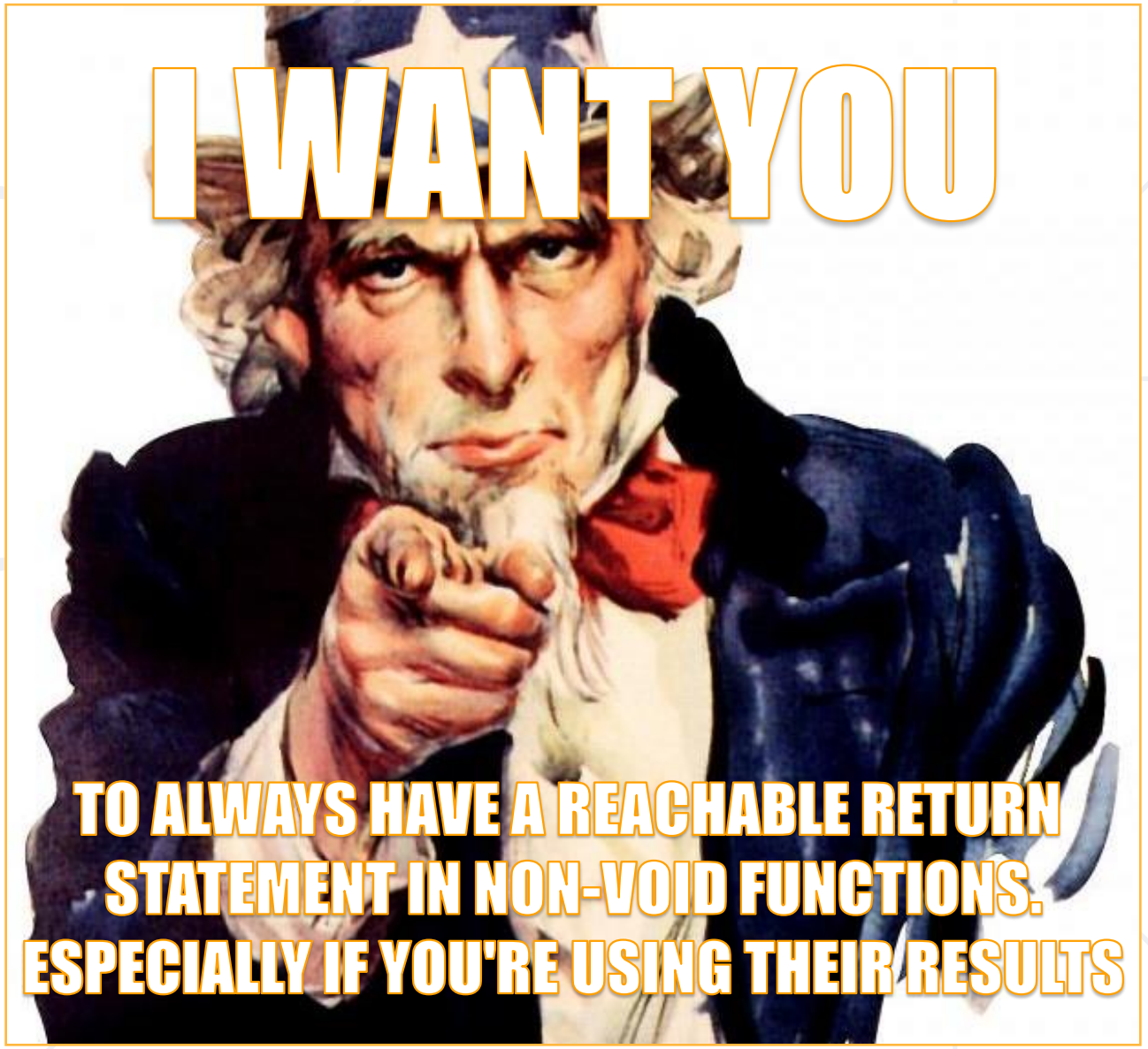
c) Behavior is undefined

Some compilers DO give compilation errors, but this is not required by the standard

C++ PITFALL: MISSING RETURN STATEMENT ON STREAM OPERATOR OVERLOAD, USED IN CHAINING

Notice the return statement is missing – hence the operator result is undefined (C++ does not give compilation errors here)

We use that undefined result in the chaining (i.e. `cin >> a >> b`, read **a** then read **b** with the resulting stream)



Comparison Operator Overload

Software University

- Comparison operators return `bool` and are binary
- `operator<` overloading is of special interest

```
class Fraction {
  int num; int denom;
public:
  Fraction(int num, int denom)
  : num(num), denom(denom) {}
  ...
  bool operator<(const Fraction& other) const {
    return this->num * other.denom < other.num * this->denom; }
};
...
set<Fraction> fractions{
  Fraction{1, 3}, Fraction{2, 10}, Fraction{2, 6}
}; // fractions will contain 2/10 and 1/3 in that order
```

Comparison Operator Overload

LIVE DEMO

Quick Quiz

- What will the following code do?

```
class Fraction {
  ...
  bool operator<(Fraction& other) {
    return this->num * other.denom < other.num * this->denom;
  }
};
```

```
set<Fraction> fractions{

  Fraction{1, 3},
  Fraction{2, 10},
  Fraction{2, 6}
};
```

a) Create a set with 2 Fractions

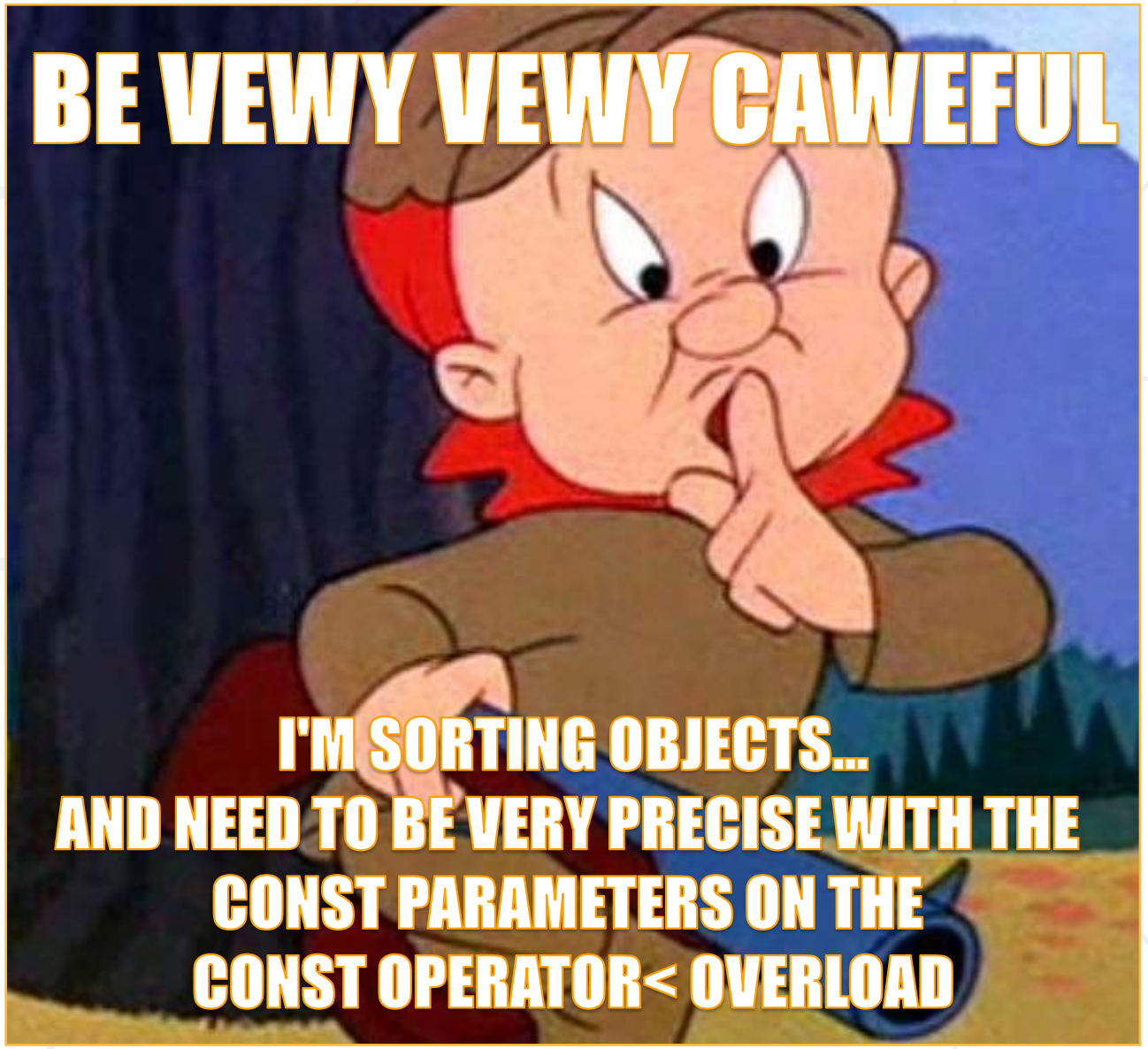
b) Give a compilation error

c) Behavior is undefined

C++ PITFALL: MISSING CONST ON PARAMETER AND/OR CONST ON OPERATOR METHOD WHEN USING WITH STL

All **`operator<`** usages in STL require the operator to be a const method with const reference parameters.

If they are not, we get a compilation error due to mismatch in parameters



Practice

Live Exercise in Class

Problem 2: Fraction Class

- Expand the Fraction class from the last examples
  - Equality comparison
  - Addition and subtraction
  - Direct **`cout`** usage
  - Direct **`cin`** usage
  - Automatically reduce (2/4 should initialize as 1/2)
  - **`operator++`** incrementation by 1

Summary

- **Namespaces** organize code and avoid name conflicts
- Static members are "global" class members
- **Friend classes/functions** can access private members
- **Operators** are just methods with special names
  - Can be overloaded by user code
  - Non-member overloads allow overloads for any class
- Don't overuse overloading – code has to be readable

Questions?

SoftUni

Software University

SoftUni Svetlina

SoftUni Creative

SoftUni Digital

SoftUni Foundation

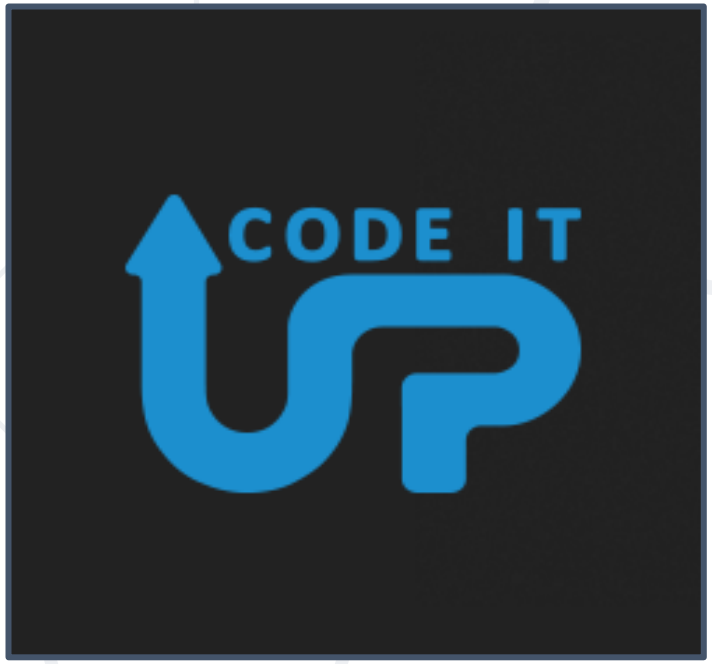
SoftUni Kids

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