

Objects and Classes

Enums, Objects, Class Definition & Members



SoftUni

SoftUni Team
Technical Trainers



Software University

<https://softuni.bg>

1. Special Types – **typedef**, **enum**
2. Representing the Real World – OOP
3. Classes and Objects
4. Defining Classes and Using Objects
 - Fields, Methods, Constructors
5. Access Modifiers
 - **public** and **private**
 - Getters and Setters





sli.do

#cpp-oop



Special Types

Typedef and Enumerations

Typedef

- **Typedefs** allows creating aliases for existing types
 - Should be used within the problem's context
 - E.g.: `map<string, vector<int> >` to **StudentScores**
- Syntax: similar to declaring a variable, place **typedef** in declaration

```
typedef string tenStrings[10];  
tenStrings words = { "the", "quick", "brown", "fox",  
"jumps", "over", "the", "lazy", "dog", "!" };
```

```
typedef map<string, vector<int> > StudentScores;  
StudentScores judgeAssignment2Scores;
```





Special Types

LIVE DEMO



Enumerations

- Enumerations contain a fixed list of special constant values
 - i.e. all possible values are known and can be written in code
 - Have some semantic meaning in the real world
- E.g. standard colors – red, green, blue, yellow, orange, etc.
- E.g. currencies – USD, BGN, GBP, etc.
- E.g. automobile fuel type – Petrol, Diesel, Electricity

- C++ has two enumeration types – **enum** and **enum class**

enum defines a list of
named constant integers

```
enum color { red, blue, pink };  
color eyeColor = blue;  
// same as color eyeColor = 1;
```

enum class defines a
new data type

```
enum class Color { red, blue, pink };  
Color eyeColor = Color::blue;  
/* Color eyeColor = 1 - invalid,  
compile time error */
```



Enumerations

LIVE DEMO



Representing the Real World

Object-Oriented Programming

- So far our data types were essentially "just numbers"
 - **int**, **float**, and **double** are obviously numbers
 - **char** is also a number, although treated like a symbol
 - arrays of the above types are still just numerical data
- The physical world CAN be represented entirely by numbers
 - Computers work with **1**s and **0**s anyway
- What matters is not the data itself, but how you interpret it

Representing the Real World in Code

- In the real world, we usually talk about "**objects**"
 - e.g.: **Peter, United Kingdom, Zhivko's Car**
 - Objects have attributes/properties, e.g.: **age, population, fuel**
 - Objects can sometimes do things, e.g.: **talk, leave EU, break**
- There are usually multiple objects of the same **type/class**
 - **Peter, Churchill, Abd al Hakim, and Hanyu** are all people
 - **United Kingdom, India, and Egypt** are all countries
- Object-oriented programming focuses on such **classes & objects**





Object-Oriented Programming

OOP Concept and C++ OOP

Object-Oriented Programming

- Introduces ways to group data into user-defined data types
 - E.g. a **Person** type, a **Country** type, a **Car** type, etc.
 - Variables defined in a user-defined type are called "**fields**"
- Such types are called "**classes**"
- Variables with such a type are called "**objects**"
- In addition to data, we can add functions to a **class**
 - Functions in a class are called **methods**





Objects and Classes

- In programming, **classes** provide the structure for **objects**
- User-defined data types
 - Act as a **template** for **objects** of the same type
- Definition contains the class "**members**":
 - Fields, Methods, Constructors, Destructors
- One class may have many instances (objects)

- Any variable of a **class**-defined data type
 - Operator **.** (dot) is used for accessing members of an object
- The **instance** is the object itself, which is created at runtime
- All instances have common **behavior**



Defining Classes

Defining Classes

- Specification of a given type of objects from the real world



Key word

```
class Dice {  
    access_modifier:  
        members...  
    access_modifier:  
        ...  
};
```

Class name

Class body

Don't forget the ; after definition

- Members of a class are variables and functions
- Access modifiers - where members can be accessed from

Defining Classes – Example

- Person class
 - Age, Name, Height
- For now, ignore access modifiers – just place **public:** at the beginning
- Notice we can use data types that are themselves objects of classes
 - name here is an object of the STL class string

Access
modifier

```
#include<string>
class Person {
public:
    std::string name;
    int age;
    double height;
};

int main(){
    Person p;
    return 0;
};
```



Defining C++ Classes

LIVE DEMO



Using Objects

Using Objects

- Creating a variable of a **class** data type
- Objects follow the same rules as normal variables
 - Can be passed as a copy to a function or by reference with **&**
 - Can be put into arrays, vectors, etc.
- Accessing members through an **operator.** (dot)
 - For access through an iterator, we use the **operator->**



Methods – Example

```
class Person {  
    class Body {  
    public:  
        double height;  
        double weight;  
    };  
    public:  
        string name;  
        int age;  
        Body body;  
};
```

```
Person person;  
person.name = "George Georgiev";  
person.age = 25;  
person.body.height = 1.82;  
person.body.weight = 87;
```

```
Person otherPerson;  
person.name = "Ana Ivanova";  
person.age = 42;  
person.body.height = 1.6;  
person.body.weight = 54;
```



Using C++ Objects

LIVE DEMO



Constructors

- **Constructors** initialize objects of a class
 - Follow same rules as functions, but without a **return** type
 - Can have overloads, default parameters, etc.

```
class Person {  
    string name; int age = 0; double height = 0;  
    Person(string pName, int pAge, double pHeight) {  
        name = pName;  
        age = pAge;  
        height = pHeight;  
    }  
};
```

Class name

Body

Parameters

- Can be called on declaration directly

```
Person peter("Peter Brown", 31, 1.69);
```

- Since C++11 can be called with `{ }` brackets too:

```
Person peter{"Peter Brown", 31, 1.69};
```

- Can be used to create objects to pass to a variable/function:

```
Person peter{"Peter Brown", 31, 1.69};  
Person ivan = Person("Ivan Ivanov", 12, 1.52));  
vector<Person> people;  
people.push_back(Person("Ana Ivanova", 43, 1.60));
```

- A constructor without parameters is a default constructor

```
Person() { name = "<unknown>"; }
```

- Called when no other constructor is called

```
Person p; Person people[3];
```

- Auto-generated if class has no other constructors
- If no default constructor for e.g. **Person**:
 - Default creation ~~Person p;~~ and ~~Person p[3];~~ won't compile
 - Some structures e.g.: ~~vector<Person> people;~~ won't compile



Constructors

LIVE DEMO

- What values will **p** have for its fields?
 - a) name empty, age==0, height==0
 - b) name=="Ary O'usure", age==42, height==1.3
 - c) name empty, age==42, height==1.3
 - d) name=="Ary O'usure", age==0, height==0
 - e) There will be a compilation error

```
class Person {  
    public:  
        string name;  
        int age = 0;  
        double height = 0;  
        Person(string name,  
                int age,  
                double height) {  
            name = name;  
            age = age;  
            height = height;  
        }  
};  
...  
Person p("Ary O'usure",  
        42, 1.3);
```


PITFALL: HIDING FIELDS WITH PARAMETERS

The parameter names match the field names here.

When there is such a conflict, the “more-local” variable hides the “less-local” variable.

So, the constructor in this case will assign the parameters with their own values and not see the fields at all.



**NAMES CONSTRUCTOR PARAMETERS
SAME AS FIELDS FOR CLARITY**

**ENDS UP HIDING AND NOT ASSIGNING
FIELDS BY FORGETTING TO USE THIS->**



The this Pointer

LIVE DEMO

- C++ gives us **this** pointer to explicitly access class members
- **this** points to whatever the current object is
- Very useful in any method where parameters match the fields

```
Person(string name, int age, double height) {  
    this->name = name;  
    this->age = age;  
    this->height = height;  
};
```

- There is a convention to always use **this**, even if not needed



Pitfall: Hiding Fields
Solution: Using `this->`
LIVE DEMO



Constructor_INITIALIZER List

- Constructor body is **always** executed **after** a member creation
- C++ constructors are typically written with initializer lists:

```
ClassName(parameters) :  
    member1(member1Parameters),  
    ...  
    memberN(memberNParameters) {  
}
```

- Executes before the body
- If a member is omitted, it is default-constructed (if possible)
- This syntax is also immune to the member-hiding problem



Constructor_INITIALIZER List

LIVE DEMO



Simple Methods

- Methods are functions declared inside a class
 - Follow the same rules as normal functions
 - Compiler knows which methods belong to which class
 - E.g.: `size()`, `begin()`, `sort()` are methods in the `list` class
- Methods can access class fields and other members directly
 - Can read and write fields, call other methods, etc.
 - Can use `this->` to explicitly refer to members

A **method** for printing information

```
void printPersonInfo() {  
    cout << "name: " << this->name  
    << ", age: " << this->age  
    << ", height: " << this->body.height  
    << ", weight: " << this->body.weight  
    << endl;  
}
```

A **method** for aging up

```
void makePersonOlder(int years) {  
    this->age += years;  
}
```



Simple Methods

LIVE DEMO

- Should a **Person** know about and access the console?
 - Low cohesion – the class knows more than its name suggests
- Should a **Person** directly access a **Body**?
 - *No, that's not what I meant!... But.. if you're interested...*
 - Bad encapsulation & high coupling – class has access to implementation details of another class
- Do we need "Person" in method names on a **Person** class?
 - They all work on the **Person** class, no need to write it everywhere

- This is somewhat better

```
class Person {  
    ...  
    void makeOlder(int years) {  
        this->age += years;  
    }  
    ...  
    string getInfo() {  
        ostringstream info;  
        info << "name: " << this->name  
            << ", age: " << this->age  
            << ", " << this->body.getInfo();  
        return info.str();  
    }  
};
```

Methods – Refactoring for better Quality

```
class Person {  
    class Body {  
        ...  
        string getInfo() {  
            ostringstream info;  
            info << "height: " << this->height  
                << ", weight: " << this->weight;  
            return info.str();  
        }  
    };  
    ...  
};
```



Refactoring Methods

LIVE DEMO



Access Modifiers

Encapsulation, Getters and Setters

Encapsulation (1)

- Do you see a problem in the following code?
 - We're updating the **radius**, but that doesn't update the **area**

```
const double PI = 3.14;

class Circle {
public:
    double radius;
    double area;

    circle(double radius) :
        radius(radius),
        area(radius * radius * PI) {}
};
```

```
int main() {
    Circle c(10);
    c.radius = 20;
    cout << c.area << endl;
    return 0
}
```

- Encapsulation – hiding internal state & operations from outside
 - And providing a controlled interface for interactions
 - *You usually don't have direct access to a car's engine - but you have pedals, a gear lever, etc.*
- A class should keep its internal state correct
 - Hide its members so external code doesn't use them incorrectly
 - Have public methods that access members correctly
- Encapsulation makes code simpler
 - *You don't need to know how a specific class works to use it*

public and private



- Access modifiers
 - Control whether external code has member access
 - **public** – access both by code "outside" & "inside" the class
 - **private** – access ONLY to code "inside" the class
- Every member has that access after an access modifier
 - Until another modifier is encountered
 - Access modifiers can set the access for multiple members

Adding Encapsulation in C++



- Let's encapsulate our **Circle**'s member fields:
 - **private** access **radius** & **area**
 - **public** constructor
 - Now we can create **Circles**, but external code can't access **radius** and **area**
- But how can we print the **area** now or change the **radius**?
 - We still need to add public methods for interaction

Adding Encapsulation in C++

```
class Circle {  
private:  
    double radius;  
    double area;  
public:  
    Circle(double radius) :  
        radius(radius),  
        area(radius * radius * PI) {}  
};
```

- "Getter" & "Setter" – common names for some **specific methods**
- Getter – **public** method returning a value of **private** member

```
double getArea() { return this->area; }
```

- Can sometimes calculate what to return (e.g. calculate **area**)
- Setter – **public** method assigning a value of **private** member
- Keeps internal state correct while giving access to external code

```
void setRadius(double radius) {  
    this->radius = radius;  
    this->area = radius * radius * PI;  
}
```

Getters and Setters – Example (1)

```
class Circle {  
private:  
    double radius;  
    double area;  
public:  
    circle(double radius) :  
        radius(radius),  
        area(radius * radius * PI) {}  
  
    double getRadius() { return this->radius; }  
    double getArea() { return this->area; }  
    void setRadius(double radius) {  
        this->radius = radius;  
        this->area = radius * radius * PI;  
    }  
};
```

Getters and Setters – Example (2)

```
int main() {  
  
    Circle c(10);  
    cout << c.getArea() << endl;  
  
    c.setRaduis(20)  
    cout << c.getArea() << endl;  
  
    return 0;  
}
```




C++ struct vs class

- In C++ **struct** and **class** mean exactly the same thing, except:
 - **class** by default uses **private:** at the start
 - **struct** by default uses **public:** at the start
 - i.e. **class** with the **public** at the start is the same as a **struct**
- The C++ community usually prefers class to actual classes
 - **struct** is sometimes used for *Plain Old Data* (POD) objects
 - no constructors, no methods, etc., only public-access fields

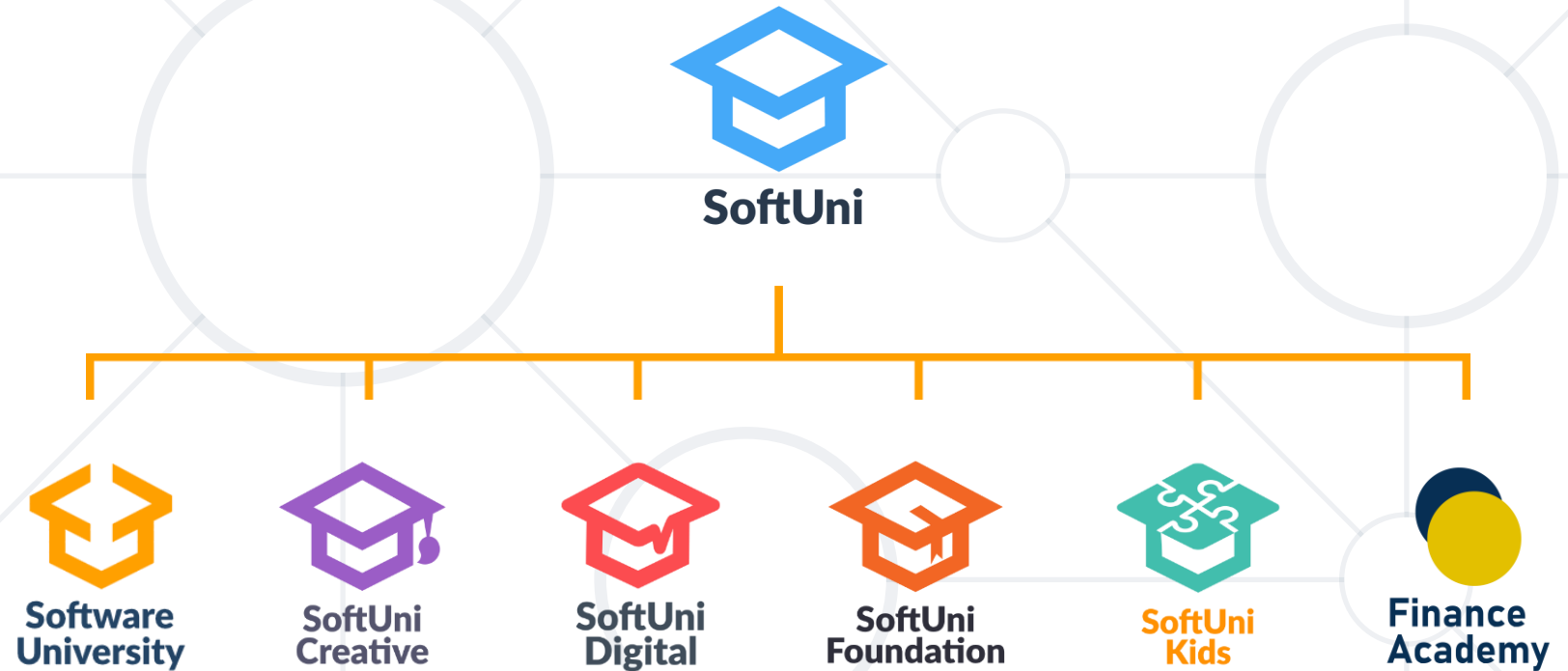
```
class C {  
    public:  
};
```

```
struct C {  
};
```

- Typedefs allow shortening code by creating **type aliases**
- Enumerations are types with user-defined values
- Classes define templates for objects
 - **Fields, Methods, Constructors, Destructors**
- Objects is an **instance** of a class
- Classes should **encapsulate** their internal state
 - And provide **methods** for **interaction**



Questions?



SoftUni Diamond Partners

**SUPER
HOSTING
.BG**



**Coca-Cola HBC
Bulgaria**



POKERSTARS
POKER | CASINO | SPORTS
a Flutter International brand

INDEAVR
Serving the high achievers



AMBITIONED

 **DRAFT
KINGS**



**SOFTWARE
GROUP**

createX



Postbank
Решения за твоето утре

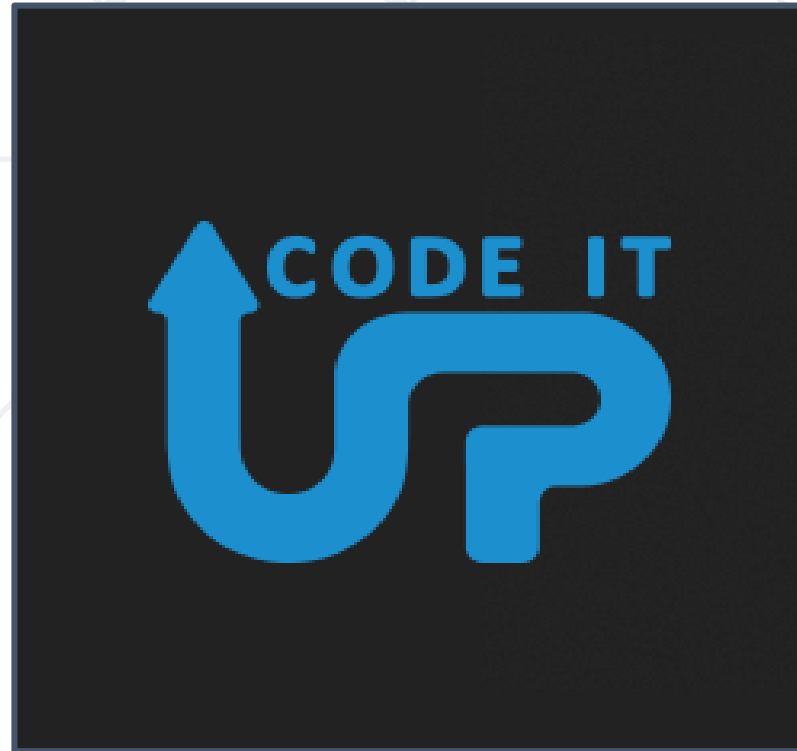


BOSCH

DXC
TECHNOLOGY



SmartIT



- 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>



- 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

