Chapter 4. Concept of object

- From C to C++, Classes and Objects
- Protection, Access
- Instantiation, Constructor, Destructor
- Overloading
- Operator “This”
- Object and modeling UML/SysML
- Automatic code generation

Parameterized Constructors:

It is also possible to create constructors with parameters to allow different ways of initializing objects based on the values you pass during object creation.

```
class Person
{public:
      Person(string name, int age)
       { this->name = name;      // operator this
          this->age = age;
}

private:
    string name;    int age;
};
```

Operator this

- The this pointer is a special pointer that points to the object for which the member function is called.
- It is used to refer to the current object instance within a member function.

Accessing Member Variables: When there is a local variable in a member function with the same name as a data member, the this pointer can be used to distinguish between the local variable and the data member.

Example:

```
class MyClass
 {private:
      int x;

   public:
     void setX(int x)
    { this->x = x; }   // Use this pointer to access the member
variable
};
```

Initialization Lists: You can use initialization lists to initialize data members in a more efficient way, especially for complex objects or objects that are not default-constructible.

EX:

```
class Student
{public:
        Student(string name, int age) :
            name(name),
            age(age)
{ // Constructor code here    }

private:
      string name;
      int age;
};
```

Implicit Constructor Calls: Constructors are called automatically when objects are created. You don't need to explicitly call the constructor like a regular function.

EX:

```
class Car
{public:
      Car(string make, string model, int year) :
         make(make),
         model(model),
         year(year)
  {cout << "Car object created." << endl;    }
      void DisplayInfo()
      {cout << "Make: " << make << ", Model: " << model << ", Year: " <<
year << endl;}
private:
  string make;   string model;    int year;};

int main()
{ Car myCar("Toyota", "Camry", 2022);
   myCar.DisplayInfo();
  return 0;
}
```

Multiple Constructors (Overloading): a class can have multiple constructors with different parameter lists. This is known as constructor overloading.

```
class Rectangle
 {public:
     Rectangle()
   { // Default constructor }

Rectangle(double width, double height)
  { // Parameterized constructor
       this->width = width;
       this->height = height; }

private:
      double width;
      double height;
};
```

Copy Constructor: A copy constructor is a special constructor that's used to create a new object as a copy of an existing object of the same class. It's called when an object is passed by value, returned by value, or explicitly copied.

```
class MyClass
   {public:
        MyClass(const MyClass& other)
     { // Copy constructor code here    }
};
```

```
class Rectangle {          // declare a class
  private:
    double length; double height;

  public:
    Rectangle(double len, double hgt)
   { length = len;
      height = hgt;  }

    // copy constructor with a Rectangle object as
parameter
    // copies data of the obj parameter
    Rectangle(Rectangle &obj) {
      length = obj.length;
      height = obj.height;
    }

    double calculateArea() {
      return length * height;
    }
};
```

```
int main() {
  // create an object of Rectangle
class
  Rectangle Rectangle1(10.5, 8.6);

  // copy contents of Rectangle1 to
Rectangle2
  Rectangle Rectangle2 = Rectangle1;

  // print areas of Rectangle1 and
Rectangle2
  cout << "Area of Rectangle 1: " <<
Rectangle1.calculateArea() << endl;
  cout << "Area of Rectangle 2: " <<
Rectangle2.calculateArea();

  return 0;
}
```

****

Output:

```
Area of Rectangle 1: 90.3
Area of Rectangle 2: 90.3
```

Recapitulation:

```
class Rectangle
{private:
            double length;
            double width;
public:
            Rectangle (double l, double h)
            {length=l;
            width=h;
            cout<<"I am the 2 parameters constructor"<<endl;}
            Rectangle (double l)
            {length=l; width=2*l;
            cout<<" I am the 1 parameter constructor "<<endl;}
            Rectangle();
//          {length=3.5; width=6.5;
//          cout<<«  I am the default constructor"<<endl;}
            double area ()
            {cout<<"my area is  "<<length*width<<endl;
            return length*width;}
            void print_o()
            {cout<<"length is: "<<length<<endl<<« width is: "<<width<<endl;}
};
```

constructeurs can be defined outside the class

```
Rectangle::Rectangle()
    {length=3.5; width=6.5;
     cout<<« I am the dafault constructor "<<endl;}
```

Here, the constructor is defined outside the class

```
int main()
{Rectangle R1; // instanciation and initialisation of the object R1with
                    // the default constructor
R1.area();
Rectangle R2(2.8); //instanciation and initialisation of the object R2 with
                    // the 1 parameter constructor

R2.area ();
Rectangle R3(2.8, 5.1); //instanciation and initialisation of the object R3
                    // with the 2 parameters constructor

R3.area();
R1.print_o();
return 0;
}
```

Result after execution:

I am the dafault constructor
my area is 22.75
I am the 1 parameter constructor
my area is 15.68
I am the 2 parameters constructor
my area is 14.28
length is: 3.5
width is: 6.5

Destructor

- A destructor is a special member function of a class that is used to clean up resources or perform any necessary actions when an object of that class goes out of scope or is explicitly deleted.
- The destructor has the same name as the class, preceded by a tilde (~).
- No return type (not even void).
- No parameters : no possible overloading.
- There is only one destructor for a class.

- Syntax:

```
class MyClass
 {public:
     // Constructor
      MyClass()
    { // Initialization code    }

 // Destructor
       ~MyClass()
    { // Cleanup code    }
};
```

The destructor is automatically called when the object goes out of scope.

Example:

```
int main()
{   // Object created
    MyClass obj;
   // Do something with obj

  // Object goes out of scope, and the destructor is
//automatically called  to clean up resources allocated by the
//object.
  return 0; // Destructor called here
}
```

If you allocate memory or resources in the constructor, it is a good practice to release those resources in the destructor to avoid memory leaks or resource leaks.

It's important to note that if you don't provide a destructor explicitly, the compiler generates a default one for you.

However, if your class needs specific cleanup operations, it's often a good idea to define your own destructor.

Example of a class with a destructor that manages dynamic memory:

```
#include <iostream>
Using namespace std;
class DynamicMemoryClass
{public:
   // Constructor
      DynamicMemoryClass()
  { data = new int[10];
    cout << "Constructor called\n";    }

 // Destructor
    ~DynamicMemoryClass()
     { delete[] data;  // Cleanup dynamic memory
       cout << "Destructor called\n";  }

private:
  int* data;};
int main()
{    // Object created
   DynamicMemoryClass obj;
 // Do something with obj
 // Object goes out of scope, and the destructor is automatically called
 // to clean up resources allocated by the object.
return 0; // Destructor called here}
```

Let’s add a user defined destructor to the Rectangle example.
The following code will be added just befor the method print_o() and without any changes to the main.

```
~Rectangle()                                    //  destructor
{cout<< "I am the destructor"<<endl;}
```

Result:

```
I am the dafault constructor
my area is  22.75
 I am the 1 parameter constructor
my area is  15.68
I am the 2 parameters constructor
my area is  14.28
length is: 3.5
width is: 6.5
I am the destructor
I am the destructor
I am the destructor
```

Object and modeling UML/SysML

- Objects and modeling using UML (Unified Modeling Language) and SysML (Systems Modeling Language) are concepts commonly used in software engineering and system design.

Objects:

In object-oriented programming (OOP), an object is an instance of a class, which is a blueprint for creating objects.

Objects encapsulate data (attributes) and behaviors (methods) that operate on the data.

Objects interact with each other through defined interfaces, and this paradigm provides a way to model and structure software systems.

```
class Car
 {public:
    // Attributes
        string brand;
         int year;

// Methods
 void startEngine() { // Code to start the engine}
 void accelerate()    {// Code to accelerate the car}
};
Int main()
{Car myCar;    // Object created
  myCar.brand = "Toyota";
  myCar.year = 2022;
  myCar.startEngine();    // Object behavior
```

UML (Unified Modeling Language):

UML is a standardized general-purpose modeling language used in software engineering for visualizing, specifying, constructing, and documenting the artifacts of a system.

UML diagrams provide a way to represent different aspects of a system, such as its structure, behavior, and interactions.

Common UML diagrams include:

Class Diagrams: Represent the structure and relationships among classes in a system. Classes are depicted as boxes with attributes and methods.

Use Case Diagrams: Depict the interactions between a system and its external entities (actors) to achieve specific goals.

Sequence Diagrams: Show the chronological sequence of interactions between objects or components in a system.

Activity Diagrams: Illustrate the flow of activities or processes within a system.

State Machine Diagrams: Model the behavior of an individual object or a system in response to external stimuli.

SysML (Systems Modeling Language):

SysML is an extension of UML that focuses on systems engineering. It provides a set of diagrams and symbols to model complex systems, including hardware, software, processes, and their interactions.

Common SysML diagrams include:

Block Definition Diagrams: Depict the structural aspects of a system, showing blocks (components) and their relationships.

Internal Block Diagrams: Detail the internal structure of a block, illustrating how parts or subsystems interact.

Requirement Diagrams: Capture and trace system requirements.

Activity Diagrams (SysML): Similar to UML activity diagrams but adapted for systems engineering.

State Machine Diagrams (SysML): Used to model the dynamic behavior of systems.

Both UML and SysML provide standardized ways to communicate and document the design of software systems and complex systems, respectively.

They are valuable tools for software engineers, system architects, and other stakeholders involved in the development and analysis of systems.

Automatic code generation

- Automatic code generation in C++ refers to the process of using tools or software to generate source code or other artifacts automatically, based on high-level specifications, models, or templates.
- This can help improve productivity, reduce errors, and maintain consistency in large software projects.
- There are several ways automatic code generation can be applied in C++ development:

Integrated Development Environments (IDEs):
Many modern IDEs include features for code generation.

For example, they may offer wizards or templates that allow developers to create common code structures or design patterns quickly.

IDEs often provide code snippets, boilerplate code, or auto-completion features to speed up coding.

Code Generators:

Specialized tools and frameworks exist that can generate C++ code based on high-level specifications.

These specifications can be in the form of models, configurations, or domain-specific languages.

Examples include tools that generate serialization/deserialization code, database access code, or code for communication between different components.

Model-Driven Development (MDD):

MDD involves creating high-level models of a system and then automatically generating the corresponding C++ code from these models.

Unified Modeling Language (UML) tools, combined with code generation capabilities, can be used for MDD.

IDL (Interface Definition Language) and RPC (Remote Procedure Call) Generators:

In distributed systems, Interface Definition Languages like IDL are used to define the interfaces between different components.

Code generators then create the necessary C++ code for communication.

Tools like Protocol Buffers or Apache Thrift use IDL to define data structures and service interfaces and generate C++ code for serialization, deserialization, and remote procedure calls.

Template Metaprogramming:

C++ itself supports a form of code generation through template metaprogramming.

Templates allow you to write generic code that is parameterized by types or values, and the compiler generates specialized code based on how the templates are instantiated.

This is a powerful feature used in libraries like the Standard Template Library (STL) and can be employed in application code for generic programming.

Protocol Buffers (protobuf):

Scenario: When you need a fast and efficient serialization mechanism for data interchange.

Description: Protocol Buffers is a language-agnostic serialization format developed by Google. You define your data structures and the services you want to expose in a .proto file, and then you use the protoc compiler to generate C++ code for serialization and deserialization.

Example .proto file:

```
syntax = "proto3";
 message  Person
 { required string name = 1;
   required int32 id = 2;
   optional string email = 3; }
```

Generated C++ code (simplified):

```
cpp
// Generated code
 class Person
 { public: // ... Constructors, accessors, etc.
 void Serialize(ostream* output) const;
bool ParseFromIstream(istream* input); };
```

Apache Thrift:

Scenario: When you are building a service-oriented architecture and need a framework for cross-language services development.

Description: Apache Thrift uses an Interface Definition Language (IDL) to define data types and service interfaces. The thrift compiler generates C++ code for servers, clients, and data serialization.

Example .thrift file:

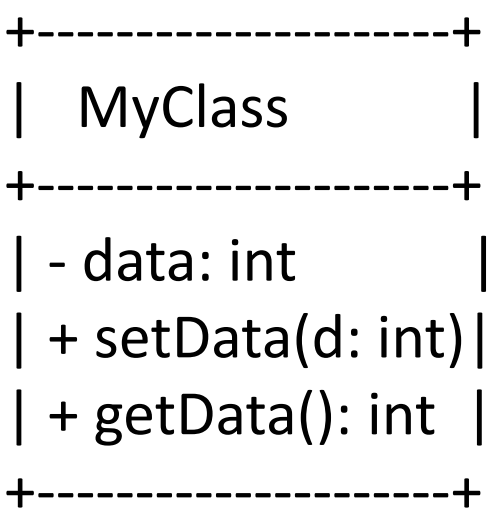
```
struct Person
{ 1: required string name;
  2: required i32 id;
  3: optional string email; }

// Generated code
class Person
 {public:    // ... Constructors, accessors, etc.
   void read(apache::thrift::protocol::TProtocol* iprot);
 void write(apache::thrift::protocol::TProtocol* oprot) const;};
```

UML Tools with Code Generation:

Scenario: When you are using UML to model your system and want to generate C++ code from your models.

Description: Some UML modeling tools offer code generation capabilities. You create UML class diagrams, activity diagrams, etc., and the tool generates corresponding C++ code.

Example UML Class Diagram:
Sql file




```
class MyClass
 { private:
        int data;
   public:
        void setData(int d);
        int getData(); };
```