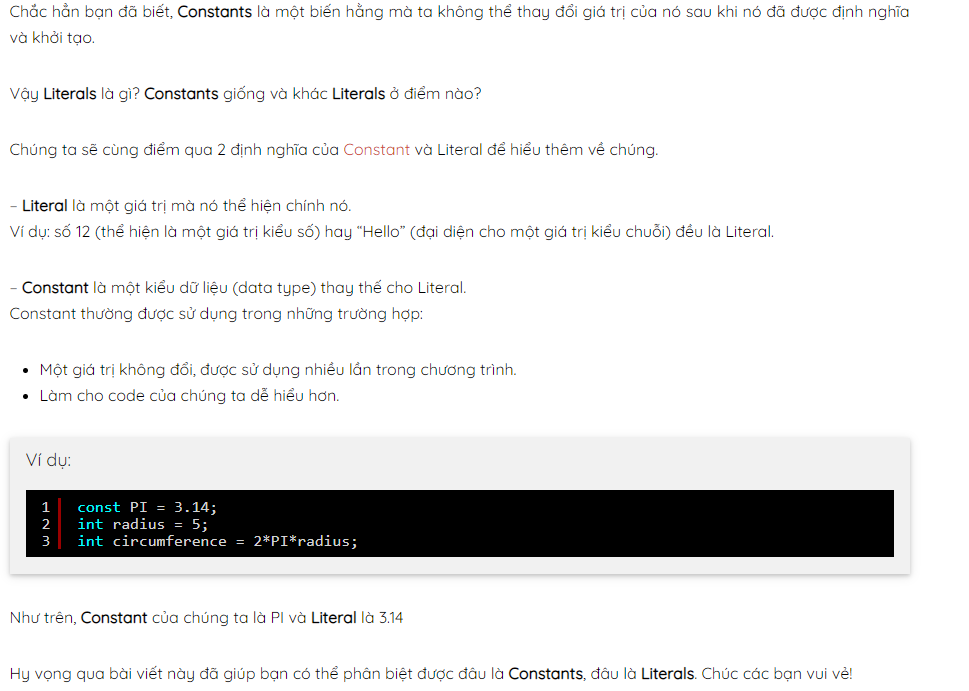
# Test one drive 2222

# Build with terminal

g++ main.cpp -o main.o



# Declaration and Definition

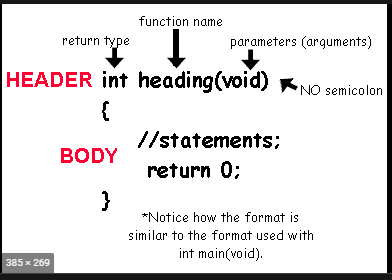
|  |  |  |
| --- | --- | --- |
| Declaration | Definition | Note |
| -Khi declare 1 biến, 1 functioc hoặc 1 class có nghĩa là: có gì đó vs cái tên này và dung kiểu dữ liệu này  -Báo cho compiler biết là thứ nãy đã được định nghĩa ở đâu đó trong project | Cung cấp đầy đủ thông tin quan trọng để tạo ra 1 cái gì đó  Compiler sẽ biết:   * Tên * Kiểu dữ liệu * Vị trí được lưu * Giá trị trả về   Defining a function means providing a function body; defining a class means giving all of the methods of the class and the fields | Có thể declare and define at the same time, nhưng k cần thiết |
|  |  |  |

In summary:

* A declaration cung cấp các thuộc tính cơ bản: its type and its name.
* A definition provides all of the details of that symbol--if it's a function, what it does; if it's a class, what fields and methods it has; if it's a variable, where that variable is stored.
* Thông thường, compiler chỉ cần có a declaration để biên dịch thành object file, và expect linker có thể tìm definition từ nơi khác( các file khác,…)
* Nếu không tìm được define, nghĩa là chỉ được declare , sẽ có lỗi tại link time : “undefined symbols”

# FUNCTION

## 103.Defining Function : 4 phần



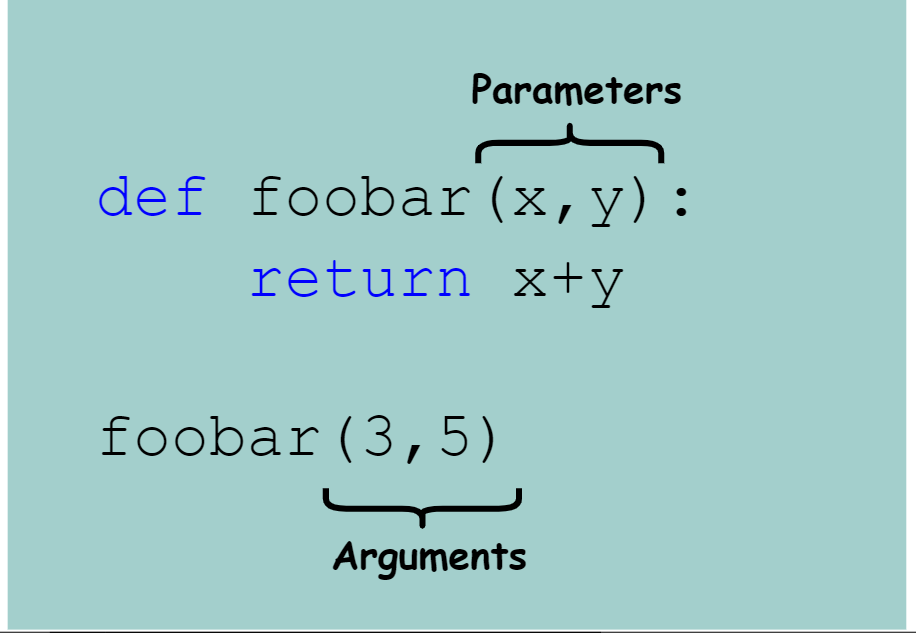
- Name

- Parameter list:

* Parameter: biến được truyền vào khi định define function, xem như để giữ chỗ chứ không có giá trị
* Argument : giá trị được truyền vào và sử dụng trong fucntion

A **parameter** is a variable in a function definition. It is a placeholder and hence does not have a concrete value.

An **argument** is a value passed during function invocation.



-Return type

-body

## 103.1. Call function

- Compiler must know the function detail BEFORE it is called!

Vd: function say\_hello được gọi trong hàm say(), nhưng trước hàm say() thì say\_hello() chưa được define hay declare, gây ra lỗi

* error: 'say\_hello' was not declared in this scope|

void say();

int main()

{

    say();

}

void say()

{

    say\_hello();

}

void say\_hello()

{

    cout << "hello" << endl;

}

Vd2: khi 1 hàm được declare, được gọi nhưng không được define (như hình)

* error: |undefined reference to `say()'|

void say();

int main()

{

    say();

}

## 104. Function Prototypes

**The compilers must know about a function before it is used**

int function\_name();            //prototype

int maifunction\_namen()

{

    statements();

    return 0;

}

Function prototype: gồm return type, function name, parameter, không có body

**Solution:**

1. Define function trước khi call
2. Use function prototype: declare function
   * Cho compiler biết function này sẽ được define ở đâu đó
   * Còn được gọi là forward declaration
   * Được đặt ở đầu chương trình

## 105.Function parameters

Data được truyền vào function

### parameters vs Argument

Trong **function call** thì gọi là **argument**

Trong **function definition** thì gọi là **parameters**

* Parameter: biến được truyền vào khi định define function, xem như để giữ chỗ chứ không có giá trị
* Argument : giá trị được truyền vào và sử dụng trong fucntion

Chúng phải match in number, order and type

### Pass by value

* When you pass **data** into a function it is passeded-by-value
* A copy of the data is passed to the function
* Whatever changed you make to the parameter in the function does NOT affect the argement that was passed in

### Formal vs Actual parameters(argument):

* + Formal parameters: the paramneter defined in the function header
  + Actual parameters( thường được gọi là arguments): the actual parameter(value) used in the function call

Vd:

## 106 Default Argument Values:

double dientich(double r, double pi = 3.14);

* chỉ được nằm trong declare hoặc d=efinition, nên nằm trong declare
* nếu không truyền vào thì sẽ nhận giá trị mặc định
* nếu được truyền vào trong function call thì sẽ nhận giá trị truyền vào đó
* double dientich(double r, double pi = 3.14);
* double dientich(double r, double pi)
* {
* return r\*r\*pi;
* }
* int main()
* {
* /\*Default argument\*/
* cout << dientich(10.1) << endl;
* /\*\*/
* cout << dientich(10.1, 3);
* }

## 107 Overloading Functions:

Nhiều function có cùng tên nhưng khác return type and parameter

int add\_numbers(int a, int b);

double add\_numbers(double a, double b);

Khi function được call, compiler will check the type of the argument in a function and chonj match function để implement

108. Passing array to function: cần truyền cả số phần tử vào

Lưu ý đối với mảng, khi khai báo một mảng ở hàm thì mạc định nó là một tham chiếu chứ không phải là tham trị.

void array\_handle(int array[])          //pass by refence

Nếu muốn không thay đổi giá trị của các phần từ trong array thì dung const

void array\_handle(const int array[])

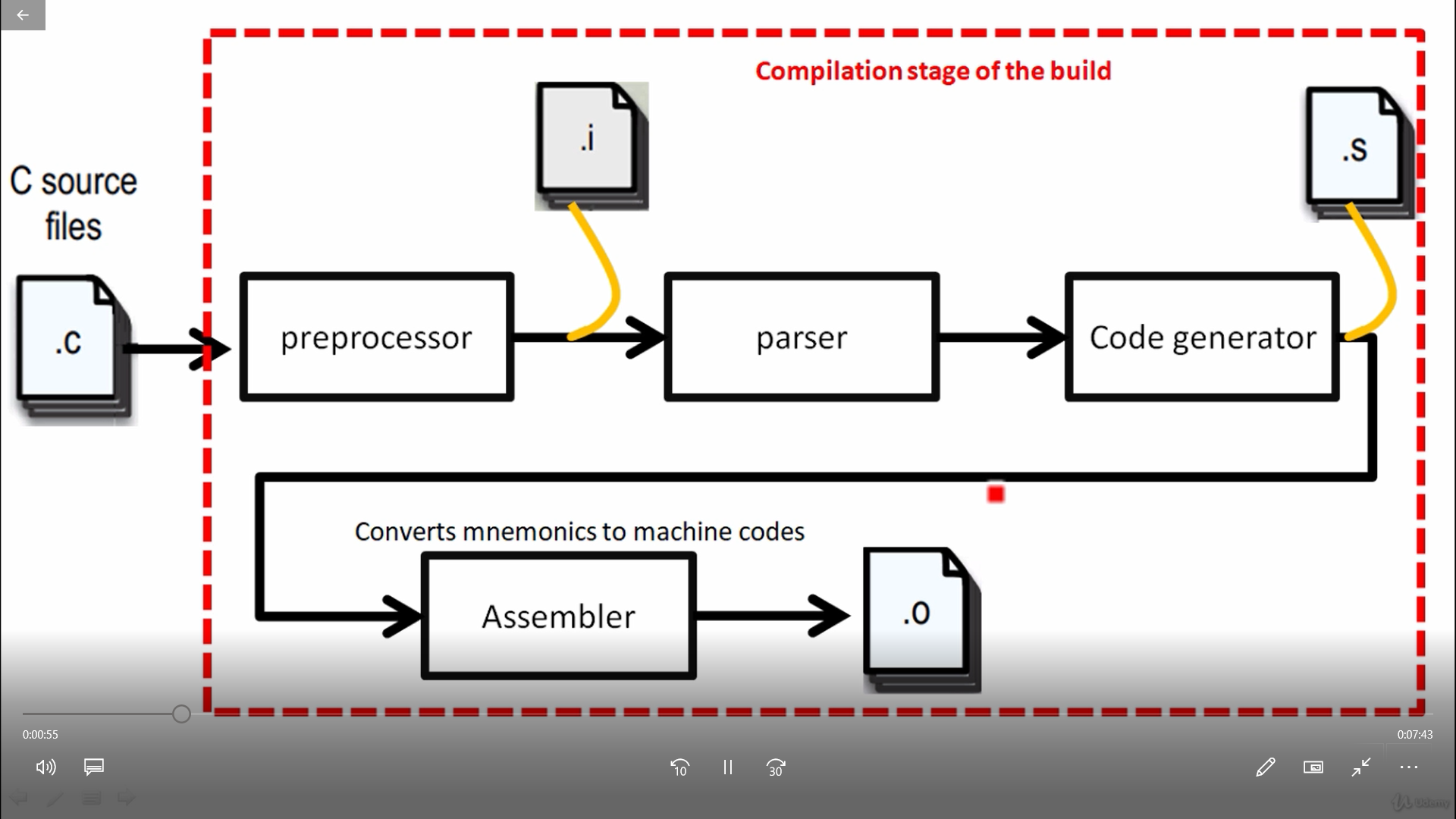
## 109. Pass by Reference

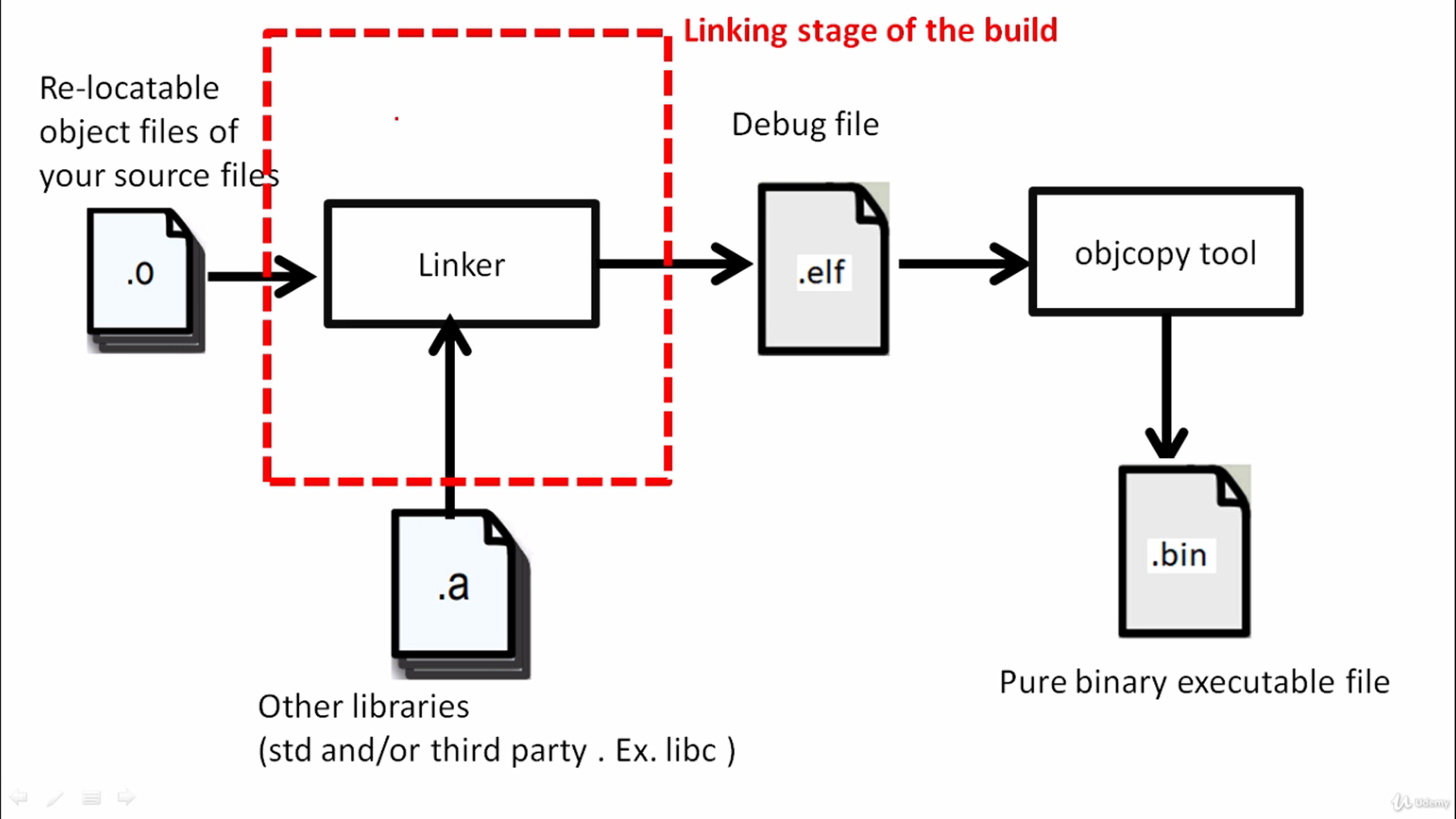
void swap2(int &a, int &b);

void swap2(int\* a, int\* b);

|  |  |
| --- | --- |
| Pass by value | Pass by reference |
| Tạo 1 copy của actucal parameter, rồi truyền vào function  Yêu cầu nhiều memory  requires more time as it involves copying  không ảnh hướng original value | Truyền địa chỉ vào function  Yêu cầu ít memory hơn  requires a less amount of time as there is no copying.  Có thay đổi sẽ ảnh hướng tới biến được truyền vào |
|  |  |
|  |  |

## 112. Inline function





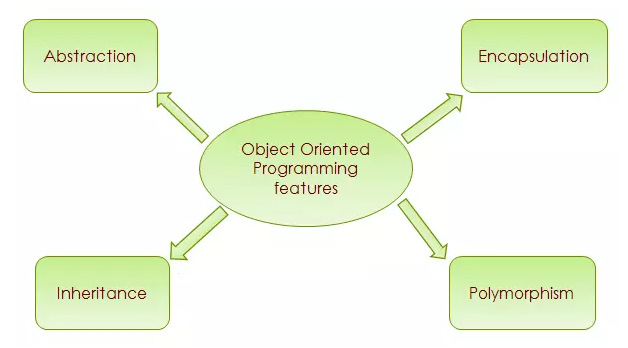
|  |  |  |
| --- | --- | --- |
| **Macro function** | **Inline Function** | **Function** |
|  |  |  |
| preprocessor |  |  |
| sd |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

## 113. Recursive Functions

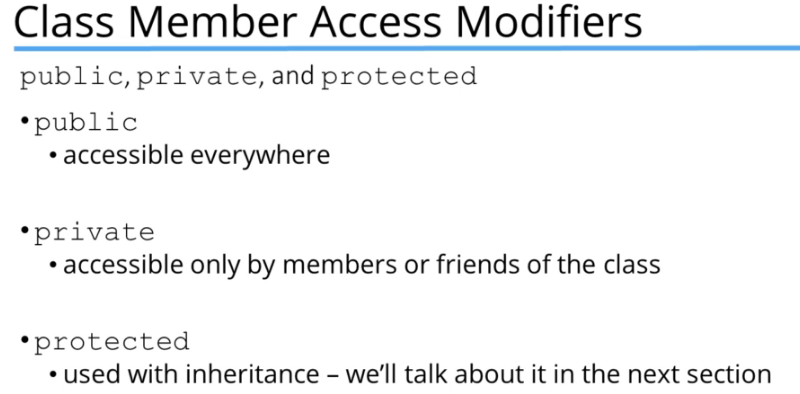
A recursive functions is a function that call itself

# OOP

1. Các đặc tính cơ bản của OOP

* **Tính đóng gói (Encapsulation)**
* **Tính kế thừa (Inheritance)**
* **Tính đa hình (Polymorphism )**
* **Tính trừu tượng(Abstraction)**
* 

## 139. Public and Private



Các biến và hàm trong private chỉ có thể dùng ở public cả class đó

|  |  |  |
| --- | --- | --- |
| Public | Private | protected |
| Ai cũng access được | Chỉ nội trong class nó khai báo | Nội trong class nó khai báo và class kế thừa class đó |

Vd:

class Lophoc

{

    public:

        int variable;

        void function1();

    private:

        int private\_variable;

        void private\_function();

};

## Constructor : hàm dựng vs Destructor

Constructor

* Tên hàm dựng giống tên class, được khai báo trong public
* Khi creat 1 object(class) thì hàm dựng auto được thực thi
* Có thể có nhiều hàm dựng, tùy thuộc vào define mà hàm dựng nào được thực thi(xem code)
* No return type
* Can be overloaded( nhiều hàm vs parameter khác nhau)

#include <iostream>

#include <string>

using namespace std;

class LophocLaptrinh

{

    public:

        LophocLaptrinh()        //ten ham dung giống tên class

        {

            cout <<"ham dung - constructor" << endl;

        }

        LophocLaptrinh(string TenGi)        //ten ham dung giống tên class

        {

            cout <<TenGi<< endl;

        }

    public:

        void SetName(string TenGi)

        {

            name = TenGi;

        }

        string GetName()

        {

            return name;

        }

    private:

        string name;

};

int main()

{

    LophocLaptrinh Lophoc1;

    LophocLaptrinh Lophoc2("ham dung - Le Quang Linh");

    return 0;

}

Destructor

* Giống với tên class nhưng có dấu ~
* No return type, no parameter
* Chỉ được có 1 destructor / 1 class- cannot overload
* Useful to relase memory and other resources
* Được thực thi khi 1 object(class) go out of scope hoặc delete a pointer to an object

Delegating Constructor: ủy quyền

Constructor này gọi constructor khác để implement

#include "Player.h"

Player::Player()

    :Player{"delegating constructor"}

{

}

## Inhetiritance: Tính kế thừa

Class cha: base class, parent class

Class con: derived class, child class

Các hàm concai có thể access các hàm biến trong các hàm chame( puclic và protected)

Syntax:

class ConCai: public ChaMe

{

    public:

        ConCai();

        ~ConCai();

        void XinChao();

    protected:

    private:

};

## ****Polymorphism: Tính đa hình****

Nếu hàm ở “class cha” có hàm bth(k phải hàm ảo), hàm con có 1 hàm cùng tên: class hoặc con trỏ kiểu “class con” thì dùng hàm của hàm con

Class hoặc con trỏ kiểu “class cha” (dù có trỏ tới địa chỉ của “class con” thì vẫn dùng hàm ở “class cha”

    ChaMe chame, \*p;        //class, con trỏ kiểu class cha

    chame.XinChao();        //lấy hàm XinChao() ở class ChaMe

    ConCai con, \*pconcai;

    con.XinChao();      //lấy hàm XinChao() ở class ConCai

    pconcai->XinChao(); //lấy hàm XinChao() ở class ConCai

    p = &con;

    p->XinChao();       //lấy hàm XinChao() ở class ChaMe

**Hàm ảo**: các class con được phép định nghĩa lại các hàm ảo thuộc class cha

virtual void Skill1(){}

**Hàm thuần ảo**: pure virtual : các class con **bắt buộc** phải định nghĩa lại hàm thuần ảo ở class cha

virtual void skill1() = 0;

**Tính đa hình:** Khai báo con trỏ ở dạng “class cha” và trỏ tới địa chỉ của “class con”

# Container\_Vector

Array: có kích thước trước

Vector : gần giống 1 array nhưng không có kích thước, đẩy vào bao nhiều thì nó sẽ tự động tăng kích thước lên để phù hợp

Iterator: xem như 1 con trỏ, trỏ tới 1 vùng nhớ( tựa tựa stack pointer)

A picture containing text, clock, screenshot

Description automatically generated

Iterator sẽ trỏ từ begin() với end()

Size() là kích thước đang được sử dụng của vector

Capacity() là kích thước thực sự của vector

Các hàm:

.**resize**(x): thay đổi kích thước **size**() của vector

Nếu x < size() thì lỗi

Nếu x > size() : sẽ tăng size() lên, capacity() sẽ tăng tương ứng

Các giá trị chưa được gán từ trước sẽ = 0

.**reserve** (x): cấp phát trước kích thước thực **capacity**() của vector

Nếu x < capacity() thì capacity() không đổi

Nếu x > capacity() thì capacity() = x