

Chương II

Vài kiến thức nâng cao về C,
C++

http://www.2shared.com/file/FwpHXtJ7/KTLT_2014.html

http://www.2shared.com/file/DFdQegtG/KTLT_2014h.html

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	= += -=	<-----

a=b=c=d=10; ???

a=b=c=d+10; ???

?: : bt1?bt2:bt3;

Ds>1ty thuong 10 %ds

>500tr 7 %

> 300 tr 4%

0

Chi dung 4 dau ;

- ++/-- Tien to : tg truoc - gan sau, thay doi gt cua bien roi tham gia vao bt

int a=10,b;

b= ++a +5;

~ a=a+1; => a=11

b = a + 5; => b=16

b = ++a + a;

=>b=22

=>Tien to : Thay doi va tra ve chinh cai bien da thay doi

- ++/-- Hậu to : gán trước - tg sau, tham gia vào biểu thức rồi mới thay đổi gt của biến

```
int a=10,b;
```

```
b= a++ +5;
```

~ $b=a+5; \Rightarrow b=15$

$a=a+1; \Rightarrow a=11;$

$b= a++ +a;$ Kq ? Như sách : $b=20$

$B=21$?

Thay đổi biến, nhưng trả về **gia trị** của biến trước khi thay đổi

Toán tử ?

- Là 1 hàm, bao giờ cũng trả về 1 giá trị
- Có thứ tự thực hiện
- Có trật tự thực hiện ->, <-
- Có số toán hạng (số tham số)
- Mất thời gian thực hiện
- () ?, ->
- Struct s {int t1; ...} St,*Pst;
- St.t1 =100;
- Pst=&St;
- *Pst.t1=100: tương đương St.t1 =100: ??

- $Pst = \&St;$
- $*Pst.t1 = 100;$ tương đương $St.t1 = 100;$??
- $(*Pst).t1 = 100; \equiv St.t1 = 100;$
- $Pst \rightarrow t1 = 100;$ (1 phép toán truy cập)
- Tiền tố : Thay đổi trước, Gán sau (tăng /giảm biến rồi tham gia vào biểu thức)
- Hậu tố : gán trước, tăng sau (tham gia vào BT rồi mới thay đổi)

Sách dạy

- Int a=10,b;
b = ++a+5;
a=a+1; => a=11
b=a+5; => b=16

- b= ++a + a;
a=a+1; a=11
b=a+a; b = 22

- Int a=10,b;
b = a++ + 5;
b=a+5; => b=15
a=a+1; => a=11

b= a++ + a;
b = 20 (sách)
b=21 (VC...) ?

$++/--$

- Tiền tố : thay đổi biến và trả về biến đã thay đổi, (có thể đứng ở trái phép =)
- Hậu tố : thay đổi biến, nhưng trả về giá trị của biến trước khi thay đổi
- Idiom ???
- $b=x + x++$ thì $b=21$ còn $b=2*x + x++$ thì $b=30$???
- Tiền tố : ok, Hậu tố : cảnh giác và tránh

Với $x = 10$

devc

Expression	Visual++	g++	Turbo C++	
$b = x + x++$	20	21	20	21
$b = x++ + x$	20	21	20	21
$b = 2 * x + x++$	30	30	30	30
$b = ++x + x++$	22	23	22	22
$b = x++ + ++x$	22	22	22	22
$b = x++$	10	10	10	10
$b = x++ + 3$	13	13	13	13
$b = x++ + x++$	20	21	20	21
$b = x++ + (++x + 2 * x)$	44	46	44	46
$b = ++x + ++x$	24	24	24	24
$b = ++x + ++x + ++x$	39	37	39	37
$b = ++x + ++x * ++x$	182	182	182	182
$++b = ++b + b++$	7	7	7	
$++b = ++b + ++b + ++b$	15	13	15	

Datatype *p

$p+n \Rightarrow$ noidungcua $p + n * \text{sizeof}(p)$

A=b=c=d=10;

A=b=c=d+1;

```
#include <conio.h>
#include <stdio.h>
int main() {
float ds;
printf("\n Vao doanh so ");
scanf("%f",&ds);
printf("\n Tien thuong = %10.2f (Trieu) ",
ds * (ds>1000?0.1:ds>500?0.07:
ds>300?0.04:0));
}
```

2.1 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];

Or **DataType** **ArrayName**
[Size1][Size2]...[Sizen];

- 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'}; hoac

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;

```
int m[10] = {1};
```

```
int m[10] = {1, 2};
```

```
int m[][] = {{1, 2}, {1, 1, 1, 1, 1}}; 4, 7
```

```
char s[6] = {'H', 'a', 'n', 'o', 'i'};
```

```
char s2[] = "Hanoi";
```


s?s2 :

H	a	n	o	i	\0
---	---	---	---	---	----

H	a	n	o	i	??
---	---	---	---	---	----

2.2 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

- 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;
```

```
j= *p;
```

```
p= & i;
```

```
*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

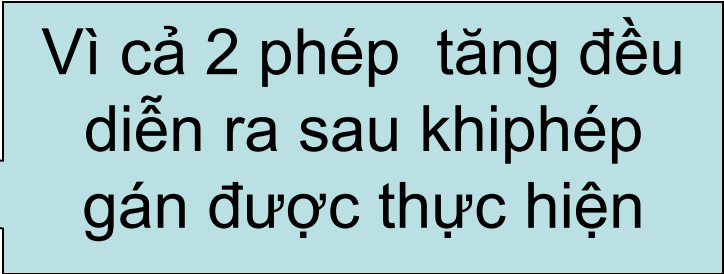
Các phép toán trên con trỏ

- 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
- $p +/- n == (\text{gia tri } p) +/- n * \text{sizeof}(*p)$
- Phép trừ giữa 2 con trỏ cho ta khoảng cách (số phần tử) giữa 2 con trỏ
- $p - q = ((\text{gia tri } p) - (\text{gia tri } q)) / \text{sizeof}(*p)$
- 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 -- hầu to 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;

- ++*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

```
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 == a[0]`

`a+i == &a[i]; => *(a+i) == a[i];`

- 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]`


```

{
    int i,n;
    int m[n];
    printf("\n nhap n"); scanf("%d",&n);
    for(i=1;i<n;i++) i[m] =i*5;
    for(i=1;i<n;i++)
        printf("\n m[%d] =%d",i,m[i]);
}

```

M[1]=5

M[2]=10

.....

M[11]=55

??? $m[i] \Rightarrow *(m+i) == *(i+m) == i[m]$

Con trỏ xâu

```
#include <stdio.h>

int main() {
    char s1[10]="ABC", *s2="ABC",*s3;
    s3="ABC";
    printf("S1= %s, s2= %s s3= %s",s1,s2,s3);
    printf("\n s1[1]= %c s2[1]= %c s3[1]=%c",s1[1],s2[2],s3[1]);
    s3=s1;
    s1[1]='D';
    // s2[1]='D';  Loi vi "ABC" la const,
    s3[2]='F';
    printf("\n s1[1]= %c s2[1]= %c s3[1]=%c",s1[1],s2[2],s3[2]);
    printf("\n S1= %s, s2= %s s3= %s",s1,s2,s3);
    gets(s1);
    // gets(s2); loi vi s2 chua duoc cap phat bo nho
    gets(s3);
    printf("\n S1= %s, s2= %s s3= %s",s1,s2,s3);
}
```

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>];

- vd : char *ds[10];

⇒ ds là 1 mảng gồm 10 ftử, mỗi ftử là 1 con trỏ kiểu char, đc dùng để trỏ tới 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"...};
- 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 ftử. 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 ftử sử dụng => tốn nhiều hơn

bt

- Viet ct nhap ds ho va ten hs cua 1 truong, biet so hs toi da =5000, ten moi hs khong qua 30 ky tu. (nhap khi du 5000 hs, hoac hovaten hs la rong).
- Sau do sap xep va in ra ds hs theo thu tu alphabe
- Ver1 : dung mang bt
- Ver 2 : dung mang con tro de sap xep
- Ver3,4,5.....

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
int main() {
    int n=0,i,j ;
    char ds[50][30], *tg,*pds[50];
    while (n<50) {
        printf("\n Vao ho ten hs thu %d ",n+1);
        gets(ds[n]); fflush(stdin);
        if (ds[n][0]='\0') break;
        pds[n]=&ds[n];
        n++;
    }
    for(i=0;i<n-1;i++)
        for(j=i+1;j<n;j++)
```

```
if (strcmp(pds[i],pds[j]) >0) {  
    tg=pds[i];  
    pds[i]=pds[j];  
    pds[j]=tg;  
}  
printf("\n DS da sap xep la ");  
for(i=0;i<n;i++)  
    puts(pds[i]);  
getch();  
}
```

- Một ưu điểm khác của mảng trả là ta có thể hoán chuyển các đối tượng (mảng con, cấu trúc..) được trả bởi con trả này bằng cách **hoán chuyển các con trả**
- Ưu điểm tiếp theo là việc truyền tham số trong hàm
- Ví dụ : Vào ds 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 nơi chứa địa chỉ hàng thứ i của mảng

$*(M+i)$ cho địa chỉ hàng thứ i của mảng

$*(M+i)+k$ là địa chỉ phần tử $[i][k]$

**Khi nào buộc phải dùng *, khi nào dùng ** và khi nào
*** ??? !!!!**

- Ví dụ : in ra 1 ma tran vuong va cong moi tu của MT 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”); }
```

```
    }
```

2.3 Bộ nhớ động – Dynamic memory

- Cho đến lúc này ta chỉ dùng bộ nhớ tĩnh : tức là khai báo mảng, biến và các đối tượng # 1 cách tường minh trước khi thực hiện ct.
- Trong thực tế nhiều khi ta không thể xđịnh 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
- Việc dùng bộ nhớ động cho phép xác định bộ nhớ cần thiết trong quá trình thực hiện của CT, 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 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

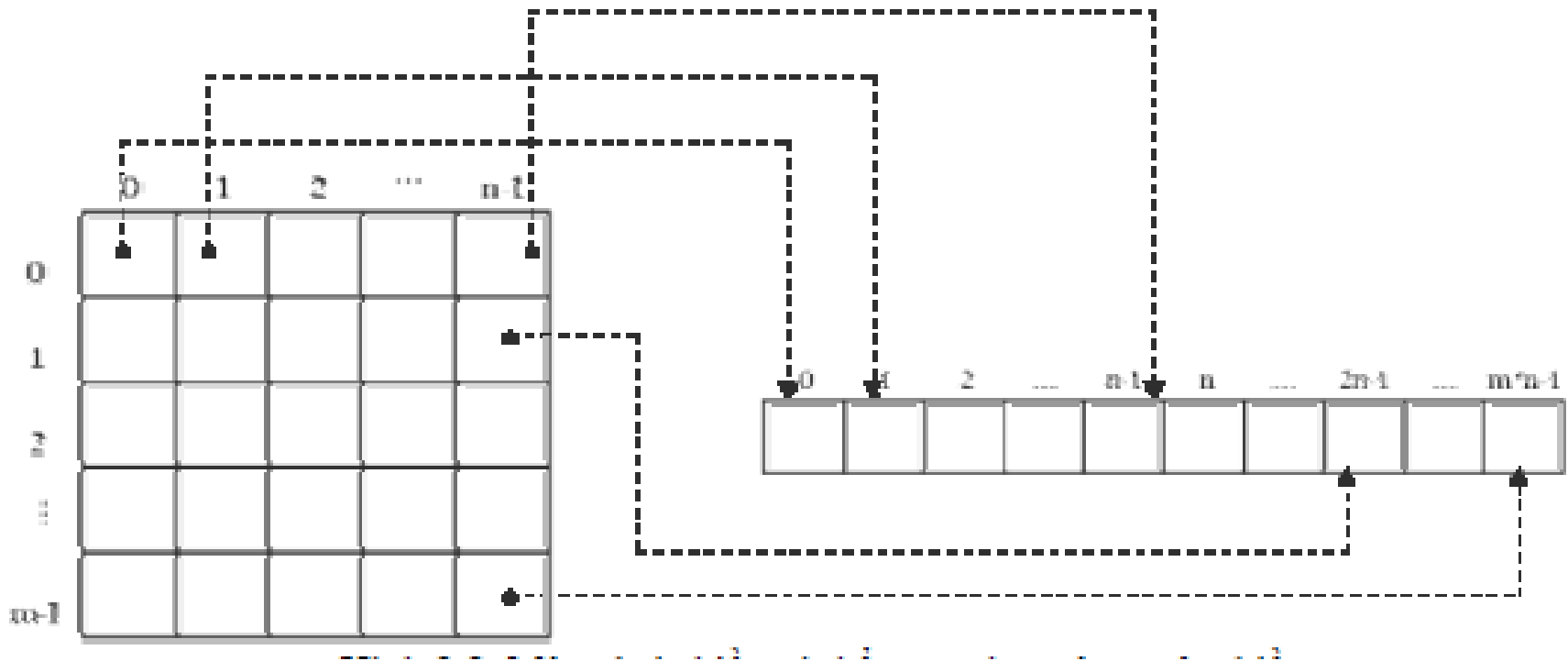
Giải phóng bộ nhớ

- 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 $X[i][j] = A[i*n + j]$

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;    ? ? ?
```

```
} // Khai báo xong. (Đoạn trên có lỗi syntax ?? )
```

Su dụng : `mt[i][j]` như bình thường. cuối cùng để giải phóng:

```
delete [] mt[0]; // xoá ? Tại sao?
```

```
delete [] mt;
```

Cách khác ???

```
// 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();
    printf("\n Nhap so dong cua ma tran: "); scanf("%d",&M);
    printf("\n Nhap so cot cua ma tran: ");  scanf("%d",&N);
    //Cấp phát vùng nhớ cho ma trận A
    if (!AllocMatrix(&A,M,N))
    {
        printf("\n Không còn đủ bộ nhớ!");
        return 1;
    }
    //Cấp phát vùng nhớ cho ma trận B
    if (!AllocMatrix(&B,M,N))
    {
        printf("\n Không còn đủ bộ nhớ!");
        FreeMatrix(A);//Giải phóng vùng nhớ A
        return 1;
    }
}
```

```

//Cấp phát vùng nhớ cho ma trận C
if (!AllocMatrix(&C,M,N))
{
    printf("\n Không còn đủ bộ nhớ! ");
    FreeMatrix(A);//Giải phóng vùng nhớ A
    FreeMatrix(B);//Giải phóng vùng nhớ B
    return 1;
}
printf("\n Nhập ma trận thu 1 ");
InputMatrix(A,M,N,'A');
printf("\n Nhập ma trận thu 2 ");
InputMatrix(B,M,N,'B');
clrscr();
printf("\n Ma trận thu 1");
DisplayMatrix(A,M,N);
printf("\n Ma trận thu 2");
DisplayMatrix(B,M,N);
AddMatrix(A,B,C,M,N);
printf("\n Tổng hai ma trận");
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)  
    {  
        printf("\n $c [%d][%d] = ",Symbol,I,J);  
        scanf("%d",&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)  
            printf(" %7d",A[I*N+J]);  
        printf("\n ");  
    }  
}
```

BTVN

- Tìm hiểu các hàm cấp phát bn động của C
- Dưa vào ví dụ trên slide, xd ct nhan 2 ma tran dung bo nho dong voi mang 1 chieu (dung $M(i * C + j)$)
- Viet lai 2 ct nhan ma tran va cong ma tran, dung bo nho dong voi con tro cua con tro (**) – dung $M[i][j]$ nhu la mang 2 chieu binh thuong
- Cai lai ct dssv thay gui, chay thu, roi them ham daoten, sxkieuvn

Modul hoa

- CtCon : Thủ tục – Hàm
- Cấu trúc CT :
 - Phân cấp –Hierarchical (pascal)
 - Đồng đẳng : C/C++
- Truyền TS :
 - Tham trị
 - Tham chiếu/ Biến
- Biến và phạm vi :
 - Global
 - Local

Hàm và truyền tham số

- Một CT C được cấu trúc thông qua các **hàm**. Mỗi hàm là một modul nhỏ trong CT, 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 cont...

- 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;
} // lỗi ???
```

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()*.

```
void vd(int *x, int y, int &z)
{
    *x+=3;
    y+=5;
    z+=10;
}

int main() {
    int x=10,y=20,z=30;
    vd(&z,x,y);
    printf("\n x= %d y=%d z= %d",x,y,z);
}
```

- **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
```

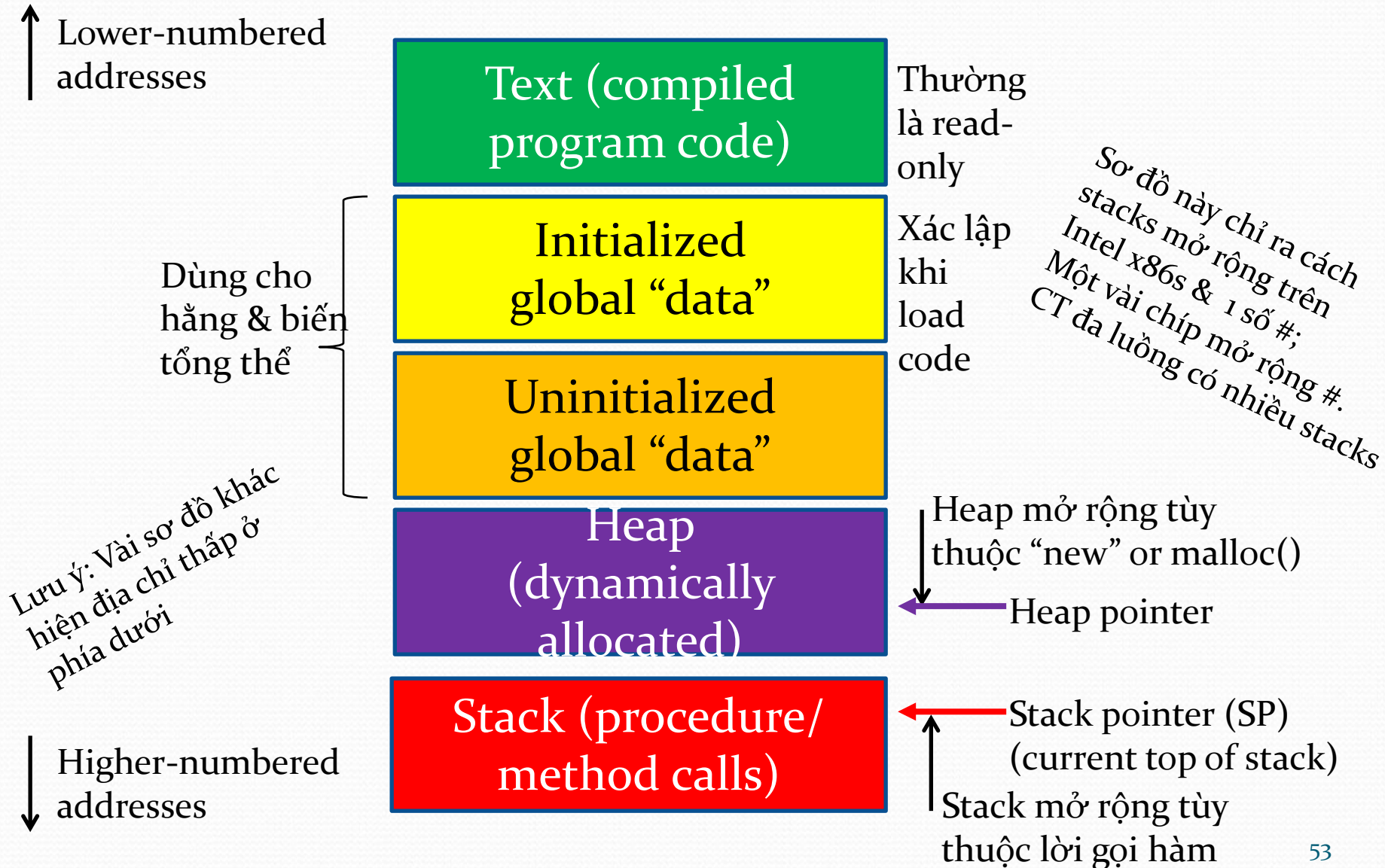
```
    cout<<"X="<<X<<endl;
```

```
    return 0;
```

```
}
```

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

Sơ đồ bộ nhớ



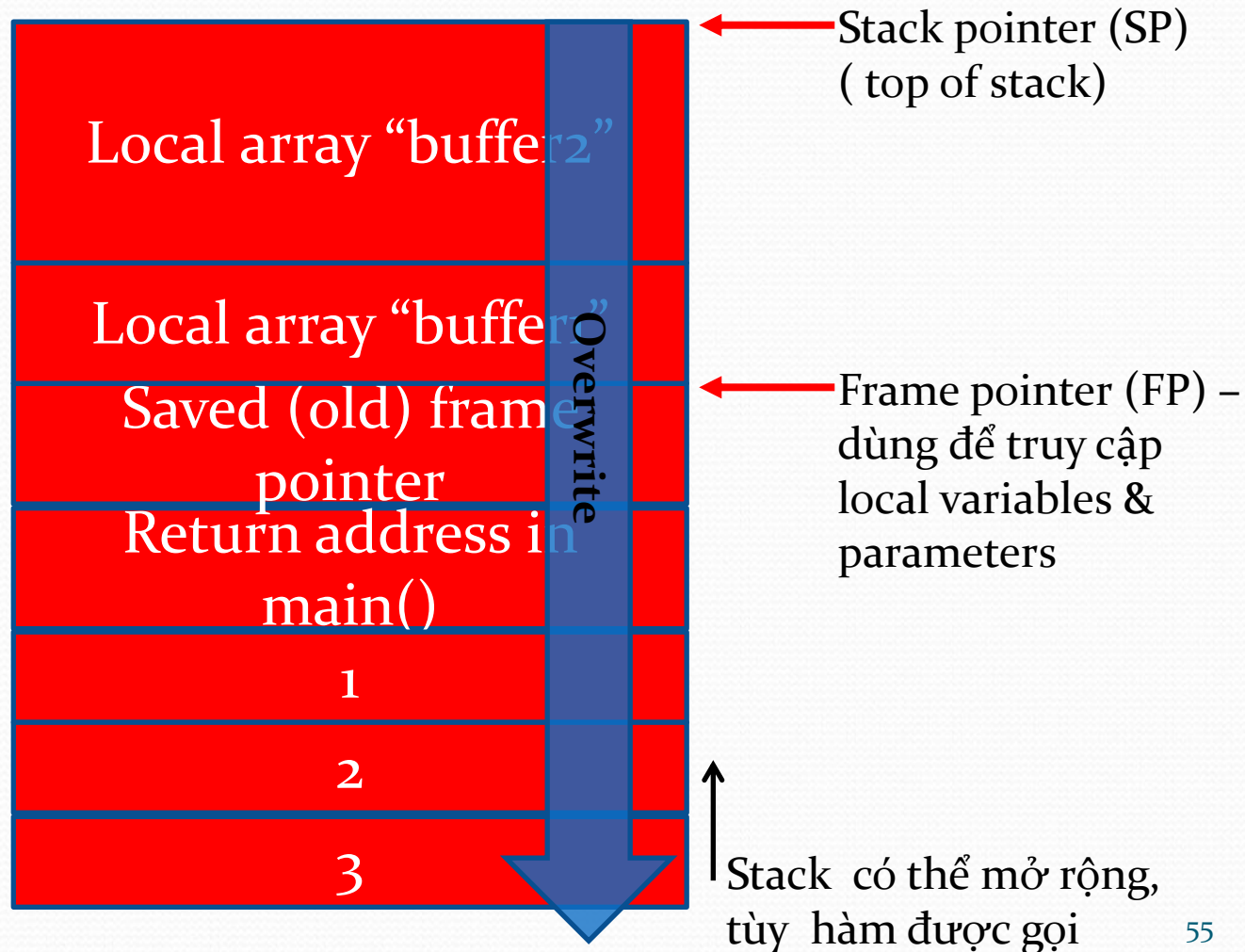
Ví dụ

```
void main() {    f(1,2,3);    }  
void f(int a, int b, int c) {  
    char buffer1[5];  
    char buffer2[10];  
    strcpy(buffer2, "This is a very long string!!!!!!");  
}
```

- LỜI gọi hàm f() sẽ thực hiện như sau:
 pushl \$3 ; đổi số 3
 pushl \$2 ; Hầu hết C _Compiler đẩy vào stack theo thứ tự ngược
 pushl \$1
 call f
- LỜI gọi f() sẽ đẩy con trỏ lệnh -instruction pointer (IP) vào stack
 - Trong ví dụ trên, là vị trí trong “main()” ngay sau f(...)
 - Con trỏ lệnh đc lưu gọi là return address (RET)
 - CPU sau đó bắt đầu thực hiện hàm

Stack:

↑ Vùng nhớ thấp



↓ Vùng nhớ cao

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 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, vd :

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

- Mỗi khi hàm *MyDelay()* được gọi chúng ta phải gởi 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 l = 0; l < Loops; ++l) ;
}
```

- `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,
- 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`

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>
#include <math.h>
long abs(long X) {
    return labs(X);
}
double abs(double X) {
    return fabs(X);
}
int main() {
    int X = -7;
    long Y = 200000l;
    double Z = -35.678;
    cout<<"Tri tuyet doi cua so nguyen "<<X<<" la " <<abs(X)<<endl;
    cout<<"Tri tuyet doi cua so nguyen "<<Y<<" la " <<abs(Y)<<endl;
    cout<<"Tri tuyet doi cua so thuc "<<Z<<" la " <<abs(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);
    printf("\n So phuc thu nhat:");      DisplaySP(C1);
    printf("\n So phuc thu hai:");      DisplaySP(C2);
    C3 = AddSP(C1,C2);
    C4 = SubSP(C1,C2);
    printf("\n Tong hai so phuc nay:");  DisplaySP(C3);
    printf("\n Hieu 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)
{printf(" %f i %f", C.THUC ,C.AO; }

```

$$C1=c2+c3+c4-c5-c6+c7-c8$$

$\Rightarrow C$

$\Rightarrow C1=\text{subsp}(\text{addsp}(\text{subsp}(\text{subsp}(\text{addsp}(\text{addsp}(c2,c3) ,c4),c5),c6),c7),c8) \quad ;$

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.
- 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); printf("\\n So phuc thu nhat:");
    DisplaySP(C1);        printf("\\n So phuc thu hai:");
    DisplaySP(C2);
    C3 = C1 + C2;
    C4 = C1 - C2;
    printf("\\n Tong hai so phuc nay:"); DisplaySP(C3);
    printf("\\n Hieu 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,C.AO); }

```

Dn phép cong so phuc voi so thuc

```
sp operator + (sp c, double d) {
```

```
    sp tg;
```

```
    tg.THUC = c.THUC+d;
```

```
    tg.AO = c.AO;
```

```
    return tg;
```

```
}
```

```
sp operator + (double d,sp c) {
```

```
    sp tg;
```

```
    tg.THUC = c.THUC+d;    tg.AO = c.AO;
```

```
    return tg;
```

```
}
```

Những điểm cần lưu ý khi 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.

Mở rộng

- Đa năng hóa toán tử +=
- Bản chất của phép toán ++, -- ?? (xd hàm đa năng hóa để hiểu rõ cơ chế !)

```

      +=
sp1+=sp2;  ~~ sp1=sp1+sp2;
SP & operator += (SP &c1, SP c2) {
    C1.THUC+=c2.THUC;
    c1.AO+=c2.AO;
    return c1; }
SP & operator += (SP &c1, double d) {
    C1.THUC+=d;
    return c1;
}

```

So thuc += so phuc ? Khong duoc, vi kq la sp

++/--

- Tien to : ++c => tang phan thuc cua sp c 1 dv
- ++c ~~ c=c+1;

```
SP &operator ++(SP &c) {  
    c.THUC=c.THUC+1;  
    return c;  
}
```


++/--

- Tien to :

```
SP &operator ++(SP &c) {  
    c.THUC=c.THUC+1;  
    return c; }
```

- Hau to : c++; Tang c 1 dv, va tra ve gia tri c truoc khi tang

```
SP operator ++(SP &c,int) {  
    SP tg=c;  
    c.THUC =c.THUC+1;  
    return tg; }
```

BTVN

- DN Cau tuc matric, roi da nang hoa cac phep toan cong va nhan voi ma tran;
- DN cau truc TIME, roi da nang hoa cac phep toan khả dụng với kiểu TIME
- DN cau truc phân số, rồi đa năng hóa các phép toán khả dụng với phân số
- Bài dssv dùng struct !!!

- $a = b + c;$
- $a += b;$ ~ $a = a + b;$ toán hạng thu 1 cũng chính là nơi nhận kq \Rightarrow tham số thu 1 phải là tham chiếu !

```
SP operator += (SP &a, SP b) {  
    a.THUC += b.THUC;  
    a.AO += b.AO;  
    return a;  
}
```

- ++, --
- tien to : dn nhu phep toan 1 ngoi

SP & operator ++ (SP &a) {

 a.THUC++;

 return a;

}

Hau to : dn nhu phep toan 2 ngoi

SP operator ++ (SP &a,int) {

 SP tg=a; // luu a truoc khi tang

 a.THUC++;

 return tg;

}

Phần phụ

- Cho khai báo : `char s[20];`
- `S="hkdhk";` ???
- Biến static
- Class ?

- Biến tổng thể và biến tĩnh : được cấp phát bộ nhớ trong vùng nhớ static
- Biến cục bộ và các tham số, được cấp phát bn trong vùng nhớ stack

Bài tập số 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 !

```
#include <string.h>
```

```
int main() {
```

```
    char ds[100][30],tg[30];
```

```
    int n,i,j;
```

```
    for(n=0;n<100;n++)
```

```
    {
```

```
        printf("\n Vao ho ten hs thu %d : ",n+1);
```

```
        gets(ds[n]);
```

```
        if (ds[n][0]=='\0') break;
```

```
    }
```

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

```
    for(
```