Chương 3: Hàm

Nội dung

1. Truyền tham số
2. Tham số ngầm định
3. Đa năng hóa hàm (function overloading)
4. Đa năng hóa toán tử (Operator Overloading)
5. Con trỏ hàm
6. Khái quát hóa hàm (function templates)
7. Biểu thức lambda - hàm nặc/vô danh

Khái niệm về hàm

- Là một nhóm các khai báo và các câu lệnh được gán một tên gọi
  - Đây là khối lệnh được đặt tên, nên sử dụng thuận tiện, hiệu quả
  - Hàm thường trả về một giá trị
- Là một chương trình con
  - Khi viết chương trình C/C++ ta luôn định nghĩa một hàm có tên là main
  - Phía trong hàm main ta có thể gọi các hàm khác
    - Bản thân các hàm này lại có thể gọi các hàm khác ở trong nó và cứ tiếp tục như vậy...

Cú pháp

```
return-type name(argument-list){
   local-declarations
   statements
   return return-value;
}
```

Ví dụ: Square

```
double square(double a){
   return a * a;
}
```

Đây là định nghĩa hàm ngoài hàm main

```
int main(void){
   double num = 0.0, sqr = 0.0;

   printf("enter a number\n");
   scanf("%lf",&num);

   sqr = square(num);

   printf("square of %g is %g\n", num, sqr);

   return 0;
}
```

Đây là chỗ gọi hàm square

Tại sao cần sử dụng hàm?

- Chia vấn đề thành nhiều tác vụ con
  - Dễ dàng hơn khi giải quyết các vấn đề phức tạp
- Tổng quát hóa được tập các câu lệnh hay lặp lại
  - Ta không phải viết cùng một thứ lặp đi lặp lại nhiều lần
  - printf và scanf là ví dụ điển hình…
- Hàm giúp chương trình dễ đọc và bảo trì hơn nhiều

Truyền tham số cho hàm

- Truyền theo giá trị (Pass by value)
  - Cách thức: Khi truyền theo giá trị, một bản sao của đối số được tạo và truyền vào hàm.
  - → Thay đổi giá trị của tham số bên trong hàm sẽ không ảnh hưởng đến giá trị của đối số ban đầu ở hàm gọi.

- Truyền theo con trỏ (Pass by pointer) (C, C++)
  - Một bản sao của con trỏ (địa chỉ) được truyền vào hàm. Tuy nhiên, vì con trỏ trỏ đến biến gốc, việc thay đổi giá trị thông qua con trỏ sẽ ảnh hưởng đến biến gốc.
  - Có thể thay đổi con trỏ để trỏ đến một địa chỉ khác, nhưng điều này không ảnh hưởng đến địa chỉ của biến gốc bên ngoài hàm.

- Truyền theo tham chiếu (Pass by reference) (C++)
  - Không có bản sao nào được tạo ra. Tham chiếu là một tên khác của biến gốc, nên mọi thay đổi trong hàm đều trực tiếp ảnh hưởng đến biến gốc.
  - Không cần giải tham chiếu, cú pháp đơn giản hơn.

Minh hoạ về bộ nhớ

Giả sử biến x nằm ở địa chỉ bộ nhớ 0x100:

- Pass by value:
  - Hàm tạo một bản sao tại địa chỉ mới (ví dụ 0x200), và thay đổi bản sao không ảnh hưởng đến 0x100.

- Pass by pointer:
  - Hàm nhận con trỏ chứa giá trị 0x100, rồi dùng * để truy cập vùng nhớ đó.

- Pass by reference:
  - Hàm không nhận địa chỉ hay bản sao, mà biến num trong hàm được trình biên dịch hiểu là chính 0x100. Mọi thao tác trên num thực chất là thao tác trên x.

Truyền theo giá trị (Pass by value)

```
#include <stdio.h>

void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}

int main() {
    int x = 10, y = 20;
    swap(x, y);
    printf("x = %d, y = %d\n", x, y); // K ế t quả: x = 10, y = 20
    return 0;
}
```

```
x = 10, y = 20
```


Truyền theo con trỏ (Pass by pointer)

```
#include <stdio.h>

void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int x = 10, y = 20;
    swap(&x, &y);
    printf("x = %d, y = %d\n", x, y); // K ế t quả: x = 20, y = 10
    return 0;
}
```

```
x = 20, y = 10
```

Truyền theo tham chiếu (Pass by reference) (C++)

```
#include <iostream>

using namespace std;

void swap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

int main() {
    int x = 10, y = 20;
    cout << "Trước khi đổi: x = " << x << ", y = " << y << endl;
    swap(x, y);
    cout << "Sau khi đổi: x = " << x << ", y = " << y << endl;
    return 0;
}
```

```
Trước khi đổi: x = 10, y = 20
Sau khi đổi: x = 20, y = 10
```


Ví dụ khác: Truyền mảng

- Mảng trong C++ thực chất là một con trỏ đến phần tử đầu tiên của mảng.
    - Do đó, khi truyền mảng vào hàm, ta thực chất đang truyền một con trỏ. Điều này có nghĩa là các thay đổi đối với mảng bên trong hàm sẽ ảnh hưởng đến mảng gốc.

```
void printArray(int arr[], int size) {
    // ...
}
```

Truyền tham chiếu

Khi một hàm trả về một tham chiếu, chúng ta có thể gọi hàm ở phía bên trái của một phép gán.

```
#include <iostream>
using namespace std;
int X = 4;
int& MyFunc(){
    return X;
}
int main(){
    cout << "X=" << X << endl;
    cout << "X=" << MyFunc() << endl;
    MyFunc() = 20; // ~X=20
    cout << "X=" << X << endl;
    return 0;
}
```

```
X=4
X=4
X=20
```


Hàm và truyền tham số

- **Trong C:**
  - Tên hàm phải là duy nhất, lời gọi hàm phải có các đối số đúng bằng và hợp tương ứng về kiểu với tham số trong định nghĩa hàm.
- **Trong C++:**
  - Có cơ chế đa năng hóa hàm, vì vậy tên hàm không phải duy nhất.
  - Có kiểu tham số ngầm định (default parameter), vì vậy số đối số trong lời gọi hàm có thể ít hơn tham số định nghĩa.

Tham số ngầm định

- Tham số mặc định cho phép gán một giá trị mặc định cho một hoặc nhiều tham số của hàm.
  - Điều này làm cho việc gọi hàm trở nên linh hoạt hơn, giảm thiểu số lượng các phiên bản hàm cần phải viết.
- Quy tắc chung:
  - Các tham số có giá trị mặc định phải đứng sau các tham số không có giá trị mặc định trong danh sách tham số của hàm.
  - Khi gọi hàm, bạn có thể truyền giá trị cho tất cả các tham số hoặc bỏ qua các tham số có giá trị mặc định. Nếu một tham số bị bỏ qua, nó sẽ tự động nhận giá trị mặc định.
- Ví dụ:

```
int MyFunc(int a = 1, int b, int c = 3, int d = 4);  //  ✖
int MyFunc(int a, int b = 2, int c = 3, int d = 4);  //  ✔
```


Tham số ngầm định - Ví dụ

```
#include <iostream>

using namespace std;

void printMessage(string message = "Hello, world!") {
    cout << message << endl;
}

int main() {
    printMessage();         // In ra: Hello, world!
    printMessage("Xin chào!"); // In ra: Xin chào!
    return 0;
}
```

```
Hello, world!
Xin chào!
```

Đa năng hóa hàm (Function Overloading)

- Cho phép định nghĩa nhiều hàm có cùng tên nhưng khác nhau về số lượng tham số hoặc kiểu dữ liệu của tham số.
  - □ Điều này giúp code trở nên linh hoạt và dễ đọc hơn.

C

```
int abs(int i);
long labs(long l);
double fabs(double d);
```

C++

```
int abs(int i);
long abs(long l);
double abs(double d);
```


Đa năng hóa hàm (Function Overloading)

```
#include <iostream>

using namespace std;

int tinhTong(int a, int b) {
    return a + b;
}

double tinhTong(double a, double b) {
    return a + b;
}

int main() {
    cout << tinhTong(2, 3) << endl;  // Gọi hàm tính tổng cho số nguyên
    cout << tinhTong(2.5, 3.14) << endl; // Gọi hàm tính tổng cho số thực
    return 0;
}
```

Đa năng hóa toán tử (Operator Overloading)

- Là một tính năng mạnh mẽ trong C++ cho phép tái định nghĩa ý nghĩa của các toán tử (như +, -, *, /, ==, !=, ...) khi sử dụng chúng với các kiểu dữ liệu tự định nghĩa (như class, struct)
- Ưu điểm:
  - Làm cho code tự nhiên hơn
    - Có thể sử dụng các toán tử quen thuộc để thực hiện các phép toán trên các đối tượng của lớp tự tạo, giúp code trở nên dễ đọc và dễ hiểu hơn.
  - Tính trừu tượng
    - Giúp tăng cường tính trừu tượng của lập trình hướng đối tượng, cho phép định nghĩa các hành vi của đối tượng một cách trực quan.
- Ví dụ: thực hiện các phép cộng, trừ số phức

Đa năng hóa toán tử trong C

```
#include <stdio.h>

typedef struct {
    double real;
    double imag;
} Complex;

Complex addComplex(Complex a, Complex b) {
    Complex result;
    result.real = a.real + b.real;
    result.imag = a.imag + b.imag;
    return result;
}

Complex subtractComplex(Complex a, Complex b) {
    Complex result;
    result.real = a.real - b.real;
    result.imag = a.imag - b.imag;
    return result;
}

int main() {
    Complex c1 = {2, 3};
    Complex c2 = {5, 7};
    Complex c3 = addComplex(c1, c2);
    printf("%lf + %lfi\n", c3.real, c3.imag);
    return 0;
}
```

```
7.000000 + 10.000000i
```

Đa năng hóa toán tử trong C++

- Một hàm định nghĩa một toán tử có cú pháp sau:

```
data_type operator operator_symbol ( parameters ){
  .....................................
}
```

Trong đó:

- `data_type`: *Kiểu trả về.*
- `operator_symbol`: *Ký hiệu của toán tử.*
- `parameters`: *Các tham số (nếu có).*

Ví dụ:

```
// Đa năng hóa toán tử +
    Complex operator+(const Complex& obj) {
        Complex temp;
        temp.real = real + obj.real;
        temp.imag = imag + obj.imag;
        return temp;
    }
```


Đa năng hóa toán tử trong C++

```
#include <iostream>
using namespace std;
typedef struct {
    double real;
    double imag;
} Complex;

// Đa năng hóa toán tử +
Complex operator+(Complex c1, Complex c2) {
    Complex temp;
    temp.real = c1.real + c2.real;
    temp.imag = c1.imag + c2.imag;
    return temp;
}
// Đa năng hóa toán tử -
Complex operator-(Complex c1, Complex c2) {
    Complex temp;
    temp.real = c1.real - c2.real;
    temp.imag = c1.imag - c2.imag;
    return temp;
}
int main() {
    Complex c1={2, 3}, c2={5, 7};
    Complex c3 = c1 + c2; // Sử dụng toán tử +
    cout << c3.real << " + " << c3.imag << "i" << endl;
    return 0;
}
```

```
7 + 10i
```

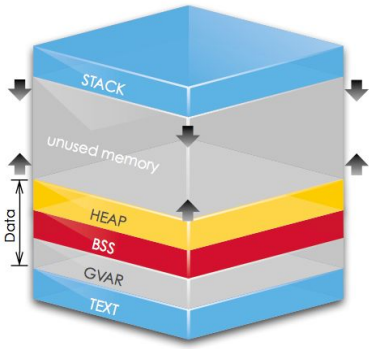
Các giới hạn của đa năng hóa toán tử

- Hầu hết các toán tử của C++ đều có thể được đa năng hóa. Các toán tử sau không được đa năng hóa là :
  - **::** Toán tử định phạm vi.
  - **.*** Truy cập đến con trỏ là trường của **struct** hay **class**.
  - **.** Truy cập đến trường của **struct** hay **class**.
  - **?:** Toán tử điều kiện
  - **sizeof**
- Không thể thay đổi thứ tự ưu tiên của một toán tử cũng như số các toán hạng của nó.
- Không thể thay đổi ý nghĩa của các toán tử khi áp dụng cho các kiểu có sẵn.
- Đa năng hóa các toán tử không thể có các tham số có giá trị mặc định.

Con trỏ hàm

```
int foo(){
    return 0;
}

int main(){
    int n = foo();
    return 0;
}
```



Cách bố trí các vùng nhớ chính trong một chương trình C/C++.
1. TEXT (Mã lệnh): Mã lệnh thực thi của chương trình, bao gồm các hàm, các câu lệnh, các hướng dẫn cho CPU.
2. GVAR (Biến toàn cục đã khởi tạo): Có tuổi thọ trong suốt thời gian thực thi của chương trình. Được khởi tạo trước khi chương trình bắt đầu chạy.
3. BSS (Biến toàn cục chưa khởi tạo): Có tuổi thọ trong suốt thời gian thực thi của chương trình.
4. HEAP (Đống): Các vùng nhớ được cấp phát động bằng các hàm như malloc, calloc, new trong C/C++. Kích thước có thể thay đổi trong quá trình thực thi. Được cấp phát và giải phóng theo yêu cầu của chương trình.
5. STACK (Ngăn xếp): Các biến cục bộ, tham số hàm, địa chỉ trả về hàm, các frame của hàm khi gọi hàm. Kích thước của ngăn xếp được hệ điều hành quản lý. Các biến cục bộ được tự động giải phóng khi kết thúc khối lệnh.

- Khi trong hàm main chạy đến dòng lệnh gọi hàm foo, hệ điều hành sẽ tìm đến địa chỉ của hàm foo trên bộ nhớ ảo và chuyển mã lệnh của hàm foo cho CPU tiếp tục xử lý
  - Con trỏ hàm là một biến chứa địa chỉ bắt đầu của một hàm. Nói cách khác, con trỏ hàm "trỏ" đến một hàm cụ thể trong chương trình.

Con trỏ hàm

```
#include <iostream>

using namespace std;

int foo(){
    return 0;
}

int main(){
    printf("%p\n", foo);//Địa chỉ bắt
    đầu của hàm foo
    return 0;
}
```

Kết quả:

```
013D1492
```


Cú pháp khai báo con trỏ hàm

```
<return_type> (*<name_of_ptr>)(<data_type_of_parameters>);
```

Ví dụ 1:

```
int foo(){
   return 0;
}
int (*pFoo) ();//Khai báo con trỏ hàm
```

Ví dụ 2:

```
void swapValue(int &value1, int &value2) {
    int temp = value1;
    value1 = value2;
    value2 = temp;
}
void(*pSwap) (int &, int &); //Khai báo con trỏ hàm
```

Ví dụ sử dụng con trỏ hàm

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

void swapValue(int &value1, int &value2){
   int temp = value1;
   value1 = value2;
   value2 = temp;
}

int main(){
   void(*pSwap) (int &, int &) = swapValue;
   int a = 1, b = 5;
   cout << "Before: " << a << " " << b << endl;
   (*pSwap)(a, b);
   cout << "After:  " << a << " " << b << endl;
   return 0;
}
```


Vai trò của con trỏ hàm

- Gọi hàm động:
  - Thay vì gọi trực tiếp một hàm bằng tên, có thể lưu địa chỉ của hàm đó vào một con trỏ hàm và sau đó gọi hàm thông qua con trỏ này. Điều này mang lại sự linh hoạt cao trong việc lựa chọn hàm cần gọi tại thời điểm chạy chương trình.
- Truyền hàm làm tham số:
  - Có thể truyền con trỏ hàm làm tham số cho một hàm khác, cho phép hàm đó sử dụng các hàm khác nhau tùy theo ngữ cảnh.
- Mảng các con trỏ hàm:
  - Tạo mảng các con trỏ hàm để lưu trữ địa chỉ của nhiều hàm khác nhau và gọi chúng một cách tuần tự hoặc theo điều kiện.
- Callback function:
  - Con trỏ hàm thường được sử dụng để thực hiện các callback function, tức là các hàm được gọi lại khi một sự kiện nào đó xảy ra.

Sắp xếp dãy số

```
#include <iostream>

using namespace std;

bool ascending(int left, int right){
   return left > right;
}
bool descending(int left, int right){
   return left < right;
}
void selectionSort(int *arr, int length, bool (*comparisonFunc)(int, int)){
   for (int i_start = 0; i_start < length; i_start++) {
      int minIndex = i_start;
      for (int i_current = i_start + 1; i_current < length; i_current++){
          if (comparisonFunc(arr[minIndex], arr[i_current])) {
             minIndex = i_current;
          }
      }
      swap(arr[i_start], arr[minIndex]); // std::swap
   }
}
void printArray(int *arr, int length){
  for (int i = 0; i < length; i++) {
     cout << arr[i] << " ";
  }
}

int main() {
   int arr[] = {1, 4, 2, 3, 6, 5, 8, 9, 7};
   int length = sizeof(arr) / sizeof(int);
   cout << "Before sorted: ";
   printArray(arr, length);
   selectionSort(arr, length, descending);
   cout << "\nAfter sorted:  ";
   printArray(arr, length);
   return 0;
}
```

```
Before sorted: 1 4 2 3 6 5 8 9 7
After sorted:  9 8 7 6 5 4 3 2 1
```


Khái quát hóa hàm (Function templates)

Ví dụ muốn tìm giá trị lớn nhất trong hai số:

- Đối với hai số nguyên:

```
int maxval(int x, int y){
     return (x > y) ? x : y;
}
```

- Đối với hai số thực:

```
double maxval(double x, double y){
     return (x > y) ? x : y;
}
```

 Cần khái quát hoá hàm: cho phép định nghĩa các hàm có thể làm việc với nhiều kiểu dữ liệu khác nhau mà không cần phải viết nhiều phiên bản hàm cho từng kiểu dữ liệu

Khái quát hóa hàm (Function templates)

Cú pháp Khai báo khuôn mẫu hàm:

```
template < parameter-list > function-declaration
```

Ví dụ:

```
template <typename T>
T maxval(T x, T y){
    return (x > y) ? x : y;
}
```


Khái quát hóa hàm (Function templates)

```
#include <iostream>
using namespace std;
template <typename T>
T maxval(T x, T y){
    return (x > y) ? x : y;
}
int main() {
    int i = maxval(3, 7); // returns 7
    cout << i << endl;
    double d = maxval(6.34, 18.523); // returns 18.523
    cout << d << endl;
    char ch = maxval('a', '6'); // returns 'a'
    cout << ch << endl;
    return 0;
}
```

```
7
18.523
a
```

Từ khóa auto

- Đối với biến (từ C++11): auto xác định kiểu của biến được khởi tạo một cách tự động từ giá trị khởi tạo của biến.

```
auto d { 5.0 }; // d will be type double
auto i { 1 + 2 }; // i will be type int
```

- Đối với hàm (từ C++14): auto tự động xác định kiểu trả về của hàm dựa vào câu lệnh return.

```
auto add(int x, int y) -> int;
auto divide(double x, double y) -> double;
auto printSomething() -> void;
auto generateSubstring(const std::string
&s, int start, int len) -> std::string;
```


Từ khóa auto

- Đối với kiểu tham số (từ C++14): auto tự động xác định kiểu của tham số dựa vào giá trị được truyền.
- Ví dụ:

```
#include <iostream>
using namespace std;
auto maxval(auto x, auto y){
    return (x > y) ? x : y;
}

int main() {
    int i = maxval(3.1, 7); // returns 7
    cout << i << endl;
    double d = maxval(6.34, 18.523); // returns 18.523
    cout << d << endl;
    char ch = maxval('a', '6'); // returns 'a'
    cout << ch << endl;
    return 0;
}
```

```
7
18.523
a
```

Hàm nặc danh - cú pháp lambda

- Lambda hay còn gọi là hàm nặc danh, nó có thể dùng để truyền vào một hàm khác và sử dụng một lần.
  - Khác với các cách dùng hàm thông thường buộc phải định nghĩa hàm sau đó dùng tên hàm truyền vào một hàm khác.
- Lợi ích của lambda là không nhất thiết phải khai báo tên hàm ở một nơi khác, mà có thể tạo ngay một hàm (dùng một lần hay hiểu chính xác hơn là chỉ có một chỗ gọi một số tác vụ nhỏ).
  - Như vậy, ta sẽ giảm được thời gian khai báo một hàm.
- Để làm rõ hơn về khái niệm này, ta sẽ xét 2 ví dụ sau.

Hàm nặc danh - cú pháp lambda

```
#include <iostream>
using namespace std;
//Phải khai báo hàm stdio_doing trước khi sử dụng
void stdio_doing(int n) {
    n = n + 10;
    cout << n << " ";
}

void for_each (int *arr, int n, void (*func)(int a)){
    for (int i = 0; i < n; i++) {
        func(*(arr + i));
    }
}

int main(){
    int arr[] ={1, 2, 3, 4, 5} , n = 5;
    for_each(arr, n, stdio_doing);
    return 0;
}
```

```
11 12 13 14 15
```

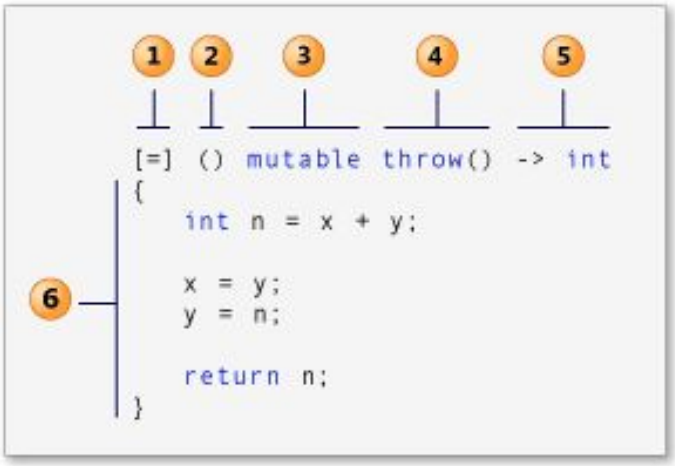
Hàm nặc danh - cú pháp lambda

```
#include <iostream>
using namespace std;
void for_each (int *arr, int n, void (*func)(int a)){
    for (int i = 0; i < n; i++)    {
        func(*(arr + i));
    }
}

int main(){
    int arr[] ={1, 2, 3, 4, 5} , n = 5;
    //Hàm stdio_doing được viết (thông qua cú pháp lambda)
    //và sử dụng trực tiếp 1 lần
    for_each(arr, n, [] (int a){
        a = a + 10;
        cout << a << " ";
    });
    return 0;
}
```

```
11 12 13 14 15
```


Hàm nặc danh - cú pháp lambda



(1) Mệnh đề bắt giữ (capture clause)
(2) Danh sách tham số
(3) Tính bền vững của lambda
(4) Ngoại lệ có thể xảy ra trong lambda.
(5) Kiểu trả về của lambda
(6) Phần thân lambda

Hàm nặc danh - cú pháp lambda

Mệnh đề bắt giữ (capture clause)

- [capture list]: Danh sách bắt giữ các biến từ ngữ cảnh xung quanh. Nó cho phép lambda truy cập vào các biến bên ngoài phạm vi của nó.
  - []: Không bắt giữ biến nào.
  - [=]: Bắt giữ tất cả các biến đã sử dụng theo giá trị.
  - [&]: Bắt giữ tất cả các biến đã sử dụng theo tham chiếu.
  - [a, &b]: Bắt giữ biến a theo giá trị và biến b theo tham chiếu.

Hàm nặc danh - cú pháp lambda

Danh sách tham số

Ngoài khả năng bắt giữ các biến bên ngoài, lambda còn có thể nhận đối số bằng cách khai báo danh sách tham số.

```
auto y = [] (int first, int second){
    return first + second;
};
```

Tính bền vững trong một lambda (mutable)

Nếu chúng ta thêm từ khóa mutable vào một lambda, nó cho phép lambda thay đổi giá trị những biến được bắt giữ theo giá trị.

Hàm nặc danh - cú pháp lambda

Kiểu trả về của một lambda

Chúng ta có thể trả về bất kỳ kiểu dữ liệu nào giống như hàm thông thường. Ví dụ:

```
// OK: return type is int
auto x1 = [](int i){ return i; };
```

Để chương trình được rõ ràng hơn, chúng ta nên viết lambda có kiểu trả về như sau:

```
auto x1 = [](int i) -> int {
    return i;
};
```

Thêm khai báo `-> int` giúp việc đọc hiểu lambda dễ dàng hơn.

Ví dụ

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

int main(){
    int m = 0;
    int n = 0;
    auto func = [&, n] (int a) mutable {
        m = ++n + a;
    };
    func(4);
    cout << m << endl << n << endl;
}
```

Kết quả:

5

0

- Lambda capture m bằng tham chiếu (&), nên thay đổi m trong lambda ảnh hưởng trực tiếp đến m trong main.
- Lambda capture n bằng giá trị (n), và nhờ mutable, nó có thể thay đổi bản sao của n bên trong lambda (từ 0 thành 1), nhưng không ảnh hưởng đến n gốc trong main.
- Khi func(4) chạy, m = ++n + a tính toán 5 (1 + 4) và gán cho m, còn n trong main giữ nguyên 0.

Tài liệu đọc thêm

1. Function templates: https://docs.microsoft.com/en-us/cpp/cpp/function-templates?view=vs-2019
2. Auto: https://docs.microsoft.com/en-us/cpp/cpp/auto-cpp?view=vs-2019
3. Lambda expression: https://docs.microsoft.com/en-us/cpp/cpp/lambda-expressions-in-cpp?view=vs-2019

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Xin cảm ơn!

soict.hust.edu.vn/ fb.com/groups/soict