Code Organization & C++ Templates

Preprocessors, Declaration vs. Definition Files, Templated Code

SoftUni Foundation

Georgi Georgiev

A guy that knows C++

Table of Contents

SoftUni Foundation

Questions

SoftUni Foundation

sli.do

#cpp-softuni

Preprocessor Directives

#include, #define, #if…

Preprocessor Directives

SoftUni Foundation

- Executed before compilation
- Instruct compiler how and what to compile
  - *Not part of the code, they modify the code*
  - **`#include`** – adds code to the compilation unit
  - **`#define`** – essentially a find-and-replace in the code
  - **`#if`**, **`#ifdef`**, **`#else`**… – use/skip code based on an expression
  - **`#pragma`** – compiler-specific settings (e.g. optimization level)

#include and #define

SoftUni Foundation

- **#include <X>** copies system **X** source in this file
  - **#include "X"** first looks for local file **X** , then for system **X**

```
#include <iostream> // directly looks for system file iostream
#include "01. Macros.h" // looks for local file "01. Macros.h"
```

- **#define X Y** – macro, replaces **X** in the code with **Y**

```
#define PI 3.14
cout << PI << endl; // prints 3.14
```

- **#define F(X) code-using-X** – macro function

```
#define SHOW(something) cout << something << endl;
SHOW("hello macros"); // prints "hello macros"
```

Quick Quiz TIME:

SoftUni Foundation

- What will the following code do?

```
#define HALF(value) value / 2
cout << HALF(4 + 2) << endl;
```

a) Print 3

b) Print **5**

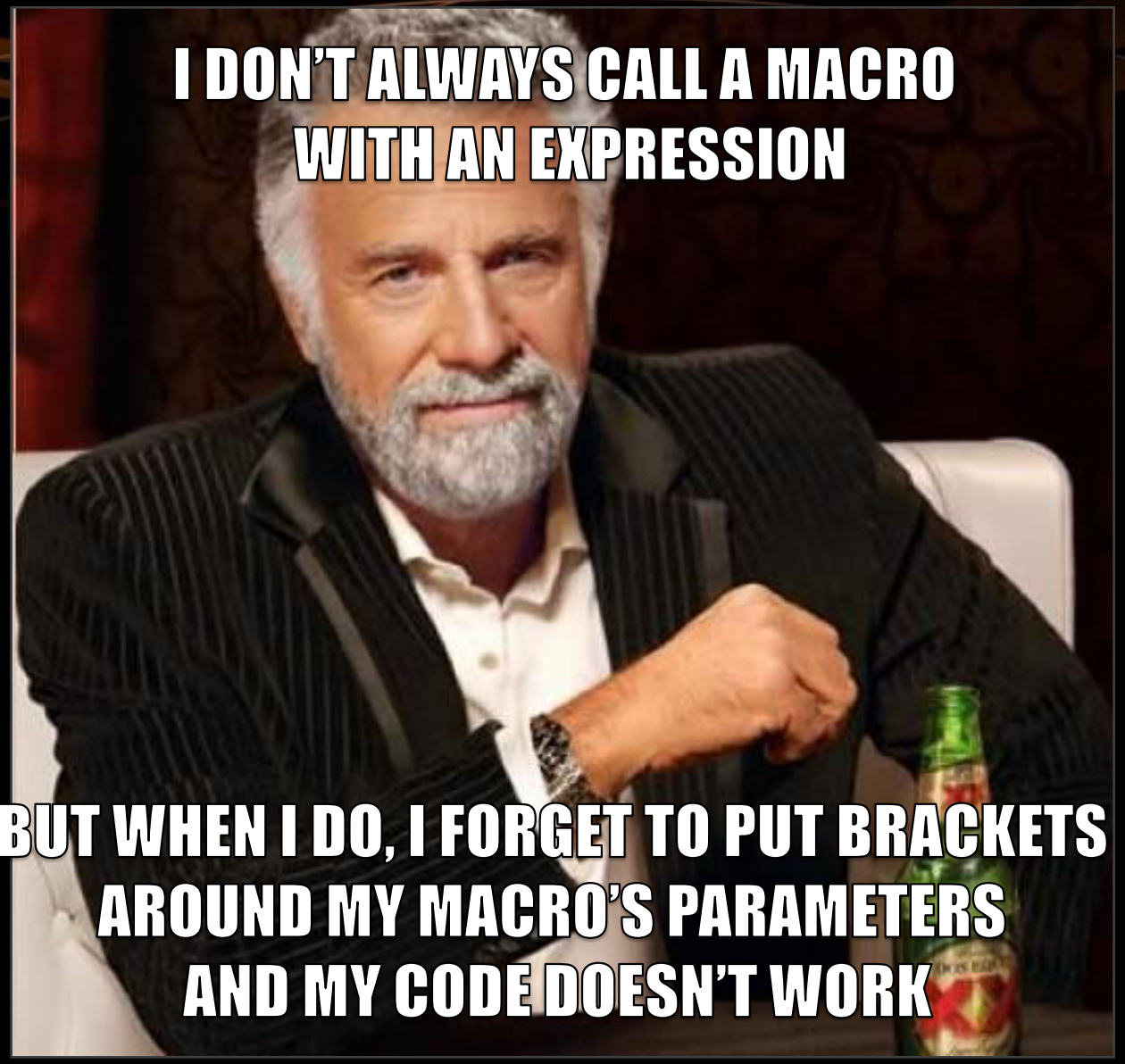
c) Cause a compilation error

d) Behavior is undefined

C++ PITFALL: MISSING BRACKETS ON MACRO PARAMETERS WHICH COULD BE EXPRESSIONS

Macro parameters are just copy-pasted into the macro code – they are not values, they are code.

A good practice is to put any macro and its parameters into brackets ( )




SoftUni
Foundation
#include and #define
LIVE DEMO

Conditional Inclusions & Header Guards

SoftUni Foundation

- Similar to if-else, when condition is NOT met, code is ignored
  - **#if**, **#elif** – "else if", **#else**, "closed" with **#endif**
  - **#ifdef X** – if macro **X** is defined, **#ifndef** – if macro **X** is NOT defined

```
#ifdef _WIN32
system("cls");
#else
system("clear");
#endif
```

- Header guards – avoid **#include**-ing code multiple times

```
#ifndef SOME_FILE_H // use any macro name unique for the file
#define SOME_FILE_H
// code here safe from multi-inclusion
#endif // !SOME_FILE_H
```

SoftUni Foundation

Conditional Inclusion & Header Guards

LIVE DEMO

SoftUni Foundation

Declaration vs. Implementation

Separating Members, Header & Source Files

Separating Declaration and Definition

SoftUni Foundation

- C++ allows separate declaration & definition/implementation
  - For functions, methods, operators and even classes
- Class members "implementation" is often separated
  - Cleaner view of class "interface"
  - Sometimes necessary – e.g. static fields or stream operators
  - Allows separate build objects for faster rebuilds & dynamic linking

Why Separate?

- *Is it easy to determine what can be called on this class?*
  - *NOTE: you need to view the presentation fullscreen in PowerPoint (shift + f5) to see the code (it's an animation)*

Why Separate? Cleaner "Interfaces"

SoftUni Foundation

- *Is it easier to view like this?*

```
class Company {
private:
int id;
string name;
vector<pair<char, char> > employees;

public:
Company(int id, string name, vector<pair<char, char> > employees);

int getId() const;
string getName() const;
vector<pair<char, char> > getEmployees() const;
string toString() const;
bool operator==(const Company& other) const;
std::string operator+(const char* s) const;
std::string operator+(const string& s);
Company& operator+=(const pair<char, char>& employee);
};
```

- *But where's the code?*

Separating Member Definitions

SoftUni Foundation

- Syntax same as member inside class, however:
  - Prefixed with namespaces & class name, joined by **`operator::`**
  - **Type *`Namesp1::Namesp2::…::ClassName`*`::member`**

```
Company::Company(int id, string name, vector<pair<char, char> > employees)
: id(id), name(name), employees(employees) {}
...
int Company::getId() const {
  return this->id;
}
...
bool Company::operator==(const Company& other) const {
  return this->id == other.id;
}
...
```


SoftUni
Foundation
Separating Member Definitions
LIVE DEMO

Header & Source Files

SoftUni Foundation

- Header files – declarations (sometimes also implementations)
  - Use header guards to avoid multi-inclusion
  - Extension doesn't matter, but usually `.h`/`.hpp`/`.h++`
- Source files – implements header declarations
  - Usually `1` per header, `#include` the header
  - Usually `.cpp` – some IDEs expect it

- NOTE: these are practices, not rules – C++ couldn't care less how you organize your code

Header & Source Files

SoftUni Foundation

Company.h

```
#ifndef COMPANY_H
#define COMPANY_H

#include <string>
#include <vector>
class Company { private:
int id; string name;
vector<pair<char, char> > employees;
public:
Company(int id, string name,
  vector<pair<char, char> > employees);
...
int getId() const;
...
bool operator==(const Company& other) const;
};
#endif // !COMPANY_H
```

Company.cpp

```
#include "Company.h"
#include <sstream> // Company::toString() uses
                   // it, not shown here
Company::Company(int id, string name,
  vector<pair<char, char> > employees)
  : id(id), name(name), employees(employees) {}
...
int Company::getId() const {
  return this->id;
}
...
bool Company::operator==(
    const Company& other) const {
  return this->id == other.id;
}
...
```

Building Multiple Sources in C++

SoftUni Foundation

- *Compilation* unit – a *file* (usually `.cpp`) the compiler works on
- C++ build process *(roughly)* for each unit:
  - `.cpp` -> expanded source (insert `#include` code, macros, etc.)
  - expanded source -> platform code -> assembly code
  - assembly code -> object code, `.o`/`.obj` (`1`'s & `0`'s)
- Linking: object code files -> linked -> final executable

Building Multiple Sources in C++

SoftUni Foundation

- Different approaches to building a multi-source "Project"
  a) Single `.cpp`, implementation in headers – compile the `.cpp`
  b) Only declaration in `.h`, multiple `.cpp` – compile & link all `.cpp`
  c) Mixed – some `.h` contain implementation – same as b)
- Compiler needs instructions which files to compile
  - As console arguments or with `makefile`, VS Solution, Qmake etc.
  - IDEs automate the process – e.g. compile & link all `.cpp` files


SoftUni
Foundation
Header & Source Files
LIVE DEMO

SoftUni Foundation

C++ Templates

Generalizing Functions/Classes for any Type

Algorithm vs. Data Type

SoftUni Foundation

- Algorithms rarely depend on a single data type
- E.g. calculate what percentage `a` is of `b`
  - `a` percent of `b` == `a * 100 / b`
  - `1` out of `4` == `a * 100 / 4` == `25%`
  - `1.5` out of `3` == `1.5 * 100 / 3` == `50%`
  - ¼ out of ½ == ¼ * 100 / ½ == 25 / ½ == 50%

Templates

SoftUni Foundation

- What should `T` be here: `T calcPercentage(T a, T b)`?
  - `int`, `double` or `Fraction`? All of them can be `T`
  - `T` here only needs `operator*` (with `int`) and `opertor/`
- Templates
  - Declare function or class with a “placeholder" type
  - Can then use with different types
  - Types should support the used methods/operators

C++ Function Templates

SoftUni Foundation

- `template<typename T>` – makes `T` a placeholder type
  - Can have multiple placeholders
  - Applies only to function/class directly after it

```
template<typename T>
T calcPercentage(const T& a, const T& b) {
  return (a * 100) / b;
}
```

```
template<class T1, class T2>
void printValues(const T1& a, const T2& b) {
  cout << a << " " << b << endl;
}
```

- `template<class T>` has same meaning[1]

- [1] there are some very specific cases where they are slightly different

Calling Templated Functions

SoftUni Foundation

- Call like normal function – C++ guesses types

```
calcPercentage(5, 10) // compiles & executes for int
```

  - If type doesn't support operations in function – compilation error

```
calcPercentage(5, " ") // compilation error in calcPercentage
                       // for operator* and operator/
```

- May need **<Type>** after name to specify type
  - E.g. **calcPercentage<double>(0.5, 1)**


SoftUni
Foundation
Function Templates
LIVE DEMO

C++ Class Templates

SoftUni Foundation

- Classes can receive templates to use as data types
  - `vector<T>`, `list<T>`, `map<K, V>` – examples we've used
- Defining class template – same as with function
  - `template<typename T> class ClassName { ... }`
  - Can use `T` for fields, methods, etc. – like any actual type
- Using class template
  - e.g. `ClassName<int> a; ClassName<Fraction> b;`

C++ Class Templates – Example

SoftUni Foundation

- Making a **`Pair`** class similar to **`std::pair`**
  - Use the same way

```
Pair<string, int> ben{
  "Ben Dover", 42
};

cout << ben.first << " "
  << ben.second;
```

```
#ifndef PAIR_H
#define PAIR_H
template<class T1, class T2>
class Pair {
public:
  T1 first; T2 second;
  Pair(T1 first, T2 second)
    : first(first)
    , second(second) {
  }
};
#endif // !PAIR_H
```


SoftUni
Foundation
Class Templates
LIVE DEMO

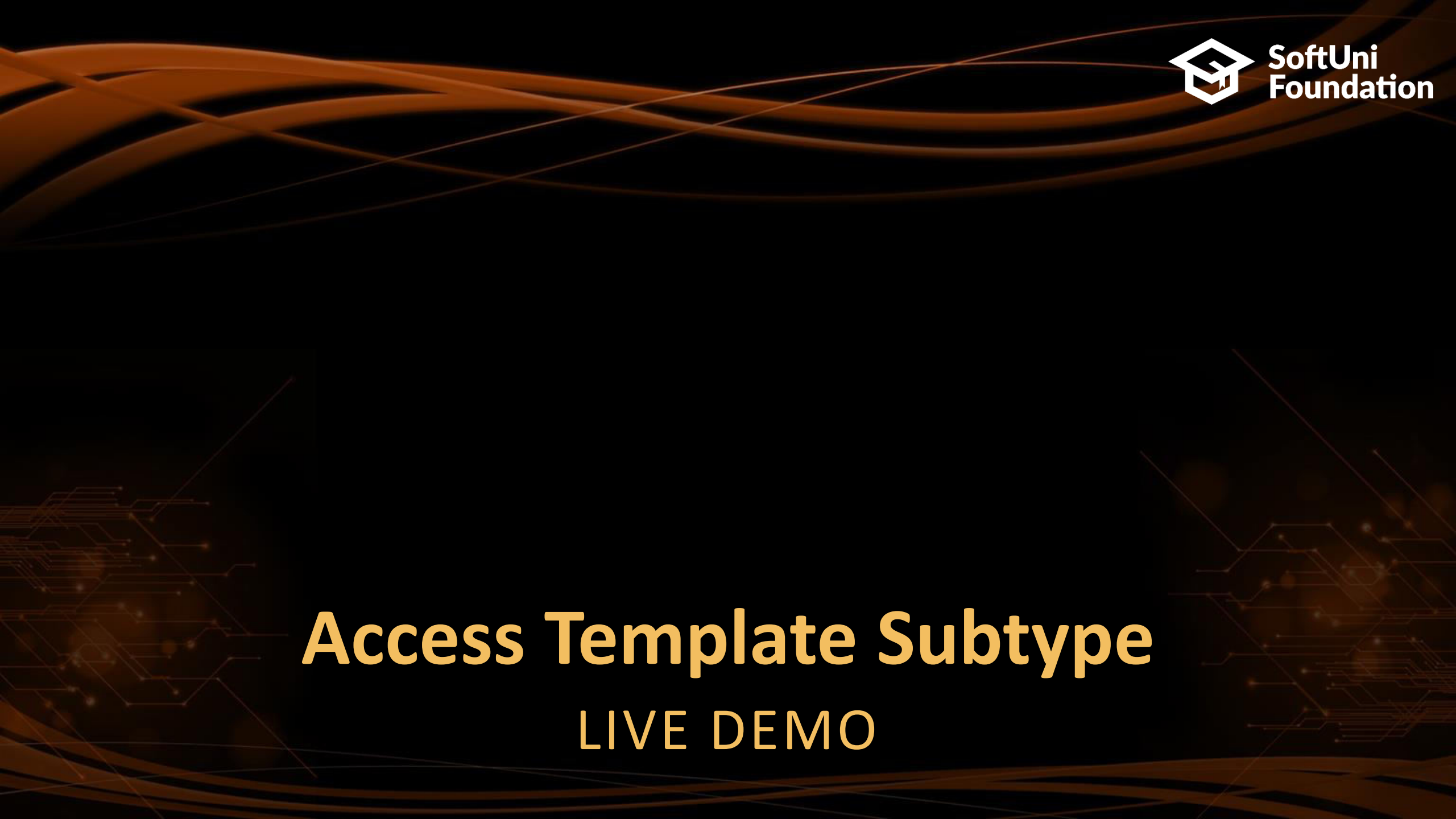
Access Template Subtype – typename

SoftUni Foundation

- operator:: to access class inside T, prefix with typename
  - typename T::SubClassName subClassObject;
  - Can also use class instead of typename[1]

```
template<typename Container> void print(Container container) {
  typename Container::iterator i;
  for (i = container.begin(); i != container.end(); i++) {
    std::cout << *i << " ";
  }
  std::cout << std::endl;
}
```

- [1] class was originally used, but caused confusion, so typename was added. But class still remains valid


SoftUni
Foundation
Access Template Subtype
LIVE DEMO

Template Specialization

- Can define different behavior for specific template value

```
template<typename T> void print(T container) {
  typename T::iterator i;
  ...
}
template<> void print<string>(string container) {
  cout << container << endl;
}
```

  - *Think "overloads" for templates*

```
vector<int> numbers{ 1, 2, 3 }; string s = "hello specialization";
print(numbers); // prints "1 2 3 "
print(s); // prints "hello specialization"
```


SoftUni
Foundation
Template Specialization
LIVE DEMO

C++ Template Specifics

SoftUni Foundation

- Template declaration and definition must be in the SAME file!
  - CAN'T separate class template in `.h` and `.cpp` files
- Templates can be constant values
  - `template<int N>` - use N as a constant in function/class
- Templates don't exist in code until used
  - When used, compiler copies template with the type
- Template metaprogramming
  - Uses templates to generate results compile-time (e.g. Fibonacci)

Summary

- Preprocessor directives
  - Execute before compilation and edit code
  - Macros, Inclusions & Header-guards
- Code is often split into header & definition/source files
  - The `.h` contains declarations, `.cpp` – implementation/definition
  - IDEs usually compile & link all `.cpp` files
- Templates allow using same code for different types
  - Functions and classes can be templates

Code Organization & C++ Templates

SoftUni Foundation

Questions?