Objects and Classes

Enums, Objects, Class Definition and Members

SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

Have a Question?

Software University

sli.do

#cpp-oop

Table of Contents

Special Types

Typedef and Enumerations

Typedef

- `Typedef` allows creating aliases for existing types
  - Should be used within the problem's context
  - `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;
```

Enumerations

Enumerations

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

Enumerations

- 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 */
```

Representing the Real World

Object-Oriented Programming

Representing the Real World in Code

Software University

- 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**"
  - Example: **Peter**, **United Kingdom**, **Zhivko's Car**
  - Objects have attributes / properties: **age**, **population**, **fuel**
  - Objects can sometimes do things: **talk**, **leave EU**, **break**

Representing the Real World in Code

- There are usually multiple objects of the same **type** / **class**
  - Example: **Peter**, **Churchill**, **Abd al Hakim** and **Hanyu** are all people
  - Example: **United Kingdom**, **India** and **Egypt** are all countries
- Object-oriented programming focuses on such **classes & objects**

Object-Oriented Programming

Object-Oriented Programming

- Introduces ways to group data into user-defined data types
  - a `Person` type, a `Country` type, a `Car` type
  - Variables defined in a user-defined type are called "**fields**"
- These user-defined types are called "**classes**"
- Instances of such user-defined types are called "**objects**"
- In addition to fields, we can add functions to a **class**
  - Functions in a class are called **methods**

Objects and Classes

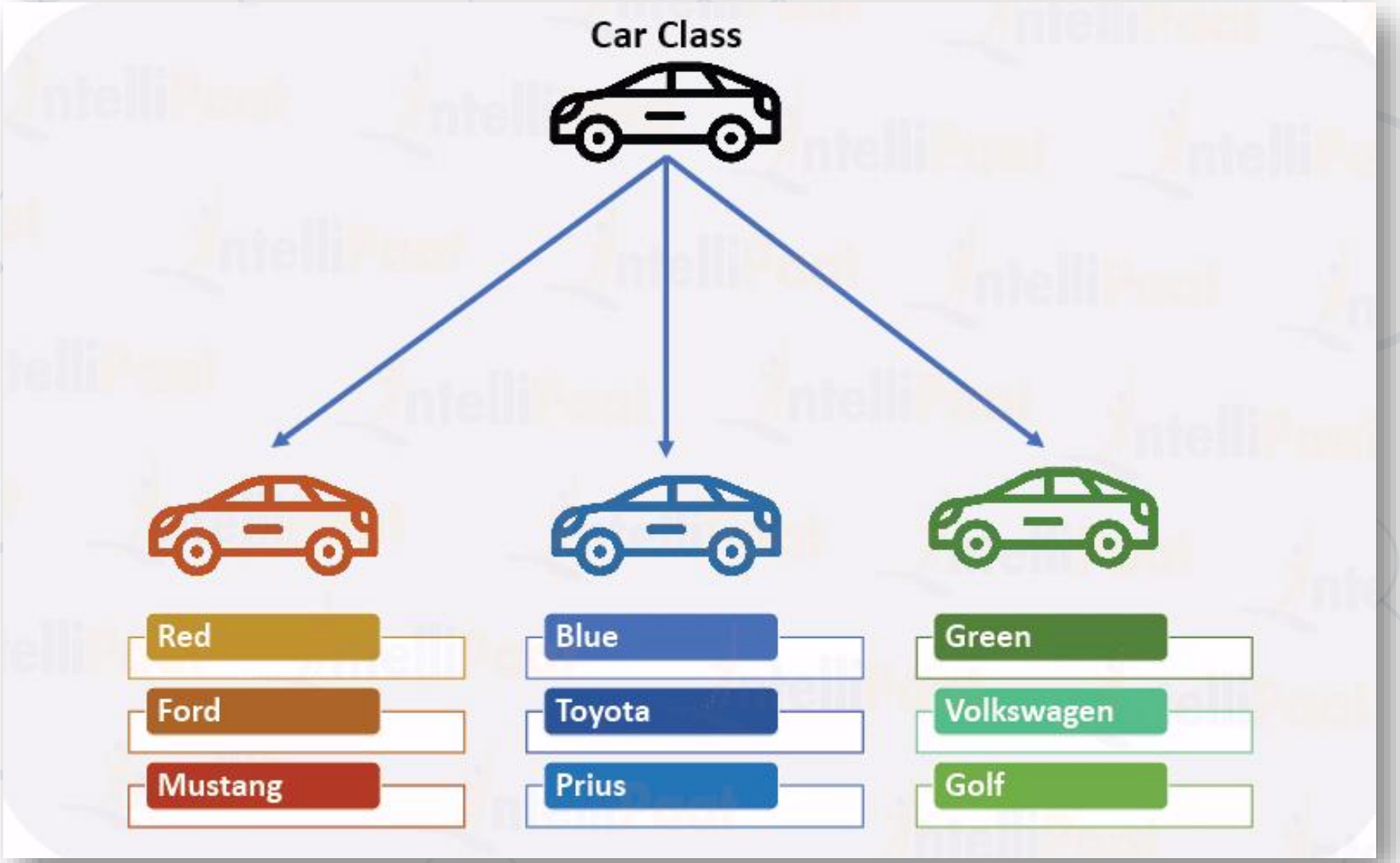
Classes

- 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, Destructor**
- One class may have many instances (objects)

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 **behaviour, defined by the class methods**

Classes and Objects



Defining Classes

Defining Classes

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

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

Class name

Key word

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;
};
```

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
- Accessing members through an **`operator.`** (dot)
  - For access through an iterator or pointer, we use the **`operator ->`**

Objects – 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;
```

Constructors

Constructors

- **Constructors** are special member functions
  - Initialize objects of a class
  - Follow same rules as functions, but without a `return` type
  - Can have overloads, default parameters

```
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

Calling Constructors

- 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));
```

Default Constructor

- 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 example `Person`:
  - Default creation ~~`Person p;`~~ and ~~`Person p[3];`~~ won't compile
  - Some structures ~~`vector<Person> people;`~~ won't compile

The this Pointer

The this Pointer

- `this` pointer explicitly access class members inside the class
- `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

Constructor Initializer List

Constructor Initializer List

- Constructor body is **always** executed **after** a member creation
- 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

Simple Methods

Methods

- Methods are functions declared inside a class
  - Follow the same rules as **normal functions**
  - Compiler knows which methods belong to which class
  - `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
  - Can use `this->` to explicitly refer to members

Methods – Example

A method for printing information

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

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

A method for aging up

Refactoring for Better Quality

Software University

- 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();
  }
};
```

Refactoring for Better Quality

Software University

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

Access Modifiers

Encapsulation, Getters and Setters

Encapsulation

- Do you see a problem in the following code?
  - We're updating the **radius**, but that does not 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
}
```

public, private, protected

- Access modifiers
  - Control whether external code has member access
  - `public` – access both by code "outside" and "inside" the class
  - `private` – access ONLY to code "inside" the class
  - `protected`
- 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

- Let's encapsulate our **`Circle`**'s member fields:
  - **`private`** access **`radius`** & **`area`**
  - **`public`** constructor
  - Now we can create **`Circle`**s, 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

Software University

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

Getters and Setters

- "Getter" and "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 * raduis * PI;
}
```

Example - Getters and Setters

Software University

```
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;
  }
};
```

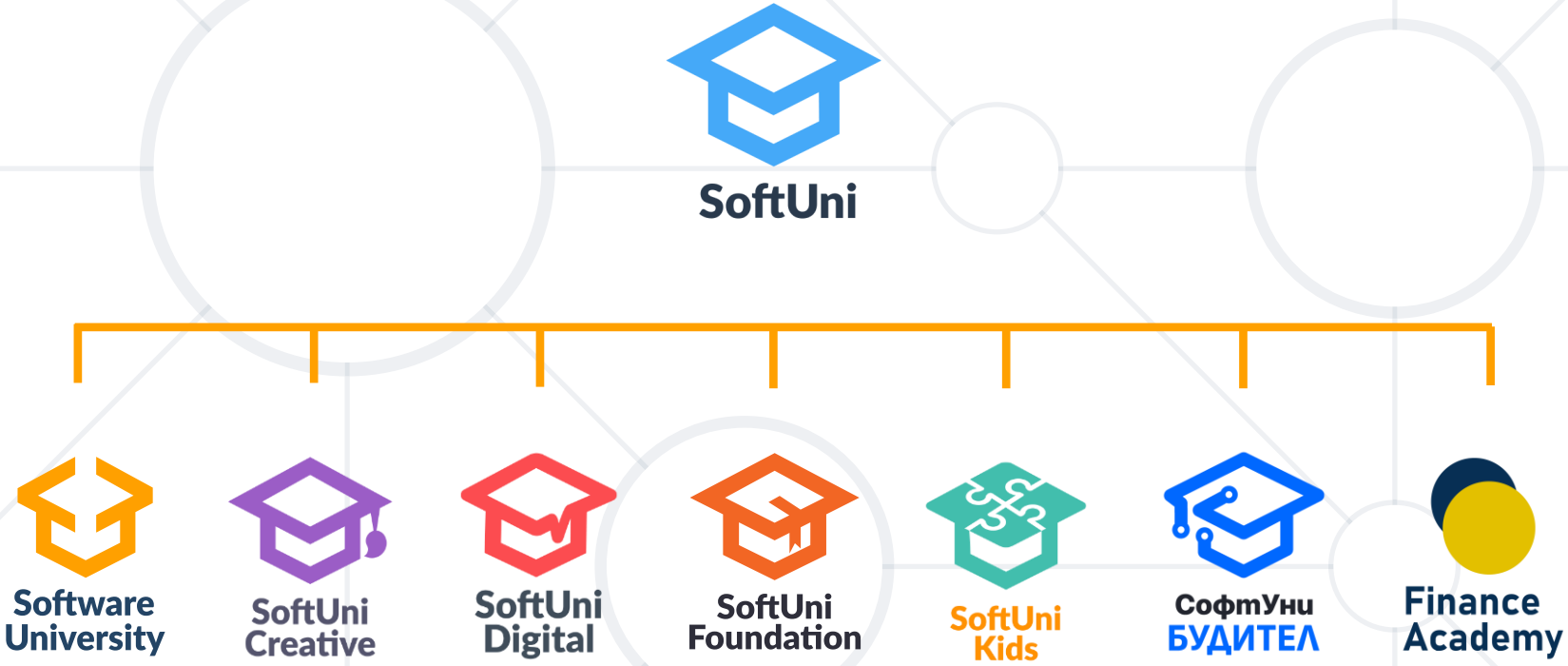
Example - Getters and Setters

```
int main() {

  Circle c(10);
  cout << c.getArea() << endl;

  c.setRaduis(20)
  cout << c.getArea() << endl;

  return 0;
}
```

Summary

- 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

Software University

SmartIT

Coca-Cola HBC Bulgaria

SUPER HOSTING .BG

createX

Postbank

top EMPLOYER България Bulgaria 2024 FOR A BETTER WORLD OF WORK

DRAFT KINGS

THE CROWN IS YOURS

VIVACOM

INDEAVR

Serving the high achievers

AMBITIONED

PHAR VISION

Trainings @ Software University (SoftUni)

- 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

License

- 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