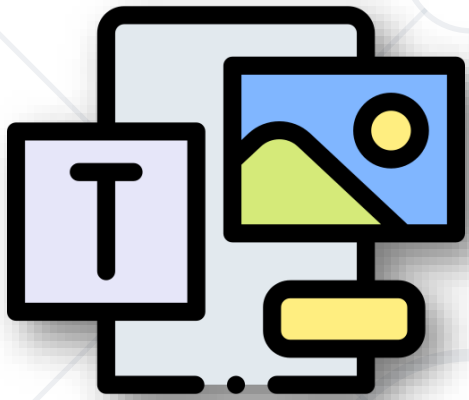


# Code Organization and Templates

Preprocessors, Header and Source Files, Templated Code



**SoftUni Team**  
**Technical Trainers**



**SoftUni**



**Software University**

<https://softuni.bg>

sli.do

**#cpp-oop**

1. Preprocessor Directives
2. Separating Declaration and Implementation
3. Header and Source Files
4. Build
5. Templates
  - Function and Class Templates





# Preprocessor Directives

`#include, #define, #if...`

# Preprocessor Directives

- 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



- **#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
```

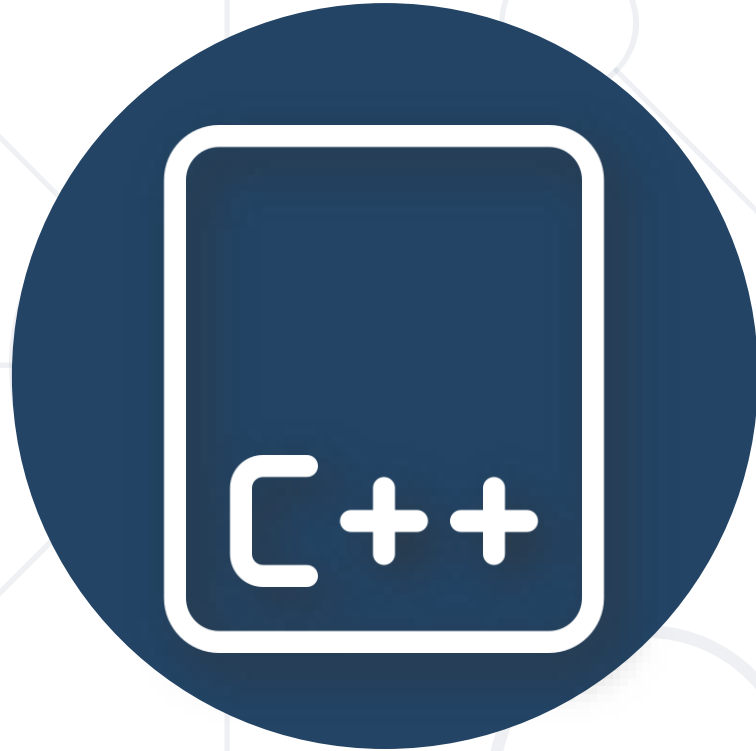
- **#define F(X) code-using-X** – macro function

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

- Similar to if-else, when condition is NOT met, code is ignored
  - **#if** and **#elif** – "else if"
  - **#else** - "closed" with **#endif**
  - **#ifdef X** – if macro **X** is defined
  - **#ifndef** – if macro **X** is NOT defined
- **Header guards** – avoid **#include**-ing code multiple times

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

```
#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
```



# **Separating Declaration and Implementation**



# Separating Declaration and Implementation

- Allows separate **declaration** and **implementation**
- For **functions**, **methods**, **operators**, **classes**
- Class members' implementation is **often separated**
  - Cleaner **view of class** "interface"
  - Sometimes **necessary** (static fields or stream operators)
  - Code needed for the implementation is **not included in the header**
  - Allows separate build objects for **faster** rebuilds and **dynamic** linking



# Why Separate?

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

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

int getId() const {
    return this->id;
}

string getName() const {
    return this->name;
} // more on the next slide ->
```

# Why Separate?

```
vector<pair<char, char> > getEmployees() const {    // PART TWO
    return this->employees;
}

string toString() {
    ostringstream stream;
    stream << id << " " << name << " ";

    for (int i = 0; i < employees.size(); i++) {
        auto initials = employees[i];
        stream << initials.first << initials.second;
        if (i < employees.size() - 1) {
            stream << " ";
        }
    }
    return stream.str();
} // more on the next slide ->
```

# Why Separate?

```
bool operator==(const Company& other) const {    // PART THREE
    return this->id == other.id;
}

string operator+(const string& s) {
    return this->toString() + s;
}

Company& operator+=(const pair<char, char>& employee) {
    this->employees.push_back(employee);

    return *this;
}

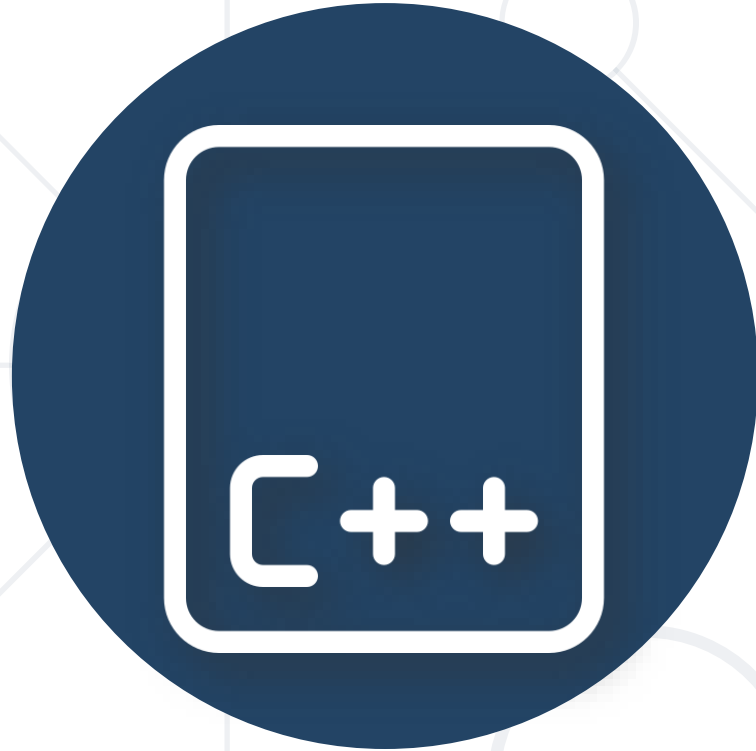
}; // end
```

# Separating Declarations and Implementation

```
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);  
};
```

- Syntax same as a member inside a class, however:
  - Prefixed with namespaces and class name, joined by **operator:**

```
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;
}
...
```



# Header and Source Files

# Header and Source Files

- **Header files** – mostly declarations
  - Use **#pragma once** to avoid multi-inclusion
  - Extension – **.h/.hpp/.h++**
- **Source files** – implements header declarations
  - Usually **1** per header, **#include** the header
  - Extension – **.cpp**





# Header Files - Company.h

```
#pragma once
```

```
#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;
```

```
};
```

# Source Files - Company.cpp

```
#include "Company.h"
```

```
Company::Company(int id, string name,  
    vector<pair<char, char> > employees)  
    : id(id), name(name), employees(employees) {}
```

```
...  
int Company::getId() const {  
    return this->id;  
}
```

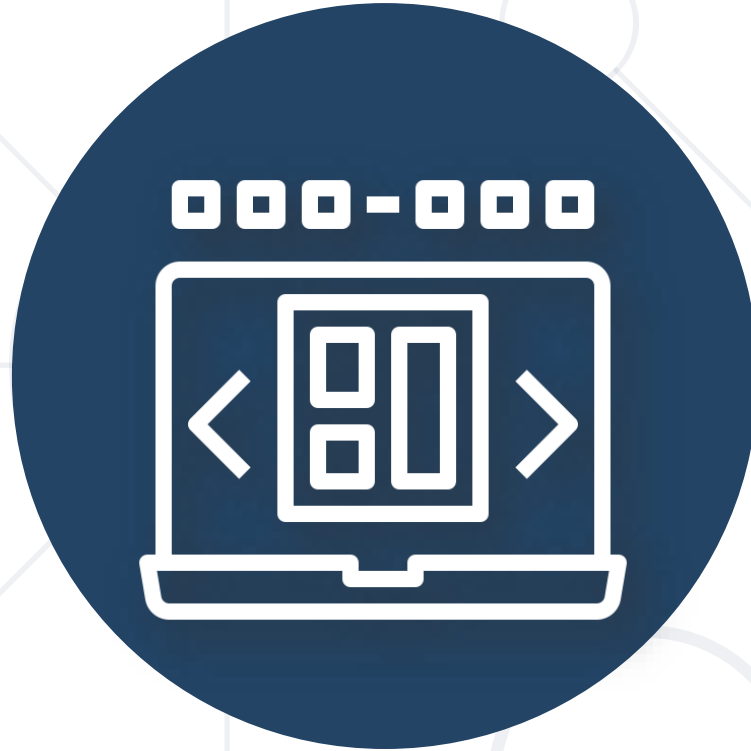
```
...  
bool Company::operator==(  
    const Company& other) const {  
    return this->id == other.id;  
}  
...
```



**Build**

- **Compilation unit** – a file (usually **.cpp**) the compiler works on
- Build process 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

- Different approaches to building a multi-source "Project"
  - Single **.cpp**, implementation in headers – compile the **.cpp**
  - Only declaration in **.h**, multiple **.cpp** – compile and link all **.cpp**
  - Mixed – some **.h** contain implementation – same as above
- Compiler needs instructions on which files to compile
  - IDEs automate the process – compile and link all **.cpp** files



# Templates

Generalizing Functions and Classes for any Type

- Algorithms rarely operate on a single data type
- E.g. calculate what percentage **a** is of **b**
  - **a out of b ==  $a * 100 / b$**
  - **1 out of 4 ==  $1 * 100 / 4 == 25\%$**
  - **1.5 out of 3 ==  $1.5 * 100 / 3 == 50\%$**
  - **$\frac{1}{4}$  out of  $\frac{1}{2}$  ==  $\frac{1}{4} * 100 / \frac{1}{2} == 25 / \frac{1}{2} == 50\%$**

- 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*` and `operator/`
- **Templates**
  - Declare function or class with a "placeholder" type
  - Can use with **different types**
  - Types should support the **used methods / operators**



# Function Templates

- **template<typename T>** – makes **T** a placeholder type
  - Can have multiple placeholders

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

- Applies only to function/class directly after it

```
template<class KeyType, class ValueType>
void printPair(const KeyType& a, const ValueType& b)
{
    cout << [ << a << ] << "->" << b << endl;
}
```

- **template<class T>** has same meaning



- Call like normal function

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

- If type doesn't support operations in function

```
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)**

# Class Templates

- Classes can receive templates to use as data types

- **vector<T>**, **list<T>**, **map<K, V>**

- Defining class template – same as with function

```
template<typename T> class ClassName { ... }
```

- Can use **T** for fields, methods, etc. – like any actual type
- Using a class template

```
ClassName<int> a;  
ClassName<Fraction> b;
```



- 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
```

- **operator::** to access class inside T, prefix with **typename**

```
typename T::SubClassName subClassObject;
```

- Can also use **class** instead of **typename**

```
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;
}
```

- 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;
}
```

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

- Template declaration and definition **must be in the same file**
  - Can not separate class template in **.h** and **.cpp** files
- Template parameters **can be constant values**
  - **template<int N>** - use N as a constant in function/class
- Templates are **not instantiated in code until used**
  - When used, compiler copies template with the type
- Template **metaprogramming**
  - Uses templates to generate results compile-time

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

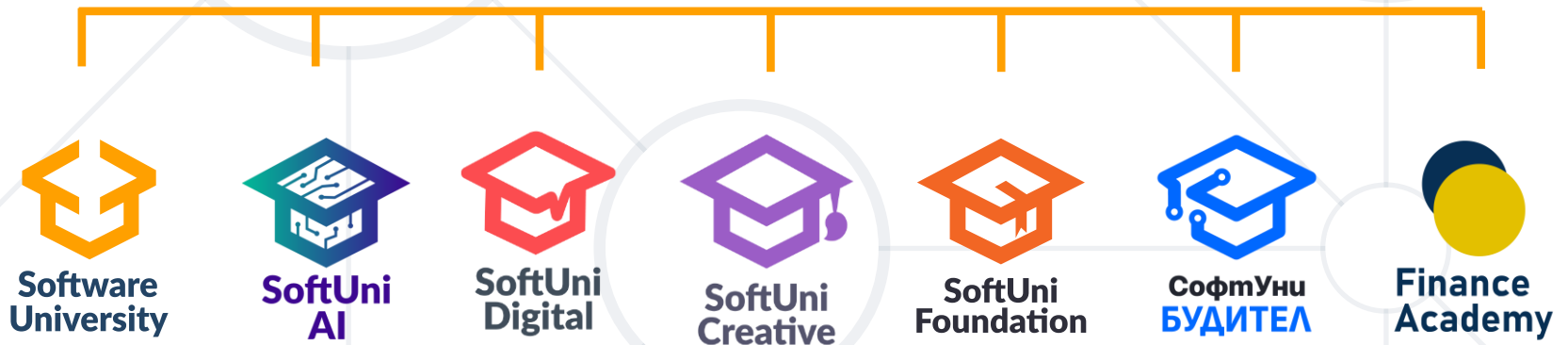




# Questions?



SoftUni



# SoftUni Diamond Partners



**SUPER  
HOSTING  
.BG**



**INDEAVR**  
Serving the high achievers



**VIVACOM**

- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)



- 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>

