Basic object-oriented programming in C++

ThanhNT

Goals

- Understand basic concept of object oriented programing (OOP) .
- Understand some basic implementation of OOP in c++.

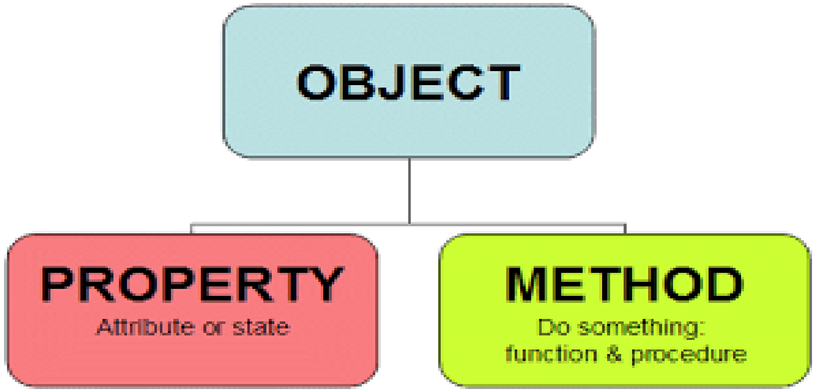
Contents

- Introduction to object oriented programming(OOP).
- Basic implementation of OOP in C++
  - Classes
  - Access specifiers
  - Access function and encapsulation
  - Constructors/Destructors
  - Static members
  - Friend function and class
  - Composition/Aggregation/Container class

Introduction to object oriented programming(OOP)

Introduction to OOP

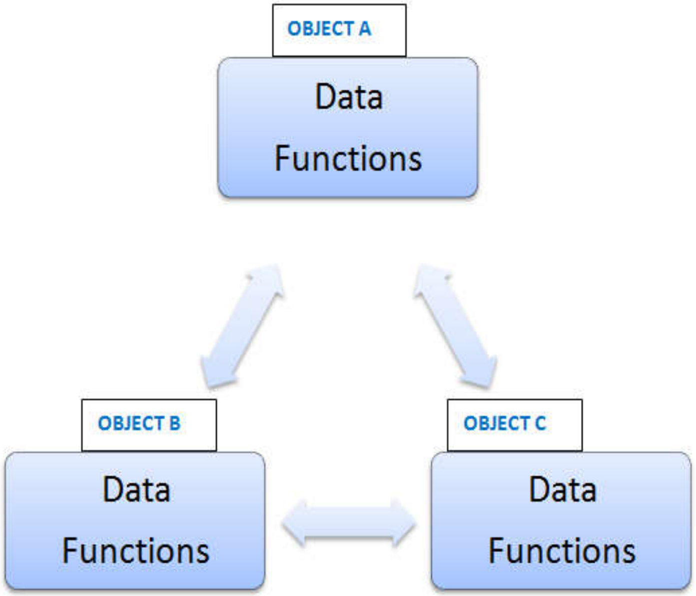
- What is OOP?



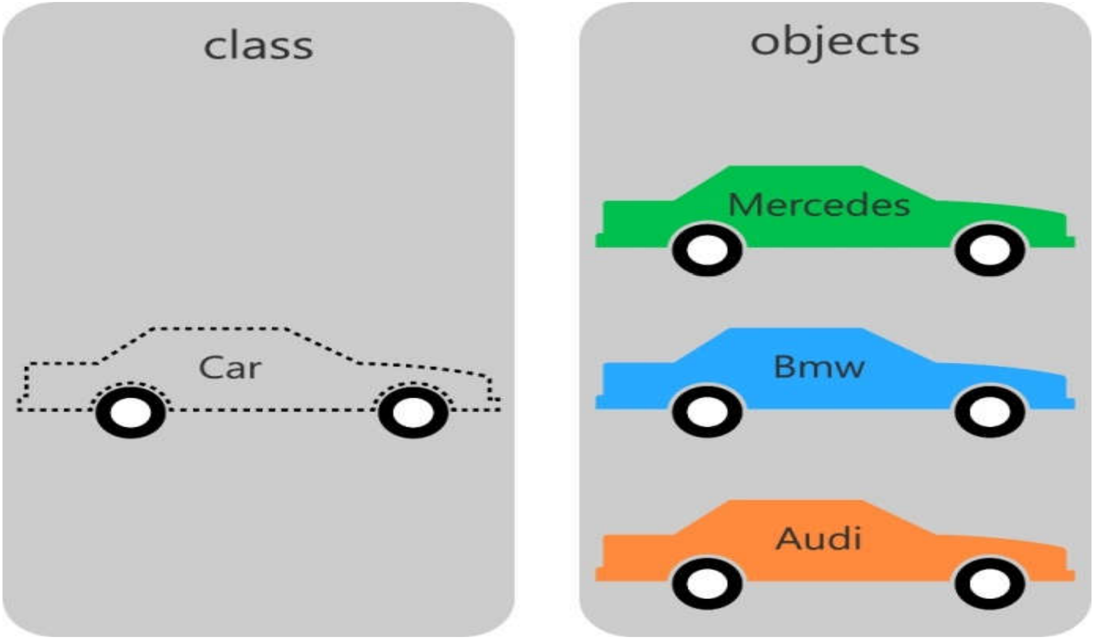
Programming paradigm based on the concept of classes and objects.

Introduction to OOP

- Object oriented programs
  - Consist of group of cooperating object.
  - Object exchange messages to achieve common objective



- Object and class



Object is an instance of a class

Introduction to OOP

- 4 major principles of OOP:
  - Encapsulation
  - Data abstraction
  - Inheritance
  - Polymorphism

Basic implementation of object oriented programming in c+

Class and class member

- Class

```
class Car{
private:
    string m_model;
    string m_colour;
    long m_price;
public:
    Car(string model, string colour, long price)
    {
        m_model = model;
        m_colour = colour;
        m_price = price;
    }
};
```

- Start with keyword *class*
- Consist of data representation(data member) and methods for manipulating that data (member function).

Access specifiers

- Private: private member can only be accessed by other members of the class.
- Public: public member can be accessed from outside of the class.
- Protected: same as private member, but can be inherited.

```
class Employee{
private:
    string m_id;
protected:
    string m_salary;
public:
    string m_name;
};

int main(){
    Employee ThanhNT7;
    ThanhNT7.m_name = "Thanh Bond";
    cout << ThanhNT7.m_name;
    ThanhNT7.m_id = "00+1/7";
    cout << ThanhNT7.m_id;
    return 0;
}
```

Encapsulation

- What?
  - is the process of keeping the details about how an object is implemented hidden away from users of the object.
- How?
  - is implemented via access specifiers.
- Why?
  - easier to use and reduce the complexity.
  - help protect your data and prevent misuse.
  - easier to change and debug.

Access function

- is a short public function whose job is to retrieve or change the value of a private member variable.
- typically come in two flavors: getters and setters.
  - Getter: return the value of a private member variable.
  - Setter: set the value of a private member variable.

```cpp
class Number{
private:
    int m_num = 0;
public:
    void setNum(int num){
        m_num = num;
    }
    int getNum(){
        return m_num;
    }
};

int main(){
    Number myNum;
    myNum.setNum(10);
    cout << myNum.getNum();
    return 0;
}
```

Constructors

- is a special kind of class member function that is automatically called when an object of that class is instantiated.
- typically used to initialize member variables of the class or do any setup steps necessary for the class to be used.
- specific rules for constructors:
  - always have the same name as the class.
  - have no return type (not even void).
  - can not be explicitly called.

Constructors

Constructor without parameter(default constructor)

```
class Car{
private:
    string m_model;
    string m_colour;
    long m_price;
public:
    Car()
    {
        m_model = "mustang";
        m_colour = "black";
        m_price = 50000;
    }
};
```

Constructor with parameter

```
class Car{
private:
    string m_model;
    string m_colour;
    long m_price;
public:
    Car(string model, string colour, long price)
    {
        m_model = model;
        m_colour = colour;
        m_price = price;
    }
};
```

Class without constructors

```cpp
class Car{
private:
        string m_model;
        string m_colour;
        long m_price;
};

int main(){
        Car myCar;
        return 0;
}
```

```cpp
class Car{
private:
      string m_model;
      string m_colour;
      long m_price;
public:
      Car(string model, string colour, long price)
      {
          m_model =model;
          m_colour = colour;
          m_price = price;
      }
};

int main(){
      Car myCar;
      return 0;
}
```

Constructor member initializer list

```cpp
class Car{
private:
    string m_model;
    string m_colour;
    long m_price;
public:
    Car(){
        // These are all assignments
        // not initializations
        m_model = "";
        m_colour  "";
        m_price = 0;
    }
};
```

```cpp
class Car{
private:
    string m_model;
    string m_colour;
    long m_price;
public:
    // directly initialize our member variables
    Car() : m_model(""), m_colour(""), m_price(0)
    {
        // No need for assignment here
    }
};
```

Destructors

- is a special kind of class member function that is executed when an object of that class is destroyed.
- designed to help clean up member variables .
- specific rules for destructors:
  - must have the same name as the class, preceded by a tilde (~).
  - can not take arguments.
  - has no return type.

Destructors

```
class Car{
private:
    string m_model;
    long * m_price;
public:
    Car(){
        m_model = "";
        m_price = new long;
    }
    Car(string model, long price){
        m_model = model;
        m_price = new long;
        *m_price = price;
    }
    ~Car(){
        delete(m_price);
    }
    long  getPrice(){return *m_price;}
};

int main(){
    Car myCar("mustang", 60000);
    cout << myCar.getPrice();
    return 0;
}
```

Resource Acquisition Is Initialization

- What?
  - is a programming technique whereby resource use is tied to the lifetime of objects with automatic duration.
- How?
  - is implemented via constructors and destructors.
  - A resource is acquired in the object’s constructor. That resource can then be used while the object is alive.
  - The resource is released in the destructor, when the object is destroyed.

RAII

- ❑ Why?
  - The primary advantage of RAII is that it helps prevent resource leaks as all resource-holding objects are cleaned up automatically.

```cpp
class Car{
private:
    string m_model;
public:
    Car(string model)
    {
        m_model =model;
        cout << "constructing car:" << m_model << endl;
    }
    ~ Car()
    {
        cout << "destructing car:" << m_model << endl;
    }
};

int main(){
    Car myCar("mustang");
    Car * myNewCar = new Car("hummer");
    delete myNewCar;
    return 0;
}
```

Class code and header files

Date.h

```cpp
#ifndef DATE_H
#define DATE_H

class Date{
private:
    int m_year;
    int m_month;
    int m_day;
public:
    Date(int year, int month, int day);

    void SetDate(int year, int month, int day);

    int getYear() { return m_year; }
    int getMonth() { return m_month; }
    int getDay()  { return m_day; }
};
#endif
```

Date.cpp

```cpp
#include "Date.h"

Date::Date(int year, int month, int day)
{
    SetDate(year, month, day);
}

void Date::SetDate(int year, int month, int day)
{
    m_month = month;
    m_day = day;
    m_year = year;
}
```

The hidden *this pointer

The function

```
A.setNum(3);
```

;when compiled, is actually converted into

```
setNum(&A, 3);
```

```
class Number{
private:
    int m_num;
public:
    Number(int num){setNum(num);}
    void setNum(int num){m_num = num;}
    int getNum(){return m_num;}
};

int main(){
    Number A(1);
    Number B(2);
    A.setNum(3);
    B.setNum(4);
    return 0;
}
```

The hidden *this pointer

- The **this pointer** is a hidden const pointer that holds the address of the object the member function was called on.

```
void setNum(int num){m_num = num;}
is converted into
void setNum(Number * const this, int num){
    this->m_num = num;
}
```

- *this always points to the object being operated on.

```
Number A(1); // *this = &A inside the Number constructor
Number B(2); // *this = &B inside the Number constructor
A.setNum(3); // *this = &A inside member function setNum
B.setNum(4); // *this = &B inside member function setNum
```

Each member function has a *this pointer parameter that is set to the address of the object being operated on.

The hidden *this pointer

```cpp
class Calc{
private:
    int m_value;

public:
    Calc() { m_value = 0; }

    Calc& add(int value) { m_value += value; return *this; }
    Calc& sub(int value) { m_value -= value; return *this; }
    Calc& mult(int value) { m_value *= value; return *this; }

    int getValue() { return m_value; }
};

int main(){
    Calc calc;
    calc.add(5).sub(3).mult(4);

    std::cout << calc.getValue() << endl;
    return 0;
}
```

Const class objects

- instantiated class objects can also be made const by using the const keyword.
- Initialization is done via class constructors.
- Once a const class object has been initialized, any attempt to modify the member variables of the object is disallowed.

```
class Number{
private:
    int m_num;
public:
    Number(): m_num(0){}
    void setNum(int num){m_num = num;}
    int getNum(){return m_num;}
};

int main(){
    const Number A;

    A.setNum(3);
    return 0;
}
```

Const member function

- Const class objects can only explicitly call *const* member functions.
- A **const member function** is a member function that guarantees it will not change any class variables or call any non-const member functions.

```
class Number{
private:
    int m_num;
public:
    Number(): m_num(0){}
    void setNum(int num){m_num = num;}
    int getNum() const {return m_num;}
    void resetNum() const {m_num = 0;}
};

int main(){
    const Number A;

    cout << A.getNum();
    return 0;
}
```

Static member variables

- Member variables of a class can be made static by using the static keyword.
- static member variables are shared by all objects of the class.
- static member variables are created when the program starts, and destroyed when the program ends.

```cpp
class Number{
private:
    static int m_num;
public:
    void setNum(int num){m_num = num;}
    int getNum() const {return m_num;}
};

int Number::m_num = 1;

int main(){
    const Number A;

    cout << A.getNum();
    return 0;
}
```

Static member variables

- Can be accessed through objects of class(red) or class itself(green).
- Must be explicitly defined outside of the class, in the global scope(orange).
- Const integer or const enum static member variavles can be initialized directly on the line in which they are declared(blue).

```
class Number{
private:
    static const int s_id = 1;
public:
    static int s_num;
    int getNum(){ return s_num; }
};

int Number::s_num = 1;

int main(){
    Number A;
    cout << A.s_num << endl;
    Number::s_num = 2;
    cout << Number::s_num;
    return 0;
}
```

Static member functions

- Static member functions are not attached to any particular object.
- Static member functions have no *this pointer.
- Static member functions can only access static member variables.
- C++ does not support static constructors.

```cpp
class Number{
private:
    static int m_num;
public:
    static void setNum(int num){m_num = num;}
    static int getNum() {return m_num;}
};

int Number::m_num = 1;

int main(){
    Number::setNum(10);
    cout << Number::getNum();
    return 0;
}
```

Friend function

```
class Accumulator{
private:
        int m_value;
   public:
        Accumulator(){m_value = 0;}
        void add(int num){m_value += num;}
        int getAcc(){return m_value;}
        friend void resetAcc(Accumulator &accumulator);
};

void resetAcc(Accumulator &accumulator){
        accumulator.m_value = 0;
}

int main(int argc, char const *argv[])
{
        Accumulator acc;
        acc.add(10);
        cout << acc.getAcc() << endl;
        resetAcc(acc);
        cout << acc.getAcc() << endl;
        return 0;
}
```

Friend function

- Is a function that can access the private members of a class as though it were a member of that class.
- may be either a normal function, or a member function of another class.
- Is declared using *friend* keyword in front of the prototype of that function.
- Can be declared in either private or public section of the class.
- can be a friend of more than one class at the same time.

Friend class

```cpp
class Accumulator{
private:
        int m_value;
 public:
        Accumulator(){m_value = 0;}
        void add(int num){m_value += num;}
        int getAcc(){return m_value;}
        friend class Reset;
};

class Reset{
private:
        bool m_key;
public:
        Reset(bool key){m_key = key;}
        void resetAcc(Accumulator &accumulator){
            if(m_key == true){
                accumulator.m_value = 0;
            }
        }
};

int main(int argc, char const *argv[])
{
        Accumulator acc;
        Reset res(true);
        acc.add(10);
        cout << acc.getAcc() << endl;
        res.resetAcc(acc);
        cout << acc.getAcc() << endl;
        return 0;
}
```

Friend class

A is a friend class of B and B is a friend class of C:

- All of the members of the A have access to the private members of B.
- A has no direct access to *this pointer of B’s objects.
- B is not a friend class of A.
- A is not a friend class of C.

Composition

```
class Frame{
private:
    string m_material;
public:
    Frame(){ m_material = "Aluminum"; }
    Frame(string material){ m_material = material; }
    string getMaterial(){ return m_material; }
};

class Car{
private:
    string m_model;
    Frame m_frame;
public:
    Car(string model, Frame frame){
        m_model = model;
        m_frame = frame;
    }
    void printCar(){
        cout << m_model << " is made of " << m_frame.getMaterial() << endl;
    }
};

int main(){
    string model = "Batmobile";
    Car myCar(model, Frame("Adamantium"));
    myCar.printCar();
    return 0;
}
```

Composition

- What?

- process of building complex objects from simpler ones is called **composition** (also known as object composition).

- Why?

1. Each individual class can be kept relatively simple and straightforward, focused on performing one task.
2. Each sub object can be self-contained, which makes them reusable.
3. The complex class can have the simple subclasses do most of the hard work, and instead focus on coordinating the data flow between the subclasses.

Aggregation

```
class Car{
private:
    string m_model;
public:
    Car(){m_model = "";}
    Car(string model){m_model =model;}
    string getModel(){return m_model;}
};

class Garage{
private:
    Car * m_car;
public:
    Garage(Car * car){
        m_car = car;
    }
    void printCar(){
        cout << m_car->getModel() << endl;
    }
};

int main(){
    Car * pCar = new Car("Audi A8");
    {
        Garage mGarage(pCar);
        mGarage.printCar();
    }
    mGarage.printCar();

    cout << pCar->getModel();
    return 0;
}
```

Aggregation

Aggregation	Composition
❑ Typically use pointer variables that point to an object that lives outside the scope of the aggregate class.	❑ Typically use normal member variables .
❑ Can use reference values that point to an object that lives outside the scope of the aggregate class.	❑ Can use pointer values if the composition class automatically handles allocation/deallocation.
❑ Not responsible for creating/destroying subclasses.	❑ Responsible for creation/destruction of subclasses .

Container class

- What?

- A **container class** is a class designed to hold and organize multiple instances of another class.

- Why?

- Container class provides the ability to help organize and store items that are put inside it.

- Types

- ***Value containers*** are composition that store copies of the objects that they are holding.
- ***Reference containers*** are aggregations that store pointers or references to other objects.

Container class

- How?

A container class should include functions that:

- Create an empty container (via a constructor)
- Insert a new object into the container
- Remove an object from the container
- Report the number of objects currently in the container
- Empty the container of all objects
- Provide access to the stored objects
- Sort the elements (optional)

Container class

```cpp
class IntArray
{
private:
    int m_nLength;
    int *m_pnData;

public:
    IntArray()
    {
        m_nLength = 0;
        m_pnData = 0;
    }

    IntArray(int nLength)
    {
        m_pnData = new int[nLength];
        m_nLength = nLength;
    }

    ~IntArray()
    {
        delete[] m_pnData;
    }

    void Erase()
    {
        delete[] m_pnData;
        m_pnData = 0;
        m_nLength = 0;
    }
```

Container class

```
    int& operator[](int nIndex)
    {
        assert(nIndex >= 0 && nIndex < m_nLength);
        return m_pnData[nIndex];
    }

    void Reallocate(int nNewLength)
    {
        Erase();
        if (nNewLength<= 0)
            return;
        m_pnData = new int[nNewLength];
        m_nLength = nNewLength;
    }

    void Resize(int nNewLength)
    {
        if (nNewLength <= 0)
        {
            Erase();
            return;
        }
        int *pnData = new int[nNewLength];
        if (m_nLength > 0)
        {
            int nElementsToCopy = (nNewLength > m_nLength) ? m_nLength : nNewLength;
            for (int nIndex=0; nIndex < nElementsToCopy; nIndex++)
                pnData[nIndex] = m_pnData[nIndex];
        }
        delete[] m_pnData;
        m_pnData = pnData;
        m_nLength = nNewLength;
    }
```

Container class

```
    void InsertBefore(int nValue, int nIndex)
    {
        assert(nIndex >= 0 && nIndex <= m_nLength);
        int *pnData = new int[m_nLength+1];
        for (int nBefore=0; nBefore < nIndex; nBefore++)
            pnData[nBefore] = m_pnData[nBefore];
        pnData[nIndex] = nValue;
        for (int nAfter=nIndex; nAfter < m_nLength; nAfter++)
            pnData[nAfter+1] = m_pnData[nAfter];
        delete[] m_pnData;
        m_pnData = pnData;
        m_nLength += 1;
    }

    void Remove(int nIndex)
    {
        assert(nIndex >= 0 && nIndex < m_nLength);
        int *pnData = new int[m_nLength-1];
        for (int nBefore=0; nBefore < nIndex; nBefore++)
            pnData[nBefore] = m_pnData[nBefore];
        for (int nAfter=nIndex+1; nAfter < m_nLength; nAfter++)
            pnData[nAfter-1] = m_pnData[nAfter];
        delete[] m_pnData;
        m_pnData = pnData;
        m_nLength -= 1;
    }
    void InsertAtBeginning(int nValue) { InsertBefore(nValue, 0); }
    void InsertAtEnd(int nValue) { InsertBefore(nValue, m_nLength); }
    int GetLength() { return m_nLength; }
};
```

Container class

```
int main()
{
    IntArray cArray(10);
    for (int i=0; i<10; i++)
        cArray[i] = i+1;
    cArray.Resize(8);
    cArray.InsertBefore(20, 5);
    cArray.Remove(3);
    cArray.InsertAtEnd(30);
    cArray.InsertAtBeginning(40);
    for (int j=0; j<cArray.GetLength(); j++)
        cout << cArray[j] << " ";
    return 0;
}
```

Q&A

Thanks for your attention!