Phạm VI TRUY CẬP : PUBIC , PRIVATE, PROTECTED

2 MEMBER : PROPERTY , METHODS( FUNCTIONS)

PUBLIC : MEMBERS IN PUBLIC CAN ACCESS

PRIVATE : THAT CLASS CAN ACCESS

PROTECTED: THAT CLASS CHILD CAN ACCESS BUT THIS OBJECT CAN’T

CẤU TRÚC CLASS: CONTRUCSTOR, DESTRUCTOR

CONTRUCSTOR: NO DEFINE SO MADE DEFAULT

DETRUCSTOR : IN FUNTIONS HAVING OBJECT AND SAVING IN STACK, AND IF STACK IS RECALLED IT’S KHỞI TẠO DESTRUCTOR RỒI MỚI THU HỒI

ENCAPSULATION: ( ĐÓNG GÓI)

EX: KHỞI TẠO CÁC PROPERTY ĐƯỢC SỬ DỤNG THÔNG QUA METHOD

VÍ DỤ PROPERTIY LÀ ID, NAME . PHẢI CÓ TÍNH ĐÓNG GÓI ĐỂ THÔNG QUA METHOD TRUY CẬP CHỨ KHÔNG ĐƯỢC THÔNG QUA TRUY CẬP PROPERTY TRỰC TIẾP TỪ OBJECT

* BẢO MẬT,

Inheritance ( kế thừa ):

EX: trong web trường có các đặc tính : tên , tuổi ,id

Khởi tạo class parent có các thuộc tính chung

* Class con kế thừa lại class parent, quá trình này complier nó đổ các thuộc tính đó từ parent -> child ( quá trình này xảy ra trong tiền xử lí )

Kế thừa public : kế thừa public hết từ cha vào con

Kế thừa proted cha sang con

Kế thừa private ko được

Kế thừa protected : cha-> protected con

Cha member pro -> con pro

Cháu kế thừa con đều pro

Kế thừa private: public cha-> private con

Proted cha -> private con

Private ko kế thừa được

Đa hình (polymorhism) :

Complier:

Giả sử viết 1 method có tên Cộng nhưng các input khác nhau

Khi objected chỏ tới method nhập 2 thông số kiểu int thì cái complier nó generate đoạn code là sum ( int,int). Giúp định nghĩa được cái sum interger

Nó tùy thuộc vào input mà gọi được hàm nào .

Template -> những input khác nhau phụ thuộc vào input mình nhập và nó generate đoạn code

Operator -> complider build đoạn code đó cho mình

RUNTIME: Được thể hiện khi mình chạy chương trình

EX

Hàm ảo có 2 loại :có định nghĩa và ko định nghĩa

Không định nghĩa thì còn được gọi là hàm thuần ảo, khởi tạo method và có giá trị =0 ( không có nội dung)

Ví dụ mình khởi tạo Class school có hàm virtual void display();

Kiểu là ko khai báo object từ class cha được,

Nhưng class con thì được ví dụ class giáo viên thì hiển thị giáo viên đang dạy

Class học sinh thì hiển thị học sinh đang học.

Và mình có thể khai báo con trỏ ptr ở object cha để trỏ đến các object con. Ví dụ Ptr = địa chỉ giáo viên. Ptr -> display();

Hoặc mình dùng vector để tạo dabase

Hàm thuần ảo ( có định nghĩa)

Ví dụ class school display có định nghĩa hiển thị trường

Thì mình khai báo object của class school có thể hiển thị tên trường

Và class sinh viên, giáo viên có thể kế thừa từ class school . Nhưng nó sử dụng display overloading của chính class con. Và nó được gọi là đa hình run time.

Trừu tượng

Ví dụ : mình khởi tạo Tính hệ số PID mà người dùng chỉ muốn nhập input là 3 thống kp,ki,kd ( họ khôgn quan tâm chương trình tính bên trong là gì)

Nên Khi mình tạo class tính toán PID, có các property các biến thì nó có tính đóng gói và các funtions tính toán được private lại.

Quá trình toán trong quá trình đó thì các object không được truy cập đến. Nên nó được gọi là tính trừu tượng

TEMPLATE

Funtions or class có thể dùng chung cho nhiều kiểu dữ liệu khác nhau

Template được sử dụng khi cần định nghĩa các hàm hoặc lớp có tính tổng quát và có thể áp dụng cho nhiều kiểu dữ liệu khác nhau. Thay vì viết các hàm hoặc lớp với nhiều phiên bản khác nhau cho từng kiểu dữ liệu, bạn có thể sử dụng template để tự động sinh mã tương ứng cho các kiểu dữ liệu cụ thể.

#include <iostream>

using namespace std;

// Khai báo một template với một kiểu dữ liệu tổng quát T

template <typename T>

T getMax(T a, T b) {

return (a > b) ? a : b;

}

int main() {

// Sử dụng hàm template với các kiểu dữ liệu khác nhau

cout << "Max of 3 and 7: " << getMax(3, 7) << endl; // int

cout << "Max of 5.4 and 2.1: " << getMax(5.4, 2.1) << endl; // double

cout << "Max of 'a' and 'b': " << getMax('a', 'b') << endl; // char

return 0;

}

<details>

<summary><strong>BÀI 1: COMPILER - MACRO</strong></summary>

# BÀI 1: COMPILER - MACRO

## 1.1 Compiler

Quá trình biên dịch là quá trình chuyển đổi từ ngôn ngữ bậc cao (C, C++, Pascal, Java, C#,...) sang ngôn ngữ máy để máy tính có thể hiểu và thực thi.

- B1: Giai đoạn tiền xử lý (Pre-processor): Nhận mã nguồn, xóa bỏ tất cả mọi chú thích, comment của chương trình, xử lý các chỉ thị tiền xử lý.

- B2: Giai đoạn dịch ngôn ngữ bậc cao sang Assembly (Compiler): Phân tích và chuyển ngôn ngữ bậc cao sang ngôn ngữ bậc thấp Assembly.

- B3: Giai đoạn dịch Assembly sang ngôn ngữ máy (Assembler): Dịch chương trình sang mã máy 0 và 1 để ra các file Object (.o).

- B4: Giai đoạn liên kết (Linker): Liên kết csac file Object tạo thành 1 file duy nhất.

- B5: Giai đoạn thực thi (Loader): File chạy cuối cùng sẽ được nạp lên RAM và thực thi bởi CPU.

![image](https://github.com/khaitq17/C-Cpp/assets/159031971/a62be2b0-2812-43a0-afa7-5c22a9e1f3d4)

## 1.2 Macro

MACRO là một từ được sử dụng để chỉ thông tin được xử lý trong trình tiền xử lý. Chia thành 3 nhóm chính:

- Tệp tiêu đề hoặc Thư viện: #include

- Thay thế macro\_name bằng macro\_value trước khi biên dịch thành tệp .bin: #define, #undef

- Chỉ thị biên dịch có điều kiện: #if, #elif, #else, #ifdef, #ifndef

### 1.2.1 Tệp tiêu đề hoặc Thư viện: #include

```

#include<stdio.h> // Thư viện

#include "yourfile.h" // Tệp tiêu đề (File header)

```

### 1.2.2 Macro #define, #undef

#define

```

#include <stdio.h>

#define MESSAGE "Hello World" // Định nghĩa macro

#define TRUE 1 // Định nghĩa macro

#define FALSE 0 // Định nghĩa macro

#define SUM(a,b) a+b // Định nghĩa macro

int main() {

printf("String: %s\n", MESSAGE);

printf("Custom boolean TRUE: %d\n", TRUE);

printf("Custom boolean FALSE: %d\n", FALSE);

printf("Arithmetic: 3+5=%d\n", SUM(3,5));

return 0;

}

```

#undef

```

#define MESSAGE "Hello World!" // Định nghĩa macro MESSAGE

#undef MESSAGE // Hủy bỏ định nghĩa macro MESSAGE

#define MESSAGE "Good luck!" // Định nghĩa lại macro MESSAGE

```

### 1.2.3 Chỉ thị tiền xử lý điều kiện (#ifdef, #ifndef, #if, #endif, #else, #elif)

```

#ifndef \_HEADERFILE\_H

#ifdef \_HEADERFILE\_H

#if (Condition 1){

//

} #elif (Condition 2){

//

} #else (Condition 3){

//

}

#endif // if

#endif // ifndef

```

</details>

<details>

<summary><strong>BÀI 2: STDARG - ASSERT</strong></summary>

# BÀI 2: STDARG - ASSERT

## 2.1 Thư viện <stdarg.h>

Thư viện stdarg cung cấp các phương thức để làm việc với các hàm có số lượng tham số đầu vào không cố định.

- `va\_list`: Là một kiểu dữ liệu để đại diện cho danh sách các đối số biến đổi.

- `va\_start`: Bắt đầu một danh sách đối số biến đổi. Nó cần được gọi trước khi truy cập các đối số biến đổi đầu tiên.

- `va\_arg`: Truy cập một đối số trong danh sách. Hàm này nhận một đối số của kiểu được xác định bởi tham số thứ hai

- `va\_end`: Kết thúc việc sử dụng danh sách đối số biến đổi. Nó cần được gọi trước khi kết thúc hàm.

Ví dụ:

```

#include <stdio.h>

#include <stdarg.h>

void display(int count, ...) {

va\_list args; // args là 1 con trỏ, dùng để các lưu địa chỉ các tham số truyền vào

va\_start(args, count); // va\_start () tạo vùng nhớ, địa chỉ đầu tiên của nó là địa chỉ biến count đc lưu trong args

for (int i = 0; i < count; i++) {

printf("Value at %d: %d\n", i, va\_arg(args,int));

}

va\_end(args);

}

int main()

{

display(5, 5, 8, 15, 10, 13); // count = 5

return 0;

}

```

Output:

```

Value at 0: 5

Value at 1: 8

Value at 2: 15

Value at 3: 10

Value at 4: 10

```

Ví dụ:

```

#include <stdio.h>

#include <stdarg.h>

typedef enum {

TEMPERATURE\_SENSOR,

PRESSURE\_SENSOR

} SensorType;

void processSensorData(SensorType type, ...) {

va\_list args;

va\_start(args, type);

switch (type) {

case TEMPERATURE\_SENSOR: {

int numArgs = va\_arg(args, int);

int sensorId = va\_arg(args, int);

float temperature = va\_arg(args, double); // float được promote thành double

printf("Temperature Sensor ID: %d, Reading: %.2f degrees\n", sensorId, temperature);

if (numArgs > 2) {

// Xử lý thêm tham số nếu có

char\* additionalInfo = va\_arg(args, char\*);

printf("Additional Info: %s\n", additionalInfo);

}

break;

}

case PRESSURE\_SENSOR: {

int numArgs = va\_arg(args, int);

int sensorId = va\_arg(args, int);

int pressure = va\_arg(args, int);

printf("Pressure Sensor ID: %d, Reading: %d Pa\n", sensorId, pressure);

if (numArgs > 2) {

// Xử lý thêm tham số nếu có

char\* unit = va\_arg(args, char\*);

printf("Unit: %s\n", unit);

}

break;

}

}

va\_end(args);

}

int main() {

processSensorData(TEMPERATURE\_SENSOR, 3, 1, 36.5, "Room Temperature");

processSensorData(PRESSURE\_SENSOR, 2, 2, 101325);

return 0;

}

```

Output:

```

Temperature Sensor ID: 1, Reading: 36.50 degrees

Additional Info: Room Temperature

Pressure Sensor ID: 2, Reading: 101325 Pa

```

## 2.2 Thư viện <assert.h>

- Cung cấp macro assert.

- Macro này được sử dụng để kiểm tra một điều kiện.

- Nếu điều kiện đúng (TRUE), không có gì xảy ra và chương trình tiếp tục thực thi.

- Nếu điều kiện sai (FALSE), chương trình dừng lại và thông báo một thông điệp lỗi.

- Dùng trong debug, dùng #define NDEBUG để tắt debug.

```

#include <stdio.h>

#include <assert.h>

int main() {

int x = 5;

assert(x == 5);

// Chương trình sẽ tiếp tục thực thi nếu điều kiện là đúng.

printf("X is: %d", x);

return 0;

}

```

Macro dùng để debug

```

#define LOG(condition, cmd) assert(condition && #cmd);

```

Ví dụ:

```

#include <assert.h>

#define ASSERT\_IN\_RANGE(val, min, max) assert((val) >= (min) && (val) <= (max))

void setLevel(int level) {

ASSERT\_IN\_RANGE(level, 1, 10);

// Thiết lập cấp độ

}

```

Ví dụ:

```

#include <assert.h>

#include <stdint.h>

#define ASSERT\_SIZE(type, size) assert(sizeof(type) == (size))

void checkTypeSizes()

{

ASSERT\_SIZE(uint32\_t, 4);

ASSERT\_SIZE(uint16\_t, 2);

// Kiểm tra các kích thước kiểu dữ liệu khác

}

```

</details>

<details>

<summary><strong>BÀI 3: POINTER</strong></summary>

# BÀI 3: POINTER

Con trỏ (Pointer) là một biến chứa địa chỉ bộ nhớ của một biến khác. Việc sử dụng con trỏ giúp chúng ta thực hiện các thao tác trên bộ nhớ một cách linh hoạt hơn.

- Khai báo:

```

int \*ptr; // Con trỏ trỏ đến kiểu int

char \*ptr; // Con trỏ trỏ đến kiểu char

float \*ptr; // Con trỏ trỏ đến kiểu float

```

- Lấy địa chỉ của một biến và truy cập giá trị

```

int a = 5;

int \*ptr = &a; // Con trỏ ptr chứa địa chỉ của a

int b = \*ptr; // b bằng giá trị mà ptr trỏ đến

```

- Kích thước của con trỏ phụ thuộc vào kiến trúc máy tính và trình biên dịch.

```

#include <stdio.h>

int main() {

int \*ptr;

printf("Size of pointer: %d bytes\n", sizeof(ptr));

return 0;

}

```

Kích thước của con trỏ là \*\*4 bytes\*\* với hệ thống \*\*32-bit\*\* hoặc \*\*8 bytes\*\* với hệ thống \*\*64-bit\*\*. Và kích thước của con trỏ sẽ \*\*KHÔNG\*\* phụ thuộc vào kiểu dữ liệu mà nó trỏ đến.

Ví dụ:

```

#include <stdio.h>

void swap(int \*a, int \*b)

{

int tmp = \*a;

\*a = \*b;

\*b = tmp;

}

int main()

{

int a = 10, b = 20;

swap(&a, &b);

printf("value a is: %d\n", a); //value a is: 20

printf("value b is: %d\n", b); //value b is: 10

return 0;

}

```

Ví dụ:

```

#include <stdio.h>

int main(){

int var = 5;

printf("Dia chi cua var la: %p\n", &var);

printf("Gia tri cua var la: %d\n", \*(&var));

int array[] = {1, 3, 5, 7, 9};

int \*ptr = array;

printf("Dia chi array: %p\n", array);

printf("Dia chi array: %p\n", ptr);

printf("Dia chi array: %p\n", &array[0]); // Giá trị của array cũng chính là địa chỉ của phần tử đầu tiên trong mảng

for(int i = 0; i<5; i++){

printf("Gia tri cua phan tu thu %d la: %d\n", i+1, \*(ptr+i));

}

return 0;

}

```

## 3.1 Void Pointer

\*\*Void Pointer\*\* thường dùng để trỏ để tới bất kỳ địa chỉ nào mà không cần biết tới kiểu dữ liệu của giá trị tại địa chỉ đó.

- Khai báo:

```

void \*ptr;

```

Ví dụ:

```

include <stdio.h>

#include <stdlib.h>

int main() {

char array[] = "Hello";

int value = 10;

double test = 17.5;

char letter = 'K';

void \*ptr = &value;

printf("value is: %d\n", \*(int\*)(ptr));

ptr = &test;

printf("value is: %f\n", \*(double\*)(ptr));

ptr = &letter;

printf("value is: %c\n", \*(char\*)(ptr));

return 0;

}

```

## 3.2 Function Pointer

\*\*Function Pointer\*\* (con trỏ hàm) là một biến mà giữ địa chỉ của một hàm, nó trỏ đến vùng nhớ trong bộ nhớ chứa mã máy của hàm được định nghĩa trong chương trình.

Trong ngôn ngữ lập trình C, con trỏ hàm cho phép truyền một hàm như là một đối số cho một hàm khác, lưu trữ địa chỉ của hàm trong một cấu trúc dữ liệu, hoặc thậm chí truyền hàm như một giá trị trả về từ một hàm khác.

Ví dụ:

```

#include <stdio.h>

void addition(int a, int b){

printf("Tong cua %d va %d la: %d\n", a, b, a+b);

}

void subtraction(int a, int b){

printf("Hieu cua %d va %d la: %d\n", a, b, a-b);

}

void multiplication(int a, int b){

printf("Tich cua %d va %d la: %d\n", a, b, a\*b);

}

void division(int a, int b){

if(b == 0){

printf("Phep chia khong hop le vi mau so bang 0!");

}

else{printf("Thuong cua %d va %d la: %.2f\n", a, b, a/(double)b);}

}

void calculation(void (\*ptr)(int, int), int a, int b){

printf("Chuong trinh tinh toan:\n");

ptr(a,b);

}

int main(){

void (\*ptr)(int, int) = &multiplication;

ptr(3,6); // Tich cua 3 va 6 la: 18

void (\*array[])(int, int) = {&addition, &subtraction, &multiplication, &division};

for(int i = 0; i < 4; i++){

array[i](17,5);

}

/\*

Tong cua 17 va 5 la: 22

Hieu cua 17 va 5 la: 12

Tich cua 17 va 5 la: 85

Thuong cua 17 va 5 la: 3.40

\*/

calculation(&addition, 17, 5); // Tong cua 17 va 5 la: 22

return 0;

}

```

## 3.3 Pointer to Constant

\*\*Pointer to Constant\*\* (Con trỏ hằng) là cách định nghĩa một con trỏ không thể thay đổi giá trị tại địa chỉ mà nó trỏ đến thông qua dereference nhưng giá trị tại địa chỉ đó có thể thay đổi.

- Khai báo:

```

int const \*ptr;

const int \*ptr;

```

Ví dụ:

```

#include<stdio.h>

int main(){

int var = 3;

int const \*ptr; // Khai báo con trỏ hằng

ptr = &var;

// \*ptr = 5; // Lỗi vì không thể thay đổi giá trị mà con trỏ trỏ đến

printf("value: %d\n", \*ptr); // value: 3

var = 10;

printf("value: %d\n", \*ptr); // value: 10

return 0;

}

```

## 3.4 Constant Pointer

\*\*Constant Pointer\*\* (Hằng con trỏ) định nghĩa một con trỏ mà giá trị nó trỏ đến (địa chỉ) không thể thay đổi, tức là khi con trỏ này được khởi tạo thì nó sẽ không thể trỏ tới địa chỉ khác.

- Khai báo:

```

int \*const ptr = &value;

```

Ví dụ:

```

#include<stdio.h>

int main(){

int value = 5;

int test = 10;

int const\* ptr = &value;

printf("value: %d\n", \*ptr);

// ptr = &test; // Lỗi vì hằng con trỏ ptr đã trỏ tới biến value nên không thể trỏ tới biến test

return 0;

}

```

## 3.5 NULL Pointer

\*\*Null Pointer\*\* là một con trỏ không trỏ đến bất kỳ đối tượng hoặc vùng nhớ cụ thể nào. Trong ngôn ngữ lập trình C, một con trỏ có thể được gán giá trị NULL để biểu diễn trạng thái null.

Sử dụng NULL Pointer thường hữu ích để kiểm tra xem một con trỏ đã được khởi tạo và có trỏ đến một vùng nhớ hợp lệ chưa. Tránh dereferencing (sử dụng giá trị mà con trỏ trỏ đến) một NULL Pointer là quan trọng để tránh lỗi chương trình.

Ví dụ:

```

#include <stdio.h>

int main() {

int \*ptr = NULL; // Gán giá trị NULL cho con trỏ 0x0000000

if (ptr == NULL) {

printf("Pointer is NULL\n");

} else {

printf("Pointer is not NULL\n");

}

int score\_game = 5;

if (ptr == NULL)

{

ptr = &score\_game;

\*ptr = 30;

ptr = NULL;

}

return 0;

}

```

Khi khai báo và sử dụng xong con trỏ, không dùng nữa thì gán con trỏ đó thành con trỏ NULL.

Ví dụ:

```

#include<stdio.h>

int main(){

int \*ptr = NULL;

printf("Dia chi: %p\n", ptr); // 0x00

ptr = NULL;

return 0;

}

```

## 3.6 Pointer to Pointer

\*\*Pointer to Pointer\*\* (Con trỏ đến con trỏ) là một kiểu dữ liệu trong ngôn ngữ lập trình cho phép bạn lưu trữ địa chỉ của một con trỏ.

Con trỏ đến con trỏ cung cấp một cấp bậc trỏ mới, cho phép thay đổi giá trị của con trỏ gốc. Cấp bậc này có thể hữu ích trong nhiều tình huống, đặc biệt là khi làm việc với các hàm cần thay đổi giá trị của con trỏ.

Ví dụ:

```

#include<stdio.h>

int main(){

int var = 3;

printf("Gia tri var: %d\n", var);

printf("Dia chi var: %p\n", &var);

printf("\n");

int \*ptr = &var;

printf("Gia tri ptr: %p\n", ptr);

printf("Dia chi ptr: %p\n", &ptr);

printf("\n");

int \*\*ptp = &ptr;

printf("Gia tri ptp: %p\n", ptp);

return 0;

}

```

Kết quả:

![image](https://github.com/user-attachments/assets/12987b23-bf07-46b8-8515-8a86eed5ddf4)

</details>

<details>

<summary><strong>BÀI 4: EXTERN - STATIC - VOLATILE - REGISTER</strong></summary>

# BÀI 4: EXTERN - STATIC - VOLATILE - REGISTER

## 4.1 Extern

- \*\*Extern\*\* được sử dụng để lấy 1 biến hoặc 1 hàm đã được khai báo trong 1 file nguồn khác vào chương trình hiện tại.

- Biến (hoặc hàm) đó là tham chiếu của một biến (hoặc hàm) cùng tên nào đó, đã được định nghĩa bên ngoài. Nó chỉ khai báo chứ không định nghĩa (cấp phát bộ nhớ cho biến).

- Biến (hoặc hàm) được tham chiếu phải được khai báo ở cấp độ cao nhất (toàn cục).

- Giúp quản lý liên kết giữa các file nguồn.

- Khi sử dụng `extern`, không được khai báo giá trị ban đầu cho biến.

Ví dụ:

File `test.c`:

```

#include<stdio.h>

int var = 5;

void tong(int a, int b){

printf("Tong cua %d va %d la: %d", a, b, a+b);

}

```

File `main.c`:

```

#include<stdio.h>

extern int var;

extern void tong(int a, int b);

int main(){

printf("var: %d\n", var);

tong(3,7);

return 0;

}

```

## 4.2 Static

### 4.2.1 Static local

- Biến \*\*Static local\*\* (Static cục bộ) được khai báo trong 1 hàm.

- Chỉ khởi tạo 1 lần duy nhất.

- Giữ giá trị của biến qua các lần gọi hàm.

- Được sử dụng hết vòng đời của chương trình.

Ví dụ:

```

#include<stdio.h>

void func(){

static int count = 0; // Biến static giữ giá trị qua các lần gọi hàm

count++;

printf("Count: %d\n", count);

}

int main(){

func(); // Count: 1

func(); // Count: 2

func(); // Count: 3

return 0;

}

```

### 4.2.2 Static global

- Biến \*\*Static global\*\*(Static toàn cục) được khai báo bên ngoài hàm.

- Hạn chế phạm vi chỉ được sử dụng trong file nguồn hiện tại.

- Không thể gọi ra và sử dụng trong file khác, kể cả khi dùng `extern`.

## 4.3 Volatile

- \*\*Volatile\*\* có nghĩa là không dự đoán được.

- Từ khóa `volatile` được sử dụng để báo hiệu cho trình biên dịch rằng một biến có thể thay đổi ngẫu nhiên, ngoài sự kiểm soát của chương trình.

- Việc này ngăn chặn trình biên dịch tối ưu hóa hoặc xóa bỏ các thao tác trên biến đó, giữ cho các thao tác trên biến được thực hiện như đã được định nghĩa.

- Trong thực tế, có 3 loại biến mà giá trị có thể bị thay đổi như vậy:

- Memory-mapped peripheral registers (thanh ghi ngoại vi có ánh xạ đến ô nhớ).

- Biến toàn cục được truy xuất từ các tiến trình con xử lý ngắt (interrupt service routine).

- Biến toàn cục được truy xuất từ nhiều tác vụ trong một ứng dụng đa luồng.

## 4.4 Register

Từ khóa `register` để cho một biến được sử dụng thường xuyên và có thể được lưu trữ trong một thanh ghi máy tính thay vì được lưu vào trong bộ nhớ RAM, nhằm tăng tốc độ truy cập.

- Trong kiến trúc của vi xử lý thì ALU (Arithmetic Logic Unit) đóng vai trò xử lý các tính toán số học. Dữ liệu đưa vào làm việc với ALU phải chứa trong một vùng đặc biệt, gọi là các thanh ghi (register), và ALU chỉ làm việc với đống thanh ghi đó.

![image](https://github.com/user-attachments/assets/fc9a0e16-a1df-40a3-acc2-4de18e65e086)

- Khi các biến khai báo trong chương trình sẽ được đặt ở bộ nhớ ngoài (RAM, …). Do đó với khai báo biến thông thường, để thực hiện một phép tính thì cần có 3 bước:

- Nạp giá trị từ vùng nhớ chứa biến vào register.

- Yêu cầu ALU xử lý register vừa được nạp giá trị.

- Đưa kết quả vừa xử lý của ALU ra ngoài vùng nhớ chứa biến.

Khi thêm từ khóa `register` để khai báo biến, thì tức là ta đã yêu cầu trình biên dịch ưu tiên đặc biệt dành luôn vùng register để chứa biến và giảm được các bước tính toán trên biến đó, giúp tăng tốc độ truy cập.

Ví dụ:

```

#include <stdio.h>

#include <time.h>

int main() {

// Lưu thời điểm bắt đầu

clock\_t start\_time = clock();

int i;

// Đoạn mã của chương trình

for (i = 0; i < 2000000; ++i) {

// Thực hiện một số công việc bất kỳ

}

// Lưu thời điểm kết thúc

clock\_t end\_time = clock();

// Tính thời gian chạy bằng miligiây

double time\_taken = ((double)(end\_time - start\_time)) / CLOCKS\_PER\_SEC;

printf("Thoi gian chay cua chuong trinh: %f giay\n", time\_taken);

return 0;

}

```

Kết quả:

```

Thoi gian chay cua chuong trinh: 0.003000 giay

```

Nhưng khi sử dụng `register int i`:

```

Thoi gian chay cua chuong trinh: 0.001000 giay

```

</details>

<details>

<summary><strong>BÀI 5: GOTO - SETJMP</strong></summary>

# BÀI 5: GOTO - SETJMP

## 5.1 Goto

- \*\*goto\*\* là một từ khóa trong ngôn ngữ lập trình C, cho phép chương trình nhảy đến một nhãn (label) đã được đặt trước đó trong cùng một hàm.

- Mặc dù nó cung cấp khả năng kiểm soát flow của chương trình, nhưng việc sử dụng `goto` thường được xem là không tốt vì nó có thể làm cho mã nguồn trở nên khó đọc và khó bảo trì.

- Trong 1 chương trình sử dụng nhiều vòng lặp lồng vòng lặp thì việc sử dụng `goto` sẽ tránh phải dùng lệnh `break;` nhiều lần.

- Lệnh `goto` chỉ dùng cục bộ trong hàm.

Ví dụ:

```

#include<stdio.h>

int main(){

int i = 0;

// Đặt nhãn start

start:

if(i >= 5){

goto end; // i >= 5 sẽ di chuyển đến nhãn end

}

printf("%d\n", i);

i++;

goto start; // Di chuyển đến nhãn start

// Đặt nhãn end

end:

printf("Ket thuc!\n");

return 0;

}

```

Kết quả:

```

0

1

2

3

4

Ket thuc!

```

## 5.2 Thư viện setjmp.h

\*\*setjmp.h\*\* là một thư viện trong ngôn ngữ lập trình C, cung cấp hai hàm chính là \*\*setjmp()\*\* và \*\*longjmp()\*\*.

Cả hai hàm này thường được sử dụng để thực hiện xử lý ngoại lệ trong C, mặc dù nó không phải là một cách tiêu biểu để xử lý ngoại lệ trong ngôn ngữ này.

- \*\*jmp\_buf\*\*: Kiểu dữ liệu được định nghĩa trong "setjmp.h"

```

jmp\_buf buffer; // Biến toàn cục

```

- \*\*int set\_jmp(BUF)\*\*: Nếu macro này trả về một cách trực tiếp từ lời gọi macro, thì nó trả về 0; nhưng nếu nó trả về từ một lời gọi hàm \*\*longjmp()\*\*, thì một giá trị khác 0 được trả về.

Ví dụ:

```

#include<stdio.h>

#include<setjmp.h>

jmp\_buf buf;

int exception\_code;

#define TRY if((exception\_code = setjmp(buf)) == 0)

#define CATCH(x) else if(exception\_code == x)

#define THROW(x) longjmp(buf, x)

double divide(int a, int b){

if(b == 0){

THROW(1);

}

return a/(double)b;

}

int main(){

double result;

TRY{

result = divide(17, 0);

printf("Result: %f\n", result);

} CATCH(1){

printf("Error: Divided by 0!\n");

} CATCH(2){

// Lỗi thứ 2

} // Các lỗi khác

return 0;

}

```

</details>

<details>

<summary><strong>BÀI 6: BITMASK</strong></summary>

# BÀI 6: BITMASK

\*\*Bitmask\*\* là một kỹ thuật sử dụng các bit để lưu trữ và thao tác với các cờ (flags) hoặc trạng thái. Có thể sử dụng bitmask để đặt, xóa và kiểm tra trạng thái của các bit cụ thể trong một từ (word).

\*\*Bitmask\*\* thường được sử dụng để tối ưu hóa bộ nhớ, thực hiện các phép toán logic trên một cụm bit, và quản lý các trạng thái, quyền truy cập, hoặc các thuộc tính khác của một đối tượng.

## 6.1 NOT bitwise

Dùng để thực hiện phép \*\*NOT bitwise\*\* trên từng bit của một số. Kết quả là bit đảo ngược của số đó.

```

int result = ~num;

~ 0 = 1

~ 1 = 0

```

## 6.2 AND bitwise

Dùng để thực hiện phép \*\*AND bitwise\*\* giữa từng cặp bit của hai số. Kết quả là 1 nếu cả hai bit tương ứng đều là 1, ngược lại là 0.

```

int result = num1 & num2;

0 & 0 = 0

0 & 1 = 0

1 & 0 = 0

1 & 1 = 1

```

## 6.3 OR bitwise

Dùng để thực hiện phép \*\*OR bitwise\*\* giữa từng cặp bit của hai số. Kết quả là 1 nếu có hơn một bit tương ứng là 1.

```

int result = num1 | num2;

0 | 0 = 0

0 | 1 = 1

1 | 0 = 1

1 | 1 = 1

```

## 6.4 XOR bitwise

Dùng để thực hiện phép \*\*XOR bitwise\*\* giữa từng cặp bit của hai số. Kết quả là 1 nếu chỉ có một bit tương ứng là 1.

```

int result = num1 ^ num2;

0 ^ 0 = 0

0 ^ 1 = 1

1 ^ 0 = 1

1 ^ 1 = 0

```

## 6.5 Shift left bitwise

Dùng để di chuyển bit sang trái, các bit ở bên phải sẽ được dịch sang trái, và các bit trái cùng sẽ được đặt giá trị 0.

```

int resultLeftShift = num << shiftAmount;

```

## 6.6 Shift right bitwise

Dùng để di chuyển bit sang phải, các bit ở bên trái sẽ được dịch sang phải, và các bit phải cùng sẽ được đặt giá trị 0 hoặc 1 tùy thuộc vào giá trị của bit cao nhất (bit dấu).

```

int resultRightShift = num >> shiftAmount;

```

Ví dụ:

```

#include <stdio.h>

#include <stdint.h>

#define ENABLE 1

#define DISABLE 0

typedef struct {

uint8\_t LED1 : 1; // LED1 sử dụng 1 bit

uint8\_t LED2 : 1; // LED2 sử dụng 1 bit

uint8\_t LED3 : 1; // LED3 sử dụng 1 bit

uint8\_t LED4 : 1; // LED4 sử dụng 1 bit

uint8\_t LED5 : 1; // LED5 sử dụng 1 bit

uint8\_t LED6 : 1; // LED6 sử dụng 1 bit

uint8\_t LED7 : 1; // LED7 sử dụng 1 bit

uint8\_t LED8 : 1; // LED8 sử dụng 1 bit

} LEDStatus;

void displayAllStatusLed(LEDStatus status) {

uint8\_t\* statusPtr = (uint8\_t\*)&status;

for (int j = 0; j < 8; j++) {

printf("LED%d: %d\n", j+1, (\*statusPtr >> j) & 1);

}

}

int main() {

LEDStatus ledStatus = {.LED7 = ENABLE};

// Bật LED 1 và 3

ledStatus.LED1 = ENABLE;

ledStatus.LED3 = ENABLE;

displayAllStatusLed(ledStatus);

return 0;

}

```

</details>

<details>

<summary><strong>BÀI 7: STRUCT - UNION</strong></summary>

# BÀI 7: STRUCT - UNION

## 7.1 Struct

\*\*Struct\*\* trong C là một kiểu dữ liệu tùy chỉnh, được sử dụng để lưu trữ các biến có kiểu dữ liệu khác nhau ở một đối tượng duy nhất.

- Một struct được định nghĩa bằng cách liệt kê tất cả các biến thành viên (member) bên trong nó, và mỗi biến thành viên có thể có một kiểu dữ liệu khác nhau.

- Khai báo:

```

struct{

kieuDuLieu1 thanhVien1;

kieuDuLieu2 thanhVien2;

// ...

} TenStruct;

```

Ví dụ: Định nghĩa kiểu dữ liệu tên là SinhVien gồm các thành phần như họ tên (kiểu char), ID (kiểu int), điểm trung bình (kiểu float), ….

```

struct{

char ten[20];

int ID;

float diemTB;

} SinhVien;

```

- Xác định kích thước của Struct:

Kích thước của Struct sẽ bằng tổng kích thước của các thành viên (members) của nó và kích thước của bộ nhớ đệm (padding) (nếu có)

Ví dụ:

```

#include<stdio.h>

#include<stdint.h>

struct{

uint8\_t var1; // 1 byte

uint32\_t var2; // 4 byte

uint16\_t var3; // 2 byte

} example;

int main(){

printf("Size: %d byte", sizeof(example));

return 0;

}

```

Kích thước của Struct phụ thuộc vào kích thước của kiểu dữ liệu lớn nhất của nó.

![image](https://github.com/user-attachments/assets/ff85a52d-378f-4179-abb6-0792fc68d174)

Kết quả:

```

Size: 12 byte

```

Khi sắp xếp lại thứ tự các biến trong Struct:

```

#include<stdio.h>

#include<stdint.h>

struct{

uint8\_t var1; // 1 byte

uint16\_t var3; // 2 byte

uint32\_t var2; // 4 byte

} example;

int main(){

printf("Size: %d byte", sizeof(example));

return 0;

}

```

![image](https://github.com/user-attachments/assets/9f9719cd-d7fa-4eb1-a3e0-76d0b6ce8a06)

Kết quả:

```

Size: 8 byte

```

Vì vậy, trong 1 struct ta có thể sắp xếp thứ tự các biến thành viên một cách hợp lí để tối ưu bộ nhớ.

Ví dụ:

```

#include<stdio.h>

#include<stdint.h>

struct{

uint8\_t arr1[5];

uint16\_t arr2[4];

uint32\_t arr3[2];

} example;

int main(){

printf("Size: %d byte", sizeof(example));

return 0;

}

```

Với struct gồm các thành viên là các mảng thì thực chất có thể được hiểu như sau:

```

struct{

// uint8\_t arr1[5];

uint8\_t arr1\_1;

uint8\_t arr1\_2;

uint8\_t arr1\_3;

uint8\_t arr1\_4;

uint8\_t arr1\_5;

// uint16\_t arr2[4];

uint16\_t arr2\_1;

uint16\_t arr2\_2;

uint16\_t arr2\_3;

uint16\_t arr2\_4;

// uint32\_t arr3[2];

uint32\_t arr3\_1;

uint32\_t arr3\_2;

} example;

```

![image](https://github.com/user-attachments/assets/7111ee2f-7965-41e6-a7be-1f69688b321e)

Kết quả:

```

Size: 24 byte

```

## 7.2 Union

\*\*Union\*\* cũng là kiểu dữ liệu mà người dùng tự định nghĩa tương tự như struct, Union cũng có các thành phần dữ liệu để mô tả thông tin của đối tượng muốn lưu trữ.

- Nếu với Struct thì các thành phần của nó có thể lưu trữ giá trị đồng thời tại cùng một thời điểm còn Union thì tại mỗi thời điểm chỉ có một thành viên của Union có thể được sử dụng.

- Tất cả các thành phần của Union sẽ chia sẻ chung 1 vùng nhớ.

- Khai báo:

```

union{

kieuDuLieu1 thanhVien1;

kieuDuLieu2 thanhVien2;

// ...

} TenUnion;

```

Ví dụ:

```

#include<stdio.h>

#include<stdint.h>

union{

uint8\_t var1;

uint16\_t var2;

uint32\_t var3;

} Data;

int main(){

printf("Size: %d byte", sizeof(example));

return 0;

}

```

Kích thước của Union chính là kích thước của thành viên có kích thước lớn nhất của nó.

Kết quả:

```

Size: 4 byte

```

Vì các thành viên của Union chia sẻ chung 1 vùng nhớ nên khi gọi 1 thành viên thì vùng nhớ sẽ như sau:

![image](https://github.com/user-attachments/assets/d2fffe3f-a2ef-423c-9cc5-d9633e9659ef)

Ví dụ:

```

#include<stdio.h>

#include<stdint.h>

union{

uint8\_t arr1[5]; // 5 byte

uint16\_t arr2[9]; // 18 byte

uint32\_t arr3[3]; // 12 byte

} Data;

int main(){

printf("Size: %d byte", sizeof(example));

return 0;

}

```

Tương tự Struct thì kích thước của Union cũng phụ thuộc vào kích thước của kiểu dữ liệu lớn nhất trong nó.

Trong ví dụ này khi quét nó sẽ quét 4 byte một lần. Và member `arr2[9]` có kích thước lớn nhất với 18 byte kích thước của nó và 2 byte padding.

Vì vậy, kích thước của Union sẽ là 20 byte.

Kết quả:

```

Size: 20 byte

```

## 7.3 Ứng dụng kết hợp của Struct và Union

```

#include <stdio.h>

#include <stdint.h>

#include <string.h>

typedef union {

struct {

uint8\_t id[2];

uint8\_t data[4];

uint8\_t check\_sum[2];

} data;

uint8\_t frame[8];

} Data\_Frame;

int main()

{

Data\_Frame transmitter\_data;

strcpy(transmitter\_data.data.id, "10");

strcpy(transmitter\_data.data.data, "1234");

strcpy(transmitter\_data.data.check\_sum, "70");

Data\_Frame receiver\_data;

strcpy(receiver\_data.frame, transmitter\_data.frame);

return 0;

}

```

</details>

<details>

<summary><strong>BÀI 8: MEMORY LAYOUT</strong></summary>

# BÀI 8: MEMORY LAYOUT

Chương trình main.exe (trên window), main.hex (nạp vào vi điều khiển) được lưu ở bộ nhớ SSD hoặc FLASH. Khi nhấn run chương trình trên Window (cấp nguồn cho vi điều khiển) thì những chương trình này sẽ được copy vào bộ nhớ RAM để thực thi.

![image](https://github.com/user-attachments/assets/e84b6278-6b83-4f30-9d88-9a7878eead10)

## 8.1 Text Segment

- Chứa tập hợp các lệnh thực thi.

- Quyền truy cập: Text Segment thường có quyền đọc và thực thi, nhưng không có quyền ghi.

- Lưu hằng số, con trỏ kiểu char.

- Tất cả các biến lưu ở phân vùng Text đều không thể thay đổi giá trị mà chỉ được đọc.

![image](https://github.com/user-attachments/assets/3e570e02-9bb4-4273-8523-8699b5e40531)

Ví dụ:

```

#include<stdio.h>

#include<string.h>

const int a = 5; // Phân vùng Text

char \*ptr = "Hello World!"; // Phân vùng Text

int main(){

a = 10; // Lỗi không thể thay đổi được hằng số

printf("a = %d\n", a);

\*ptr = 'K'; // Lỗi không thể thay đổi con trỏ kiểu char

printf("%s\n", ptr);

return 0;

}

```

## 8.2 Data Segment

\*\*Initialized Data Segment\*\* (Dữ liệu đã khởi tạo):

- Chứa các biến toàn cục được khởi tạo với giá trị khác 0.

- Chứa các biến static được khởi tạo với giá trị khác 0.

- Quyền truy cập là đọc và ghi, tức là có thể đọc và thay đổi giá trị của biến.

- Tất cả các biến sẽ được thu hồi \*\*sau khi chương trình kết thúc\*\*.

![image](https://github.com/user-attachments/assets/43b01e88-f724-4e13-bdf7-7e9875fde4e0)

Ví dụ:

```

#include<stdio.h>

/\*

Các biến phân vùng Data

int a = 5; // Biến toàn cục được khởi tạo khác 0

static int b = 10; // Biến static toàn cục được khởi tạo khác 0

void test(){

static int x = 3; // Biến static cục bộ được khởi tạo khác 0

}

\*/

int main(){

return 0;

}

// Sau khi kết thúc chương trình các biến phân vùng Data sẽ được thu hồi

```

## 8.3 BSS Segment

\*\*Uninitialized Data Segment\*\* (Dữ liệu Chưa Khởi Tạo):

- Chứa các biến toàn cục khởi tạo với giá trị bằng 0 hoặc không gán giá trị.

- Chứa các biến static với giá trị khởi tạo bằng 0 hoặc không gán giá trị.

- Quyền truy cập là đọc và ghi, tức là có thể đọc và thay đổi giá trị của biến.

- Tất cả các biến sẽ được thu hồi \*\*sau khi chương trình kết thúc\*\*.

![image](https://github.com/user-attachments/assets/5d1eb88e-fba4-4129-913f-52d8323d3bb6)

Ví dụ:

```

#include<stdio.h>

/\*

Các biến phân vùng BSS

int a = 0; // Biến toàn cục khởi tạo với giá trị bằng 0

static int b; // Biến static không khởi tạo giá trị ban đầu

\*/

int main(){

a = 10; // Khi gán giá trị a = 10 thì biến a vẫn nằm trong phân vùng BSS chứ không chuyển sang phân vùng Data.

return 0;

}

```

## 8.4 Stack

- Chứa các biến cục bộ, tham số truyền vào.

- Quyền truy cập: đọc và ghi, nghĩa là có thể đọc và thay đổi giá trị của biến trong suốt thời gian chương trình chạy.

- Các biến sẽ được thu hồi vùng nhớ \*\*sau khi ra khỏi hàm\*\*.

![image](https://github.com/user-attachments/assets/23e16469-2639-4ffc-86bd-386b75a47584)

Ví dụ:

```

#include<stdio.h>

int tong(int a, int b){

const int x = 10; // Biến cục bộ là hằng số nhưng vẫn được lưu ở phân vùng Stack và chỉ có quyền đọc.

int c; // Các biến cục bộ a, b, c được lưu ở phân vùng Stack

c = a + b;

return c;

}

int main(){

tong(3, 5);

// Sau khi kết thúc hàm thì các biến phân vùng Stack sẽ bị thu hồi vùng nhớ.

return 0;

}

```

## 8.5 Heap

Cấp phát động:

- \*\*Heap\*\* được sử dụng để cấp phát bộ nhớ động trong quá trình thực thi của chương trình.

- Điều này cho phép chương trình tạo ra và giải phóng bộ nhớ theo nhu cầu, thích ứng với sự biến đổi của dữ liệu trong quá trình chạy.

- Các hàm như `malloc()`, `calloc()`, `realloc()`, và `free()` được sử dụng để cấp phát và giải phóng bộ nhớ trên heap.

![image](https://github.com/user-attachments/assets/ddcb9a31-2ad6-4743-afbb-dfd686b066ce)

Ví dụ:

```

#include<stdio.h>

#include<stdint.h>

#include<stdlib.h>

int main(){

uint16\_t \*ptr = NULL;

ptr = (uint16\_t\*)malloc(sizeof(uint16\_t) \* 6); // Dùng malloc() cấp phát động 1 mảng có kích thước 2 \* 6 = 12 byte

for (int i = 0; i < 6; i++)

{

ptr[i] = i + 1;

}

for (int i = 0; i < 6; i++)

{

printf("%d\n", ptr[i]);

}

ptr = (uint16\_t\*)realloc(ptr, sizeof(uint16\_t) \* 10); // Dùng realloc() cấp phát động lại mảng với kích thước mới 2 \* 10 = 20 byte

ptr[10] = 17; // Ô nhớ chưa được cấp phát động nên biến a có thể truy cập đến và sử dụng

int a; // Các ô nhớ được cấp phát động thì biến a không thể truy cập đến

for (int i = 0; i < 10; i++)

{

ptr[i] = i + 1;

}

for (int i = 0; i < 11; i++)

{

printf("%d\n", ptr[i]);

}

free(ptr); // Thu hồi vùng nhớ để tránh tràn bộ nhớ

return 0;

}

```

## 8.6 Hàm malloc() và calloc()

### 8.6.1 Hàm malloc()

Hàm \*\*malloc()\*\* là viết tắt của từ \*\*memory allocation\*\* tức là cấp phát động vùng nhớ, hàm này được sử dụng để xin cấp phát khối bộ nhớ theo kích thước byte mong muốn.

Giá trị trả về của hàm là một con trỏ kiểu `void`, nên ép kiểu sang kiểu dữ liệu cần dùng.

Các giá trị trong các ô nhớ được cấp phát là giá trị rác.

Cú pháp: `ptr = (cast\_type\*)malloc(byte\_size)`

Trong đó :

- `ptr` là con trỏ lưu trữ ô nhớ đầu tiên của vùng nhớ được cấp phát

- `cast\_type\*` là kiểu con trỏ muốn ép kiểu sang

- `byte\_size` là kích thước theo byte muốn cấp phát

### 8.6.2 Hàm calloc()

Hàm \*\*calloc()\*\* viết tắt của \*\*contiguous allocation\*\*, tương tự như \*\*malloc()\*\* sử dụng để cấp phát vùng nhớ động nhưng các giá trị của các vùng nhớ được cấp phát sẽ có giá trị mặc định là 0.

Cú pháp : `ptr = (cast\_type\*) calloc(n, element\_size)`

Trong đó :

- `ptr` là con trỏ lưu trữ ô nhớ đầu tiên của vùng nhớ được cấp phát

- `cast\_type\*` là kiểu con trỏ muốn ép kiểu sang

- `n` là số lượng phần tử muốn cấp phát

- `element\_size` là kích thước theo byte của 1 phần tử

### 8.6.3 So sánh malloc() và calloc()

|\*\*malloc\*\*|\*\*calloc()\*\*|

|:---------|:-----------|

|Tạo một khối bộ nhớ có kích thước do người dùng chỉ định.|Có thể gán nhiều khối bộ nhớ cho một biến.|

|Chứa giá trị rác.|Khối bộ nhớ được cấp phát luôn được khởi tạo bằng 0.|

|Số lượng input là 1.|Số lượng input là 2.|

|Nhanh hơn calloc().|Chậm hơn malloc().|

|Không an toàn so với calloc().|An toàn để sử dụng so với malloc().|

|Hiệu quả về thời gian cao hơn calloc().|Hiệu quả về thời gian thấp hơn malloc().|

|Chỉ trả về địa chỉ bắt đầu và không biến nó thành 0.|Trả về địa chỉ bắt đầu và biến nó thành 0 trước khi cấp phát địa chỉ.|

|Không thực hiện khởi tạo bộ nhớ.|Thực hiện khởi tạo bộ nhớ.|

- Trường hợp nên sử dụng \*\*malloc()\*\*:

- Khi phải cấp phát bộ nhớ khi chạy.

- Khi phải phân bổ các đối tượng phải tồn tại ngoài việc thực thi khối bộ nhớ hiện tại.

- Nếu cần phân bổ bộ nhớ lớn hơn kích thước của ngăn xếp đó.

- Trường hợp nên sử dụng \*\*calloc()\*\*:

- Khi phải đặt bộ nhớ được phân bổ về 0.

- Khi cần khởi tạo các phần tử về 0 để trả về một con trỏ vào bộ nhớ.

- Để tránh tràn có thể xảy ra với malloc().

## 8.7 Stack và Heap

| | \*\*Stack\*\* | \*\*Heap\*\* |

|:-:|:------------|:-----------|

|Đặc điểm|- Dùng để lưu trữ các biến cục bộ trong hàm, tham số truyền vào... Truy cập vào bộ nhớ này rất nhanh và được thực thi khi chương trình được biên dịch.<br>- Được quản lý bởi hệ điều hành, dữ liệu được lưu trong Stack sẽ tự động giải phóng khi hàm thực hiện xong công việc của mình.|- Dùng để lưu trữ vùng nhớ cho những biến được cấp phát động bởi các hàm malloc - calloc - realloc (trong C).<br>- Được quản lý bởi lập trình viên, dữ liệu trong Heap sẽ không bị hủy khi hàm thực hiện xong, vì vậy phải tự tay giải phóng vùng nhớ bằng câu lệnh free (trong C), và delete hoặc delete [] (trong C++), nếu không sẽ xảy ra hiện tượng rò rỉ bộ nhớ.|

|Kích thước vùng nhớ|Cố định, tùy thuộc vào hệ điều hành|Không cố định, có thể tăng giảm để đáp ứng được nhu cầu lưu trữ dữ liệu của chương trình.|

|Lỗi xảy ra đối với vùng nhớ|Nếu chương trình bạn sử dụng quá nhiều bộ nhớ vượt quá khả năng lưu trữ của Stack chắc chắn sẽ xảy ra tình trạng tràn bộ nhớ Stack (Stack overflow), các trường hợp xảy ra như khởi tạo quá nhiều biến cục bộ, hàm đệ quy vô hạn,...|- Nếu liên tục cấp phát vùng nhớ mà không giải phóng thì sẽ bị lỗi tràn vùng nhớ Heap (Heap overflow).<br>- Nếu khởi tạo một vùng nhớ quá lớn mà vùng nhớ Heap không thể lưu trữ một lần được sẽ bị lỗi khởi tạo vùng nhớ Heap thất bại.|

Sử dụng \*\*Stack\*\* nếu biết chính xác lượng dữ liệu phân bổ trước khi biên dịch và dữ liệu không quá lớn. Ngược lại, nên sử dụng \*\*Heap\*\*.

</details>

<details>

<summary><strong>BÀI 9: JSON</strong></summary>

# BÀI 9: JSON

- \*\*JSON\*\* là viết tắt của "JavaScript Object Notation" (Ghi chú về Đối tượng JavaScript). Đây là một định dạng truyền tải dữ liệu phổ biến trong lập trình và giao tiếp giữa các máy chủ và trình duyệt web, cũng như giữa các hệ thống khác nhau.

- \*\*JSON\*\* được thiết kế để dễ đọc và dễ viết cho con người, cũng như dễ dàng để phân tích và tạo ra cho máy tính. Nó sử dụng một cú pháp nhẹ dựa trên cặp \*\*key\*\* - \*\*value\*\*, tương tự như các đối tượng và mảng trong JavaScript. Mỗi đối tượng JSON bao gồm một tập hợp các cặp "key" và "value", trong khi mỗi mảng JSON là một tập hợp các giá trị.

Ví dụ:

1 Object JSON:

```

{

"name" : "Nguyen Van A", // value là 1 string

"age" : 18, // value là 1 number

"address" : null, // value là null

"isStudent": true, // value là boolean

"score" : [9, 10, 7], // value là 1 array

"contact" : { // value là 1 object

"email" : "nva@gmail.com",

"phoneNumber" : "123456789"

}

}

```

Trong đó:

- `"name" : "Nguyen Van A"` là 1 member

- `"name"` là 1 key

- `"Nguyen Van A"` là 1 value

1 Object JSON mở đầu bằng "{" kết thúc bằng "}".

Trong đó:

- \*key\* phải là 1 \*\*string\*\*, ngăn cách với \*value\* bởi dấu `:`

- \*value\* có thể là:

- Null: Đại diện cho 1 giá trị không tồn tại hoặc không xác định

- Boolean: `true` hoặc `false`

- Number: Có thể là số nguyên (int) hoặc số thực (float, double),...

- String: Chuỗi ký tự

- Array: Mảng có thể bao gồm nhiều kiểu dữ liệu

- Object: Đối tượng thuộc 1 \*key\* có cấu trúc như ví dụ trên

- Mỗi member trong object ngăn cách nhau bằng dấu `,`

```

typedef enum{

JSON\_NULL,

JSON\_BOOLEAN,

JSON\_NUMBER,

JSON\_STRING,

JSON\_ARRAY,

JSON\_OBJECT

} JsonType;

```

</details>

<details>

<summary><strong>BÀI 10: LINKED LIST</strong></summary>

# BÀI 10: LINKED LIST

\*\*Linked list\*\* hay \*\*danh sách liên kết\*\* (DSLK) là một cấu trúc dữ liệu được sử dụng để tổ chức và lưu trữ dữ liệu.

- Một linked list bao gồm một chuỗi các "nút" (nodes), mỗi nút chứa một giá trị dữ liệu và một con trỏ (pointer) đến nút tiếp theo trong chuỗi.

- Cấu trúc 1 node:

```

typedef struct node{

int value; // Dữ liệu kiểu int

struct node\* next; // Con trỏ trỏ đến nút tiếp theo

} node;

```

![image](https://github.com/user-attachments/assets/fedc69d4-9da0-497e-bba9-795e00b2b824)

- Tạo 1 node mới:

```

node \*createNode(int value){

node \*ptr = (node\*)malloc(sizeof(node));

ptr->value = value;

ptr->next = NULL;

return ptr;

}

```

- Tính chất:

- Phần tử cuối cùng trong DSLK trỏ đến NULL (con trỏ NULL).

- Mỗi node trong DSLK đều được cấp phát động và số lượng phần tử trong DSLK phụ thuộc vào bộ nhớ Heap.

- Có thể thay đổi kích thước qua việc thêm, xóa phần tử.

- Truy cập phần tử trong DSLK cần truy cập tuần tự, \*\*KHÔNG\*\* thể truy cập qua chỉ số.

- Mỗi node trong DSLK thực chất là một con trỏ, nó là địa chỉ một ô nhớ mà ô nhớ đó được sử dụng để lưu trữ thông tin về một node.

</details>

<details>

<summary><strong>BÀI 11: STACK - QUEUE</strong></summary>

# BÀI 11: STACK - QUEUE

## 11.1 Stack

\*\*Stack\*\* (ngăn xếp) là một cấu trúc dữ liệu tuân theo nguyên tắc \*\*"Last In, First Out"\*\* (LIFO), nghĩa là phần tử cuối cùng được thêm vào Stack sẽ là phần tử đầu tiên được lấy ra.

![image](https://github.com/user-attachments/assets/0d3e8537-9c59-4ac1-9c76-bc41506de6e3)

Định nghĩa 1 Stack:

```

typedef struct {

int\* items; // Mảng chứa các giá trị trong ngăn xếp

int size; // Kích thước của mảng

int top; // Chỉ số của phần tử trên cùng

} Stack;

```

Các thao tác cơ bản trên Stack bao gồm:

- \*\*initialize\*\*: Khởi tạo ngăn xếp rỗng.

```

void initialize(Stack \*stack, int size) {

stack->items = (int\*) malloc(sizeof(int) \* size);

stack->size = size;

stack->top = -1;

}

```

- \*\*is\_empty\*\*: Kiểm tra xem ngăn xếp có rỗng không.

```

int is\_empty(Stack stack) {

return stack.top == -1;

}

```

- \*\*is\_full\*\*: Kiểm tra xem ngăn xếp có bị đầy (tràn) hay không.

```

int is\_full(Stack stack) {

return stack.top == stack.size - 1;

}

```

- \*\*push\*\*: Thêm một phần tử vào ngăn xếp.

```

void push(Stack \*stack, int value) {

if (!is\_full(\*stack)) {

stack->items[++stack->top] = value;

} else {

printf("Stack overflow\n");

}

}

```

- \*\*pop\*\*: Lấy ra phần tử ở đỉnh ngăn xếp.

```

int pop(Stack \*stack) {

if (!is\_empty(\*stack)) {

return stack->items[stack->top--];

} else {

printf("Stack underflow\n");

return -1;

}

}

```

- \*\*top\*\*: Trả về phần tử ở đỉnh ngăn xếp.

```

int top(Stack stack) {

if (!is\_empty(stack)) {

return stack.items[stack.top];

} else {

printf("Stack is empty\n");

return -1;

}

}

```

## 11.2 Queue

\*\*Queue\*\* (hàng đợi) là một cấu trúc dữ liệu tuân theo nguyên tắc \*\*"First In, First Out"\*\* (FIFO), nghĩa là phần tử đầu tiên được thêm vào hàng đợi sẽ là phần tử đầu tiên được lấy ra.

![image](https://github.com/user-attachments/assets/a3ee63b2-079c-45ad-81c3-6460c96bb085)

\*\*Queue\*\* trong lập trình nhúng: Hoạt động kiểu Circular Queue

![image](https://github.com/user-attachments/assets/a7628eb3-8f15-42a7-ac0d-94d34342252d)

Định nghĩa 1 Queue:

```

typedef struct {

int\* items; // Mảng chứa các giá trị trong ngăn xếp

int size; // Kích thước của mảng

int front, rear; // Chỉ số của phần tử đầu và cuối

} Queue;

```

Các thao tác cơ bản trên hàng đợi bao gồm:

- \*\*initialize\*\*: Khởi tạo một hàng đợi rỗng.

```

void initialize(Queue \*queue, int size)

{

queue->items = (int\*) malloc(sizeof(int)\* size);

queue->front = -1;

queue->rear = -1;

queue->size = size;

}

```

- \*\*is\_empty\*\*: Kiểm tra hàng đợi có rỗng hay không.

```

int is\_empty(Queue queue) {

return queue.front == -1;

}

```

- \*\*is\_full\*\*: Kiểm tra hàng đợi đã bị đầy chưa.

```

int is\_full(Queue queue) {

return (queue.rear + 1) % queue.size == queue.front;

}

```

- \*\*front\*\*: Trả về giá trị của phần tử ở đầu hàng đợi.

```

int front(Queue queue) {

if (!is\_empty(queue)) {

return queue.items[queue.front];

} else {

printf("Queue is empty\n");

return -1;

}

}

```

- \*\*enqueue\*\*: Đẩy một phần tử vào cuối hàng đợi.

```

void enqueue(Queue \*queue, int value) {

if (!is\_full(\*queue)) {

if (is\_empty(\*queue)) {

queue->front = queue->rear = 0;

} else {

queue->rear = (queue->rear + 1) % queue->size;

}

queue->items[queue->rear] = value;

} else {

printf("Queue overflow\n");

}

}

```

- \*\*dequeue\*\*: Loại bỏ một phần tử ở đầu hàng đợi và trả về phần tử đó.

```

int dequeue(Queue \*queue) {

if (!is\_empty(\*queue)) {

int dequeued\_value = queue->items[queue->front];

if (queue->front == queue->rear) {

queue->front = queue->rear = -1;

} else {

queue->front = (queue->front + 1) % queue->size;

}

return dequeued\_value;

} else {

printf("Queue underflow\n");

return -1;

}

}

```

</details>

<details>

<summary><strong>BÀI 12: BINARY SEARCH - FILE OPERATIONS</strong></summary>

# BÀI 12: BINARY SEARCH - FILE OPERATIONS

## 12.1 Binary search

- \*\*Binary search\*\* (Tìm kiếm nhị phân) là một thuật toán tìm kiếm xác định vị trí của một phần tử trong một mảng đã được sắp xếp.

- Thuật toán này hoạt động bằng cách chia không gian tìm kiếm thành hai phần bằng cách so sánh giá trị cần tìm với giá trị ở giữa mảng. Dựa vào kết quả so sánh, thuật toán tiếp tục tìm kiếm trong nửa phía bên trái hoặc bên phải của mảng.

```

#include <stdio.h>

#include <stdlib.h>

int binarySearch(int\* arr, int l, int r, int x) // x là phần tử cần tìm kiếm

{ // l (left), r (right) lần lượt là vị trí của phần tử đầu tiên và cuối cùng trong mảng

if (r >= l)

{

int mid = (r + l) / 2;

if (arr[mid] == x) return mid; // So sánh x với phần tử ở vị trí giữ mảng

if (arr[mid] > x) return binarySearch(arr, l, mid - 1, x); // Đệ quy với mảng mới là nửa bên phải của mảng ban đầu

return binarySearch(arr, mid + 1, r, x); // Đệ quy với mảng mới là nửa bên trái của mảng ban đầu

}

return -1;

}

void swap(int\* a, int\* b) // Hàm hoán đổi vị trí 2 số

{

int temp = \*a;

\*a = \*b;

\*b = temp;

}

void bubbleSort(int arr[], int n) // Hàm sắp xếp mảng theo thứ tự tăng dần

{

int i, j;

for (i = 0; i < n - 1; i++)

{

for (j = 0; j < n - i - 1; j++)

{

if (arr[j] > arr[j + 1])

swap(&arr[j], &arr[j + 1]);

}

}

}

int main()

{

int n, x, i;

printf("Nhap so phan tu cua mang: ");

scanf\_s("%d", &n);

int\* arr = (int\*)malloc(n \* sizeof(int));

printf("Nhap cac phan tu cua mang: ");

for (i = 0; i < n; i++)

{

scanf\_s("%d", &arr[i]);

}

bubbleSort(arr, n);

for (int i = 0; i < n; i++)

{

printf\_s("i = %d\n", arr[i]);

}

printf\_s("Nhap gia tri can tim: ");

scanf\_s("%d", &x);

int result = binarySearch(arr, 0, n - 1, x);

if (result == -1)

printf\_s("Khong tim thay %d trong mang.\n", x);

else

printf\_s("Tim thay %d tai vi tri %d trong mang.\n", x, result);

free(arr);

return 0;

}

```

## 12.2 File operations

Ngôn ngữ lập trình C cung cấp một số thư viện và hàm tiêu biểu để thực hiện các thao tác với file.

```

FILE \*file = fopen(const char \*file\_name, const char \*access\_mode);

```

Trong đó:

- `const char \*file\_name` là tên file

- `const char \*access\_mode` là chế độ sử dụng

### 12.2.1 Các chế độ sử dụng

|\*\*Chế độ\*\*|\*\*Mô tả\*\*|

|:--------:|:--------|

|"r"|Mở file với chế độ chỉ đọc file. Nếu mở file thành công thì trả về địa chỉ của phần tử đầu tiên trong file, nếu không thì trả về NULL.|

|"rb"|Mở file với chế độ chỉ đọc file theo định dạng binary. Nếu mở file thành công thì trả về địa chỉ của phần tử đầu tiên trong file, nếu không thì trả về NULL.|

|"w"|Mở file với chế độ ghi vào file. Nếu file đã tồn tại, thì sẽ ghi đè vào nội dung bên trong file. Nếu file chưa tồn tại thì sẽ tạo một file mới. Nếu không mở được file thì trả về NULL.|

|"wb"|Mở file với chế độ ghi vào file theo định dạng binary. Nếu file đã tồn tại thì sẽ ghi đè vào nội dung bên trong file. Nếu file chưa tồn tại thì sẽ tạo một file mới. Nếu không mở được file thì trả về NULL.|

|"a"|Mở file với chế độ nối. Nếu mở file thành công thì trả về địa chỉ của phần tử cuối cùng trong file. Nếu file chưa tồn tại thì sẽ tạo một file mới. Nếu không mở được file thì trả về NULL.|

|"ab"|Mở file với chế độ nối dưới định dạng binary. Nếu mở file thành công thì trả về địa chỉ của phần tử cuối cùng trong file. Nếu file chưa tồn tại sẽ tạo một file mới. Nếu không mở được file thì trả về NULL.|

|"r+"|Mở file với chế độ đọc và ghi file. Nếu mở file thành công thì trả về địa chỉ của phần tử đầu tiên trong file, nếu không thì trả về NULL.|

|"rb+"|Mở file với chế độ đọc và ghi file dưới định dạng binary. Nếu mở file thành công thì trả về địa chỉ của phần tử đầu tiên trong file, nếu không thì trả về NULL.|

|"w+"|Mở file với chế độ ghi và đọc file. Nếu file đã tồn tại thì trả về địa chỉ của phần tử đầu tiên của file. Nếu file chưa tồn tại thì sẽ tạo một file mới.|

|"wb+"|Mở file với chế độ ghi và đọc file dưới định dạng binary. Nếu file đã tồn tại thì trả về địa chỉ của phần tử đầu tiên của file. Nếu file chưa tồn tại thì sẽ tạo một file mới.|

|"a+"|Mở file với chế độ nối và đọc file. Nếu file đã tồn tại thì trả về địa chỉ của phần tử cuối cùng của file. Nếu file chưa tồn tại thì sẽ tạo một file mới.|

|"ab+"|Mở file với chế độ nối và đọc file dưới định dạng binary. Nếu file đã tồn tại thì trả về địa chỉ của phần tử cuối cùng của file. Nếu file chưa tồn tại thì sẽ tạo một file mới.|

### 12.2.2 Đọc file

|\*\*Tên hàm\*\*|\*\*Mô tả\*\*|

|:---------:|:--------|

|`fscanf()`|Sử dụng chuỗi được định dạng và danh sách đối số biến để lấy đầu vào từ một File|

|`fgets()`|Copy nội dung trong File vào mảng dùng để lưu trữ với tối đa số lượng phần tử của mảng hoặc tới khi gặp ký tự xuống dòng.|

|`fgetc()`|Lấy giá trị tại địa chỉ hiện tại của file, sau đó di chuyển tới địa chỉ tiếp theo. Kiểu trả về là char|

|`fread()`|Đọc một số lượng byte được chỉ định từ File .bin|

### 12.2.3 Ghi file

|\*\*Tên hàm\*\*|\*\*Mô tả\*\*|

|:---------:|:--------|

|`fprintf()`|Ghi chuỗi vào File, và có thể thêm danh sách các đối số|

|`fputs()`|Ghi chuỗi vào File|

|`fputc()`|Ghi một ký tự vào File|

|`fwrite()`|Ghi một số byte được chỉ định vào File .bin|

\*\*Một số hàm khác\*\*

- `fclose()`: Đóng File đã mở

- `feof()`: Để kiểm tra địa chỉ hiện tại có phải ký tự cuối cùng của File hay chưa

</details>

<details>

<summary><strong>BÀI 13: CLASS</strong></summary>

# BÀI 13: CLASS

Trong C++, từ khóa `class` được sử dụng để định nghĩa một lớp, là một cấu trúc dữ liệu tự định nghĩa có thể chứa dữ liệu và các hàm thành viên liên quan.

## 13.1 Phạm vi truy cập

Có 3 phạm vi truy cập trong class, bao gồm `private`, `protected`, `public`.

```

class ClassName {

private:

// Các thành phần riêng tư (private) chỉ có thể truy cập bên trong lớp

// Dữ liệu thành viên, hàm thành viên, ...

protected:

// Các thành phần bảo vệ (protected) tương tự như private, nhưng có thể truy cập từ lớp kế thừa

public:

// Các thành phần công khai (public) được truy cập từ bên ngoài lớp

// Dữ liệu thành viên, hàm thành viên, ...

// Hàm thành viên và các phương thức khác có thể được định nghĩa tại đây

// ...

};

```

|Phạm vi truy cập|Truy cập bên trong class?|Truy cập bên trong lớp con?|Truy cập bên ngoài class?|

|:--------------:|:-----------------------:|:-------------------------:|:-----------------------:|

|`private`|Có|Không|Không|

|`protected`|Có|Có|Không|

|`public`|Có|Có|Có|

Ví dụ:

```

#include <iostream>

#include <string>

using namespace std;

class SinhVien {

public:

int id;

string name;

int score;

void display(){

cout << "MSSV: " << id << endl;

cout << "Ten: " << name << endl;

cout << "Diem: " << score << endl;

}

};

int main(){

SinhVien sv;

sv.id = 2024;

sv.name = "Nguyen Van A";

sv.score = 10;

sv.display();

return 0;

}

```

Trong đó:

- `SinhVien` là 1 lớp (class)

- `sv` là 1 đối tượng (object)

- `id`, `name`, `score` là các thuộc tính (property)

- `display()` là phương thức (method)

Method có thể được khai báo như sau:

```

class SinhVien {

public:

int id;

string name;

int score;

void display();

};

void SinhVien::display(){

cout << "MSSV: " << id << endl;

cout << "Ten: " << name << endl;

cout << "Diem: " << score << endl;

}

```

## 13.2 Constructor (Hàm khởi tạo)

\*\*Constructor\*\* trong C++ là một method sẽ được tự động chạy đầu tiền khi khởi tạo object. Constructor không định nghĩa kiểu trả về và có tên trùng với tên của class.

```

#include <iostream>

#include <string>

using namespace std;

class HinhChuNhat {

public:

double chieuDai;

double chieuRong;

HinhChuNhat(int dai, int rong);

double tinhDienTich();

};

HinhChuNhat::HinhChuNhat(int dai, int rong)

{

chieuDai = dai;

chieuRong = rong;

}

double HinhChuNhat::tinhDienTich()

{

return chieuDai \* chieuRong;

}

int main(){

HinhChuNhat hcn(5, 3);

cout << "Chieu dai: " << hcn.chieuDai << endl;

cout << "Chieu rong: " << hcn.chieuRong << endl;

cout << "Dien tich: " << hcn.tinhDienTich() << endl;

return 0;

}

```

Nếu một lớp không được khai báo constructor thì chương trình sẽ tự động tạo ra một constructor mặc định (không có tham số đầu vào). Ở ví dụ trên, nếu không khai báo constructor `HinhChuNhat(int dai, int rong);` thì chương trình sẽ tự động thêm vào một constructor mặc định (constructor rỗng) với phạm vi truy cập `public` vào bên trong class.

```

class HinhChuNhat {

public:

HinhChuNhat() {}

//

};

```

Nếu dùng từ khóa `private` hoặc `protected` để khai báo constructor thì sẽ không thể khởi tạo được đối tượng của lớp này.

## 13.3 Destructor (Hàm hủy)

\*\*Destructor\*\* trong C++ là một method sẽ được tự động gọi khi object được giải phóng. Destructor sẽ có tên trùng với tên của class và thêm ký tự `~` ở phía trước tên.

```

#include <iostream>

#include <string>

using namespace std;

class HinhChuNhat {

public:

double chieuDai;

double chieuRong;

HinhChuNhat(int dai, int rong);

~HinhChuNhat();

double tinhDienTich();

};

HinhChuNhat::HinhChuNhat(int dai, int rong)

{

chieuDai = dai;

chieuRong = rong;

}

HinhChuNhat::~HinhChuNhat(){

cout << "Ket thuc" << endl;

}

double HinhChuNhat::tinhDienTich()

{

return chieuDai \* chieuRong;

}

int main(){

HinhChuNhat hcn(5, 3);

cout << "Chieu dai: " << hcn.chieuDai << endl;

cout << "Chieu rong: " << hcn.chieuRong << endl;

cout << "Dien tich: " << hcn.tinhDienTich() << endl;

return 0;

}

```

## 13.4 Static trong class

Khi một thuộc tính được khai báo với từ khóa `static`, tất cả các object sẽ sử dụng chung thuộc tính này với một địa chỉ duy nhất.

Ví dụ:

```

#include <iostream>

#include <string>

using namespace std;

class HinhChuNhat {

public:

double chieuDai;

double chieuRong;

static int var; // Thuộc tính sử dụng cho tất cả các object

};

int HinhChuNhat::var; // Thuộc tính với từ khóa static phải được khởi tạo để lấy đại chỉ trước

int main()

{

HinhChuNhat hinh1;

HinhChuNhat hinh2;

HinhChuNhat hinh3;

cout << "address of chieu dai: " << &hinh1.chieuDai << '\n';

cout << "address of chieu dai: " << &hinh2.chieuDai << '\n';

cout << "address of chieu dai: " << &hinh3.chieuDai << '\n';

cout << "address of var: " << &hinh1.var << '\n';

cout << "address of var: " << &hinh2.var << '\n';

cout << "address of var: " << &hinh3.var << '\n';

return 0;

}

```

Kết quả:

![image](https://github.com/user-attachments/assets/2444357a-5576-4758-9a89-c4f15fbb5788)

Khi một method trong class được khai báo với từ khóa static:

- Method này độc lập với bất kỳ đối tượng nào của lớp.

- Method này có thể được gọi ngay cả khi không có đối tượng nào của class tồn tại.

- Method này có thể được truy cập bằng cách sử dụng tên class thông qua toán tử :: .

- Method này có thể truy cập các static property và các static method bên trong hoặc bên ngoài class.

- Method có phạm vi bên trong class và không thể truy cập con trỏ đối tượng hiện tại.

</details>

<details>

<summary><strong>BÀI 14: OOP</strong></summary>

# BÀI 14: OOP

## 14.1 Tính đóng gói (Encapsulation)

\*\*Tính đóng gói (Encapsulation)\*\* là ẩn đi các property “ mật” khỏi người dùng. Và để làm được điều này, ta sẽ khai báo các property ở quyền truy cập private (tức là không thể truy cập trực tiếp tới các property này).

Trong trường hợp ta muốn đọc hoặc ghi các property này, thì ta cung cấp các method ở quyền truy cập public. Mục đích chính của tính đóng gói là giúp hạn chế các lỗi khi phát triển chương trình.

Ví dụ:

```

class Student {

public:

string name;

int age;

double gpa;

}

```

Các thuộc tính của lớp `Student` đều là `public` nên các thuộc tính của lớp này có thể được truy xuất trực tiếp từ nơi khác.

Để đảm bảo tính đóng gói cho lớp `Student` thì nên để phạm vi truy cập của các thuộc tính này là `private` và truy xuất tới các thuộc tính này thông qua các phương thức `public`.

```

class Student {

private:

string name;

int age;

double gpa;

public:

string getName();

void setName(string name);

int getAge();

void setAge(int age);

double getGpa();

void setGpa(double gpa);

};

string Student::getName() {

return name;

}

void Student::setName(string name) {

this->name = name;

}

int Student::getAge() {

return age;

}

void Student::setAge(int age) {

this->age = age;

}

double Student::getGpa() {

return gpa;

}

void Student::setGpa(double gpa) {

this->gpa = gpa;

}

```

Tính đóng gói giúp:

- Hạn chế được các truy xuất không hợp lệ tới các thuộc tính của đối tượng.

- Giúp cho trạng thái của các đối tượng luôn đúng.

- Giúp ẩn đi những thông tin không cần thiết về đối tượng.

- Cho phép thay đổi cấu trúc bên trong lớp mà không ảnh hưởng tới lớp khác.

## 14.2 Tính kế thừa (Inheritance)

\*\*Tính kế thừa (Inheritance)\*\* là khả năng sử dụng lại các property và method của một class trong một class khác. Ta chia chúng làm 2 loại là class cha và class con. Để kế thừa từ class khác, ta dùng ký tự `:`.

Tất cả những property và method có quyền truy cập là public và protected ở class cha sẽ được class con kế thừa.

Ví dụ:

```

#include<iostream>

using namespace std;

class Person {

private:

string name;

int age;

public:

string getName();

void setName(string name);

int getAge();

void setAge(int age);

};

string Person::getName() {

return name;

}

void Person::setName(string name) {

this->name = name;

}

int Person::getAge() {

return age;

}

void Person::setAge(int age) {

this->age = age;

}

class Student : public Person {

private:

double gpa;

public:

double getGpa();

void setGpa(double gpa);

};

double Student::getGpa() {

return gpa;

}

void Student::setGpa(double gpa) {

this->gpa = gpa;

}

```

Tính kế thừa giúp:

- Tái sử dụng lại code.

- Tăng khả năng mở rộng của chương trình.

Có 3 kiểu kế thừa là public, private và protected. Những property và method được kế thừa từ class cha sẽ nằm ở quyền truy cập của class con tương ứng với kiểu kế thừa.

![image](https://github.com/user-attachments/assets/ded04c2e-eabe-4fe7-843f-c0247c989276)

\*\*Ghi đè phương thức\*\*

Trong kế thừa, khi lớp con khai báo phương thức có tên trùng với phương thức ở lớp cha thì phương thức của lớp cha sẽ bị thay thế bởi phương thức ở lớp con.

Ví dụ:

```

#include<iostream>

using namespace std;

class SuperClass {

public:

void display();

};

void SuperClass::display() {

cout << "Hello from SuperClass" << endl;

}

class SubClass : public SuperClass {

public:

void display();

};

void SubClass::display() {

cout << "Hello from SubClass" << endl;

}

int main() {

SubClass s;

s.display();

return 0;

}

```

Kết quả:

```

Hello from SubClass

```

Trong trường hợp phương thức của lớp cha bị ghi đè vẫn có thể gọi tới nó bằng toán tử `::`.

```

#include<iostream>

using namespace std;

class SuperClass {

public:

void display();

};

void SuperClass::display() {

cout << "Hello from SuperClass" << endl;

}

class SubClass : public SuperClass {

public:

void display();

};

void SubClass::display() {

SuperClass::display();

cout << "Hello from SubClass" << endl;

}

int main() {

SubClass s;

s.display();

return 0;

}

```

Kết quả:

```

Hello from SuperClass

Hello from SubClass

```

\*\*Virtual function\*\*

Ví dụ:

```

#include <iostream>

using namespace std;

class Animal {

public:

void sound() {

cout << "some sound" << endl;

}

};

class Cat : public Animal {

public:

void sound() {

cout << "meow meow" << endl;

}

};

class Dog : public Animal {

public:

void sound() {

cout << "woof woof" << endl;

}

};

int main() {

Cat cat;

Animal\* animal1 = &cat;

animal1->sound();

Dog dog;

Animal\* animal2 = &dog;

animal2->sound();

return 0;

}

```

Trong ví dụ trên, con trỏ `animal1` và `animal2` trỏ tới đối tượng `cat` và đối tượng `dog` nhưng phương thức được gọi lại không phải là phương thức trong 2 đối tượng này mà là phương thức ở lớp `Animal`. Điều này là do khi biên dịch thì trình biên dịch không thể xác định được đối tượng mà con trỏ `animal` đang trỏ tới là đối tượng thuộc lớp nào, dẫn tới mặc định các phương thức được gọi sẽ là phương thức ở class cha.

Kết quả:

```

some sound

some sound

```

Để gọi tới được các phương thức ở class con thì bạn cần thêm từ khóa `virtual` vào phương thức ở class cha để chỉ rõ cho trình biên dịch biết rằng phương thức cần được gọi sẽ được xác định tại thời điểm runtime. Tức là khi có 1 hàm trong class cha mà ở các class con muốn tái sử dụng hàm đó thì phải thêm từ khóa `virtual` vào hàm ở class cha.

Sau khi thêm từ khóa `virtual`:

```

class Animal {

public:

virtual void sound() {

cout << "some sound" << endl;

}

};

```

Kết quả:

```

meow meow

woof woof

```

## 14.3 Tính đa hình (Polymorphism)

\*\*Tính đa hình (Polymorphism)\*\* có nghĩa là "nhiều dạng" và nó xảy ra khi chúng ta có nhiều class có liên quan với nhau thông qua tính kế thừa.

Tính đa hình là cách dùng những method được kế thừa để thực hiện các tác vụ khác nhau. Điều này giúp chúng ta thể hiện 1 hành động theo nhiều cách khác nhau. (Function overriding)

Function overloading cung cấp nhiều định nghĩa cho 1 function bằng cách thay đổi số lượng input parameter, kiểu dữ liệu của input parameter.

Ví dụ:

```

#include <iostream>

#include <string>

using namespace std;

int sum(int a, int b) {

return a+b;

}

int sum(int a, int b, int c) {

return a+b+c;

}

double sum(double a, double b) {

return a+b;

}

int main()

{

cout << "Sum: " << sum(1,2) << endl;

cout << "Sum: " << sum(2.5,3.7) << endl;

cout << "Sum: " << sum(1,2,3) << endl;

return 0;

}

```

Ta thấy 1 method `sum` có thể được định nghĩa với số lượng và kiểu dữ liệu của input parameter khác nhau.

Kết quả:

```

Sum: 3

Sum: 6.2

Sum: 6

```

## 14.4 Tính trừu tượng (Abstraction)

\*\*Tính trừu tượng (Abstraction)\*\* đề cập đến việc ẩn đi các chi tiết cụ thể của một đối tượng và chỉ hiển thị những gì cần thiết để sử dụng đối tượng đó.

Ví dụ:

```

#include <iostream>

#include <string>

#include <cmath>

using namespace std;

class GiaiPhuongTrinh {

private:

double a;

double b;

double c;

double x1;

double x2;

double delta;

void tinhNghiem()

{

delta = b\*b - 4\*a\*c;

if (delta < 0)

{

delta = -1;

}

else if (delta == 0)

{

x1 = x2 = -b/ (2\*a);

}

else if (delta > 0)

{

x1 = (-b + sqrt(delta))/(2\*a);

x2 = (-b - sqrt(delta))/(2\*a);

}

}

public:

void enterNumber(double num\_a, double num\_b, double num\_c);

void printResult();

};

void GiaiPhuongTrinh::enterNumber(double num\_a, double num\_b, double num\_c)

{

a = num\_a;

b = num\_b;

c = num\_c;

}

void GiaiPhuongTrinh::printResult()

{

tinhNghiem();

if (delta == -1)

{

cout << "PT vo nghiem" << endl;

}

else if (delta == 0)

{

cout << "PT co nghiem chung: " << x1 << endl;

}

else if (delta > 0)

{

cout << "PT co 2 nghiem: \n";

cout << "x1: " << x1 << endl;

cout << "x2: " << x2 << endl;

}

}

int main()

{

GiaiPhuongTrinh phuongtrinh1;

phuongtrinh1.enterNumber(1,5,4);

phuongtrinh1.printResult();

return 0;

}

```

Trong ví dụ giải phương trình bậc hai trên, ta chỉ cần sử dụng các hàm như nhập hệ số và in ra kết quả, còn quá trình tính toán bên trong là không cần thiết nên ta có thể ẩn nó đi bằng cách cho vào trong `private`.

</details>

<details>

<summary><strong>BÀI 15: STANDARD TEMPLATE LIBRARY</strong></summary>

# BÀI 15: STANDARD TEMPLATE LIBRARY

\*\*Standard Template Library (STL)\*\* là một thư viện trong ngôn ngữ lập trình C++ cung cấp một tập hợp các template classes và functions để thực hiện nhiều loại cấu trúc dữ liệu và các thuật toán phổ biến. STL đã trở thành một phần quan trọng của ngôn ngữ C++ và làm cho việc lập trình trở nên mạnh mẽ, linh hoạt và hiệu quả.

Một số thành phần chính của STL:

- Container

- Iterator

- Algorithms

- Functor

## 15.1 Container

Một container là một cấu trúc dữ liệu chứa nhiều phần tử theo một cách cụ thể. STL (Standard Template Library) cung cấp một số container tiêu biểu giúp lưu trữ và quản lý dữ liệu.

### 15.1.1 Vector

\*\*Vector\*\* là một trong những container quan trọng nhất trong STL của C++. Nó cung cấp một mảng động với khả năng thay đổi kích thước một cách linh hoạt.

\*\*Một số đặc điểm chính của vector\*\*:

- Vector là một mảng động, tức là có khả năng thay đổi kích thước một cách linh hoạt.

- Truy cập ngẫu nhiên: Việc truy cập các phần tử của vector có thể được thực hiện bằng cách sử dụng chỉ số.

- Hiệu suất chèn và xóa: Chèn và xóa phần tử ở cuối vector có hiệu suất tốt. Tuy nhiên, chèn và xóa ở vị trí bất kỳ có thể đòi hỏi di chuyển một số phần tử.

\*\*Một số method của vector\*\*:

- `at()`: Truy cập vào phần tử của vector

- `size()`: Trả về kích thước của vector

- `resize()`: Thay đổi kích thước của vector

- `begin()`: Địa chỉ của phần tử đầu tiên của vector

- `end()`: Địa chỉ của phần tử cuối cùng của vector

- `push\_back()`: Thêm phần tử vào vị trí cuối của vector

Ví dụ:

```

#include <iostream>

#include <vector>

using namespace std;

int main(){

vector<int> array = {1, 2, 3, 4};

array.at(0) = 5; // 5 2 3 4

array.resize(6); // 5 2 3 4 0 0

array.push\_back(7); // 5 2 3 4 0 0 7

for (int i = 0; i < array.size(); i++)

{

cout << array.at(i) << " ";

}

return 0;

}

```

### 15.1.2 List

\*\*List\*\* là một container trong STL của C++, triển khai dưới dạng danh sách liên kết hai chiều.

\*\*Một số đặc điểm quan trọng của list\*\*:

- Truy cập tuần tự: Truy cập các phần tử của list chỉ có thể thực hiện tuần tự, không hỗ trợ truy cập ngẫu nhiên.

- Hiệu suất chèn và xóa: Chèn và xóa ở bất kỳ vị trí nào trong danh sách có hiệu suất tốt hơn so với vector. Điều này đặc biệt đúng khi thêm/xóa ở giữa danh sách.

\*\*Một số method của list\*\*:

- `push\_back()`: Thêm node vào vị trí cuối của list

- `pop\_back()`: Xóa node ở vị trí cuối của list

- `insert()`: Chèn một node vào list

- `erase()`: Xóa một node của list

- `size()`: Trả về kích thước của list

- `begin()`: Địa chỉ của node đầu tiên của list

- `end()`: Địa chỉ của node cuối cùng của list

Ví dụ:

```

#include <iostream>

#include <list>

using namespace std;

int main(){

list<int> array = {1, 3, 5, 7, 9};

array.pop\_back(); // 1 3 5 7

array.push\_back(11); // 1 3 5 7 11

array.push\_back(13); // 1 3 5 7 11 13

list<int>::iterator it; // Con trỏ it để truy cập phần tử

int i = 0;

// Chèn node vào vị trí thứ 2

for (it = array.begin(); it != array.end(); it++)

{

if (i == 2)

{

array.insert(it, 15); // 1 3 15 5 7 11 13

break;

}

i++;

}

// Xóa node ở vị trí thứ 5

i = 0;

for (it = array.begin(); it != array.end(); it++)

{

if (i == 5)

{

array.erase(it); // 1 3 15 5 7 13

break;

}

i++;

}

for (auto var : array)

{

cout << var << " ";

}

return 0;

}

```

\*\*Khi nào sử dụng vector? Khi nào sử dụng list?\*\*

| |Trường hợp sử dụng|

|:----:|:-----------------|

|Vector|- Cần truy cập ngẫu nhiên đến các phần tử. <br> - Thực hiện nhiều thao tác chèn/xóa ở cuối danh sách. <br> - Dung lượng có thể biết trước hoặc thay đổi ít.|

|List|- Thực hiện nhiều thao tác chèn/xóa ở bất kỳ vị trí nào trong danh sách. <br> - Cần thực hiện nhiều thao tác chèn/xóa mà không làm ảnh hưởng đến các iterators hiện có. <br> - Dung lượng không quan trọng hoặc thay đổi thường xuyên.|

### 15.1.3 Map

\*\*Map\*\* là một container trong STL của C++, cung cấp một cấu trúc dữ liệu ánh xạ key-value.

\*\*Một sô đặc điểm chính của map\*\*:

- Map lưu trữ các phần tử dưới dạng cặp key-value, trong đó mỗi key phải là duy nhất trong map.

- Ta có thể thêm phần tử mới vào map bằng cách sử dụng toán tử `[]` hoặc `hàm insert()`. Để xóa phần tử, bạn có thể sử dụng hàm `erase()`.

- Ta có thể sử dụng `iterator` để duyệt qua các phần tử của map.

Ví dụ:

```

#include <iostream>

#include <map>

#include <string>

using namespace std;

int main(){

map<string, int> mp;

map<string, int>::iterator it;

// Thêm phần tử vào map

mp["one"] = 1;

mp["two"] = 2;

mp["three"] = 3;

mp.insert(make\_pair("four", 4));

// Xóa phần tử trong map

mp.erase("three");

// In ra các phần tử trong map

for (auto var : mp)

{

cout << "Key: " << var.first << ", Value: " << var.second << endl;

}

/\*

Cách in khác

for (it = mp.begin(); it != mp.end(); it++)

{

cout << "Key: " << (\*it).first << ", Value: " << (\*it).second << endl;

}

\*/

return 0;

}

```

Kết quả:

```

Key: four, Value: 4

Key: one, Value: 1

Key: two, Value: 2

```

### 15.1.4 Array

\*\*Array\*\* là một container có kích thước cố định và có sẵn trong thư viện STL (Standard Template Library).

\*\*Một số đặc điểm của array\*\*:

- Array có kích thước cố định được xác định tại thời điểm biên dịch và không thể thay đổi sau khi được khai báo.

- Array hỗ trợ truy cập ngẫu nhiên vào các phần tử thông qua toán tử `[]`.

Ví dụ:

```

#include <iostream>

#include <array>

using namespace std;

int main()

{

array <int, 5> arr = {2,3,1,7,6};

// Kiểm tra xem hàm có rỗng hay không

if (arr.empty())

{

cout << "Array is empty" << endl;

}

else

{

cout << "Array it not empty" << endl;

}

// In ra các phần tử trong array

for (int i = 0; i < arr.size(); i++)

{

cout << "Value: " << arr.at(i) << endl;

}

return 0;

}

```

Kết quả:

```

Array it not empty

Value: 2

Value: 3

Value: 1

Value: 7

Value: 6

```

## 15.2 Iterator

\*\*Iterator\*\* cung cấp một cách chung để duyệt qua các phần tử của một container mà không cần biết chi tiết về cách container được triển khai.

Iterator là một đối tượng cho phép truy cập tuần tự qua các phần tử của một container. Nó giống như con trỏ, cho phép di chuyển qua các phần tử trong container.

Ví dụ:

```

#include <iostream>

#include <map>

#include <string>

using namespace std;

int main(){

map<string, int> mp;

map<string, int>::iterator it; // Con trỏ it để truy cập phần tử

// Thêm, xóa phần tử

// In ra các phần tử thông qua iterator

for (it = mp.begin(); it != mp.end(); it++)

{

cout << "Key: " << (\*it).first << ", Value: " << (\*it).second << endl;

}

return 0;

}

```

## 15.3 Algorithm

Thư viện STL (Standard Template Library) cung cấp một số thuật toán tiêu biểu thông qua algorithm. Các thuật toán này hoạt động trên các phạm vi hoặc các loại dữ liệu khác nhau, giúp thực hiện các nhiệm vụ như sắp xếp, tìm kiếm, chuyển đổi dữ liệu, và nhiều thao tác khác.

Ví dụ:

```

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

int main()

{

vector <int> arr = {3,5,7,4,1};

for (auto const var : arr)

{

cout << "Vector: " << var << endl;

}

sort(arr.begin(), arr.end()); // Hàm sắp xếp tăng dần

cout << "vector after sort ascending: " << endl;

for (auto const var : arr)

{

cout << "Vector: " << var << endl;

}

sort(arr.begin(), arr.end(), greater<int>()); // Hàm sắp xếp giảm dần

cout << "vector after sort descending: " << endl;

for (auto const var : arr)

{

cout << "Vector: " << var << endl;

}

return 0;

}

```

Kết quả:

```

Vector: 3

Vector: 5

Vector: 7

Vector: 4

Vector: 1

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

vector after sort ascending:

Vector: 1

Vector: 3

Vector: 4

Vector: 5

Vector: 7

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

vector after sort descending:

Vector: 7

Vector: 5

Vector: 4

Vector: 3

Vector: 1

```

Ví dụ:

```

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

int main()

{

vector <int> arr1 = {2,5,7,4,9};

auto it = find(arr1.begin(), arr1.end(), 3); // Hàm tìm kiếm phần tử

if (it != arr1.end())

{

cout << "Found" << endl;

}

else

{

cout << "Not found" << endl;

}

return 0;

}

```

Kết quả:

```

Not found

```

</details>

<details>

<summary><strong>BÀI 16: GENERIC PROGRAMMING</strong></summary>

# BÀI 16: GENERIC PROGRAMMING

## 16.1 Function templates

\*\*Function templates\*\* trong C++ là một tính năng mạnh mẽ giúp viết các function hoặc class chung có thể được sử dụng cho nhiều kiểu dữ liệu khác nhau mà không cần phải triển khai nhiều phiên bản của cùng một function hoặc class.

Ví dụ:

```

#include <iostream>

using namespace std;

template <typename T>

T sum(T a, T b) {

return a + b;

}

int main() {

int tong1 = sum(3, 7); // T là int

double tong2 = sum(1.7, 0.5); // T là double

}

```

Ví dụ:

```

#include <iostream>

using namespace std;

template <typename T1, typename T2>

T2 sum(T1 a, T2 b) {

return (T2)a + b;

}

int main() {

double tong = sum(3, 7.5); // T1 là int, T2 là double

}

```

## 16.2 Class templates

\*\*Class templates\*\* trong C++ là một khái niệm tương tự như function templates, nhưng được áp dụng cho class thay vì function. Class templates cho phép bạn viết một lớp chung mà có thể được sử dụng với nhiều kiểu dữ liệu khác nhau.

Ví dụ:

```

#include <iostream>

using namespace std;

template<typename T>

class Sensor {

private:

T data;

public:

Sensor(T initData) {

this->data = initData;

}

void updateData(T newData) {

data = newData;

}

T getData() const {

return data;

}

};

int main(){

Sensor<double> temperaturSensor(36.5);

temperaturSensor.updateData(35);

cout << "Temperature: " << temperaturSensor.getData() << endl;

Sensor<int> speedSensor(16);

cout << "Speed: " << speedSensor.getData() << endl;

return 0;

}

```

</details>

<details>

<summary><strong>BÀI 17: NAMESPACE</strong></summary>

# BÀI 17: NAMESPACE

\*\*Namespace\*\* là từ khóa trong C++ được sử dụng để định nghĩa một phạm vi nhằm mục đích phân biệt các hàm, lớp, biến, ... cùng tên trong các thư viện khác nhau.

Ví dụ:

```

#include <iostream>

namespace NamespaceA {

int var = 5;

void display() {

std::cout << "Namespace A" << endl;

}

// ...

}

namespace NamespaceB {

int var = 10;

void display() {

std::cout << "Namespace B" << endl;

}

// ...

}

int main() {

std::cout << display(); // Lỗi vì trình biên dịch không biết hàm display() muốn gọi là hàm nào

std::cout << "var = " << NamespaceA::var << endl;

NamespaceA::display();

std::cout << "var = " << NamespaceB::var << endl;

NamespaceB::display();

return 0;

}

```

Kết quả:

```

var = 5;

Namespace A

var = 10;

Namespace B

```

- Ta cũng có thể mở rộng thư viện trong C++ bằng cách thêm các hàm, lớp, biến, ... vào trong namespace của thư viện đó và có thể mở rộng ở bất kỳ file nào. Ví dụ muốn mở rộng thư viện `std`:

```

#include <iostream>

namespace std {

void display() {

std::cout << "Hello!" << endl;

}

// ...

}

int main(){

std::display();

return 0;

}

```

- Nếu muốn khai báo hàm, biến thông qua namespace 1 cách ngắn gọn, ta có thể sử dụng `using namespace`:

```

using namespace std;

int main() {

cout << "Hello world!" << endl;

}

```

- Tuy nhiên, khi gọi 2 hàm cùng tên và cùng sử dụng từ khóa `using namespace` thì chương trình sẽ báo lỗi.

```

#include <iostream>

using namespace NamespaceA;

using namespace NamespaceB;

namespace NamespaceA {

int var = 5;

void display() {

std::cout << "Namespace A" << endl;

}

// ...

}

namespace NamespaceB {

int var = 10;

void display() {

std::cout << "Namespace B" << endl;

}

// ...

}

int main() {

display(); // Chương trình báo lỗi

return 0;

}

```

- Namespace lồng nhau:

Khi không dùng `usingnamespace`:

```

#include <iostream>

namespace NamespaceA {

int var = 5;

namespace NamespaceB {

int var1 = 10;

}

}

int main() {

std::cout << "var1 = " << NamespaceA::NamespaceB::var1 << endl;

return 0;

}

```

Khi dùng `using namespace`:

```

#include <iostream>

using namespace NamespaceA;

namespace NamespaceA {

int var = 5;

namespace NamespaceB {

int var1 = 10;

}

}

int main() {

std::cout << "var1 = " << NamespaceB::var1 << endl;

return 0;

}

```

Hoặc

```

#include <iostream>

using namespace NamespaceA::NamespaceB;

namespace NamespaceA {

int var = 5;

namespace NamespaceB {

int var1 = 10;

}

}

int main() {

std::cout << "var1 = " << var1 << endl;

return 0;

}

```

</details>

<details>

<summary><strong>BÀI 18: LAMBDA FUNCTION</strong></summary>

# BÀI 18: LAMBDA FUNCTION

\*\*Lambda function\*\* là một tính năng mạnh mẽ được thêm vào ngôn ngữ lập trình C++ từ phiên bản C++11. Lambda cho phép định nghĩa hàm ngắn gọn (anonymous function) mà không cần phải viết một hàm riêng biệt. Cú pháp của lambda rất linh hoạt và có thể được sử dụng để viết mã ngắn gọn và dễ đọc.

Khai báo:

```

[capture](parameters) -> return\_type {

// function body

}

```

Trong đó:

- \*\*Capture\*\*: Cho phép bắt giữ biến từ môi trường xung quanh vào lambda.

- `[]`: Không bắt giữ bất kỳ biến nào từ môi trường xung quanh.

- `[var]`: Bắt giữ biến var theo giá trị.

- `[&var]`: Bắt giữ biến var theo tham chiếu.

- `[=]`: Bắt giữ tất cả biến theo giá trị.

- `[&]`: Bắt giữ tất cả biến theo tham chiếu.

- \*\*Parameters (parameters)\*\*: Tương tự như định nghĩa hàm, có thể bao gồm các tham số của lambda.

- \*\*Return type (return\_type)\*\*: Kiểu dữ liệu trả về của lambda. Có thể bị bỏ qua nếu không cần.

- \*\*Function body\*\*: Đặt trong dấu ngoặc nhọn {} và chứa mã nguồn thực thi của lambda.

Ví dụ:

```

#include <iostream>

using namespace std;

int main(){

int x = 10; // Biến môi trường

int y = 20; // Biến môi trường

auto display = [x, y]() { // Bắt giữ biến môi trường x, y, không có tham số đầu vào

cout << "Hello world" << endl;

cout << "x = " << x << ", y = " << y << endl;

};

display();

auto tong = [](int a, int b) -> int { // Không bắt giữ biến môi trường, tham số đầu vào int a, int b, trả về kiểu int

return a + b;

};

cout << "Tong: " << tong(10, 5) << endl;

return 0;

}

```

Kết quả:

```

Hello world

x = 10, y = 20

Tong: 15

```

\*\*Lambda function\*\* là 1 hàm cục bộ, bản chất là truyền vào con trỏ hàm và chỉ sử dụng 1 lần duy nhất.

</details>

<details>

<summary><strong>BÀI 19: SMART POINTER</strong></summary>

# BÀI 19: SMART POINTER

## 19.1 Khái niệm

Trong C++, \*\*smart pointer\*\* là một cơ chế quản lý bộ nhớ tự động giúp giảm thiểu rủi ro của lỗi liên quan đến quản lý bộ nhớ và tránh được việc quên giải phóng bộ nhớ đã được cấp phát.

Ta có thể tự tạo ra 1 smart pointer dùng cho mọi đối tượng bằng cách dùng class template:

```

#include <iostream>

using namespace std;

template <typename T>

class SmartPointer {

private:

T\* ptr; // Con trỏ quản lý bộ nhớ

public:

// Constructor: Nhận vào 1 con trỏ thô

explicit SmartPointer(T\* p = nullptr) : ptr(p) {} // Yêu cầu dùng hàm khởi tạo này

// Destructor: Giải phóng bộ nhớ khi đối tượng bị hủy

~SmartPointer() {

if (ptr) {

delete ptr;

}

}

// Xóa bỏ khả năng copy để đảm bảo chỉ có 1 con trỏ sở hữu đối tượng

SmartPointer(const SmartPointer&) = delete;

SmartPointer& operator = (const SmartPointer&) = delete;

// Cho phép chuyển nhượng quyền sở hữu (move semantics)

SmartPointer(SmartPointer&& other) noexcept : ptr(other.ptr) {

other.ptr = nullptr; // Chuyển quyền sở hữu để đối tượng cũ không còn trỏ đến dữ liệu

}

SmartPointer& operator = (SmartPointer&& other) noexcept {

if (this != other) {

// Giải phóng tài nguyên hiện tại

if (ptr) {

delete ptr;

}

// Chuyển quyền sở hữu

ptr = other.ptr;

other.ptr = nullptr;

}

return \*this;

}

// Toán tử dereference để truy cập giá trị

T& operator\*() const {

return \*ptr;

}

// Toán tử -> để truy cập thành viên của con trỏ

T\* operator->() const {

return ptr;

}

// Trả về con trỏ thô bên trong (nếu cần)

T\* get() const {

return ptr;

}

// Giải phóng quyền sở hữu và trả về con trỏ thô, không xóa đối tượng

T\* release() {

T\* temp = ptr;

ptr = nullptr;

return temp;

}

// Thay thế con trỏ hiện tại bằng 1 con trỏ mới

void reset(T\* p = nullptr) {

if (ptr) {

delete ptr;

}

ptr = p;

}

};

int main(){

SmartPointer<int> uptr(new int(10));

cout << "Value: " << \*uptr << endl;

}

```

\*\*Các loại Smart Pointer:\*\*

- Unique Pointer

- Shared Pointer

- Weak Pointer

## 19.2 Unique Pointer

\*\*Unique Pointer (unique\_ptr)\*\* là một loại smart pointer trong C++, giúp quản lý bộ nhớ động và tự động giải phóng bộ nhớ khi không còn cần thiết.

Đặc điểm chính của unique\_ptr:

- Một unique\_ptr chỉ có thể sở hữu một đối tượng hoặc mảng.

- Khi một unique\_ptr bị hủy, bộ nhớ của đối tượng sẽ được tự động giải phóng.

- Có thể gán một đối tượng khác bằng cách xóa đối tượng hiện tại khỏi con trỏ.

Ví dụ:

```

#include <iostream>

#include <memory>

using namespace std;

class Rectangle {

int length;

int width;

public:

Rectangle(int length, int width) : length(length), width(width) {}

int area() {

return length \* width;

}

};

int main() {

unique\_ptr<Rectangle> P1 = make\_unique<Rectangle>(10, 5);

cout << P1->area() << endl;

//unique\_ptr<Rectangle> P2 = P1; // Lỗi vì chỉ có 1 con trỏ có thể sở hữu đối tượng

unique\_ptr<Rectangle> P2;

P2 = move(P1); // Thay thế con trỏ hiện tại bằng 1 con trỏ mới

cout << P2->area() << endl;

return 0;

}

```

## 19.3 Shared Pointer

\*\*Shared Pointer (shared\_ptr)\*\* là một smart pointer khác trong C++ và cũng giúp quản lý bộ nhớ động.

Điểm đặc biệt của shared\_ptr:

- Bằng cách sử dụng shared pointer nhiều con trỏ có thể trỏ đến một đối tượng tại một thời điểm.

- Sử dụng một bộ đếm tham chiếu để theo dõi số lượng shared\_ptr đang tham chiếu đến một đối tượng.

- Chỉ giải phóng bộ nhớ khi không còn shared\_ptr nào tham chiếu đến nó.

Ví dụ:

```

#include <iostream>

#include <memory>

using namespace std;

int main(){

shared\_ptr<int> ptr1 = make\_shared<int>(10);

shared\_ptr<int> ptr2 = ptr1;

shared\_ptr<int> ptr3 = ptr1;

cout << "Value: " << \*ptr1 << endl;

cout << "Count: " << ptr1.use\_count() << endl; // Đếm xem có bao nhiêu object trỏ đến vùng nhớ

int \*ptr = ptr1.get(); // Trả về con trỏ thô

}

```

Kết quả:

```

Value: 10

Count: 3

```

Khi khai báo shared pointer trong 1 hàm nhỏ thì sau khi kết thúc hàm nó sẽ thu hồi con trỏ tham chiếu đến đối tượng.

```

#include <iostream>

#include <memory>

using namespace std;

int main(){

shared\_ptr<int> ptr1 = make\_shared<int>(10);

shared\_ptr<int> ptr2 = ptr1;

{

shared\_ptr<int> ptr3 = ptr1;

cout << "Count: " << ptr1.use\_count() << endl;

}

// Sau khi kết thú hàm ptr3 sẽ bị thu hồi

cout << "Value: " << \*ptr1 << endl;

cout << "Count: " << ptr1.use\_count() << endl;

}

```

Kết quả:

```

Count: 3

Value: 10

Count: 2

```

## 19.4 Weak Pointer

\*\*Weak Pointer (weak\_ptr)\*\* là một cơ chế giữ tham chiếu yếu (weak reference) đến một đối tượng được quản lý bởi shared\_ptr. Nó cung cấp một cách an toàn để theo dõi một đối tượng mà không tăng bộ đếm tham chiếu của shared\_ptr và weak\_ptr không trực tiếp truy cập đến đối tượng (object) mà nó theo dõi.

weak\_ptr có một phương thức là lock(), mà trả về một shared\_ptr. Nếu shared\_ptr mà weak\_ptr theo dõi vẫn tồn tại, lock() sẽ trả về một shared\_ptr hợp lệ có thể sử dụng để truy cập đối tượng. Ngược lại, nếu shared\_ptr đã bị giải phóng, lock() sẽ trả về một shared\_ptr rỗng.

Ví dụ:

```

#include <iostream>

#include <memory>

using namespace std;

int main() {

shared\_ptr<int> ptr1 = make\_shared<int>(10);

weak\_ptr<int> ptr2 = ptr1;

cout << ptr1.use\_count() << endl; // 1

//cout << "ptr2: " << \*ptr2 << endl; // Weak pointer không đọc được

cout << "ptr2: " << \*(ptr2.lock()) << endl; // Dùng lock để chuyển weak pointer -> shared pointer

// Weak pointer dùng để truy cập đến địa chỉ, đọc giá trị và không thể sửa được giá trị tại địa chỉ đó

cout << ptr2.expired(); // Trả về true nếu vùng nhớ bị thu hồi, và ngược lại

}

```

Kết quả:

```

Count: 1

ptr2: 10

0

```

</details>

<details>

<summary><strong>BÀI 20: DESIGN PATTERN</strong></summary>

# BÀI 20: DESIGN PATTERN

\*\*Design Pattern\*\* là một giải pháp thiết kế, một công thức đã được chứng minh bằng thực tế để giải quyết một vấn đề thường gặp trong phát triển phần mềm.

Mỗi một Design Pattern sẽ đưa ra các interfaces, classes, mối quan hệ giữa chúng và cách thức chúng hoạt động để giải quyết được vấn đề một cách linh hoạt, đảm bảo tính mở rộng, bảo trì, sử dụng lại.

## 20.1 Observer Pattern

\*\*Observer Pattern\*\* là design pattern dùng để tạo ra mối quan hệ phụ thuộc giữa các đối tượng. Khi một đối tượng thay đổi trạng thái, tất cả các phụ thuộc của nó sẽ được thông báo và cập nhật tự động.

```

#include <iostream>

#include <string>

#include <vector>

#include <algorithm>

using namespace std;

// Interface for observers (display, logger, etc.)

class Observer {

public:

virtual void update(float temperature, float humidity, float light) = 0;

};

// Subject (SensorManager) holds the state and notifies observers

class SensorManager {

float temperature;

float humidity;

float light;

vector<Observer\*> observers;

public:

void registerObserver(Observer\* observer) {

observers.push\_back(observer);

}

void removeObserver(Observer\* observer) {

observers.erase(remove(observers.begin(), observers.end(), observer), observers.end());

}

void notifyObservers() {

for (auto observer : observers) {

observer->update(temperature, humidity, light);

}

}

void setMeasurements(float temp, float hum, float lightLvl) {

temperature = temp;

humidity = hum;

light = lightLvl;

notifyObservers();

}

};

// Display component (an observer)

class Display : public Observer {

public:

void update(float temperature, float humidity, float light) override {

cout << "Display: Temperature: " << temperature

<< ", Humidity: " << humidity

<< ", Light: " << light << endl;

}

};

// Logger component (an observer)

class Logger : public Observer {

public:

void update(float temperature, float humidity, float light) override {

cout << "Logging data... Temp: " << temperature

<< ", Humidity: " << humidity

<< ", Light: " << light << endl;

}

};

int main() {

SensorManager sensorManager;

Display display;

Logger logger;

sensorManager.registerObserver(&display);

sensorManager.registerObserver(&logger);

sensorManager.setMeasurements(25.0, 60.0, 700.0);

}

```

## 20.2 Singleton Pattern

\*\*Singleton Pattern\*\* là design pattern mà chỉ có tối đa một đối tượng cụ thể của class được khởi tạo xuyên suốt chương trình. Khi tạo ra nhiều object thì nó sẽ chỉ dùng chung 1 địa chỉ và giúp tối ưu bộ nhớ.

```

#include <iostream>

void gpioInit() {

//

}

void gpioSetPin(int pin, bool value) {

//

}

int gpioReadPin(int pin) {

//

return 0;

}

class GpioManager {

private:

GpioManager() {} // Constructor private

static GpioManager\* instance; // Con trỏ tĩnh instance

void init() {

gpioInit();

}

public:

static GpioManager\* getInstance() {

if (!instance) {

instance = new GpioManager();

instance->init();

}

return instance;

}

void setPin(int pin, bool value) {

gpioSetPin(pin, value);

}

int readPin(int pin) {

return gpioReadPin(pin);

}

};

GpioManager\* GpioManager::instance = nullptr;

int main() {

GpioManager\* gpioManager = GpioManager::getInstance();

gpioManager->setPin(5, true);

int pinValue = gpioManager->readPin(5);

std::cout << "Pin 5 value: " << pinValue << std::endl;

return 0;

}

```

## 20.3 Factory Pattern

\*\*Factory Pattern\*\* là design pattern dùng để khởi tạo 1 object mà lớp con quyết định loại đối tượng nào.

```

#include <iostream>

using namespace std;

typedef enum {

TEMPERATURE\_SENSOR,

HUMIDITY\_SENSOR

//

} SensorType;

class Sensor {

public:

virtual void readData() = 0;

};

// Cảm biến nhiệt độ

class TemperatureSensor : public Sensor {

public:

void readData() override {

cout << "Reading temperature data: " << endl;

}

};

// Cảm biến độ ẩm

class HumiditySensor : public Sensor {

public:

void readData() override {

cout << "Reading humidity data: " << endl;

}

};

// Factory Pattern

class SensorFactory {

public:

static Sensor\* creatSensor(const SensorType type) {

switch (type)

{

case TEMPERATURE\_SENSOR:

return new TemperatureSensor();

break;

case HUMIDITY\_SENSOR:

return new HumiditySensor();

break;

default:

return nullptr;

break;

}

}

};

int main() {

Sensor\* sensor = SensorFactory::creatSensor(TEMPERATURE\_SENSOR);

sensor->readData();

}

```

## 20.4 Decorator Pattern

\*\*Decorator Patten\*\* là design pattern thêm tính năng mới vào object mà không làm thay đổi cấu trúc của class.

```

#include <iostream>

using namespace std;

class Sensor {

public:

virtual void readData() = 0;

};

// Cảm biến nhiệt độ

class TemperatureSensor : public Sensor {

public:

void readData() override {

cout << "Reading temperature data: " << endl;

}

};

// Cảm biến độ ẩm

class HumiditySensor : public Sensor {

public:

void readData() override {

cout << "Reading humidity data: " << endl;

}

};

// Decorator Pattern

class SensorDecorator : public Sensor {

protected:

Sensor\* wrappedSensor;

public:

SensorDecorator(Sensor\* sensor) : wrappedSensor(sensor) {}

virtual void readData() override {

wrappedSensor->readData();

}

};

// Thêm tính năng Log

class LoggingSensor : public SensorDecorator {

public:

LoggingSensor(Sensor\* sensor) : SensorDecorator(sensor) {}

void readData() override {

cout << "LOG: Sensor data" << endl;

SensorDecorator::readData();

}

};

int main() {

Sensor\* sensor = new TemperatureSensor();

Sensor\* log = new LoggingSensor(sensor);

log->readData();

}

```

</details>

<details>

<summary><strong>BÀI 21: PROCESS - THREAD</strong></summary>

# BÀI 21: PROCESS - THREAD

## 21.1 Khái niệm

- \*\*Process\*\* là một thực thể thực thi của một ứng dụng. Khi khởi tạo 1 process thì nó sẽ có 1 phân vùng RAM riêng, các process khác không thể truy cập đến vùng RAM của process này để chỉnh sửa được.

- \*\*Thread\*\* là một phần thực thi trong một process. Một process có thể chứa nhiều thread và các thread trong cùng 1 process thì có thể chia sẻ tài nguyên (địa chỉ bộ nhớ).

```

#include <iostream>

#include <thread>

#include <chrono>

using namespace std;

void delay(uint64\_t time) {

this\_thread::sleep\_for(chrono::milliseconds(time));

}

void task1(int time) {

int i = 0;

for (int j = 0; j < 5; j++)

{

cout << "Task 1, i: " << i++ << endl;

delay(time);

}

}

void task2(int time) {

int i = 0;

for (int j = 0; j < 5; j++)

{

cout << "Task 2, i: " << i++ << endl;

delay(time);

}

}

int main() {

thread t1(task1, 500);

thread t2(task2, 1000);

t1.join(); // Tạo ra 1 luồng

t2.join(); // Tạo ra 1 luồng

// t1.detach();

// t2.detach();

cout << "Ket thuc!" << endl;

return 0;

}

```

| |join|detach|

|:--:|:--:|:----:|

|Khối chương trình|Luồng chính bị chặn, chờ thread con kết thúc|Luồng chính tiếp tục, thread con tự chạy độc lập|

|Khả năng kiểm soát|Có thể theo dõi tiến trình và kết quả của thread|Không thể kiểm soát hoặc đồng bộ thread|

|Ứng dụng|Khi cần đảm bảo thread con hoàn thành trước|Khi thread con thực hiện tác vụ dài và độc lập|

Kiểm tra `joinable`:

- Trả về `true` nếu thread có thể join.

- Trả về `false` nếu thread không thể join, tức là:

- Thread đã được join.

- Thread đã được detach.

- Thread chưa được gán (khởi tạo với trạng thái rỗng, không có thread nào chạy)

Ví dụ:

```

#include <iostream>

#include <thread>

void threadFunction() {

std::cout << "Thread is running!" << std::endl;

}

int main() {

std::thread t(threadFunction);

if (t.joinable()) {

std::cout << "Thread is joinable!" << std::endl;

t.join(); // Kết hợp thread con với luồng chính

} else {

std::cout << "Thread is not joinable!" << std::endl;

}

return 0;

}

```

Kết quả:

```

Thread is joinable!

Thread is running!

```

Ví dụ:

```

#include <iostream>

#include <thread>

void threadFunction() {

std::cout << "Thread is running!" << std::endl;

}

int main() {

std::thread t;

if (t.joinable()) {

t.join();

} else {

std::cout << "Thread is not joinable because it is default-constructed!" << std::endl;

}

std::thread t2(threadFunction);

t2.detach(); // Tách thread ra

if (t2.joinable()) {

t2.join();

} else {

std::cout << "Thread is not joinable because it is detached!" << std::endl;

}

return 0;

}

```

Kết quả:

```

Thread is not joinable because it is default-constructed!

Thread is running!

Thread is not joinable because it is detached!

```

## 21.2 Race condition

\*\*Race condition (điều kiện tranh chấp)\*\* xảy ra trong lập trình đa luồng khi hai hoặc nhiều luồng cùng truy cập và thao tác trên tài nguyên dùng chung (shared resource) mà không có sự đồng bộ hóa thích hợp, dẫn đến hành vi không xác định hoặc không mong muốn.

Ví dụ:

```

#include <iostream>

#include <thread>

using namespace std;

void display(int id) {

while(1) {

cout << "This is task display: " << id << endl;

delay(500);

}

}

int main() {

thread t1(display, 1);

thread t2(display, 2);

thread t3(display, 3);

t1.join();

t2.join();

t3.join();

return 0;

}

```

Khi chạy chương trình có thể sẽ có 2 luồng ghi đè lên nhau do không đồng bộ.

Vì vậy, có thể sử dụng \*\*mutex\*\* để đảm bảo chỉ một luồng có thể truy cập tài nguyên tại một thời điểm.

```

#include <iostream>

#include <thread>

#include <chrono>

#include <mutex>

using namespace std;

mutex mtx;

void delay(uint64\_t time) {

this\_thread::sleep\_for(chrono::milliseconds(time));

}

void display(int id) {

while(1) {

lock\_guard<mutex> lock(mtx); // Tự động khóa và mở khóa mutex

cout << "This is task display: " << id << endl;

delay(100);

cout << "Task: " << id << endl;

}

}

int main() {

thread t1(display, 1);

thread t2(display, 2);

thread t3(display, 3);

t1.join();

t2.join();

t3.join();

return 0;

}

```

- `std::unique\_lock` cũng dùng để quản lý việc khóa và mở khóa mutex nhưng cung cấp tính linh hoạt cao hơn so với `std::lock\_guard`. `std::unique\_lock` cho phép khóa và mở khóa thủ công, cũng như chuyển quyền sở hữu mutex giữa các đối tượng.

Ở ví dụ trên, giả sử thread t1 chạy dòng `cout << "This is task display: " << id << endl;` và muốn thread 2 chạy nhưng thread 2 phải đợi chạy xong hàm `display` mới có thể chạy. Vì vậy, có thể dùng `unique\_lock` để mở khóa thủ công để thread 2 có thể chạy luôn mà không cần đợi chạy xong hàm `display`.

```

#include <iostream>

#include <thread>

#include <chrono>

#include <mutex>

using namespace std;

mutex mtx;

void delay(uint64\_t time) {

this\_thread::sleep\_for(chrono::milliseconds(time));

}

void display(int id) {

while(1) {

unique\_lock<mutex> lock(mtx); // Khóa mutex

cout << "This is task display: " << id << endl;

lock.unlock(); // Mở khóa thủ công

delay(200);

lock.lock(); // Khóa thủ công

cout << "Task: " << id << endl;

lock.unlock(); // Mở khóa thủ công

}

}

int main() {

thread t1(display, 1);

thread t2(display, 2);

thread t3(display, 3);

t1.join();

t2.join();

t3.join();

return 0;

}

```

- `atomic` đảm bảo các thao tác trên biến nguyên tử (atomic operations), tức là không thể bị gián đoạn bởi luồng khác. `std::atomic` được cung cấp bởi thư viện <atomic>.

- Đảm bảo tính nguyên tử (atomicity):Mỗi thao tác đọc/ghi hoặc sửa đổi trên biến nguyên tử là toàn vẹn (không bị gián đoạn bởi các luồng khác) và loại bỏ các vấn đề race condition trong các thao tác cơ bản mà không cần sử dụng mutex.

- Hiệu suất cao hơn so với mutex: Không cần phải khóa toàn bộ tài nguyên, do đó giảm chi phí đồng bộ hóa.

```

#include <iostream>

#include <thread>

#include <chrono>

#include <mutex>

#include <atomic>

using namespace std;

mutex mtx;

atomic<int> counter(0);

void delay(uint64\_t time) {

this\_thread::sleep\_for(chrono::milliseconds(time));

}

void display() {

while(1) {

cout << "Counter: " << ++counter << endl;

delay(200);

}

}

int main() {

thread t1(display);

thread t2(display);

thread t3(display);

t1.join();

t2.join();

t3.join();

return 0;

}

```

## 21.3 Condition variable

\*\*Condition Variable\*\* trong C++ là một cơ chế đồng bộ hóa trong lập trình đa luồng, được sử dụng để cho phép một hoặc nhiều luồng chờ trong khi một luồng khác thông báo (notify) khi điều kiện cụ thể được thỏa mãn. `std::condition\_variable` là công cụ chính trong C++ để thực hiện điều này, được cung cấp trong thư viện <condition\_variable>.

```

#include <iostream>

#include <thread>

#include <mutex>

#include <condition\_variable>

using namespace std;

int sensorData = 0;

bool checkData = false;

mutex mtx;

condition\_variable cv;

void sensorRead() {

while (1)

{

this\_thread::sleep\_for(chrono::seconds(2)); // Delay 2s

unique\_lock<mutex> lock(mtx); // Khóa mutex

sensorData = rand() % 100;

checkData = true;

cout << "Read data done!" << endl;

lock.unlock(); // Mở khóa thủ công

cv.notify\_one(); // Gửi thông báo đến một luồng đang chờ

//cv.notify\_all(); // Gửi thông báo đến tất cả các luồng đang chờ

}

}

void processData() {

while (1)

{

unique\_lock<mutex> lock(mtx); // Khóa mutex

cv.wait(lock, []{return checkData;}); // Đợi đọc giá trị cảm biến mới, tức là checkData trả về true

cout << "Data: " << sensorData << endl;

checkData = false;

lock.unlock(); // Mở khóa thủ công

}

}

int main() {

thread task1(sensorRead);

thread task2(processData);

task1.join();

task2.join();

return 0;

}

```

## 21.4 Bất đồng bộ

\*\*Bất đồng bộ (asynchronous)\*\* đề cập đến việc một tác vụ được thực thi mà không cần phải chờ tác vụ đó hoàn thành trước khi tiếp tục các tác vụ khác. Nó cho phép chương trình thực hiện nhiều công việc đồng thời, giúp tận dụng tài nguyên tốt hơn và tăng hiệu suất, đặc biệt trong các ứng dụng đòi hỏi xử lý đồng thời hoặc I/O.

Ví dụ:

```

#include <iostream>

#include <thread>

#include <future>

#include <chrono>

using namespace std;

int asyncTask() {

this\_thread::sleep\_for(chrono::seconds(3));

return 100;

}

void processResult(int id, future<int> ft) {

cout << "Task: " << id << ": " << ft.get() << endl;

}

int main() {

future<int> ft = async(launch::async, asyncTask);

thread t1(processResult, 1, move(ft)); // Với future phải dùng lệnh std::move để truy cập

t1.join();

return 0;

}

```

Ví dụ:

```

#include <iostream>

#include <thread>

#include <future>

#include <chrono>

using namespace std;

int asyncTask() {

this\_thread::sleep\_for(chrono::seconds(3));

return 100;

}

void processResult(int id, shared\_future<int> ft) {

cout << "Task: " << id << ": " << ft.get() << endl;

}

int main() {

//future<int> ft = async(launch::async, asyncTask);

//shared\_future<int> shared\_ft = ft.share();

shared\_future<int> shared\_ft = async(launch::async, asyncTask).share();

thread t1(processResult, 1, shared\_ft);

thread t2(processResult, 2, shared\_ft);

t1.join();

t2.join();

return 0;

}

```

\*\*Một số hàm khác\*\*

- `wait()`:

```

#include <iostream>

#include <thread>

#include <future>

#include <chrono>

using namespace std;

int asyncTask() {

this\_thread::sleep\_for(chrono::seconds(3));

return 100;

}

void processResult(int id, shared\_future<int> ft) {

cout << "Task: " << id << ": " << ft.get() << endl;

}

int main() {

future<int> ft = async(launch::async, asyncTask);

cout << "LOG 1" << endl;

ft.wait(); // Chờ tác vụ bất đông bộ hoàn thành

cout << "LOG 2: " << ft.get() << endl;

return 0;

}

```

- `wait\_for()`:

```

#include <iostream>

#include <thread>

#include <future>

#include <chrono>

using namespace std;

int asyncTask() {

this\_thread::sleep\_for(chrono::seconds(3));

return 100;

}

void processResult(int id, shared\_future<int> ft) {

cout << "Task: " << id << ": " << ft.get() << endl;

}

int main() {

future<int> ft = async(launch::async, asyncTask);

cout << "LOG 1" << endl;

// Chờ một khoảng thời gian nhất định trước khi trả lại kết quả, thay vì đợi vô thời hạn

if (ft.wait\_for(std::chrono::seconds(1)) == std::future\_status::ready) {

cout << "Task finished before timeout." << endl;

} else {

cout << "Task did not finish within the timeout." << endl;

}

return 0;

}

```

- `wait\_until()`:

```

#include <iostream>

#include <thread>

#include <future>

#include <chrono>

using namespace std;

int asyncTask() {

this\_thread::sleep\_for(chrono::seconds(3));

return 100;

}

void processResult(int id, shared\_future<int> ft) {

cout << "Task: " << id << ": " << ft.get() << endl;

}

int main() {

future<int> ft = async(launch::async, asyncTask);

cout << "LOG 1" << endl;

auto timeout = chrono::system\_clock::now() + chrono::seconds(10);

// Chờ đến một thời điểm cụ thể thay vì chờ trong một khoảng thời gian nhất định.

if (ft.wait\_until(timeout) == std::future\_status::ready) {

cout << "Task finished before timeout." << endl;

} else {

cout << "Task did not finish before the deadline." << endl;

}

return 0;

}

```

- `valid()`:

```

#include <iostream>

#include <thread>

#include <future>

#include <chrono>

using namespace std;

int asyncTask() {

this\_thread::sleep\_for(chrono::seconds(3));

return 100;

}

void processResult(int id, shared\_future<int> ft) {

cout << "Task: " << id << ": " << ft.get() << endl;

}

int main() {

future<int> ft = async(launch::async, asyncTask);

// Kiểm tra xem tác vụ bất đồng bộ đã hoàn thành hay chưa, hoặc nếu có một vấn đề xảy ra khi khởi tạo future

if (ft.valid()) {

cout << "Future is valid." << endl;

} else {

cout << "Future is not valid." << endl;

}

return 0;

}

```

</details>