Chương 3: Hàm

Nội dung

1. Truyền tham trị, tham biến và tham số ngầm định
2. Đa năng hóa hàm (function overload)
3. Con trỏ hàm và tham số hóa hàm
4. Khái quát hóa hàm (function templates)
5. Biểu thức lambda và hàm nặc 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

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 đn hàm. C chỉ có duy nhất 1 cách truyền tham số: tham trị (kể cả dùng địa chỉ cũng vậy).
- **Trong C++:** ngoài truyền tham trị, C++ còn cho phép truyền tham chiếu. Tham số trong C++ còn 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. Đồng thời C++ còn có cơ chế đa năng hóa hàm, vì vậy tên hàm không phải duy nhất.

Truyền tham chiếu

- Hàm nhận tham số là con trỏ

```
void Swap(int *X, int *Y) {
   int Temp = *X;
   *X = *Y;
   *Y = Temp;
}
```

- Để hoán đổi giá trị hai biến A và B

```
Swap(&A, &B);
```

Truyền tham chiếu

- Hàm nhận tham số là tham chiếu

```
void Swap(int &X, int &Y){
   int Temp = X;
   X = Y;
   Y = Temp;
}
```

- Để hoán đổi giá trị hai biến A và B

```
Swap(A, B);
```


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;
}
```


Tham số ngầm định

- Định nghĩa các giá trị tham số mặc định cho các hàm
- Ví dụ

```
void MyDelay(long Loops = 1000){
   for(int i = 0; i < Loops; ++i) ;
}
```

- `MyDelay();` // Loops có giá trị là 1000
- `MyDelay(5000);` // Loops có giá trị là 5000

Tham số ngầm định

- Nếu có prototype, các tham số có giá trị mặc định chỉ được cho trong prototype của hàm và không được lặp lại trong định nghĩa hàm.
- Một hàm có thể có nhiều tham số có giá trị mặc định. Các tham số có giá trị mặc định cần phải được nhóm lại vào các tham số cuối cùng (hoặc duy nhất) của một hàm.
- Khi gọi hàm có nhiều tham số có giá trị mặc định, chúng ta chỉ có thể bỏ bớt các tham số theo thứ tự từ phải sang trái và phải bỏ liên tiếp nhau
- 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);  //  ✔
```

Đa năng hóa hàm (Overloading)

- Cung cấp nhiều hơn một định nghĩa cho tên hàm đã cho trong cùng một phạm vi.
- Trình biên dịch sẽ lựa chọn phiên bản thích hợp của hàm hay toán tử dựa trên các tham số mà nó được gọi.

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);
```



```
#include <iostream>
#include <math.h>
Using namespace std;
int MyAbs(int X){
    return abs(X);
}
long MyAbs(long X){
    return labs(X);
}
double MyAbs(double X){
    return fabs(X);
}
int main(){
    int X = -7;
    long Y = 2000001;
    double Z = -35.678;
    cout << "Tri tuyet doi cua so nguyen "  << X <<" la " <<   MyAbs(X) <<
endl;
    cout << "Tri tuyet doi cua so nguyen "  << Y <<" la " << MyAbs(Y) << endl;
    cout << "Tri tuyet doi cua so thuc "  << Z <<" la " << MyAbs(Z) << endl;
    return 0;
}
```


Đa năng hoá toán tử

- Định nghĩa lại chức năng của các toán tử đã có sẵn
  - Thể hiện các phép toán một cách tự nhiên hơn
- Ví dụ: thực hiện các phép cộng, trừ số phức
  - Trong C: Cần phải xây dựng các hàm `AddSP(), TruSP()`
  - Không thể hiện được phép cộng và trừ cho các biểu thức như: `a=b+c-d+e+f-h-k`


```
#include <stdio.h>
struct  SP {
   double real;
   double img;
};
SP SetSP(double real, double img);
SP AddSP(SP C1,SP C2);
SP SubSP(SP C1,SP C2);
void DisplaySP(SP C);
int main(void) {
   SP C1,C2,C3,C4;
   C1 = SetSP(1.0,2.0);
   C2 = SetSP(-3.0,4.0);
   cout << "\nSo phuc thu nhat:";   DisplaySP(C1);
   cout << "\nSo phuc thu hai:";    DisplaySP(C2);
   C3 = AddSP(C1,C2);
   C4 = SubSP(C1,C2);
   cout << "\nTong hai so phuc nay:"; DisplaySP(C3);
   cout << "\nHieu hai so phuc nay:"; DisplaySP(C4);
   return 0;
}
```

```
SP SetSP(double real, double img) {
   SP tmp;
   tmp.real = real; tmp.img = img;
   return tmp;
}
SP AddSP(SP C1,SP C2) {
   SP tmp;
   tmp.real = C1.real + C2.real;
   tmp.img = C1.img + C2.img;
   return tmp;
}
SP SubSP(SP C1,SP C2) {
   SP tmp;
   tmp.real = C1.real - C2.real;
   tmp.img = C1.img - C2.img;
return tmp;
}
void DisplaySP(SP C) {
   cout << C.real << " i "  << C.img;
}
```


C++

- C++ cho phép chúng ta có thể định nghĩa lại chức năng của các toán tử đã có sẵn một cách tiện lợi và tự nhiên. Điều này gọi là đa năng hóa toán tử.
- 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).*


```
#include <iostream>
using namespace std;
typedef struct { double real; double img;} SP;
SP SetSP(double real, double img);
void DisplaySP(SP C);
SP operator + (SP C1, SP C2);
SP operator - (SP C1, SP C2);
int main() {
    SP C1,C2,C3,C4;
    C1 = SetSP(1.1,2.0);
    C2 = SetSP(-3.0,4.0);
    cout << "\nSo phuc thu  nhat:"; DisplaySP(C1);
    cout << "\nSo  phuc thu hai:";  DisplaySP(C2);
    C3 = C1 + C2;
    C4 = C1 - C2;
    cout << "\nTong hai so phuc nay:"; DisplaySP(C3);
    cout << "\nHieu hai so phuc nay:"; DisplaySP(C4);
    return 0;
}
```



```
SP SetSP(double real,double img) {
    SP tmp;
    tmp.real = real; tmp.img = img;
    return tmp;
}
SP operator + (SP C1,SP C2) {
    SP tmp;
    tmp.real = C1.real + C2.real;
    tmp.img = C1.img + C2.img;
    return tmp;
}
SP operator - (SP C1,SP C2) {
    SP tmp;
    tmp.real = C1.real - C2.real;
    tmp.img = C1.img - C2.img;
    return tmp;
}
void DisplaySP(SP C) {
    cout << "(" << C.real << "," << C.img << ")";
}
```

Các giới hạn của đa năng hóa toán tử

- Không thể định nghĩa các toán tử mới.
- 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**.

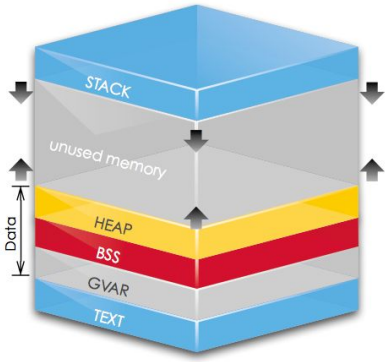
  ?: Toán tử điều kiện

  sizeof

  Các ký hiệu tiền xử lý.
- 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;
}
```



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

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

int main(){
    printf("%p\n", 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) ();
```

Ví dụ 2:

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

Ví dụ sử dụng con trỏ hàm

```
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;
}
```


Sắp xếp dãy số

```
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
   }
}
```


Sắp xếp dãy số

```
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 << "After sorted:  ";
   printArray(arr, length);
   return 0;
}
```


Sắp xếp dãy số

```
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, ascending);
   cout << "After sorted:  ";
   printArray(arr, length);
   return 0;
}
```

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;
}
```


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;
}
```


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:

```
auto maxval(auto x, auto 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;
}
```


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.

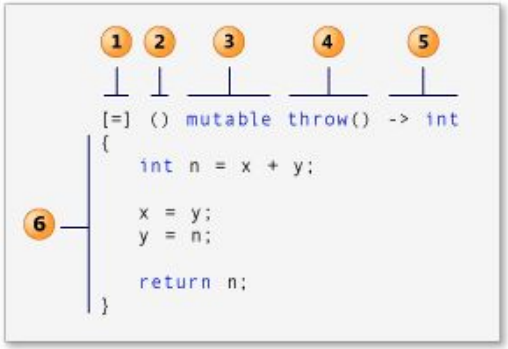
Hàm nặc danh - cú pháp lambda

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

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;
}
```


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;
   for_each(arr, n, [] (int a){
      a = a + 10;
      cout << a << " ";
   });
   return 0;
}
```

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)

- Một biểu thức lambda có thể khai báo thêm biến mới bên trong nó (từ chuẩn C++14 trở đi), và nó còn có thể truy cập, hoặc tham chiếu đến những biến bên trong khối lệnh chứa nó.
- Một lambda luôn bắt đầu với cặp ngoặc vuông [ ], và những biến cần được bắt giữ sẽ khai báo bên trong đó. Ký hiệu (&) là biến được truy cập bằng tham chiếu, bỏ ký hiệu (&) hoặc sử dụng cách khai báo [=] sẽ được hiểu là truy cập giá trị.
- Phần này có thể được bỏ trống, và được hiểu rằng lambda này không truy cập biến nào trong khối lệnh chứa nó.

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; };
```

Tuy nhiên, 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.

Hàm nặc danh - cú pháp lambda

Phần thân của một lambda

Phần thân của một lambda có thể:

- sử dụng những biến được bắt giữ trong mệnh đề bắt giữ.
- sử dụng các tham số.
- sử dụng các biến được khai báo bên trong struct/class chứa nó thông qua con trỏ this (OOP).
- sử dụng các biến toàn cục, biến static.

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

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