Classes

Terms

Access modifiers
Classes
Constructors
Copy constructor
Data hiding
Default constructor
Destructors
Getters and setters
Encapsulation
Header (interface) file
Immutable objects
Implementation file
Member initializer list
Object-oriented programming
Objects
Programming paradigms
Static members
Unified Modeling Language (UML)

Summary

- A *programming paradigm* is a style or way of writing software. The two most popular programming paradigms are *object-oriented* and *functional programming*.
- An *object* is a software entity that contains *attributes* (*properties*) and *functions* (*methods*). In C++, these are called *member variables* and *member functions* respectively.
- A *class* is a blueprint for creating objects. An object is an instance of a structure or class.
- We often use structures as simple data containers and classes for creating objects that can do things.
- *Encapsulation* means combining the data and functions that operate on the data into a single unit (class/object). The words class and object are often used interchangeably.
- UML is short for *Unified Modeling Language*. It's a visual language for representing the classes of an application.

- To create a class, we need two files: a *header (interface) file* with **.h** extension, and an *implementation file* with **.cpp** extension.

- *Access modifiers* control how members of a class are accessed. *Public* members are accessible everywhere. *Private* members are only accessible within a class.

- Once we create an object from a class, that object has a *state* (the data that it stores). As our program runs, the state of an object may change.

- Objects should protect their internal state and provide functions for accessing the state. This is referred to as *data* or *information hiding* in object-oriented programming.

- *Getters* (*accessors*) allow us to read the values stored in member variables.

- *Setters* (*mutators*) allow us to change the values of member variables.

- A *constructor* is a special function inside a class that is used for initializing objects. It gets automatically called when an instance of a class is created.

- Member variables can be initialized in the constructor and/or using the *member initializer list*. Initializing member variables using a member initializer list is more efficient because variables are created and initialized in a single operation.

- A *default constructor* is a constructor with zero parameters. The C++ compiler automatically generates a default constructor for a class with no constructors. This allows us to create instances of that class without providing an argument.

- A *copy constructor* is used to create an object as a copy of an existing object. It's called when we declare and initialize an object as well as when we pass an object to a function (by value) and return it (by value). The C++ compiler automatically generates a copy constructor for our classes unless we define one.

- A *destructor* is another special function inside classes that is used for releasing system resources (eg memory, file handles, etc). Destructors are automatically called when objects are destroyed. C++ automatically destroyed objects declared on the stack when they go out of scope. Objects declared on the heap (free store) should be explicitly released using the delete operator.

- *Static* members of a class are shared by all objects of the class. Static functions of a class cannot access instance members because they don't know about the existence of any instances.

- If we declare an object using the **const** keyword, all its member variables will become constant as well. We refer to this object as *immutable* (unchangeable).

Creating a Class

```
// Declaring a class (in Rectangle.h)
class Rectangle {
public:
    int getArea();
    void draw();
private:
    int width;
    int height;
};

// Implementation of the class (in Rectangle.cpp)
#include "Rectangle.h"

int Rectangle::getArea() {
    return width * height;
}

void Rectangle::draw() {
    cout << "Drawing a Rectangle";
}
```

Getters and setters

```
class Rectangle {
public:
    int getWidth() const;
    void setWidth(int width);
private:
    int width;
};


int Rectangle::getWidth() {
    return width;
}

void Rectangle::setWidth(int width) {
    if (width < 0)
        throw invalid_argument("width");
    this->width = width;
}
```

Constructors and destructor

```
class Rectangle {
public:
    // Default constructor
    Rectangle() = default;
    // Copy constructor
    Rectangle(const Rectangle& source);
    // Constructor with parameters
    Rectangle(int width, int height);
    // Destructor
    ~Rectangle();
};
```

Member initializer list

```
Rectangle::Rectangle(int width, int height)
    : width{width}, height{height} {

}
```

Constructor delegation

```
Rectangle::Rectangle(int width, int height, const string& color)
    : Rectangle(width, height) {
    this->color = color;
}
```

Static members

```
class Rectangle {
public:
    static int getObjectsCount();
private:
    static int objectsCount;
};


int main() {
    cout << Rectangle::getObjectsCount();
    return 0;
}
```