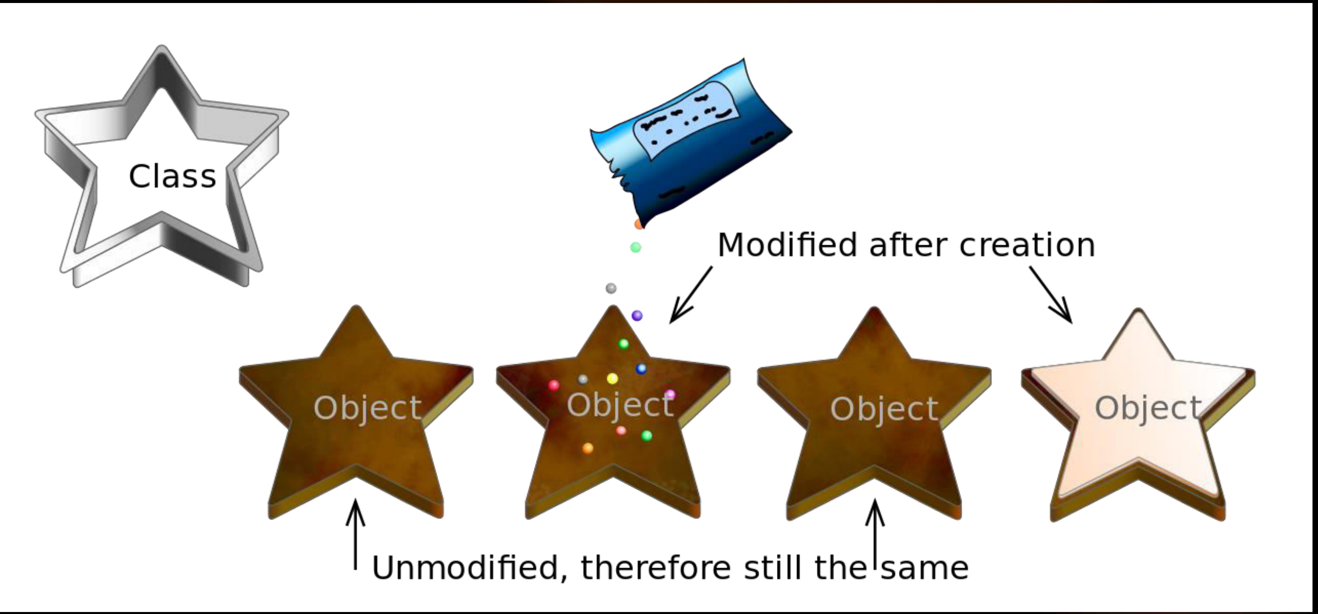
Classes and Objects

Enums, Namespaces, Objects, Class Definition & Members



SoftUni Foundation

Georgi Georgiev
A guy that knows C++

Table of Contents

Special Types

Enumerations, Typedefs

Typedefs

- Typedefs allow creating aliases for existing types
  - Should be used within the problem's context
  - E.g.: **map<string, vector<int> >** to **StudentScores**
- Syntax: like 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;
```

typedef

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++ Enumerations

SoftUni Foundation

- 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` in C++11 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

Representing the Real World in Code

- 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**, **George'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**, **Hanyu** are all people
  - **United Kingdom**, **India**, **Egypt** are all countries
- Object-oriented programming focuses on such **classes & objects**

Object-Oriented Programming

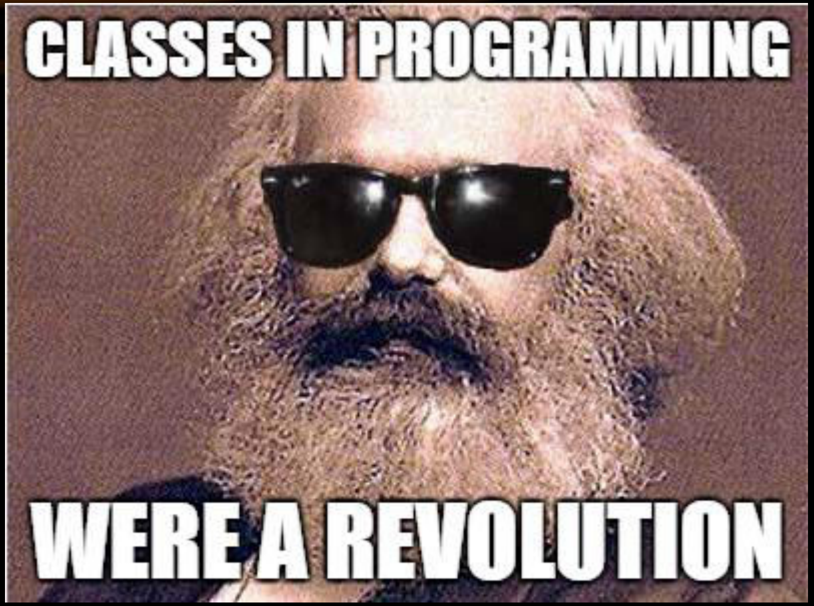
OOP Concept and C++ OOP

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 of a **class** are called "**objects**"
- In addition to data, we can add functions to a **class**
  - Functions in a class are called **methods**

Classes and Objects in C++

- Classes – user-defined data types
  - `class` keyword, followed by a name
  - Followed by definition in `{}` brackets
  - Definition contains the class "**members**" – **fields**, **methods**, **constructors**, **destructors**



- Objects – any variable of a `class`-defined data type
  - Work similar to variables of primitive data types
  - Accessing members of an object is done with `operator.` (dot)

Defining C++ Classes

- Syntax: `class` keyword followed by name you want and `{}`

```
class ClassName {
access_modifier:
      members…
access_modifier:
      …
}; //don't forget the ; after definition
```

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

Defining C++ Classes – Example

- Let's define a Person class
  - With an age, a name and a height
  - For now, ignore access modifiers – just place **`public:`** at the beginning
- Notice we can use data types which are themselves classes
  - **name** here is a **`string`**, which is also a **`class`**

```
#include<string>

class Person {
    public:
    std::string name;
    int age;
    double heightMeters;
};

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

Defining C++ Classes

LIVE DEMO

Using C++ Objects

SoftUni Foundation

- We get C++ objects by creating a variable of a `class` data type
  - We've done that before – creating `string`s, `vector`s, `map`s, etc.
- Objects follow the same rules as normal variables
  - Can be passed as copy to a function or by reference with `&`
  - Can be put into arrays, `vector`s, etc.
- Accessing members is done through `operator.` (dot)
  - If accessing through an iterator, we use the by `operator->`

Using C++ Objects – Example

- Let's create a Person object and assign their fields some values
  - NOTE: classes can be defined inside other classes – see **Body** here

```
class Person {
    class Body {
        public:
        double heightMeters;
        double weightKgs;
    };

    public:
    string name;
    int age;

    Body body;
};
```

```
Person joro;
person.name = "George Georgiev";
person.age = 25;
person.body.heightMeters = 1.82;
person.body.weightKgs = 87;

Person otherPerson;
otherPerson.name = "Lorem Ipsum";
otherPerson.age = 42;
otherPerson.body.heightMeters = 1.3;
otherPerson.body.weightKgs = 69;
```

Using C++ Objects

LIVE DEMO

C++ Simple Constructors

- **Constructors** initialize objects of a class (shortened: "**ctor**")
  - Follow same rules as functions, but without a **`return`** type
  - Can have overloads, default parameters, etc.
- Syntax: **`ClassName(`*`parameters`*`) { `*`body`*` }`**

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

C++ Constructors – Calling

- Can be called on declaration directly

```
Person ben("Ben Dover", 42, 1.69);
```

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

```
Person ben{"Ben Dover", 42, 1.69};
```

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

```
Person ben{"Ben Dover", 42, 1.69};
Person chucky = Person("Chuck Norris", 77, 1.78);
vector<Person> people;
people.push_back(Person("Joe Bishop", 23, 1.90));
```

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 e.g. **Person**:
  - Default creation ~~**Person p;**~~ and ~~**Person p[3];**~~ won't compile
  - Some structures e.g.: ~~**vector<Person> people;**~~ won't compile

Simple Constructors

LIVE DEMO

Quick Quiz TIME:

- 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

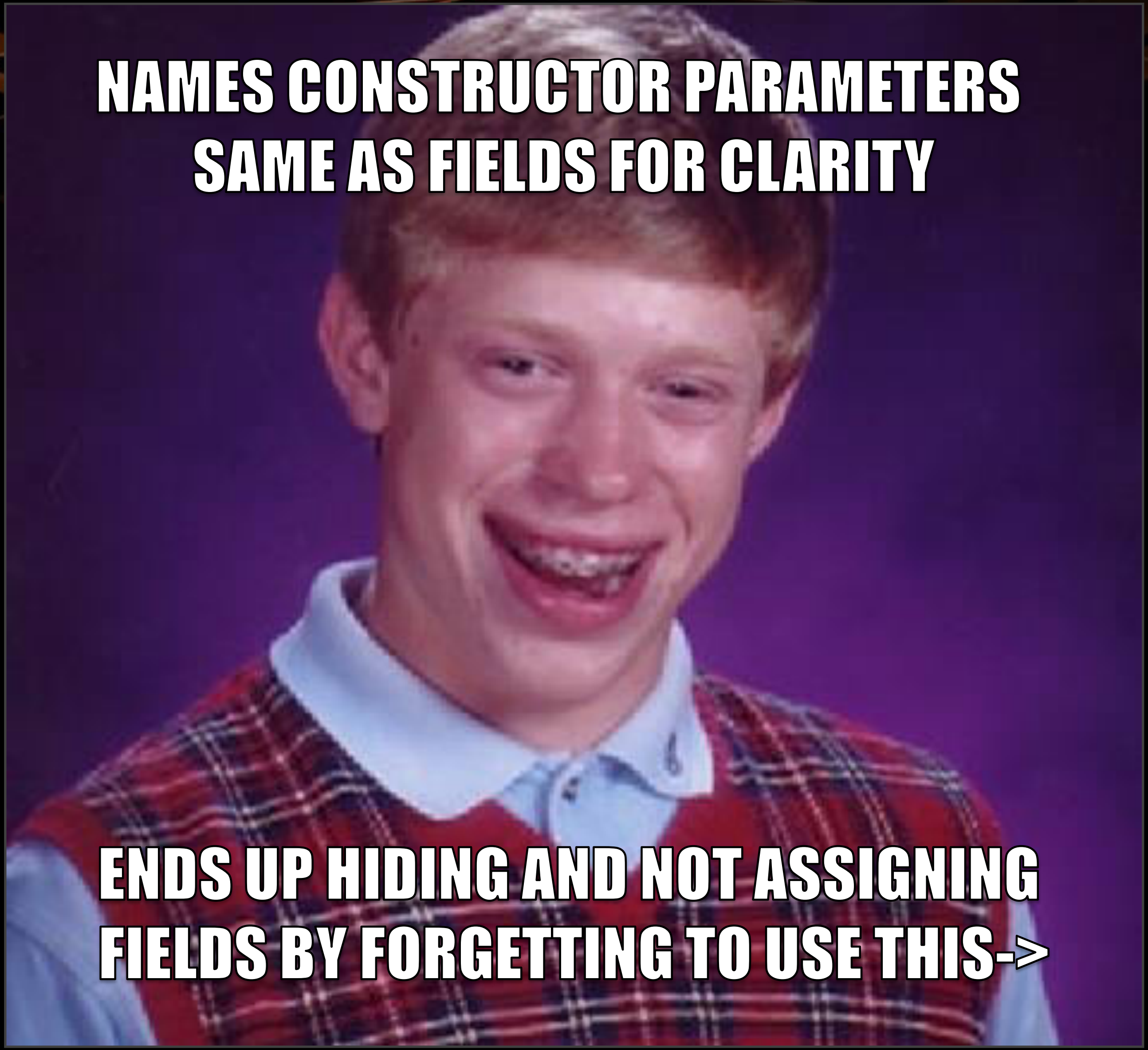
```cpp
class Person {
    public:
    string name;
    int age = 0;
    double height = 0;
    Person(string name,
           int age,
           double height) {
        name = name;
        age = age;
        height = height;
    }
};
...
Person p("Ary O'usure",
         42, 1.3);
```

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.



The this Pointer

- C++ gives us the **`this`** pointer to explicitly access class members
- **`this`** points to whatever the current object is
  - i.e. it gives you the context in which you are working
- 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

C++ Constructor Initializer List

- Constructor body is ALWAYS executed AFTER member creation
  - What if members can't default-construct (e.g. no default ctor)?
- Use initializer list – executes before body:

```
ClassName(parameters) :
      member1(member1Params), …
      memberN(memberNParams) {
}
```

  - 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

Methods

- **Methods** are simply 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

Methods – Example

SoftUni Foundation

- A function which works on an object of a class
  - Should be a member of that class *(usually)*
- Let's make some **Person** methods:
  - A method for printing info
  - A method for aging
- We add the functions inside the **Person** class

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

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

Simple Methods

LIVE DEMO

Code Quality Issues of the Last Example

SoftUni Foundation

- 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

Methods – Refactoring for better Quality

- 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();
  }
};
```

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

Refactoring Methods

LIVE DEMO

Encapsulation

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

```
const double PI = 3.1415;

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

- 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*

Encapsulation in C++ – `public` & `private`

SoftUni Foundation

- `public` and `private` are access modifiers in C++
  - 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 after an access modifier has that access
  - Until another modifier is encountered
  - i.e. access modifiers can set the access for multiple members

Adding Encapsulation in C++

SoftUni Foundation

- 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**

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

- *But how can we print the **area** now or change the **radius**?*
  - *We still need to add public methods for interaction*

Encapsulation – Getters and Setters

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

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

  - Can sometimes calculate what to return (e.g. calculate **area**)
- Setter – **public** method assigning 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;
}
```

Encapsulation with Getters & Setters

- Here's the Circle using encapsulation, getters and setters

```cpp
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;
  }
};
```

```cpp
int main() {
  Circle c(10);

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

  c.setRadius(20);

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

  return 0;
}
```

Getters and Setters

LIVE DEMO

C++ struct vs class

SoftUni Foundation

- 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 **`public`** at the start is the same as a **`struct`**

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

```cpp
struct C {
};
```

- The C++ community usually prefers class for actual classes
  - **`struct`** is sometimes used for *Plain Old Data* (POD) objects – no constructors, no methods, etc., only public-access fields

Summary

- Typedefs allow shortening code by creating type aliases
- Enumerations are types with user-defined values
- Objects and Classes mimic the real-world
  - A class is a collection of data and operations
  - An object is a particular instance of a class (e.g. variable of a class)
- Classes should encapsulate their internal state
  - And provide methods for interaction

Classes and Objects

SoftUni Foundation

xssoftware

SmartIT

NETPEAK
SEO and PPC for Business

Questions?

SUPERHOSTING.BG

INDEAVR
Serving the high achievers

telenor

SOFTWARE GROUP

INFRAGISTICS
DESIGN / DEVELOP / EXPERIENCE

https://softuni.bg/courses/