

Objects and Classes Advanced

Advanced Class Members



SoftUni Team
Technical Trainers



SoftUni



Software University

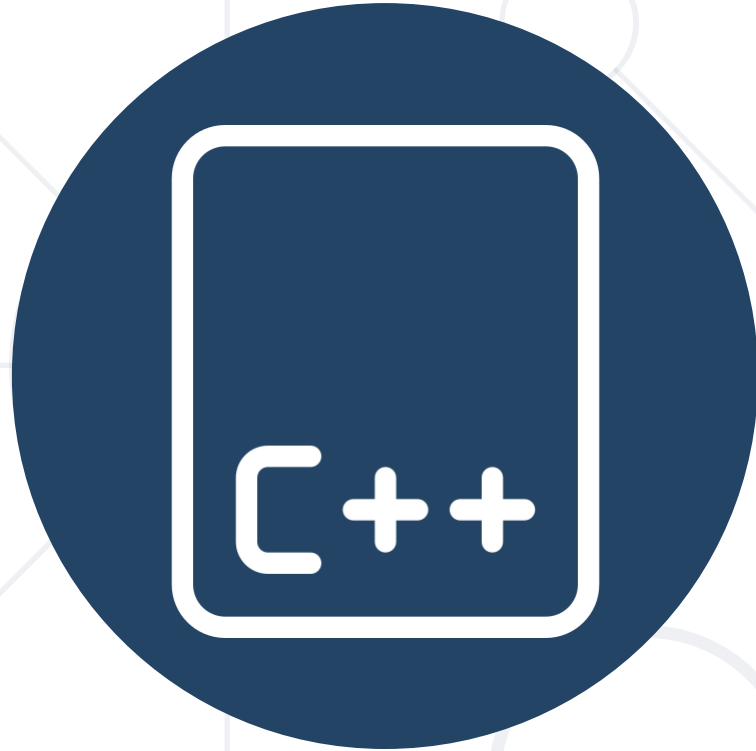
<https://softuni.bg>

sli.do

#cpp-oop

1. Namespaces
2. Members
 - **static**, **const** and **mutable**
3. Friend Functions and Classes
4. Operator Overloading
 - Modifying STL Behavior





Namespaces

Organizing Code into Named Groups

- Named groups of variables, functions, classes

```
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", ... };  
    }  
}
```

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

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

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

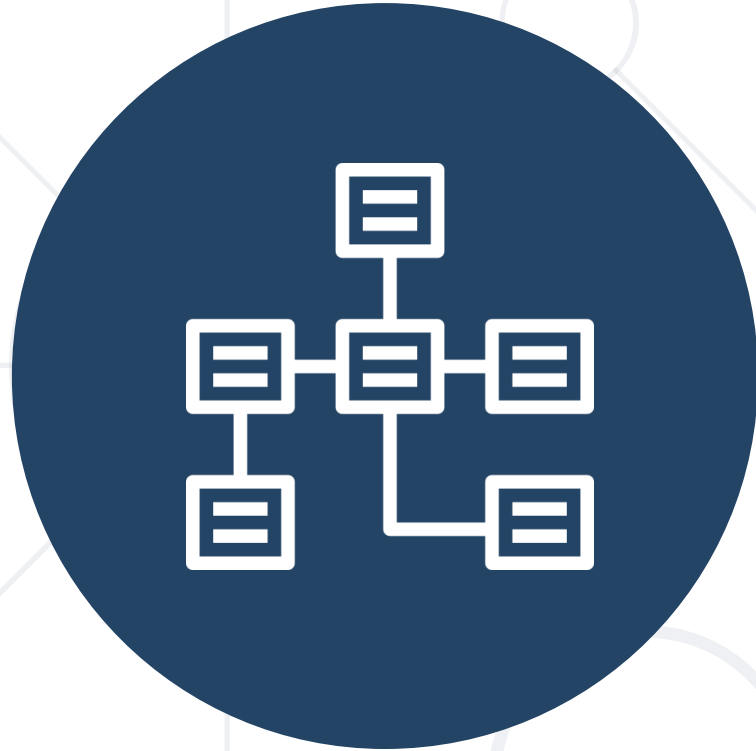
```
int main() {  
    using namespace SoftUni::CppFundamentals;  
    for (const std::string& s : lectures)  
        std::cout << s << std::endl;  
}
```

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
```





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 };
    ...
}
```

- Exist in the class, not in each object
- Defined and initialized outside 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  
    ...  
}
```

- A static data member may be declared **inline**
- **Inline static data** member can be defined directly in the class
- It does not need an out-of-class definition (**since C++ 17**)

```
class Company {  
public:  
    inline static int CREATED_COMPANIES = 0;  
};
```

- 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{ "GOOGLINC.", ... };  
cout << c->id << endl;           // prints GOOGLINC.  
c->id = "thiswontcompile";       // compilation error
```

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;  
    }  
};
```

Return
type

Method name

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
```

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

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



Friend Functions and Classes

Sharing Access to Private Members

The Friend Keyword

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



```
friend Type functionName();
```

Defining a friend function

```
friend className;
```

Defining a friend class

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

The Friend Keyword 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);
```



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" }
```

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

- 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) {  
    ostreamstream 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;
```



Comparison Operator Overload

- 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
```

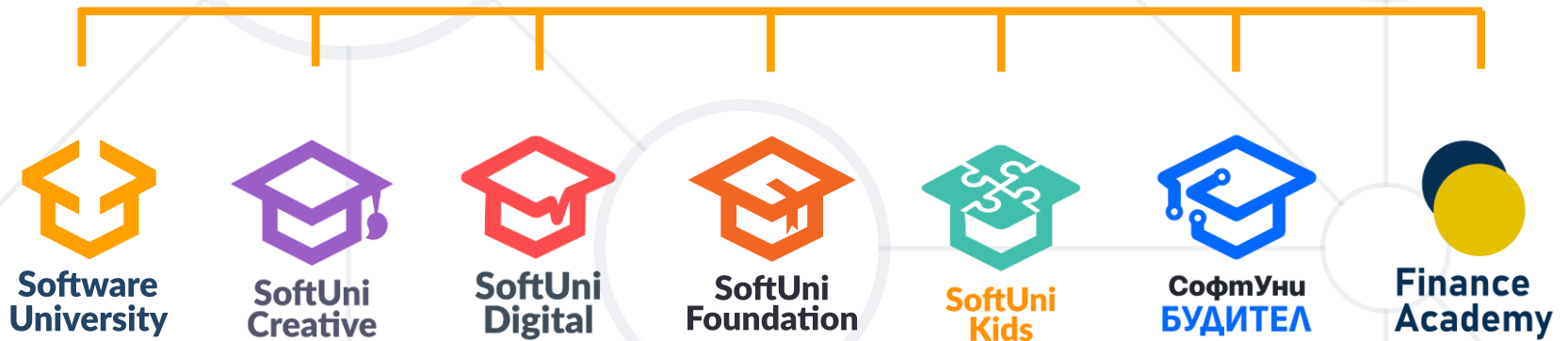
- **Namespaces** organize code and avoid name conflicts
- Static members are "global" class members
- **Friend classes / functions** can access private members
- **Operators** are just methods with special names
 - Can be overloaded by user code
 - Non-member overloads allow overloads for any class
-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



Questions?



SoftUni



SoftUni Diamond Partners



- 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



- 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>

