

Bài 2

C/C++ NÂNG CAO

Trịnh Thành Trung
trungtt@soict.hust.edu.vn

THỨ TỰ ƯU TIÊN CÁC PHÉP TOÁN

Mức	Các toán tử	Trật tự kết hợp
1	() [] . -> ++ (hậu tố) -- hậu tố	----->
2	! ~ ++ (tiền tố) -- (tiền tố) - *	<-----
	& sizeof	
3	* / %	----->
4	+ -	----->
5	<< >>	----->
6	< <= > >=	----->
7	== !=	----->
8	&	----->
9	^	----->
10		----->
11	&&	----->
12		----->
13	?:	<-----
14	= += -=	<-----



1

MẢNG

Mảng

- Là một dãy hữu hạn các phần tử liên tiếp có cùng kiểu và tên
- Có thể là 1 hay nhiều chiều, C không giới hạn số chiều của mảng
- Khai báo theo syntax sau :
 - `DataType ArrayName[size];`
- hoặc
 - `DataType ArrayName[size 1][size 2]...[size n];`

Mảng

- Khởi tạo giá trị cho mảng theo 2 cách

- C1. Khi khai báo :

```
float y[5] = { 3.2, 1.2, 4.5, 6.0, 3.6 }  
int m[6][2] = { { 1, 1 }, { 1, 2 }, { 2, 1 }, { 2, 2 }, {  
3, 1 }, { 3, 2 } };  
char s1[6] = { 'H', 'a', 'n', 'o', 'i', '\0' }; //hoặc  
char s1[6] = "Hanoi";  
char s1[] = "Dai hoc Bach Khoa Hanoi"; //L = 24  
int m[][] = { { 1, 2, 3 }, { 4, 5, 6 } };
```

- C2. Khai báo rồi gán giá trị cho từng phần tử của mảng.

- Ví dụ : `int m[4];`

```
m[0] = 1; m[1] = 2; m[2] = 3; m[3] = 4;
```



2

CON TRỎ

Con trỏ

- Khái niệm : Giá trị các biến được lưu trữ trong bộ nhớ MT, có thể truy cập tới các giá trị đó qua tên biến, đồng thời cũng có thể qua địa chỉ của chúng trong bộ nhớ.
- Con trỏ thực chất là 1 biến mà nội dung của nó là địa chỉ của 1 đối tượng khác (Biến, hàm, nhưng không phải 1 hằng số).
- Có nhiều kiểu biến với các kích thước khác nhau, nên có nhiều kiểu con trỏ. Con trỏ int để trỏ tới biến hay hàm kiểu int.
- Việc sử dụng con trỏ cho phép ta truy nhập tới 1 đối tượng gián tiếp qua địa chỉ của nó.
- Trong C, con trỏ là một công cụ rất mạnh, linh hoạt

Con trỏ

- Khai báo con trỏ :
- Syntax : `dataType * PointerName;`
 └─ Chỉ rằng đây là con trỏ
- Sau khi khai báo, ta được con trỏ NULL, vì nó chưa trỏ tới 1 đối tượng nào.
- Để sử dụng con trỏ, ta dùng toán tử lấy địa chỉ &
 - `PointerName = & VarName`Ví dụ :
 - `int a; int *p; a=10;`
 - `p= &a;`
- Để lấy nội dung biến do con trỏ trỏ tới, ta dùng toán tử lấy nội dung *
 - `* PointerName`

Ví dụ :

```
int i,j,*p;  
i= 5;  p= & i;  
j= *p; *p= j+2;
```

100		i
102		j
104		p

Gán i=5

100	5	i
102		j
104		p

gán p = & i

100	5	i
102		j
104	100	p

gán j = *p

100	5	i
102	5	j
104	100	p

*p = j+2

100	7	i
102	5	j
104	100	p

Chú ý

- Một con trỏ chỉ có thể trỏ tới 1 đối tượng cùng kiểu
- Toán tử 1 ngôi * và & có độ ưu tiên cao hơn các toán tử số học
- Ta có thể viết *p cho mọi nơi có đối tượng mà nó trỏ tới xuất hiện

```
int x = 5, *p; p = & x; =>  
x=x+10; ~ *p = *p+10;
```

- Ta cũng có thể gán nội dung 2 con trỏ cho nhau: khi đó cả hai con trỏ cùng trỏ tới 1 đối tượng

```
int x=10, *p, *q;  
p = &x; q = p;  
=> p và q cùng trỏ tới x
```

Các phép toán trên con trỏ

- Một biến trỏ có thể cộng hoặc trừ với 1 số nguyên n để cho kết quả là 1 con trỏ cùng kiểu, là địa chỉ mới trỏ tới 1 đối tượng khác nằm cách đối tượng đang bị trỏ n phần tử
- Phép trừ giữa 2 con trỏ cho ta khoảng cách (số phần tử) giữa 2 con trỏ
- Không có phép cộng, nhân, chia 2 con trỏ
- Có thể dùng các phép gán, so sánh các con trỏ, nhưng cần chú ý đến sự tương thích về kiểu.
- Ví dụ : `char *pchar; short *pshort; long *plong;`
- sau khi xác lập địa chỉ cho các con trỏ, nếu :
- `pchar ++; pshort ++; plong ++;` và các địa chỉ ban đầu tương ứng của 3 con trỏ là 100, 200 và 300, thì kết quả ta có các giá trị tương ứng là : 101, 202 và 304 tương ứng

- Nếu viết tiếp :

`plong += 5; => plong = 324`

`pchar -=10; => pchar = 91`

`pshort +=5; => pshort = 212`

- Chú ý : ++ và -- có độ ưu tiên cao hơn * nhưng `*p++ ~ *(p++)` tức là tăng địa chỉ mà nó trỏ tới chứ không phải tăng giá trị mà nó chứa.

- `*p++ = *q++` sẽ tương đương :

`*p = *q;`

`p=p+1;`

`q=q+1;`

Vì cả 2 phép tăng đều diễn ra sau khi phép gán được thực hiện

- `++*p = ++*q; //???`

=> Cần dùng dấu () để tránh nhầm lẫn

Con trỏ void*

- Là con trỏ không định kiểu (void *). Nó có thể trỏ tới bất kì một loại biến nào.
- Thực chất một con trỏ void chỉ chứa một địa chỉ bộ nhớ mà không biết rằng tại địa chỉ đó có đối tượng kiểu dữ liệu gì. => không thể truy cập nội dung của một đối tượng thông qua con trỏ void.
- Để truy cập được đối tượng thì trước hết phải ép kiểu biến trỏ void thành biến trỏ có định kiểu của kiểu đối tượng

Con trỏ void*

```
float x; int y;  
void *p; // khai báo con trỏ void  
p = &x; // p chứa địa chỉ số thực x  
*p = 2.5; // báo lỗi vì p là con trỏ void  
/* cần phải ép kiểu con trỏ void trước khi  
truy cập đối tượng qua con trỏ */  
*((float*)p) = 2.5; // x = 2.5  
p = &y; // p chứa địa chỉ số nguyên y  
*((int*)p) = 2; // y = 2
```

```
(float) *p=2.5;  
*p= (float *) 2.5;  
*(float)p =2.5;  
(float *) p =2.5;  
(float *) *p=2.5;  
*((float *) p )=2.5;
```

Con trỏ và mảng

- Giả sử ta có : `int a[30];` thì `&a[0]` là địa chỉ phần tử đầu tiên của mảng đó, đồng thời là địa chỉ của mảng.
- Trong C, tên của mảng chính là **1 hằng địa chỉ** = địa chỉ của phần tử đầu tiên của mảng
 - `a = &a[0];`
 - `a+i = &a[i];`

Con trỏ và mảng

- Tuy vậy cần chú ý rằng `a` là 1 hằng => không thể dùng nó trong câu lệnh gán hay toán tử tăng, giảm như `a++`;
- Xét con trỏ: `int *pa;`
`pa = & a[0];`
=> `pa` trỏ vào phần tử thứ nhất của mảng và :
 - `pa + 1` sẽ trỏ vào phần tử thứ 2 của mảng
 - `*(pa+i)` sẽ là nội dung của `a[i]`

Con trỏ xâu

- Ta có : `char tinhthanh[30] = "Da Lat";`
- Tương đương :
`char *tinhthanh;`
`tinhthanh = "Da lat";`
- Hoặc : `char *tinhthanh = "Da lat";`
- Ngoài ra các thao tác trên xâu cũng tương tự như trên mảng
`*(tinhthanh+3) = "l"`
- Chú ý : với xâu thường thì không thể gán trực tiếp như dòng thứ 3

Mảng các con trỏ

- Con trỏ cũng là một loại dữ liệu nên ta có thể tạo một mảng các phần tử là con trỏ theo dạng thức.
 <kiểu> *<mảng con trỏ>[<số phần tử>];
- Ví dụ: `char *ds[10];`
- ds là 1 mảng gồm 10 phần tử, mỗi phần tử là 1 con trỏ kiểu char, được dùng để lưu trữ được của 10 xâu ký tự nào đó
- Cũng có thể khởi tạo trực tiếp các giá trị khi khai báo
 `char * ma[10] = {"mot", "hai", "ba"...};`

Mảng các con trỏ

- Chú ý: cần phân biệt mảng con trỏ và mảng nhiều chiều. Mảng nhiều chiều là mảng thực sự được khai báo và có đủ vùng nhớ dành sẵn cho các phần tử. Mảng con trỏ chỉ dành không gian nhớ cho các biến trỏ (chứa địa chỉ). Khi khởi tạo hay gán giá trị: cần thêm bộ nhớ cho các phần tử sử dụng => tốn nhiều hơn

- Một ưu điểm khác của mảng trỏ là ta có thể hoán chuyển các đối tượng (mảng con, cấu trúc..) được trỏ bởi con trỏ này bằng cách **hoán chuyển các con trỏ**
- Ưu điểm tiếp theo là việc truyền tham số trong hàm
- Ví dụ: Vào danh sách lớp theo họ và tên, sau đó sắp xếp để in ra theo thứ tự ABC.

```
#include <stdio.h>
#include <string.h>
#define MAXHS 50
#define MAXLEN 30
```

```
int main () {
    int i, j, count = 0;    char ds[MAXHS][MAXLEN];
    char *ptr[MAXHS], *tmp;
    while ( count < MAXHS) {
        printf(" Vao hoc sinh thu : %d  ",count+1);
        gets(ds[count]);
        if (strlen(ds[count] == 0) break;
        ptr[count] = ds +count;
        ++count;
    }
    for ( i=0;i<count-1;i++)
        for ( j =i+1;j < count; j++)
            if (strcmp(ptr[i],ptr[j])>0) {
                tmp=ptr[i]; ptr[i] = ptr[j]; ptr[j] = tmp;
            }
    for (i=0;i<count; i++)
        printf("\n %d :  %s", i+1,ptr[i]);
}
```

Con trỏ trỏ tới con trỏ

- Bản thân con trỏ cũng là 1 biến, vì vậy nó cũng có địa chỉ và có thể dùng 1 con trỏ khác để trỏ tới địa chỉ đó.
 <Kiểu DL> **<Tên biến trỏ>;
- Ví dụ : `int x = 12;`
 `int *p1 = &x;`
 `int **p2 = &p1;`
- Có thể mô tả 1 mảng 2 chiều qua con trỏ của con trỏ theo công thức :
 `M[i][k] = *(* (M+i) + k)`
- Với
 - `M+i` là địa chỉ của phần tử thứ `i` của mảng
 - `*(M+i)` cho nội dung phần tử trên
 - `*(M+i)+k` là địa chỉ phần tử `[i][k]`

- Ví dụ : in ra 1 ma trận vuông và cộng mỗi phần tử của ma trận với 10

```
#include <stdio.h>
```

```
#define hang 3
```

```
#define cot 3
```

```
int main() {
```

```
    int mt[hang][cot] = {{7,8,9},{10,13,15},{2,7,8}};
```

```
    int i,j;
```

```
    for (i=0;i<hang;i++) {
```

```
        for (j=0;j<cot;j++)    printf(" %d ",mt[i][j]);
```

```
        printf("\n");
```

```
    }
```

```
    for (i=0; i<hang;i++) {
```

```
        for (j=0;j<cot;j++) {
```

```
            (*(mt+i)+j)=(*(mt+i)+j) +10;
```

```
            printf(" %d ", (*(mt+i)+j);    }
```

```
            printf("\n"); }
```

```
}
```


3

QUẢN LÝ BỘ NHỚ

Bộ nhớ động

- Cho đến lúc này ta chỉ dùng bộ nhớ tĩnh: tức là khai báo mảng, biến và các đối tượng khác một cách tường minh trước khi thực hiện chương trình.
- Trong thực tế nhiều khi ta không thể xác định trước được kích thước bộ nhớ cần thiết để làm việc, và phải trả giá bằng việc khai báo dự trữ quá lớn
- Nhiều đối tượng có kích thước thay đổi linh hoạt

Bộ nhớ động

- Việc dùng bộ nhớ động cho phép xác định bộ nhớ cần thiết trong quá trình thực hiện của chương trình, đồng thời giải phóng chúng khi không còn cần đến để dùng bộ nhớ cho việc khác
- Trong C ta dùng các hàm **malloc**, **calloc**, **realloc** và **free** để xin cấp phát, tái cấp phát và giải phóng bộ nhớ. Trong C++ ta chỉ dùng **new** và **delete**

Xin cấp phát bộ nhớ : new va delete

- Để xin cấp phát bộ nhớ ta dùng :
<biên trở> = new <kiểu dữ liệu>;
hoặc <biến trở> = new <kiểu dữ liệu>[Số phần tử];
dòng trên xin cấp phát một vùng nhớ cho một biến đơn,
còn dòng dưới : cho một mảng các phần tử có cùng
kiểu với kiểu dữ liệu.
- Bộ nhớ động được quản lý bởi hệ điều hành, và với môi trường đa nhiệm (multitask interface) thì bộ nhớ này sẽ được chia sẻ giữa hàng loạt các ứng dụng, vì vậy có thể không đủ bộ nhớ. Khi đó toán tử new sẽ trả về con trỏ NULL.
- Ví dụ:

```
int *pds;  
pds = new int [200];  
if (pds == NULL) { // thông báo lỗi và xử lý
```

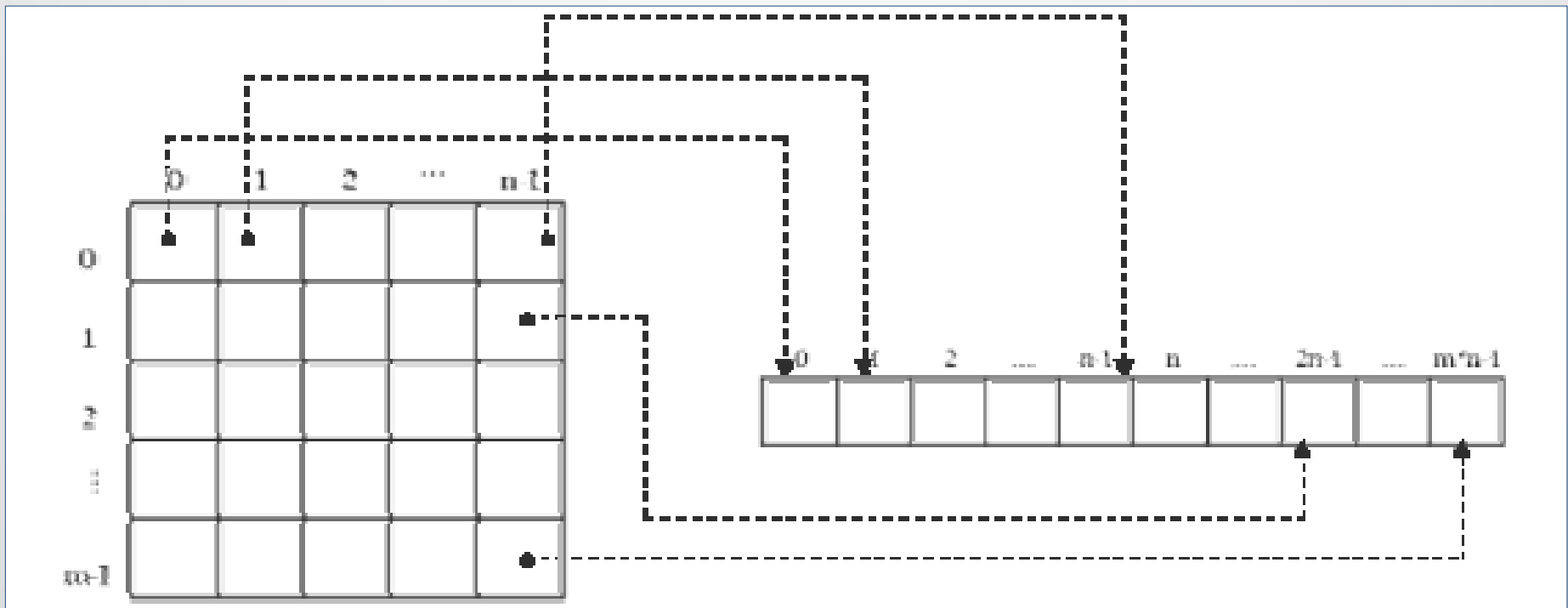
Giải phóng bộ nhớ

- `delete ptr;` // xóa 1 biến đơn
- `delete [] ptr;` // xóa 1 biến mảng
- Ví dụ :

```
#include <stdio.h>
int main() {
    int i,n; long total=100,x,*ds;
    printf(" Vao so ptu "); scanf("%d",&n);
    ds = new long [n];
    if (ds==NULL) exit(1);
    for (i=0;i<n;i++){
        printf("\n Vao so thu  %d : ", i+1 );
        scanf("%d",&ds[i] );
    }
    printf("Danh sach cac so : \n");
    for (i=0;i<n;i++)    printf("%d",ds[i]);
    delete []ds;
    return 0;
}
```

Dùng bộ nhớ động cho mảng 2 chiều

- Ta có thể coi một mảng 2 chiều là 1 mảng 1 chiều như hình sau:



Gọi X là mảng hai chiều có kích thước m dòng và n cột.
A là mảng một chiều tương ứng ,thì $X[i][j] = A[i \cdot n + j]$

Dùng con trỏ của con trỏ cho ma trận

- Với mảng số nguyên 2 chiều có kích thước là $R * C$ ta khai báo như sau :

```
int **mt;  
mt = new int *[R];  
int *temp = new int [R*C];  
for (i=0; i< R; ++i) {  
    mt[i] = temp;  
    temp += C;  
}
```

Để giải phóng:

```
delete [] mt[0];  
delete [] mt;
```

```
// Nhap R,C;  
float ** M = new float *[R];  
for(i=0; i<R;i++)  
    M[i] = new float[C];  
//dung M[i][j] nhu binh thuong
```

```
// giai phong  
for(i=0; i<R;i++)  
    delete []M[i];    // giai phong cac hang  
delete []M;
```


CT cộng hai ma trận với mỗi ma trận được cấp phát động

```
#include <iostream.h>
#include <conio.h>
int main()
{
    int M,N;
    int *A = NULL,
        *B = NULL,
        *C = NULL;
    clrscr();
    cout<<"Nhap so dong cua ma
tran:"; cin>>M;
    cout<<"Nhap so cot cua ma
tran:"; cin>>N;
```

```
//Cấp phát vùng nhớ cho
ma trận A
if(!AllocMatrix(&A,M,N))
{
    cout<<"Khong con du bo
nho!"<<endl;
    return 1;
}
//Cấp phát vùng nhớ cho
ma trận B
if(!AllocMatrix(&B,M,N))
{
    cout<<"Khong con du
bo nho!"<<endl;
    FreeMatrix(A); //Giải
phóng vùng nhớ A
    return 1;
}
}
```

```

//Cấp phát vùng nhớ cho ma
trận C
if (!AllocMatrix(&C,M,N))
{
    cout<<"Khong con du
bo nho!"<<endl;
    FreeMatrix(A);//Giải
phóng vùng nhớ A
    FreeMatrix(B);//Giải
phóng vùng nhớ B
    return 1;
}
cout<<"Nhap ma tran thu
1"<<endl;
InputMatrix(A,M,N,'A');
cout<<"Nhap ma tran thu
2"<<endl;
InputMatrix(B,M,N,'B');
clrscr();

```

```

cout<<"Ma tran thu
1"<<endl;
    DisplayMatrix(A,M,N);
    cout<<"Ma tran thu
2"<<endl;
    DisplayMatrix(B,M,N);
    AddMatrix(A,B,C,M,N);
    cout<<"Tong hai ma
tran"<<endl;
    DisplayMatrix(C,M,N);
    FreeMatrix(A);//Giải
phóng vùng nhớ A
    FreeMatrix(B);//Giải
phóng vùng nhớ B
    FreeMatrix(C);//Giải
phóng vùng nhớ C
    return 0;
}

```

```

//Cộng hai ma trận
void AddMatrix(int *A,int
*B,int*C,int M,int N)
{
    for(int
I=0;I<M*N;++I)
        C[I] = A[I] + B[I];
}
//Cấp phát vùng nhớ cho ma
trận
int AllocMatrix(int
**A,int M,int N)    // chú ý
: **
{
    *A = new int [M*N];
    if (*A == NULL)
        return 0;
    return 1;
}

```

```

//Giải phóng vùng nhớ
void FreeMatrix(int *A)
{
    if (A!=NULL)
        delete [] A;
}

```

```

//Nhập các giá trị của ma trận
void InputMatrix(int *A,int M,int N,char Symbol)
{
    for(int I=0;I<M;++I)
    for(int J=0;J<N;++J)
    {
        cout<<Symbol<<"["<<I<<"]["<<J<<"]=";
        cin>>A[I*N+J];
    }
}

//Hiển thị ma trận
void DisplayMatrix(int *A,int M,int N)
{
    for(int I=0;I<M;++I)
    {
        for(int J=0;J<N;++J)
        {
            out.width(7);//canh le phai voi chieu
dai 7 ky tu
            cout<<A[I*N+J];
        }
        cout<<endl;
    }
}

```

4

HÀM VÀ THAM SỐ

Hàm và truyền tham số

- Một chương trình C được cấu trúc thông qua các **hàm**. Mỗi hàm là một modul nhỏ trong chương trình, có thể được gọi nhiều lần.
- C **chỉ có hàm**, có thể coi thủ tục là một hàm không có dữ liệu trả về. C cũng **không có khái niệm hàm con**, tất cả các hàm kể cả hàm chính (main) **đều có cùng một cấp duy nhất** (cấu trúc hàm đồng cấp). Một hàm có thể gọi một hàm khác bất kì của chương trình.
- syntax :
[<kiểu trả về>] <tên hàm>([<danh sách tham số>])
{
 <thân hàm>
}

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++ thì khác: 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.

Phép tham chiếu

- Trong C, hàm nhận tham số là con trỏ đòi hỏi chúng ta phải thận trọng khi gọi hàm. Chúng ta cần viết hàm hoán đổi giá trị giữa hai số như sau:

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

- Để hoán đổi giá trị hai biến A và B thì chúng ta gọi hàm như sau:

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

- Rõ ràng cách viết này không được thuận tiện lắm

Dùng tham chiếu với C++

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

- Chúng ta gọi hàm như sau :
 `Swap(A, B);`
- Với cách gọi hàm này, C++ truyền tham chiếu của *A* và *B* làm tham số cho hàm *Swap()*.

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

```
#include <iostream.h>
int X = 4;
int & MyFunc()
{
    return X;
}

int main()
{
    cout<<"X="<<X<<endl;
    cout<<"X="<<MyFunc()<<endl;
    MyFunc() = 20; //Nghĩa là X = 20
    cout<<"X="<<X<<endl;
    return 0;
}
```

- Lưu ý : liên hệ với các phép toán ++, -- sau này

Hàm với tham số ngầm định

- Một trong các đặc tính nổi bật nhất của C++ là khả năng định nghĩa các giá trị tham số mặc định cho các hàm. Bình thường khi gọi một hàm, chúng ta cần gởi một giá trị cho mỗi tham số đã được định nghĩa trong hàm đó
- Ví dụ

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

Hàm với tham số ngầm định

- Mỗi khi hàm MyDelay() được gọi chúng ta phải gửi cho nó một giá trị cho tham số Loops. Tuy nhiên, trong nhiều trường hợp chúng ta có thể nhận thấy rằng chúng ta luôn luôn gọi hàm MyDelay() với cùng một giá trị Loops nào đó ?.

```
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

Chú ý:

- 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 (Vì trình biên dịch sẽ dùng các thông tin trong prototype chứ không phải trong định nghĩa hàm để tạo một lệnh gọi).
- 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

Chú ý:

- Chẳng hạn chúng ta có đoạn chương trình như sau:
 - `int MyFunc(int a= 1, int b, int c = 3, int d = 4); //prototype sai!!!`
 - `int MyFunc(int a, int b = 2, int c = 3, int d = 4); //prototype đúng`



5

ĐA NĂNG HÓA (OVERLOADING)

Phép đa năng hóa (Overloading)

- Với ngôn ngữ C++, chúng ta có thể đa năng hóa các hàm và các toán tử (operator). Đa năng hóa là phương pháp 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.

Đa năng hóa các hàm (Functions overloading)

- Trong C ta phải dùng 3 hàm để tính trị tuyệt đối:

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

- C++ cho phép chúng ta tạo ra các hàm khác nhau có cùng một tên.

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

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

- Trong NNLT C, với 1 kiểu dữ liệu mới, chúng ta thực hiện các thao tác liên quan đến kiểu dữ liệu đó thường thông qua các hàm => không thoải mái.
- Ví dụ: cài đặt các phép toán cộng và trừ số phức

```

#include <stdio.h> /* Dinh nghia so phuc */
struct SP {
    double THUC;
    double AO;
};

SP SetSP(double R,double I);
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 R,double I) {  
    SP Tmp;  
    Tmp.THUC = R; Tmp.AO = I;  
    return Tmp; }
```

```
SP AddSP(SP C1,SP C2) {  
    SP Tmp;  
    Tmp.THUC = C1.THUC+C2.THUC;  
    Tmp.AO = C1.AO+C2.AO;  
    return Tmp; }
```

```
SP SubSP(SP C1,SP C2) {  
    SP Tmp;  
    Tmp.THUC = C1.THUC-C2.THUC;  
    Tmp.AO = C1.AO-C2.AO;  
    return Tmp; }
```

```
void DisplaySP(SP C) {  
    cout <<C.THUC <<' i ' <<C.AO; }
```

C++

- Trong ví dụ trên, ta dùng hàm để cài đặt các phép toán cộng và trừ hai số phức => phức tạp, không thoải mái khi sử dụng, vì thực chất thao tác cộng và trừ là các toán tử chứ không phải là hàm.
 - Ví dụ : viết biểu thức : $a=b+c-d+e+f-h-k$;
- 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 hơn rất nhiều. Điều này gọi là đa năng hóa toán tử.

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).

```
#include <iostream.h> //Dinh nghia so phuc
typedef struct SP { double THUC; double AO; } ;
SP SetSP(double R,double I);
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; }
```



```
SetSP(double R,double I) {  
    SP Tmp;  
    Tmp.THUC = R; Tmp.AO = I; return Tmp; }
```

```
//Cong hai so phuc  
SP operator + (SP C1,SP C2) { SP Tmp;  
    Tmp.THUC = C1.THUC+C2.THUC;  
    Tmp.AO = C1.AO+C2.AO; return Tmp; }
```

```
//Tru hai so phuc  
    SP operator - (SP C1,SP C2) { SP Tmp;  
    Tmp.THUC = C1.THUC-C2.THUC;  
    Tmp.AO = C1.AO-C2.AO; return Tmp; }
```

```
//Hien thi so phuc  
void DisplaySP(SP C) {  
    printf("\n %f , %f ",C.THUC); }
```

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 :
 1. `::` Toán tử định phạm vi.
 2. `.*` Truy cập đến con trỏ là trường của struct hay class.
 3. `.` Truy cập đến trường của struct hay class.
 4. `?:` Toán tử điều kiện
 5. **`sizeof`**
 6. Các ký hiệu tiền xử lý

Các giới hạn của đa năng hóa toán tử

- 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.

Bài tập 2

- Xây dựng cấu trúc thời gian Time và đa năng hóa các toán tử cần thiết
- Sử dụng cấp phát động, xây dựng cấu trúc matrix và đa năng hóa các toán tử $+$, $*$ ma trận
- Hãy sử dụng các toán tử trên và kiểm thử trên các dạng biểu thức khác nhau để phát hiện các trường hợp ngoại lệ và đề ra giải pháp
- Sử dụng class thay cho struct và chỉ ra các ưu điểm nếu có !