Advanced C++ Class Members

Operator Overloading, Friends, static & const, Modifying STL

SoftUni Foundation

Georgi Georgiev

A guy that knows C++

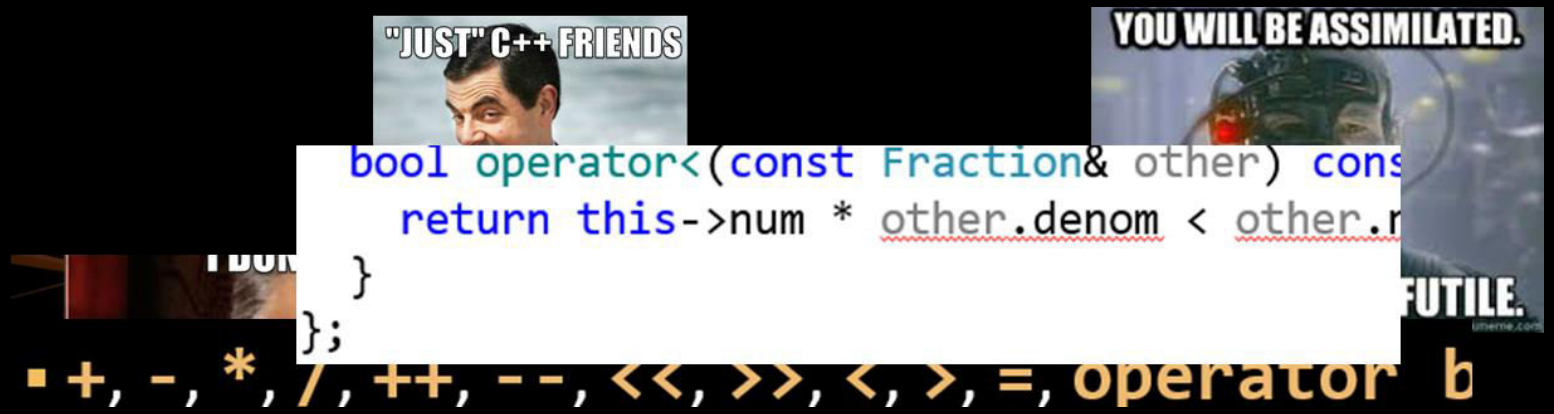


Table of Contents

Questions

SoftUni Foundation

sli.do

#cpp-softuni

SoftUni Foundation

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

SoftUni Foundation

- 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;`** – *check for all identifiers in* ***`std`***

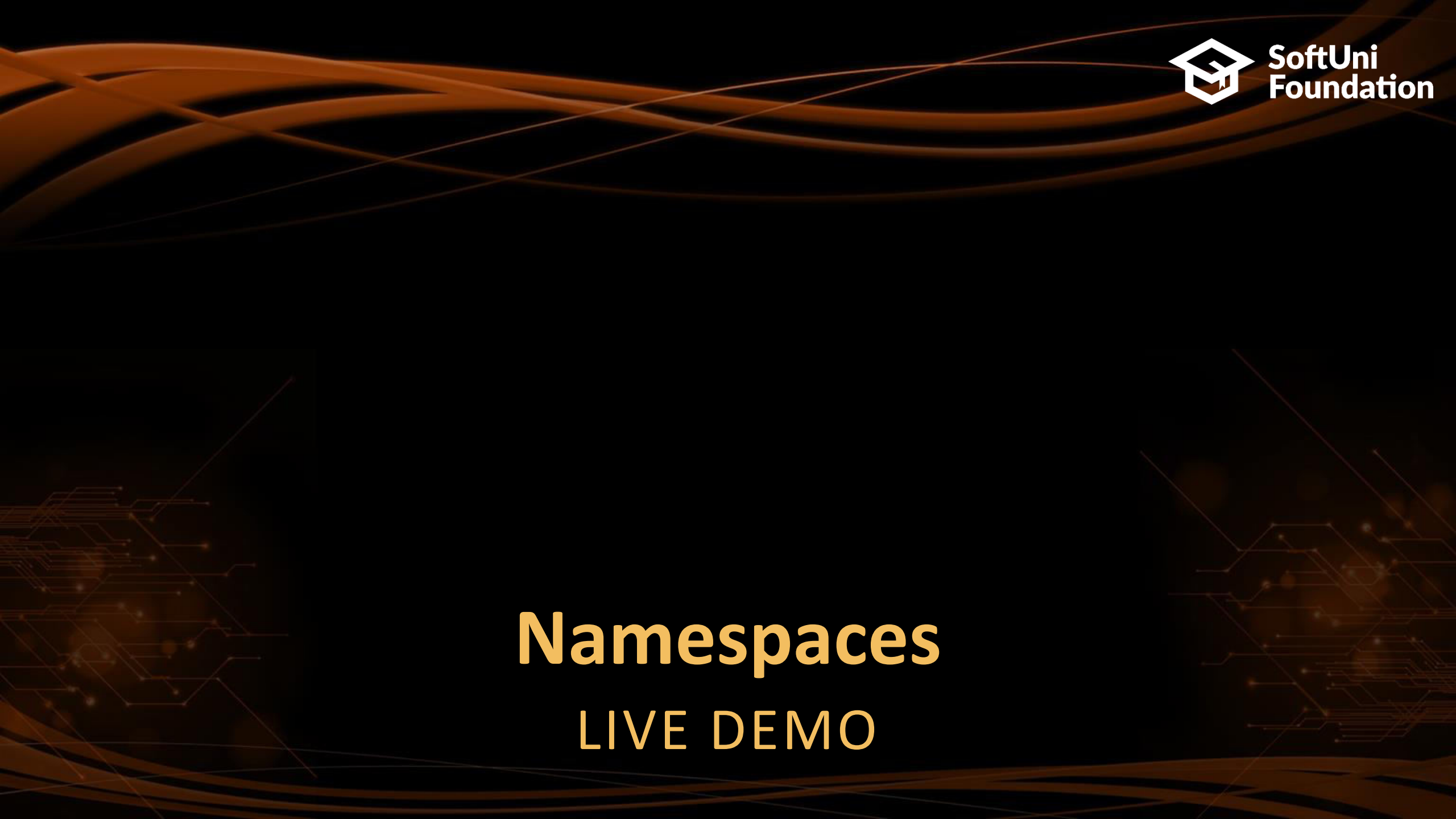
```
int main() {
  using namespace SoftUni::CppFundamentals;
  for (std::string s : lectures)
    std::cout << s << std::endl;
}
```

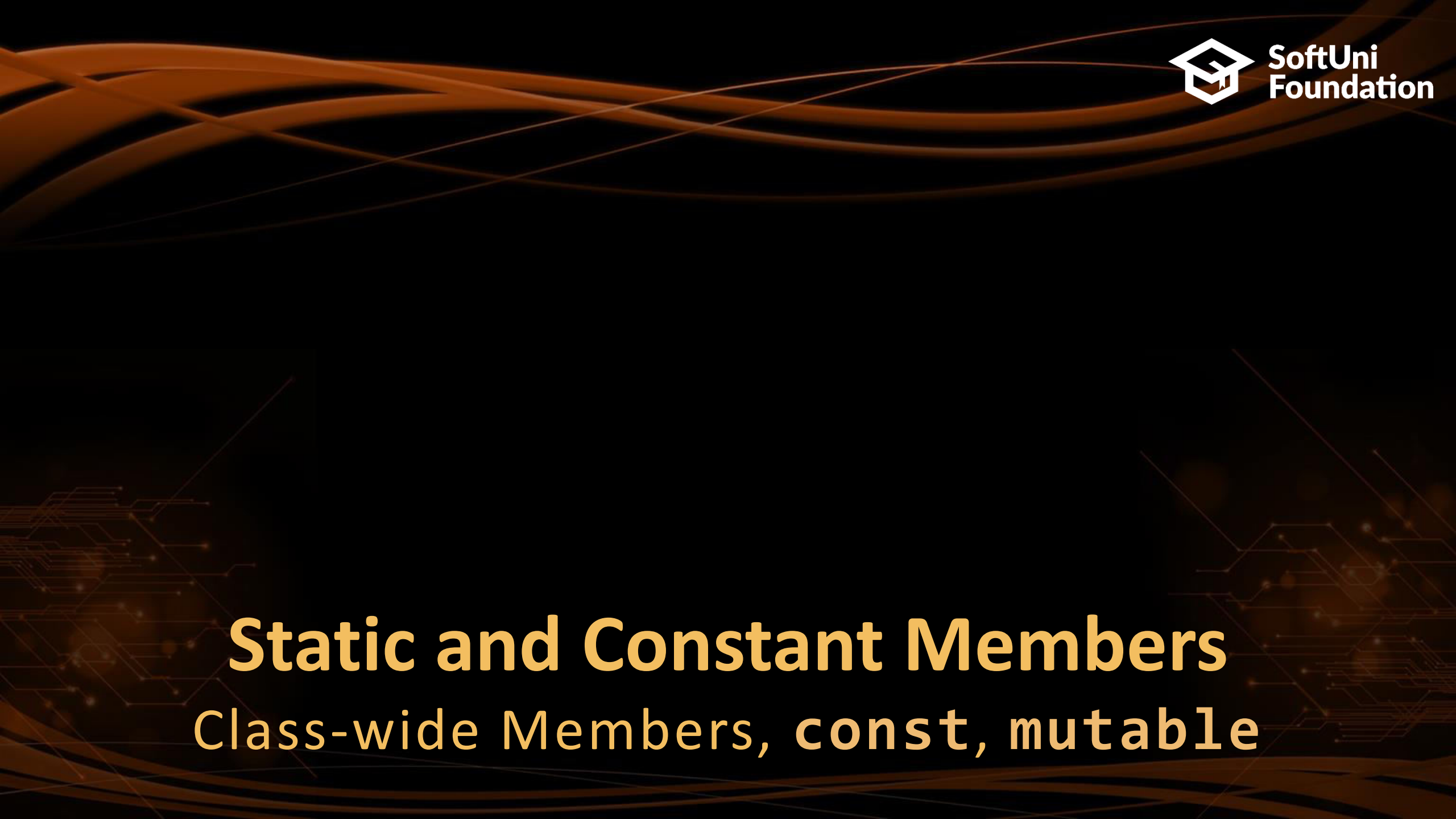
Namespaces Application

SoftUni Foundation

- 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
```


SoftUni
Foundation
Namespaces
LIVE DEMO



Static and Constant Members

Class-wide Members, `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 & **operator::**

```
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

SoftUni Foundation

- Exist on the class, not on each object
- Defined & initialized outside[1] class, in a `.cpp` file[2]
  - `Type ClassName::field = ...;` in same scope

```
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
...
```

- [1] `static const int/bool/char` can be initialized inside class; [2] extension doesn't matter, but the file must be a compilation unit, not just `#include`


SoftUni
Foundation
Static Members
LIVE DEMO

C++ const Fields

SoftUni Foundation

- 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

- *ReturnType methodName() const { ... }*
  - Methods with **const** can NOT change fields
  - **const** object/ref/pointer can only call **const** methods

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

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


SoftUni
Foundation
Constant Members
LIVE DEMO

Quick Quiz TIME:

SoftUni Foundation

- 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

SoftUni Foundation

- 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*

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

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

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

```
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;
  }
};
```

Exercise 1: Rolling Sticks

SoftUni Foundation

- 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 `included` by the `RollingSticksMain.cpp` file

SoftUni Foundation

Friends

Sharing Access to Private Members

The C++ friend Keyword

- Allows access to private members
  - Declared inside the "sharing" class
  - The friend can access the "sharing" class
- Can be function or class:
  - `friend Type functionName(...);`
  - `friend ClassName;`
- "Sharing" is one-way – from declaring class to friend

The C++ friend Usage

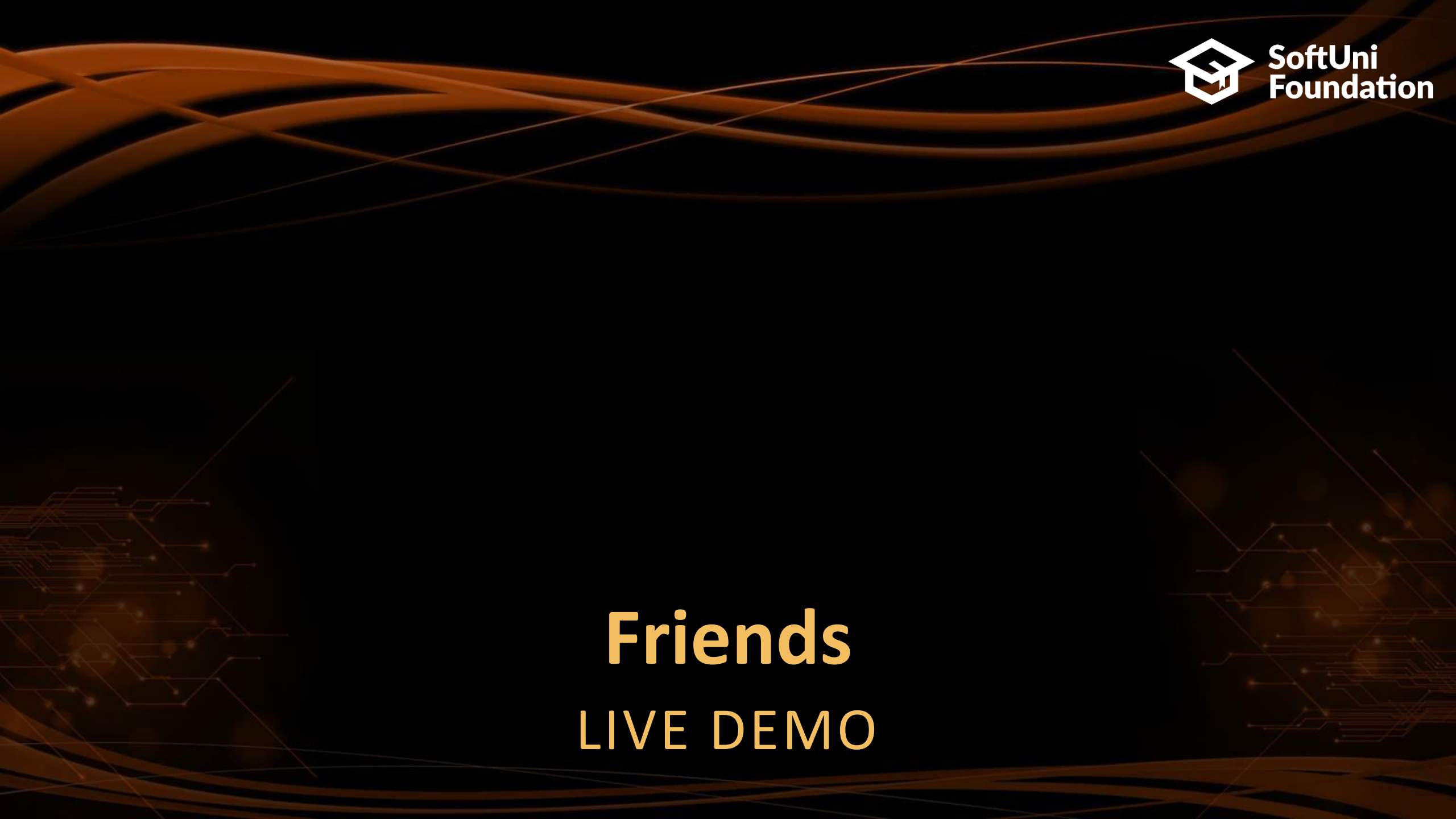
- Friend functions often used for directly reading fields of a class

```
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);
```

- Friends can usually be changed to members
  - *Prefer assimilating friends as class members… i.e. be like the Borg ☺*


SoftUni
Foundation
Friends
LIVE DEMO

SoftUni Foundation

Operator Overloading

No, that's not a Cheat-Code for StarCraft

Operator Overloading

SoftUni Foundation

- 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<(…)`, etc.
- As members – first operand `this`, others are parameters
- As non-members – all operands are parameters

Member Operator Overload

SoftUni Foundation

- Syntax (replace `@` with the operator, e.g. `+`, `-`, `<`, ...)
  - Binary: `ResultT operator@(RighthandT r)`
  - Unary: `ResultT operator@()`

```
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

SoftUni Foundation

- Syntax (replace `@` with the operator, e.g. `+`, `-`, `<`, ...)
  - Binary: `ResultT operator@(LefthandT l, RighthandT r)`
  - Unary: `ResultT operator@(T operand)`

```
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

SoftUni Foundation

- Non-member overloads allow any left-hand class
- Can be used to define operators for "other" types
  - E.g. operator appending a user class to **`string`**
  - E.g. operator writing a user class to a stream

```
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

SoftUni Foundation

- `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, right operand user object

Overloading Stream Read/Write

SoftUni Foundation

- 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;
```

SoftUni Foundation

Non-Member Operator Overload

LIVE DEMO

Quick Quiz TIME:

SoftUni Foundation

- What will the following code do (assuming **Price** is as in previous slides)?

```
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

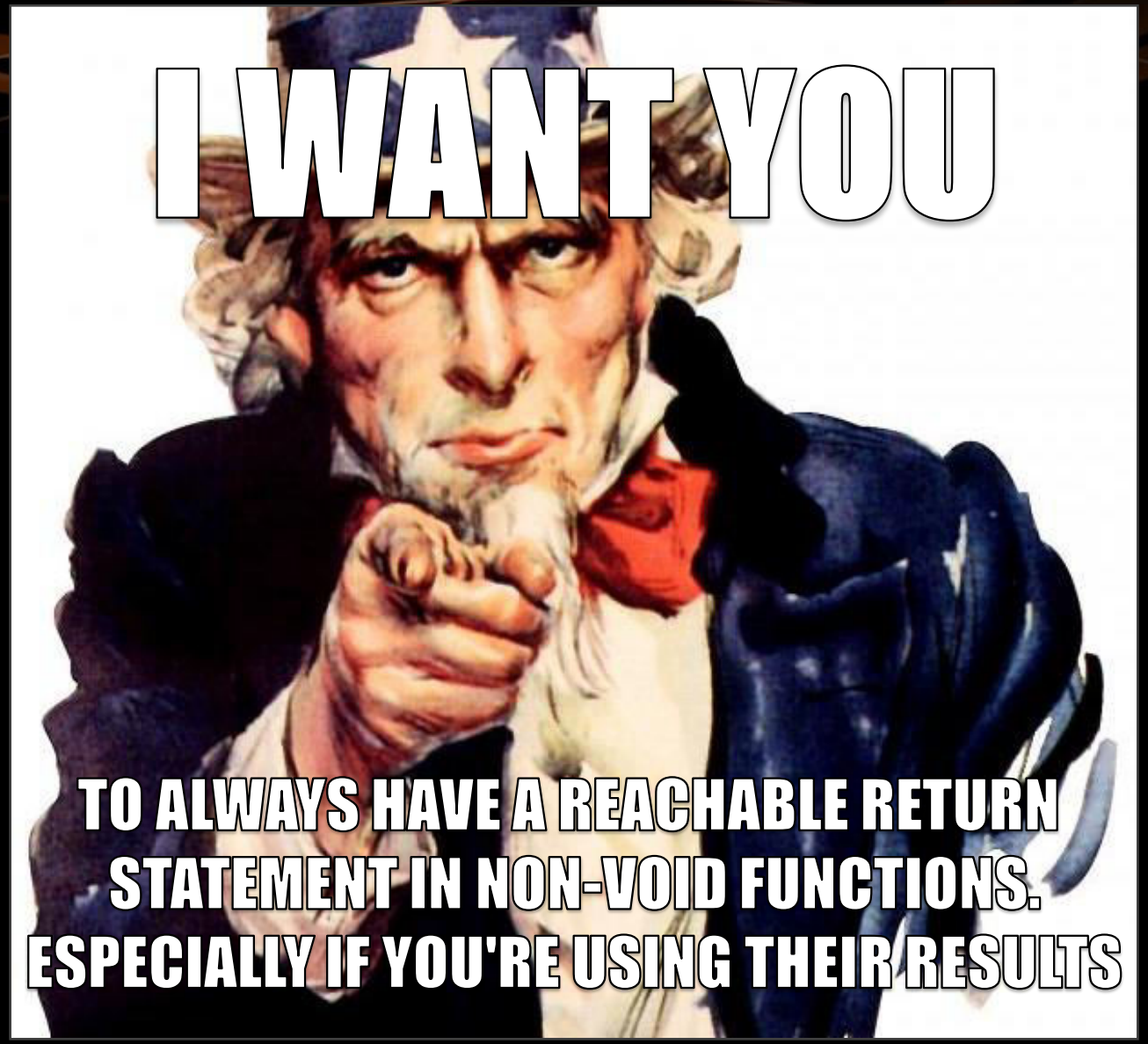
Note: some compilers DO give compilation errors, but this is not required by the standard

c) Behavior is undefined

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

SoftUni Foundation

- Comparison operators return **`bool`** and are binary
- **`operator<`** overloading is of special interest
  - Sorting & ordered containers require it by default[1]
  - *use only if class has a "natural" ordering*

```
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
```

- [1] this can be changed through functors – discussed later


SoftUni
Foundation
Comparison Operator Overload
LIVE DEMO

Quick Quiz TIME:

SoftUni Foundation

- What will the following code do (`Fraction` is as in previous slides)?

```
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}
}; // fractions will contain 2/10 and 1/3 in that order
```

a) Create a set with 2 **Fractions**

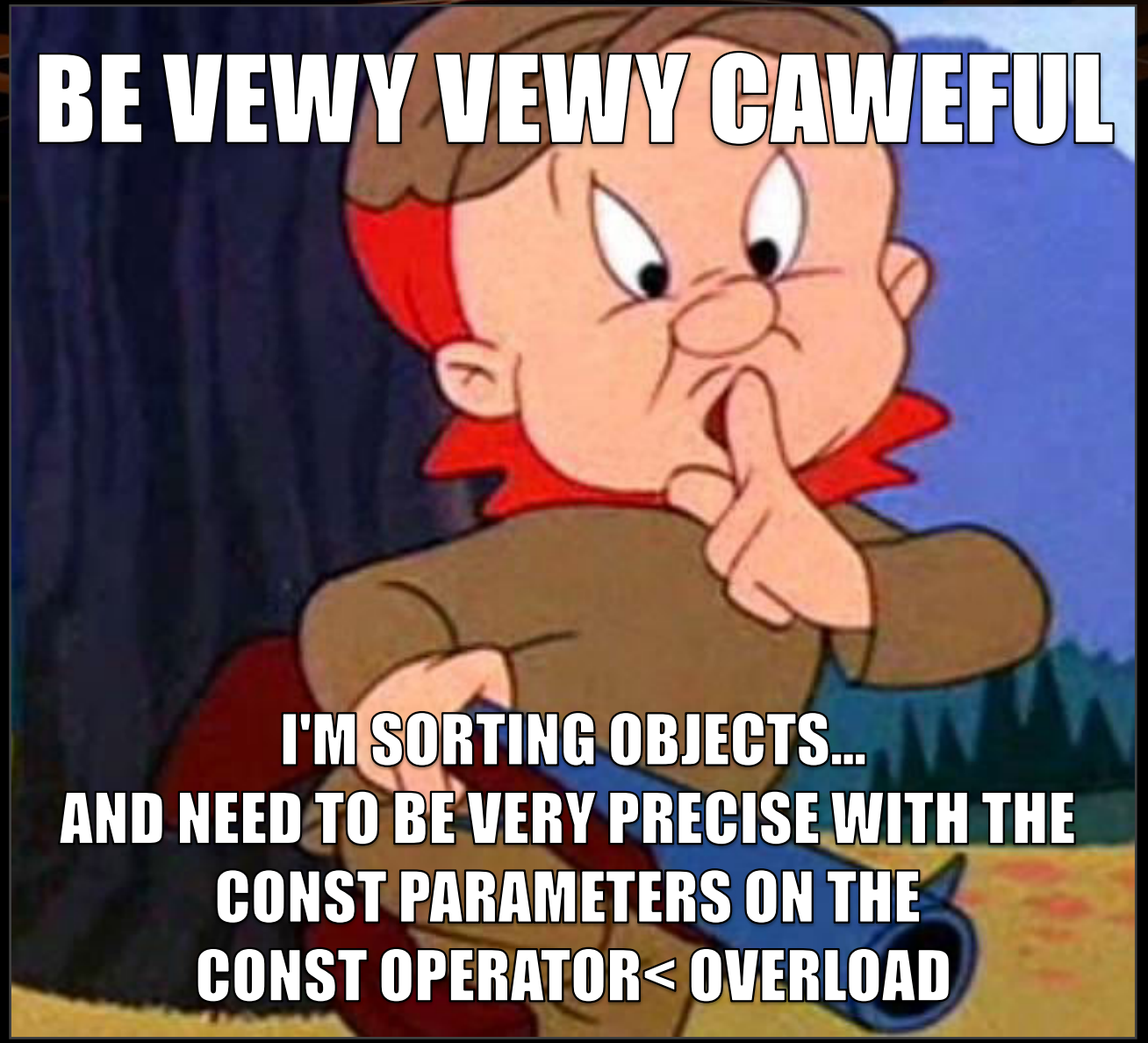
b) Give a compilation error

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



Exercise 2: Fraction Class

SoftUni Foundation

- Expand the `Fraction` class from the last examples
  - Equality comparison
  - Addition and subtraction
  - Direct `cout` usage
  - Direct `cin` usage
  - Automatically reduce – e.g. `2/4` should initialize as `1/2`
  - `operator++` incrementation by `1` (be careful with the math)

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
  - Avoid overloads unless meaning is obvious

Advanced C++ Class Members

SoftUni Foundation

Questions?

https://softuni.bg/courses/