ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC BÁCH KHOA

KHOA KHOA HỌC - KỸ THUẬT MÁY TÍNH

REPORT

Lập Trình Hướng Đối Tượng
OOP - Object-Oriented Programming

SV thực hiện: *Nguyễn Đức An - 2010102*
Email: an.nguyenduc1406@hcmut.edu.vn
Lớp: L02

Tp.HCM, 12/09/2021

Contents

1 Giới thiệu về hai thời kỳ lập trình **3**

2 Class & Object **3**

3 Access Specifiers **6**

4 Constructor **6**

5 Destructor **10**

6 Operator Overloading **10**

7 Friendship **12**

8 This Pointer **14**

9 Static variables and methods **14**

10 Tính đóng gói (Encapsulation) **15**

11 Tính trừu tượng (Abstraction) **16**

12 Tính kế thừa (Inheritance) **17**

13 Tính đa hình (Polymorphism) **19**

14 Template **20**

15 Exception Handling **21**

16 Tài liệu tham khảo **22**

1 Giới thiệu về hai thời kỳ lập trình

Structure Programming:

- Đây là lập trình lấy ***hành động*** làm trung tâm.
- Phân chương trình thành các công việc nhỏ hơn để dễ dàng giải quyết, gọi là chương trình con (hàm).
- Hàm là xương sống của toàn bộ chương trình.

Object-Oriented Programming:

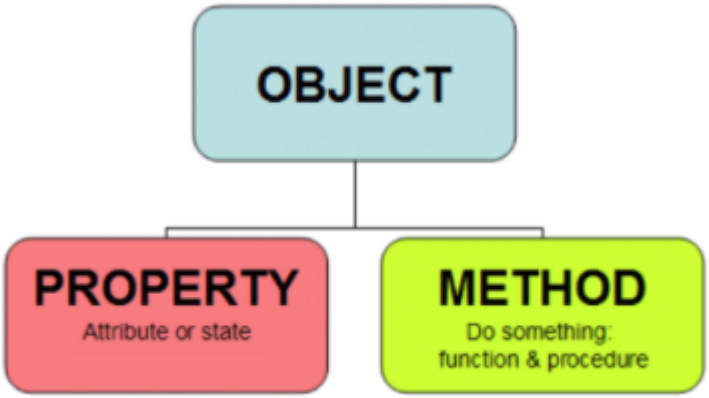
- Là phương pháp lập trình dựa trên kiến trúc lớp(**class**) và đối tượng (**object**), trong đó lấy ***đối tượng*** làm trung tâm.
- Bốn tính chất của lập trình hướng đối tượng: Tính trừu tượng (**Abstraction**), Tính đóng gói và che giấu dữ liệu (**Encapsulation & Data Hiding**). Tính kế thừa (**Inheritance**). Tính đa hình (**Polymorphism**)..

2 Class & Object

Lớp(Class): Đây là **kiểu dữ liệu** do người dùng tự định nghĩa để nhóm các thông tin lại với nhau.

Đối Tượng (Object): Đối tượng là trường hợp cụ thể của một Class.

- Thuộc tính (**attribute**): mô tả tính chất của đối tượng
- Phương thức (**method**): mô tả hành động của đối tượng đó



Implement a Class & Object example

```
#include <iostream>
using namespace std;
class MyClass {     // The class
private:            // Access specifier
int myNum;       // Attribute (int variable)
string myString; // Attribute (string variable)
public:
void setNum(int myNum) {       //method to set myNum
    this->myNum = myNum;
}
void setString(string myString) { //method to set myString
    this->myString = myString;
}
void print() { //method to print information of an object
    cout << "myNum is: " << myNum << " && myString is: " << myString;
}
};

int main() {
    MyClass myObj;       // Create an object of MyClass
    myObj.setNum(2010102); // set attribute myNum of object myObj
    myObj.setString("Nguyen Duc An"); // set attribute myString of
        object myObj
    myObj.print();       // print information of an object
    return 0;
}
```

- Có hai cách để định nghĩa cho các **Method** thuộc Class

- **Inside Class:**

```
#include<iostream>
using namespace std;
class MyClass {      // The class
    public:            // Access specifier
    void myMethod() { // Method/function defined inside the
        class
        cout << "Hello World!";
    }
};
int main() {
    MyClass myObj;   // Create an object of MyClass
    myObj.myMethod(); // Call the method
    return 0;
}
```

- **Outside Class:**

```
#include<iostream>
using namespace std;
class MyClass {      // The class
    public:            // Access specifier
    void myMethod();
};
void MyClass::myMethod(){ // Method/function outside the class
    cout << "Hello World!";
}
int main() {
    MyClass myObj;   // Create an object of MyClass
    myObj.myMethod(); // Call the method
    return 0;
}
```

3 Access Specifiers

- **Public:** các thuộc tính hoặc các phương thức có thể truy xuất ra bên ngoài class.
- **Private:** các thuộc tính các phương thức không thể truy xuất ra bên ngoài class, nó chỉ được gọi trong phạm vi của chính class đó.
- **Protected:** các thuộc tính các phương thức không thể truy xuất ra bên ngoài class, nó chỉ được gọi trong class và class kế thừa.

```
#include<iostream>
using namespace std;
class FirstClass {
   int x;  // Private attribute
};
struct SecondClass {
   int x; // Public attribute
};
int main(){
    FirstClass a;
    SecondClass b;
    a.x = 5 // Error because x is private
    b.x = 5 // Can assign b.x = 5 because x is public
}
```

4 Constructor

Định nghĩa: Trong lập trình hướng đối tượng, phương thức khởi tạo (**Constructor**) được xây dựng nhằm khởi tạo các thành phần dữ liệu của đối tượng
- Các **Constructor** có tên hàm trùng với tên của Class và không có kiểu dữ liệu trả về, ký hiệu **Class_Name()**

```
#include <iostream>
using namespace std;
class MyClass{
private:
    int x, y;
public:
    MyClass(): x(0), y(0) {} // Constructor
};
```

1. **Default Constructor**
   Phương thức thiết lập mặc định có tên Tiếng Anh được gọi là: **Default Constructor.**

   - Là phương thức khởi tạo không có tham số đầu vào.
   - Được tự động gọi khi không có một phương thức khởi tạo nào được gọi trước đó.
   - Khi khởi tạo đối tượng, nếu không có constructor do người dùng tự định nghĩa thì trình biên dịch sẽ tự động tạo ra bên trong nó một cái **Default Constructor.**

```
class MyClass {   // The class
    public:         // Access specifier
    MyClass() {   // Constructor
      cout << "Hello World!";
    }
};
int main() {
    MyClass myObj;  // Create an object of MyClass (this will
        call the constructor)
    MyClass* newObj; // newObj is a pointer which is not
        initialized so it is not call Default Constructor
    return 0;
}
// Output: Hello World!
```

2. Parameterized Constructor

Phương thức thiết lập có tham số đầu vào có tên Tiếng Anh thường gọi là: **Parameterized Constructor.**

- Là phương thức khởi tạo có tham số đầu vào mà số lượng tham số đầu vào tùy người sử dụng quyết định.
- Có tên phương thức trùng với tên lớp.
- Khi ta khởi tạo **Parameterized Constructor** thì lúc này chức năng tự tạo **Default Constructor** của trình biên dịch không còn nữa, lúc này nếu có khởi tạo đối tượng theo kiểu mặc định thì phải tự tạo cho nó một cái **Default Constructor**.

COMMON MISTAKES

```
#include<iostream>
using namespace std;
class MyClass {
    private:
     int x;
    public:
     MyClass(int x) { // Parameterized Constructor
        this->x = x;
     }
};
int main() {
   MyClass obj; // Error because we don't have default
      constructor
   return 0;
}
```

3. Copy Constructor

Phương thức thiết lập sao chép có tên Tiếng Anh thường gọi là: **Copy Constructor.**

- Là phương thức khởi tạo nhận tham số đầu vào là một đối tượng cùng thuộc về lớp.
- Phương thức khởi tạo sao chép đem thông tin đối tượng đầu vào cho đối tượng đang khởi tạo.
- Parameter khi truyền vào Copy Constructor phải là truyền tham chiếu **(const MyClass& obj)**.
- Nếu người dùng không tự định nghĩa **Copy Constructor** thì trình biên dịch cũng sẽ tự động tạo ra trong nó một cái **Copy Constructor.**

```
    #include<iostream>
    #include<string.h>
    using namespace std;
    class Book {
        char* name;
    public:
        Book(char* name){
            this->name = new char[strlen(name)];
            strcpy(this->name, name);
        }
        Book(const Book& MyBook) {
            this->name = new char[strlen(MyBook.name)];
            strcpy(this->name, MyBook.name);
        }
    };
int main() {
    char first_book[] = "C++ How To Program";
    char second_book[] = "Algorithm";
    Book a(first_book);
    Book b(second_book);
    a = b;        // Copy Constructor
    return 0;
}
```

COMMON MISTAKES

- Khi **attributes** là một con trỏ như (**int* a, double* b, char* c**) thì ta nên tự tạo cho nó một **Copy Constructor** và tự khởi tạo một vùng nhớ mới(toán tử **new**) để copy những attributes (có dạng con trỏ) vào vùng nhớ mới đó. Nếu không làm vậy thì sẽ bị mắc những lỗi do **Shallow Copy.**
- Trong trường hợp trên ta phải sử dụng **Deep Copy.**
- Các bạn có thể tìm hiểu thêm về **Shallow Copy** và **Deep Copy** tại đường link sau:
`https://www.geeksforgeeks.org/shallow-copy-and-deep-copy-in-c/`

5 Destructor

Định nghĩa: Trong lập trình hướng đối tượng, phương thức phá hủy (**Destructor**) được xây dựng nhằm thu hồi lại bộ nhớ đã cấp phát cho đối tượng thuộc lớp ngay khi đối tượng hết phạm vi hoạt động.
- Cũng như **Constructor**, **Destructor** cũng không có kiểu dữ liệu trả về, ký hiệu là ∼ **Class_Name()**

```
#include <iostream>
using namespace std;
class MyClass{
private:
    int* x;
public:
    ~MyClass(){       // Destructor
        delete x;
    }
};
```

- Phương thức có tên trùng với tên của lớp đối tượng và có dấu phía trước.
- Không có kiểu trả về, không có tham số đầu vào.
- Tự động được gọi thực hiện khi đối tượng hết phạm vi sử dụng.
- Phương thức phá hủy chỉ được gọi 1 lần duy nhất trong quá trình sống của đối tượng.
- Chỉ có thể có duy nhất một **Destructor** trong một Class.
- Nếu ta không tự khởi tạo một **Destructor** thì trình biên dịch sẽ tự tạo ra trong nó một **Default Destructor** với nội dung rỗng.
- **Destructor** có thể là ảo **(virtual)** nhưng **Constructor** không thể là **virtual**

6 Operator Overloading

Hàm có tên là keyword **operator** với **toán tử** đứng sau nó dùng để nạp chồng cho toán tử đó.
Ví dụ : **operator+** được sử dụng để nạp chồng cho toán tử (+)

Restrictions on Operator Overloading

Operators that can be overloaded							
+	-	*	/	%	^	&	|
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&	||	++
--	->*	,	->	[]	()	new	delete
new[]	delete[]						

Operators that cannot be overloaded				
.	.*	::	?:	sizeof

Implement an operator+ between two functions

```
#include <iostream>
using namespace std;

class PhanSo {
private:
   int tuso;
   int mauso;
public:
   PhanSo(): tuso(0), mauso(1) {}
   PhanSo(int tu, int mau): tuso(tu), mauso(mau){}
   int findGCD(int tuso, int mauso){ // find GCD to simplify PhanSo
      if (tuso == 0 || mauso == 0) return tuso + mauso;
      return findGCD(mauso, tuso % mauso);
   }
   PhanSo operator+(const PhanSo& ps) { //Overloading opearator+ PhanSo
      PhanSo res;
      res.tuso = tuso * ps.mauso + mauso * ps.tuso;
      res.mauso = ps.mauso * mauso;
      res.tuso /= findGCD(res.tuso, res.mauso);
      res.mauso /= findGCD(res.tuso, res.mauso);
      return res;
   }
   void print() {
      cout << tuso << "/" << mauso;
   }
};
int main() {
   PhanSo a(1, 2);
   PhanSo b(3, 4);
   PhanSo res = a + b;
   res.print();           // output = 5/4 ( 1/2 + 3/4 = 5/4)
}
```

1. **Toán tử một ngôi:** sử dụng một đối tượng để thay đổi đối tượng đó: operator+, operator-, operator!, operator ++ (tăng sau), operator- - (giảm sau), ++operator (tăng trước), - -operator (giảm trước)
2. **Toán tử hai ngôi:** thao tác đến 2 đối tượng để tìm kết quả mong muốn:
   - operator-(class), +, *, /
   - operator+=(class), -=, *=, /=
3. **Toán tử quan hệ:** sử dụng để so sánh các đối tượng với nhau: operator >, <, ==, >=, <=, !=
4. **Toán tử gán:** dùng để gán đối tượng này cho đối tượng kia, tương tự như một copy constructor: operator =
5. **Toán tử gọi hàm:** dùng để tạo ra một phương thức với nhiều tham số đầu vào khác nhau : operator ( )
6. **Toán tử truy xuất:** dùng để truy xuất giá trị tại một index: operator[ ]
7. **Toán tử nhập:**

```
friend istream& operator>>(istream& is, PhanSo& ps) {
  cout << "Nhap tu: ";
  is >> ps.tuso;
  cout << "Nhap mau: ";
  is >> ps.mauso;
  return is;
}
```

8. **Toán tử xuất:**

```
friend ostream& operator<<(ostream& os, PhanSo ps) {
  os << ps.tuso << "/" << ps.mauso;
   return os;
}
```

7 Friendship

Một số tính chất của keyword **friend**:

- Có thể truy xuất thành phần của lớp.

- Được khai báo bên trong lớp.
- Định nghĩa bên ngoài lớp.

1. **Friend Functions:**
   - **Hàm bạn** là một hàm có thể truy cập đến các thành viên private (gồm cả các biến thành viên và các hàm thành viên) của một class, như thể nó là một thành viên của class đó.

```
#include <iostream>
using namespace std;
class Accumulator {
private:
    int m_value;
public:
    Accumulator() { m_value = 0; }
    void add(int value) { m_value += value; }
// Make the reset() function a friend of this class
    friend void reset(Accumulator &accumulator);
};

    // reset() is now a friend of the Accumulator class
void reset(Accumulator &accumulator){
// And can access the private data of Accumulator objects
    accumulator.m_value = 0;
}
int main(){
    Accumulator acc;
    acc.add(5); // add 5 to the accumulator
    reset(acc); // reset the accumulator to 0
    return 0;
}
```

2. **Friend Class:**
   - **Lớp bạn** là lớp mà mọi thành viên của nó có thể truy cập đến những thành phần **private** và **protected** của các lớp khác là bạn của nó.

```
class Lecturer;
class Student {
    friend class Lecturer; //Lecturer is a friend class
};
```

8 This Pointer

Con trỏ this: giữ địa chỉ của đối tượng đang gọi phương thức.

```
PhanSo& operator+=(const PhanSo& b) {
   *this = *this + b;
   return *this;
}
```

9 Static variables and methods

- Biến **static** không thuộc về bất kỳ đối tượng nào và giá trị của nó được chia sẻ cho tất cả các đối tượng trong một chương trình.

Function to count the number of objects in a Class

```
#include <iostream>
using namespace std;
class Student {
private:
   string name;
   int age;
public:
  static int numberOfStudents;  // static member
   Student(string name, int age) {
      this->name = name;
      this->age = age;
      numberOfStudents++;
   }
};
int Student::numberOfStudents = 0; //initialized numberOfStudents = 0
int main() {
    Student s1("An", 19);
    Student s2("Binh", 20);
    Student s3("Chi", 19);
    cout << s1.numberOfStudents;      // Output: 3
}
```

NOTES:

- Vì không thuộc đối tượng nào nên **static** chỉ có thể gọi phương thức **static** khác, không thể gọi các phương thức **non-static** trong cùng một lớp.

10 Tính đóng gói (Encapsulation)

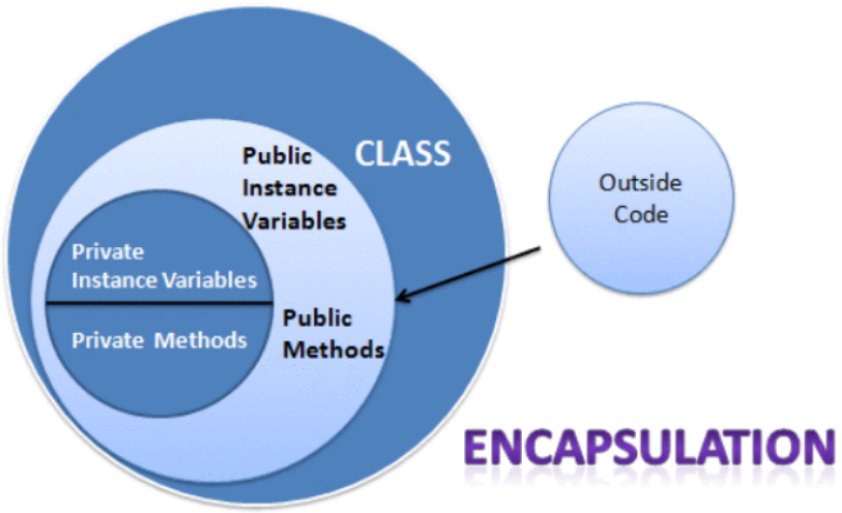
- Các **dữ liệu** và **phương thức** có liên quan với nhau được **đóng gói** thành các lớp để tiện cho việc quản lý và sử dụng. Tức là mỗi lớp được xây dựng để thực hiện một nhóm chức năng đặc trưng của riêng lớp đó.
- Ngoài ra, đóng gói còn để che giấu một số thông tin và chi tiết cài đặt nội bộ để bên ngoài không thể nhìn thấy.

Encapsulation in a Rectangle Class

```
class Rectangle{
    private:
        double length;
        double width;
    public:
    Rectangle(){
        this->length = 0;
        this-> width = 0;
    }
    Rectangle(double length, double width){
        this->length = length;
        this->width = width;
    }
    void setLength(double length){  // function to set length
        this->length = length;
    }
    double getLength(){             // function get length
        return length;
    }
    void setWidth(double width){   // function set width
        this->width = width;
    }
    double getWidth(){             // function get width
        return width;
    }
    double getArea(){              // function get Rectangle Area
        return length * width;
    }
    double getPerimeter(){       // function get Rectangle Parameter
        return 2 * (length + width);
    }
};
```

Khi nói đến một đối tượng **"Hình Chữ Nhật"** ta cần quan tâm đến chiều dài, chiều rộng, chu vi, diện tích của hình chữ nhật.

Ta đóng gói các thuộc tính đó lại thành một **Class Rectangle**, rất hiệu quả trong việc lưu trữ, cập nhật, tái sử dụng.



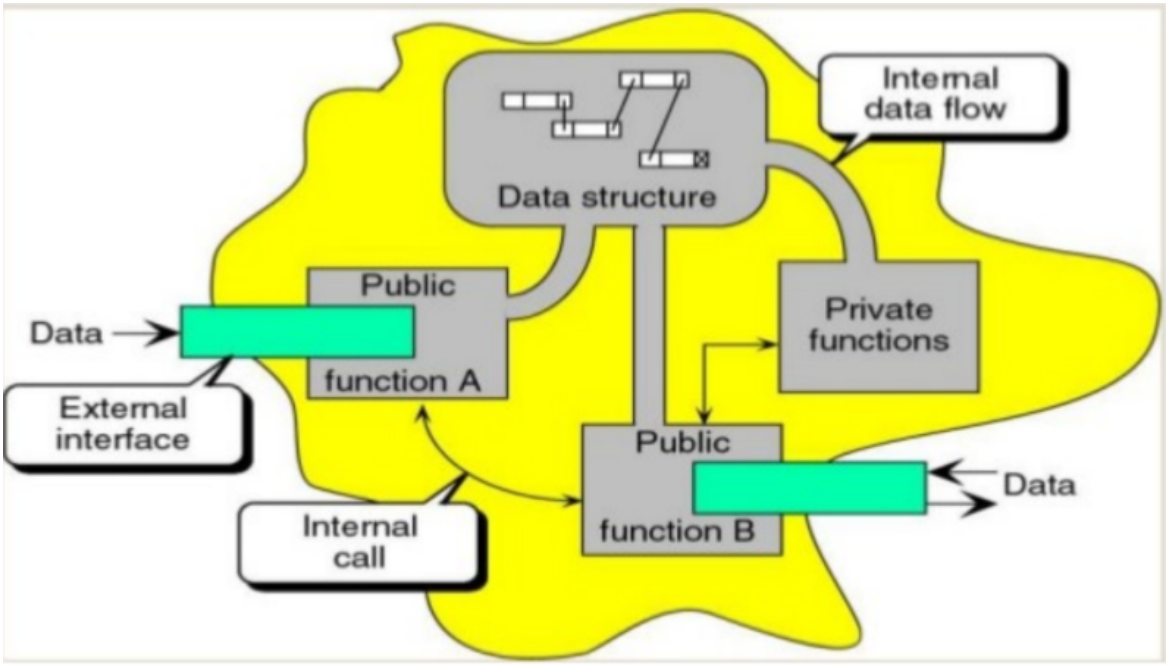
11 Tính trừu tượng (Abstraction)

- **Trừu tượng(Abstraction)** có nghĩa là tổng quát hóa một cái gì đó lên, không cần chú ý chi tiết bên trong. Tức là người dùng không cần biết một hàm được implement như thế nào nhưng vẫn có thể gọi nó ra để sử dụng.
  Ví dụ : Một đối tượng thuộc class **Rectangle** người dùng không cần biết công thức tính chu vi hay diện tích hình chữ nhật tuy nhiên có thể dùng toán tử **(.)** (đối với đối tượng tĩnh) hoặc **(→)**(đối với đối tượng động) để gọi ra giá trị chu vi/ diện tích được lập trình sẵn trong class **Rectangle.**

Trong class **Rectangle** trên ta sử dụng các **public** Method **get()**, **set()** để thao tác với object tạo ra bởi class **Rectangle**. Trong khi đó các **attributes** của class được hidden bằng keyword **private** và người dùng không thể thay đổi được.

```
class Rectangle;
int main(){
    Rectangle obj;
   obj.setLength(5);
   obj.setWidth(4);
   cout << obj.getPerimeter() << " " << obj.getArea(); // 18 20
   return 0;
}
```

- Với **Abstraction** ta có thể kiểm soát được hành vi của người dùng, chỉ cấp phép cho họ sử dụng các chức năng trong phạm vi của mình, bảo mật được mã nguồn nhưng vẫn đáp ứng được nhu cầu sử dụng.



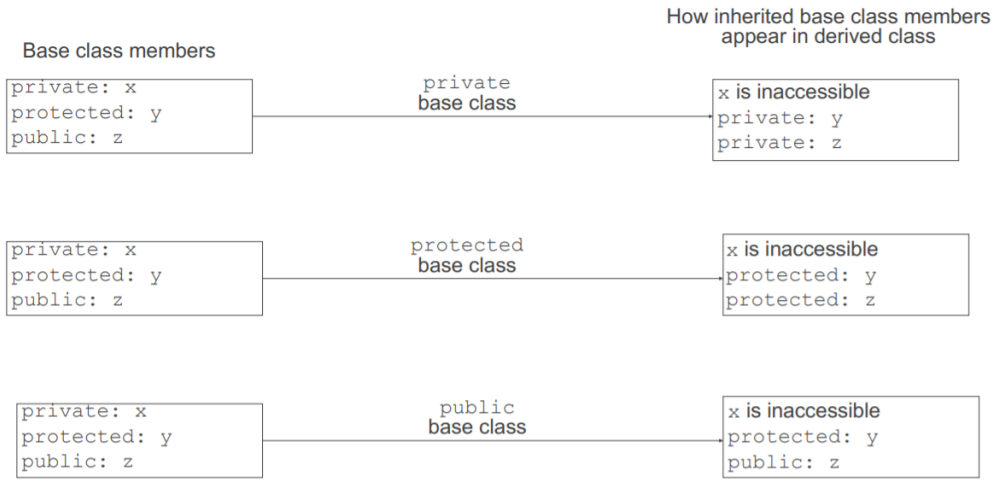
12 Tính kế thừa (Inheritance)

- **Tính kế thừa(Inheritance)** cho phép xây dựng một lớp mới dựa trên các định nghĩa của lớp đã có. Có nghĩa là lớp cha **(base class)** có thể chia sẻ dữ liệu và phương thức cho các lớp con **(derived class)**. Các lớp con khỏi phải định nghĩa lại, ngoài ra có thể mở rộng các thành phần kế thừa và bổ sung thêm các thành phần mới.
- Tái sử dụng mã nguồn 1 cách tối ưu, tận dụng được mã nguồn.
- Một số loại kế loại kế thừa thường gặp: đơn kế thừa, đa kế thừa, kế thừa đa cấp, kế thừa thứ bậc.
- C++ là một ngôn ngữ **đa kế thừa**

Inheritance of Student Class from Person Class

```cpp
// Base Class
class Person{
private:
    string name;
    int age;
public:
    Person(string name, int age){
        this->name = name;
        this->age = age;
    }
};
// Derived Class
class Student : public Person{
private:
    double gpa;
public:
    Student(string name, int age, double gpa) : Person(name,age){
        this-> gpa = gpa;
    }
};
```

Inheritance Access



Access Scope

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

Một số thành phần Derived Class không thể thừa kế từ Base Class

- Constructors, Destructors và Copy Constructors của Base Class.
- Overloaded Operators của Base Class.
- Hàm bạn của Base Class.

13 Tính đa hình (Polymorphism)

- **Đa hình (Polymorphism)** là khái niệm với cùng một phương thức cùng tên ở nhiều lớp con khác nhau, lớp cơ sở sẽ hiểu được là lớp dẫn xuất nào đang gọi để thi hành phương thức của lớp dẫn xuất đó.
  Ví dụ: Ta có 2 con vật: chó, mèo. Cả 2 con vật này đều là lớp **Animals**. Nhưng khi ta bảo cả 2 động vật kêu thì con chó sẽ kêu "gâu gâu", con mèo sẽ kêu "meo meo".

Implement Polymorphism in Animal Sounds

```
class Animal {
public:
  virtual void sound() {
     cout << "some sound" << endl;
  }
};
class Dog : public Animal {
public:
  void sound() {
     cout << "wow wow" << endl;
  }
};
class Cat : public Animal {
public:
  void sound() {
     cout << "meow meow" << endl;
  }
};
int main() {
  Animal* animals[3];
  animals[0] = new Animal();     //some sound
  animals[1] = new Dog();       // wow wow
  animals[2] = new Cat();       // meow meow
  for (int i = 0; i < 3; i++) {
     animals[i]->sound();
  }
  return 0;
}
```

Virtual Keyword:

- **Virtual:** là từ khóa để khai báo một phương thức ảo.
- **Virtual Method:** là phương thức chỉ có ở lớp cơ sở các lớp dẫn xuất có thể có hoặc không, nếu có thì khi khai báo sử dụng, nó sẽ sử dụng của chính nó. Nếu không có phương thức ảo của lớp cha sẽ được sử dụng.
- **Virtual Function:** Dùng để lớp cơ sở (cha) điều hướng phương thức đó về đúng lớp dẫn xuất (con) mà phương thức đó thuộc về.
- **Pure Virtual:** là phương thức chỉ có ở lớp cở sở, không bắt buộc lớp cha phải định nghĩa. Nhưng ở lớp dẫn xuất con bắt buộc phải có phương thức cùng tên với phương thức ảo của lớp cơ sở.

```
class Animal{
    public:
        virtual void sound() = 0; // Pure Virtual
}
```

Overloading & Overriding:

- **Overloading:** tức là tạo ra nhiều phương thức ở lớp cơ sở, có cùng tên nhưng khác nhau về danh sách tham số đầu vào (argument).
- **Overriding:** tạo ra một phương thức trong lớp dẫn xuất có cùng loại với một phương thức trong lớp cơ sở. Và lớp dẫn xuất sẽ định nghĩa lại phương thức đó cho riêng mình.

14 Template

- **Template (khuôn mẫu)** là một từ khóa được sử dụng để tạo ra các phương thức các lớp với nhiều loại kiểu dữ liệu khác nhau mà không cần phải viết lại nhiều lần.
- **Class Template:** Khuôn mẫu lớp cho phép tạo các lớp khuôn mẫu tổng quát, bao gói dữ liệu và xử lý dữ liệu vào cùng một nơi, sử dụng một cách tổng quát.
- **Function Template:** Khuôn mẫu hàm cho phép định nghĩa các hàm/phương thức tổng quát dùng cho các kiểu dữ liệu tùy ý.

Implement Stack using template class<T>

```
template <class T>
class Stack {
   private:
      vector<T> st;  // elements
   public:
      void push(T const&); // push element
      void pop();             // pop element
      T top() const;          // return top element
      bool empty() const {    // return true if empty.
         return st.empty();
      }
};
template <class T>
void Stack<T>::push (T const& elem) {
   st.push_back(elem);
}

template <class T>
void Stack<T>::pop () {
   if (st.empty()) {
      throw out_of_range("Stack<>::pop(): empty stack");
   }
   // remove last element
   st.pop_back();
}

template <class T>
T Stack<T>::top () const {
   if (st.empty())
      throw out_of_range("Stack<>::top(): empty stack");
   // return copy of last element
   return st.back();
}
```

15 Exception Handling

- **Try(thử)**: tìm kiếm những Exception (ngoại lệ) từ người dùng và Throw (ném) nó về một nơi để xử lý.
- **Throw(ném)** Dùng để cho người dùng ném một ngoại lệ. Người dùng có thể throw một câu thông báo hoặc 1 biến bất kỳ. Mỗi "throw" phải có ít nhất một "catch".
- **Catch (bắt):** Khi ngoại lệ được ném đi thì đây là nơi bắt ngoại lệ để xử lý hoặc thông báo.

Sử dụng **Exception Handling** để chủ động **throw** ra lỗi khi input đầu vào không hợp lệ, lập tức kết thúc chương trình chứ không để chương trình bị crash khi chạy.

Implement an Exception Devide by 0

```
#include <iostream>
using namespace std;
int main() {
   try {
      while (true) {
         int a, b;
         cin >> a >> b;
         if (b == 0)
            throw b;
         cout << a / b << endl;
      }
   }
   catch (int exception){
      cout << "Cannot division by zero !";
   }
   return 0;
}
```

16 Tài liệu tham khảo

- **Geeksforgeeks C++:** https://www.geeksforgeeks.org/c-plus-plus/
- **Web learncpp.com:** https://www.learncpp.com/
- **Tutorialspoint:** https://www.tutorialspoint.com/cplusplus/index.htm
- **Slide Fundamental Programming, HCMUT**