KỸ THUẬT LẬP TRÌNH

Chương 3

HÀM - Function

Liên hệ GV: thuong.bui@ut.edu.vn

3.1 Khái niệm về hàm

Hàm (function) trong lập trình là một đoạn mã được thiết kế để thực hiện một nhiệm vụ cụ thể nào đó trong chương trình.

Lợi ích khi sử dụng hàm trong chương trình:

- **Tái sử dụng mã:** Thay vì viết lại cùng một đoạn mã nhiều lần, ta chỉ cần tạo một hàm và gọi nó khi cần.
- **Tổ chức chương trình:** Hàm giúp chia nhỏ chương trình thành các phần nhỏ, dễ quản lý và hiểu hơn.
- **Truyền tham số:** Hàm có thể nhận dữ liệu truyền vào để thực hiện các tính toán khác nhau.
- **Trả về kết quả:** Hàm có thể trả về một giá trị sau khi thực hiện xong nhiệm vụ.

3.2 Quy trình hoạt động của hàm

- **Khai báo hàm**: Bạn định nghĩa hàm bằng cách khai báo kiểu trả về, tên hàm và danh sách các tham số (nếu có). Thân hàm chứa các câu lệnh mà hàm sẽ thực hiện.
- **Gọi hàm**: Khi bạn muốn sử dụng hàm, bạn gọi tên hàm kèm theo các đối số (giá trị thực tế truyền vào cho các tham số). Khi gọi hàm, chương trình sẽ thực hiện các bước sau:
  - Đánh giá các đối số.
  - Truyền giá trị của các đối số vào các tham số tương ứng trong hàm.
  - Chuyển điều khiển chương trình đến dòng lệnh đầu tiên của hàm.
- **Thực thi hàm**: Chương trình sẽ thực hiện tuần tự các lệnh bên trong hàm. Nếu hàm có giá trị trả về, kết quả sẽ được gán cho biến hoặc biểu thức ở vị trí gọi hàm.
- **Kết thúc hàm**: Khi gặp lệnh return hoặc khi kết thúc hàm, chương trình sẽ quay trở lại vị trí gọi hàm và tiếp tục thực hiện các lệnh còn lại.

3.3 Khai báo và định nghĩa hàm

Hàm trước khi sử dụng cần được khai báo và định nghĩa. Trong đó khai báo hàm (còn gọi là khai báo nguyên mẫu hàm) là 1 phần quan trọng để hạn chế lỗi phát sinh cho chương trình khi gọi đến hàm (thường ta chỉ cần khai báo nguyên mẫu hàm cho hàm A trong trường hợp có một hàm B nào đó được định nghĩa trước hàm A và trong B chứa lệnh gọi hàm A).

Khai báo nguyên mẫu hàm:

Một khai báo nguyên mẫu hàm sẽ cung cấp cho trình biên dịch mô tả về một hàm sẽ được định nghĩa ở một vị trí nào đó trong chương trình.

Vị trí khai báo nguyên mẫu hàm thường là đầu chương trình, ngay sau các lệnh tiền xử lý #include.

Cú pháp:

<Kiểu_trả_về> <tên_hàm> ([kiểu và danh_sách_tham_số]);

Ví dụ:

```
#include<iostream>
using namespace std;
int factorial(int n);//Khai báo nguyên mẫu hàm factorial
```

3.3 Khai báo và định nghĩa hàm

Định nghĩa hàm:

```
<Kiểu_trả_về> <Tên_hàm>(<Kiểu_tham_số> <Tên_tham_số>){
    //Các câu lệnh
    return <Biểu_thức>;//Nếu hàm có giá trị trả về
}
```

Kiểu_trả_về - Kiểu dữ liệu mà hàm sẽ trả về (ví dụ: int, float, bool, void). Nếu hàm không trả về giá trị nào, kiểu trả về là void.

Tên_hàm - được đặt tên theo quy tắc định danh.

Danh_sách_tham_số: Các giá trị được truyền vào hàm khi gọi hàm. Nếu có nhiều tham số thì chúng phải cách nhau bởi dấu phẩy và phải khai báo riêng biệt nhau.

Câu lệnh **return** dùng để kết thúc việc thực hiện của một hàm (nếu hàm có giá trị trả về), trả kết quả và chuyển quyền điều khiển về nơi gọi hàm. Giá trị kết quả này phải có kiểu phù hợp với **kiểu_trả_về** đã được khai báo ở dòng tiêu đề.

3.3 Khai báo và định nghĩa hàm

Sử dụng hàm (lệnh gọi hàm):

```
Tên_hàm(danh_sách_đối_số);
```

Ví dụ: Chương trình sau có dùng hàm factorial để tính giai thừa của một số nguyên dương:

```
#include<iostream>
using namespace std;
long factorial(int n){//Phần định nghĩa hàm
    if (n<0) return -1;
    long result = 1;
    for(int i=1; i<=n; i++){
        result *= i;
    }
    return result;
}
int main(){
    cout<<factorial(5);//In ra giao thừa của 5
    system("pause");
}
```

3.3 Khai báo và định nghĩa hàm

Ví dụ: Minh họa hàm không có giá trị trả về với tham số mặc định và các cách sử dụng

```
#include<iostream>
using namespace std;
void drawLine(int n=20){
    for(int i=0;i<n;i++){
        cout<<"=";
    }
    cout<<"\n";
}
int main(){
    int x = 15;
    drawLine(10);//Truyền đối số là hằng
    drawLine(x); //Truyền đối số là biến
    drawLine();//Không truyền đối số
    system("pause");
}
```

Kết quả chạy chương trình

```
==========
===============
====================
```

3.3 Khai báo và định nghĩa hàm

Định nghĩa hàm bằng #define:

```
#define <tên_hàm>(danh_sách_biến) (biểu_thức_trả_về)
```

Ví dụ: Định nghĩa hàm tính giá trị lớn nhất của 3 số bằng câu lệnh #define:

```
#include<iostream>
using namespace std;
#define MAX(a,b,c) (a>b?(a>c?a:c):(b>c?b:c))
int main(){
    cout<<MAX(3,5,9);//Kết quả in ra là 9
}
```

3.4 Quy trình thực thi hàm trong máy tính

Đẩy thông tin lên ngăn xếp (Stack): Khi một hàm được gọi, địa chỉ trở về, các tham số và các biến cục bộ của hàm đó sẽ được lần lượt đẩy lên đỉnh của ngăn xếp.

Phân bổ bộ nhớ cho các biến cục bộ được khai báo trong hàm.

Thực thi hàm: CPU sẽ thực hiện các lệnh trong hàm, sử dụng các giá trị đã được lưu trên ngăn xếp.

Trả về kết quả (nếu có): Khi hàm kết thúc, các giá trị trên đỉnh ngăn xếp (liên quan đến hàm vừa thực thi) sẽ được loại bỏ. Địa chỉ trở về được lấy ra và chương trình sẽ quay lại vị trí gọi hàm.

Hồi phục trạng thái trước khi gọi hàm: Lấy địa chỉ trở về ra khỏi ngăn xếp ----> Giải phóng bộ nhớ được phân bổ cho các biến cục bộ của hàm ---- > Quay trở lại vị trí gọi hàm và tiếp tục thực thi lệnh tiếp theo sau lời gọi hàm.

Ghi chú: *Ngăn xếp (stack) là một cấu trúc dữ liệu tuân theo nguyên tắc "vào sau ra trước" (LIFO - Last In, First Out). Hình dung như một chồng đĩa, đĩa nào đặt vào sau cùng sẽ được lấy ra trước. Trong lập trình, ngăn xếp được sử dụng rộng rãi để quản lý bộ nhớ trong quá trình thực thi chương trình, đặc biệt là khi gọi hàm.*

3.5 Biến và tham chiếu

Phạm vi của biến (scope):

- **Toàn cục (global scope):** Khai báo bên ngoài tất cả các hàm (kể cả hàm main) và có tác dụng lên toàn bộ chương trình.
- **Cục bộ (local scope):** Khai báo trong hàm hoặc khối { } và chỉ có tác dụng trong bản thân hàm hoặc khối đó (kể cả khối con nó). Biến cục bộ sẽ bị xóa khỏi bộ nhớ khi kết thúc khối khai báo nó.

```
#include<iostream>
using namespace std;
int main(){
    int i=2,j=2;
    for(int i=1; i<=10; i++, j++){
        cout<<i<<" ";
    }
    cout<<"\n------------------\n";
    cout<<"i = "<<i<<endl;
    cout<<"j = "<<j<<endl;
    system("pause");
}
```

Kết quả chạy chương trình

```
1 2 3 4 5 6 7 8 9 10
------------------
i = 2
j = 12
```

3.5 Biến và tham chiếu

Biến tĩnh và từ khóa static

- Biến tĩnh được tạo ra bên trong một khối lệnh (cục bộ) có khả năng lưu giữ giá trị của nó cho dù chương trình đã chạy ra bên ngoài khối lệnh chứa nó.
- Biến tĩnh chỉ cần được khai báo một lần duy nhất, và tiếp tục được duy trì sự tồn tại xuyên suốt cho đến khi chương trình kết thúc.

```
#include<iostream>
using namespace std;
void staticVariablesPrint() {
    static int s = 1;
    ++s;
    cout<<s << '\n';
}
int main() {
    staticVariablesPrint();
    staticVariablesPrint() ;
    staticVariablesPrint() ;
    system("pause");
}
```

Kết quả chạy chương trình

```
2
3
4
```

3.5 Biến và tham chiếu

Địa chỉ của biến (Address of a variable):

Khi bạn khai báo một biến trong chương trình, hệ thống sẽ cấp phát một vùng nhớ trong RAM để lưu trữ giá trị của biến đó. Địa chỉ của biến chính là vị trí bắt đầu của vùng nhớ này. Bạn có thể hình dung RAM như một dãy các ô nhớ, mỗi ô có một địa chỉ duy nhất (số hệ 16). Địa chỉ biến cho biết chính xác ô nhớ nơi giá trị của biến được lưu trữ.

Để lấy địa chỉ của biến ta dùng toán tử một ngôi & (address of operator) : **&tên_biến;**

```
#include<iostream>
using namespace std;
int main(){
    int x=5, y;
    cout<<"Dia chi cua bien x la: "<<&x<<endl;
    cout<<"Dia chi cua bien y la: "<<&y<<endl;
    system("pause");
}
```

Kết quả chạy chương trình

```
Dia chi cua bien x la: 0x61ff0c
Dia chi cua bien y la: 0x61ff08
```

3.5 Biến và tham chiếu

Tham chiếu (reference):

- Tham chiếu hay là bí danh (alias), tên gọi khác của một biến có sẵn. Về bản chất là 2 biến sử dụng chung 1 vùng nhớ.
- Biến tham chiếu và biến được tham chiếu phải cùng kiểu, duy nhất và được khởi tạo khi khai báo.
- Khai báo: **kiểu &biến_tham_chiếu = biến_được_tham_chiếu;**

```
#include<iostream>
using namespace std;
int main(){
    int x=5, &y = x;
    cout<<"x = "<<x<<endl;
    cout<<"y = "<<y<<endl;
    x = 8;
    cout<<"-----\n";
    cout<<"x = "<<x<<endl;
    cout<<"y = "<<y<<endl;
    system("pause");
}
```

Kết quả chạy chương trình

```
x = 5
y = 5
-----
x = 8
y = 8
```

3.6 Truyền đối số cho hàm

Truyền theo giá trị (call/pass by value):

- Đối số được truyền ở dạng giá trị (sao chép giá trị vào cho tham số thực sự).
- Có thể truyền hằng, biến, biểu thức nhưng hàm chỉ nhận giá trị.
- Được sử dụng khi không có nhu cầu thay đổi giá trị của đối số sau khi thực hiện hàm.

```
#include<iostream>
using namespace std;
void swap(int a, int b){
    int t=a; a=b; b=t;
}
int main(){
    int x=5, y=10;
    swap(7, 12);
    swap(2*x+y, y-2);
    swap(x, y);
    cout<<"x = "<<x<<endl<<"y = "<<y<<endl;
    system("pause");
}
```

Kết quả chạy chương trình

```
x = 5
y = 10
```

3.6 Truyền đối số cho hàm

Truyền theo tham chiếu (call/pass by reference):

- Khi truyền đối số chỉ là biến.
- Có nhu cầu thay đổi giá trị của đối số sau khi thực hiện hàm.
- Khi khai báo hàm thì các tham số phải có &.
- Chú ý: có thể viết int& x hoặc int & x; int &x.

```
#include<iostream>
using namespace std;
void swap(int& a, int & b){
    int t=a; a=b; b=t;
}
int main(){
    int x=5, y=10;
    swap(x, y);
    cout<<"x = "<<x<<endl<<"y = "<<y<<endl;
    system("pause");
}
```

Kết quả chạy chương trình

```
x = 10
y = 5
```

3.6 Truyền đối số cho hàm

Truyền theo địa chỉ (call/pass by address):

- Đối số phải là địa chỉ của biến.
- Khai báo hàm thì các tham số phải khai báo con trỏ.
- Có nhu cầu thay đổi giá trị của đối số sau khi thực hiện hàm.

```
#include<iostream>
using namespace std;
void swap(int *a, int *b){
    int t=*a; *a=*b; *b=t;
}
int main(){
    int x=5, y=10;
    swap(&x, &y);
    cout<<"x = "<<x<<endl<<"y = "<<y<<endl;
    system("pause");
}
```

Kết quả chạy chương trình

```
x = 10
y = 5
```

3.7 Hàm đệ quy (Recursion function)

Hàm đệ quy là hàm mà từ một điểm trong thân của nó có thể gọi lại chính nó. Hàm đệ quy thường dùng để giải quyết các bài toán đệ quy.

Cấu trúc tổng quát của hàm đệ quy:

```
if (trường hợp cơ sở){
    //giả định đã có cách giải
     Trình bày cách giải
} else {
    //trường hợp tổng quát
     Gọi lại hàm với tham đối "bé" hơn
}
```

Ví dụ: Viết hàm đệ quy tính s = 1 + 2 +3 +...+n

```
int tinhTong(int n){
    if (n==0) return 0;
    return tinhTong(n-1)+n;
}
```

3.7 Hàm đệ quy (Recursion function)

Ví dụ: Hàm đệ quy tính giai thừa

```
#include<iostream>
using namespace std;
long factorial(int n){
    if(n==0 || n==1) return 1;
    return n*factorial(n-1);
}
int main(){
    int n;
    INPUT:
    cout<<"Enter the number: "; cin>>n;
    if(n<0) {
        cout<<"Enter the positive number!\n";
        goto INPUT;
    } else {
        cout<<n<<"! = "<<factorial(n)<<endl;
    }
    system("pause");
}
```

Kết quả chạy chương trình

```
Enter the number: -4
Enter the positive number!
Enter the number: 5
5! = 120
```

3.7 Hàm đệ quy (Recursion function)

Các loại hàm đệ quy cơ bản:

Dựa trên số lượng lời gọi đệ quy và cấu trúc của các lời gọi đó, ta có thể phân loại hàm đệ quy thành một số loại chính sau:

❖ **Đệ quy Tuyến Tính (Linear Recursion)**: Chỉ có một lời gọi đệ quy trong hàm.

- Ưu điểm: Dễ hiểu và triển khai.
- Nhược điểm: Có thể kém hiệu quả với các bài toán lớn do đệ quy sâu.

❖ **Đệ quy Nhị Phân (Binary Recursion)**: Có hai lời gọi đệ quy trong hàm, thường chia bài toán thành hai nửa bằng nhau.Ví dụ: Tìm kiếm nhị phân, sắp xếp nhanh (Quicksort).

- Ưu điểm: Hiệu quả hơn đệ quy tuyến tính cho nhiều bài toán.
- Nhược điểm: Cần thiết kế cấu trúc dữ liệu phù hợp để tận dụng.

❖ **Đệ quy Đa Tuyến**: Có nhiều hơn hai lời gọi đệ quy.

❖ **Đệ quy Lồng**: Một hàm đệ quy gọi một hàm đệ quy khác.

❖ **Đệ quy Tương Hỗ**: Hai hoặc nhiều hàm gọi lẫn nhau.

3.7 Hàm đệ quy (Recursion function)

Ưu điểm của đệ quy:

- ❖ Dễ hiểu: Mã nguồn thường ngắn gọn và dễ đọc hơn so với các giải pháp vòng lặp.
- ❖ Tự nhiên: Phù hợp với các bài toán có cấu trúc đệ quy.
- ❖ Mạnh mẽ: Có thể giải quyết các bài toán phức tạp một cách hiệu quả.

Nhược điểm của đệ quy:

- ❖ Hiệu suất: Có thể kém hiệu quả do overhead của việc gọi hàm và quản lý stack.
- ❖ Tràn stack: Nếu đệ quy quá sâu, có thể gây ra lỗi tràn stack.
- ❖ Khó debug: Việc theo dõi quá trình thực thi của hàm đệ quy có thể phức tạp.

3.7 Hàm đệ quy (Recursion function)

Khi nào nên sử dụng đệ quy?

- ❖ Các bài toán có cấu trúc đệ quy rõ ràng.
- ❖ Các bài toán đòi hỏi tính ngắn gọn và dễ hiểu của mã nguồn.
- ❖ Các bài toán mà đệ quy mang lại hiệu suất tốt hơn so với các giải pháp khác.

Lưu ý khi dùng đệ quy:

- ❖ **Điều kiện dừng**: Mỗi hàm đệ quy cần có một điều kiện dừng để tránh vòng lặp vô hạn.
- ❖ **Độ phức tạp**: Cần phân tích độ phức tạp của thuật toán đệ quy để đảm bảo hiệu suất.
- ❖ **Tối ưu hóa:** Có thể sử dụng các kỹ thuật tối ưu hóa như đệ quy đuôi để cải thiện hiệu suất (đệ quy đuôi là trường hợp lời gọi đề quy là hành động cuối cùng được thực hiện trong hàm)

3.8 Ứng dụng của hàm

Hàm là một công cụ vô cùng hữu ích trong lập trình, giúp bạn viết code hiệu quả, dễ đọc và dễ bảo trì hơn.

Các ứng dụng của hàm:

- **Thư viện:** Hầu hết các ngôn ngữ lập trình đều có các thư viện hàm sẵn có, giúp bạn thực hiện các tác vụ phức tạp mà không cần viết lại mã.
- **Lập trình hướng đối tượng:** Hàm là một phần không thể thiếu trong lập trình hướng đối tượng, chúng được gọi là phương thức (method) của một đối tượng.

3.9 Một số hàm chuẩn trong C/C++

Tên hàm	Khai báo thư viện	Ý nghĩa
rand()	stdlib.h	Cho 1 giá trị ngẫu nhiên từ 0 đến 32767
random(x)	stdlib.h	Cho 1 giá trị ngẫu nhiên từ 0 đến x
pow(x,y)	math.h	Tính x mũ y
sqrt(x)	math.h	Tính căn bậc 2 của x
sin(x), cos(x), tan(x)	math.h	Tính sin, cosin, tang của góc x có số đo x radian
abs(a)	stdlid.h	Cho giá trị tuyệt đối của số nguyên a
labs(a)	stdlid.h	Cho giá trị tuyệt đối của số nguyên dài a
fabs(a)	stdlid.h	Cho giá trị tuyệt đối của số thực
exp(x)	math.h	Tính e mũ x
log(x)	math.h	Tính logarit cơ số e của x
log10(x)	math.h	Tính logarit cơ số 10 của x
ceil(x)	math.h	Phần nguyên nhỏ nhất không nhỏ hơn x
floor(x)	math.h	Phần nguyên lớn nhất không lớn hơn x

3.9 Một số hàm chuẩn trong C/C++

Tên hàm	Khai báo thư viện	Ý nghĩa
atof(str)	stdlib.h	Chuyển chuỗi str sang giá trị số thực
atoi(str)	stdlib.h	Chuyển chuỗi str sang số nguyên
itoa(x,str,y)	stdlib.h	Chuyển số nguyên x sang chuỗi str theo cơ số y
tolower(x)	ctype.h	Đổi chữ hoa sang chữ thường
toupper(x)	ctype.h	Đổi chữ thường sang chữ hoa
strcat(a,b)	string.h	Thêm chuỗi b vào sau chuỗi a
strcpy(a,b)	string.h	Sao chép chuỗi b vào a
strlwr(s)	string.h	Chuyển chuỗi s sang chữ thường
strupr(s)	string.h	Chuyễn chuỗi s sang chữ hoa
strlen(s)	string.h	Trả về độ dài chuỗi s
strcmp(s,t)	string.h	Trả về hiệu của 2 kí tự khác nhau đầu tiên trong chuỗi, nếu hàm trả về giá trị 0 thì s bằng t.

BÀI TẬP CHƯƠNG 3

Bài 1. Viết hàm kiểu bool để kiểm tra xem một số nguyên dương có phải là số nguyên tố hay không?

Bài 2. Viết hàm đếm số chữ số của một số nguyên bất kỳ.

Bài 3. Viết hàm hiển thị các ước của một số nguyên dương bất kỳ lên màn hình.

Bài 4. Viết hàm tính tổng các chữ số của số nguyên dương bất kỳ.

Bài 5. Viết hàm trả về phần tử thứ n của dãy Fibonacci (*Dãy Fibonacci là một dãy số vô hạn các số tự nhiên bắt đầu bằng hai phần tử 0 và 1, các phần tử sau đó được xác định bằng cách lấy tổng của hai phần tử liền trước của nó*).

Bài 6. Viết hàm tìm UCLN của 2 số nguyên dương.

Bài 7. Viết hàm tính BCNN của 2 số nguyên dương.

Bài 8. Viết hàm kiểm tra xem một số có phải là số chính phương hay không?

Bài 9. Viết hàm chuyển đổi 1 ký tự chữ cái thường thành chữ cái in hoa, nếu không phải chữ cái thì giữ nguyên.

Bài 10. Viết hàm trả về số đảo ngược của 1 số nguyên dương (Ghi chú: số đảo ngược của 123 là 321).