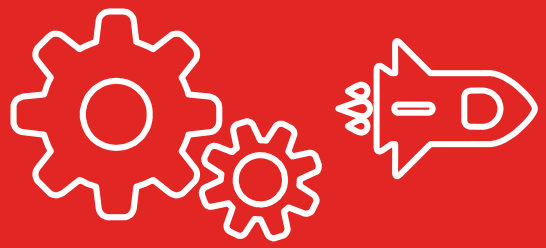
Trịnh Thành Trung (ThS)
trungtt@soict.hust.edu.vn

Bài 2
C/C++ nâng cao

Nội dung



1. Con trỏ
2. Quản lý bộ nhớ
3. Hàm và tham số
4. Đa năng hóa

1.
Con trỏ

Pointer

Con trỏ
Pointer

- Khái niệm
  - *Giá trị các biến được lưu trữ trong bộ nhớ máy tính, 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.*
- 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).
  - Việc sử dụng con trỏ cho phép ta truy nhập tới 1 đối tượng gián tiếp qua địa chỉ của nó.
- Có nhiều kiểu biến với các kích thước khác nhau, nên có nhiều kiểu con trỏ.
  - Ví dụ: Con trỏ int để trỏ tới biến hay hàm kiểu int.

Con trỏ
Pointer

- Khai báo con trỏ :
  - Syntax : `dataType * PointerName;`
    Chỉ rằng đây là con trỏ
- Sau khi khai báo, ta được con trỏ NULL (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

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	= += -=	<------

Các phép toán trên con trỏ

- Cộng hoặc trừ với 1 số nguyên n trả về 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
- Trừ 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
  - Chú ý đến sự tương thích về kiểu.

Ví dụ

```
char *pchar; short *pshort; long  *plong;
pchar ++;    pshort ++;     plong ++;
```

Giả sử các địa chỉ ban đầu tương ứng của 3 con trỏ là 100, 200 và 300, kết quả ta có các giá trị 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 `*` nên `*p++` tương đương với `*(p++)` tức là tăng địa chỉ mà nó trỏ tới chứ không phải tăng giá trị mà nó chứa.

`*p++ = *q++` sẽ tương đương với

```
*p = *q;
p=p+1;
q=q+1;
```

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

Dùng **( )** để tránh nhầm lẫn

Con trỏ
void*

- Là con trỏ không định kiểu. 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
```

Ví dụ

```
(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 thẻ 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 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

Mảng các con trỏ

- 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]

Con trỏ
trỏ tới con trỏ

- Ví dụ: in ra 1 ma trận vuông và công 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=o; 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.
Quản lý bộ nhớ

Memory management

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 dùng `new` và `delete`

Xin cấp phát bộ nhớ
new và delete

- Để xin cấp phát bộ nhớ ta dùng :

```
<biên trỏ> = new <kiểu dữ liệu>;
hoặc <biến trỏ> = 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.

- Giải phóng bộ nhớ

```
delete ptr;    // xóa 1 biến đơn
delete [] ptr; // xóa 1 biến mảng
```

Xin cấp phát bộ nhớ new và delete

- Bộ nhớ động được quản lý bởi hệ điều hành, 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:

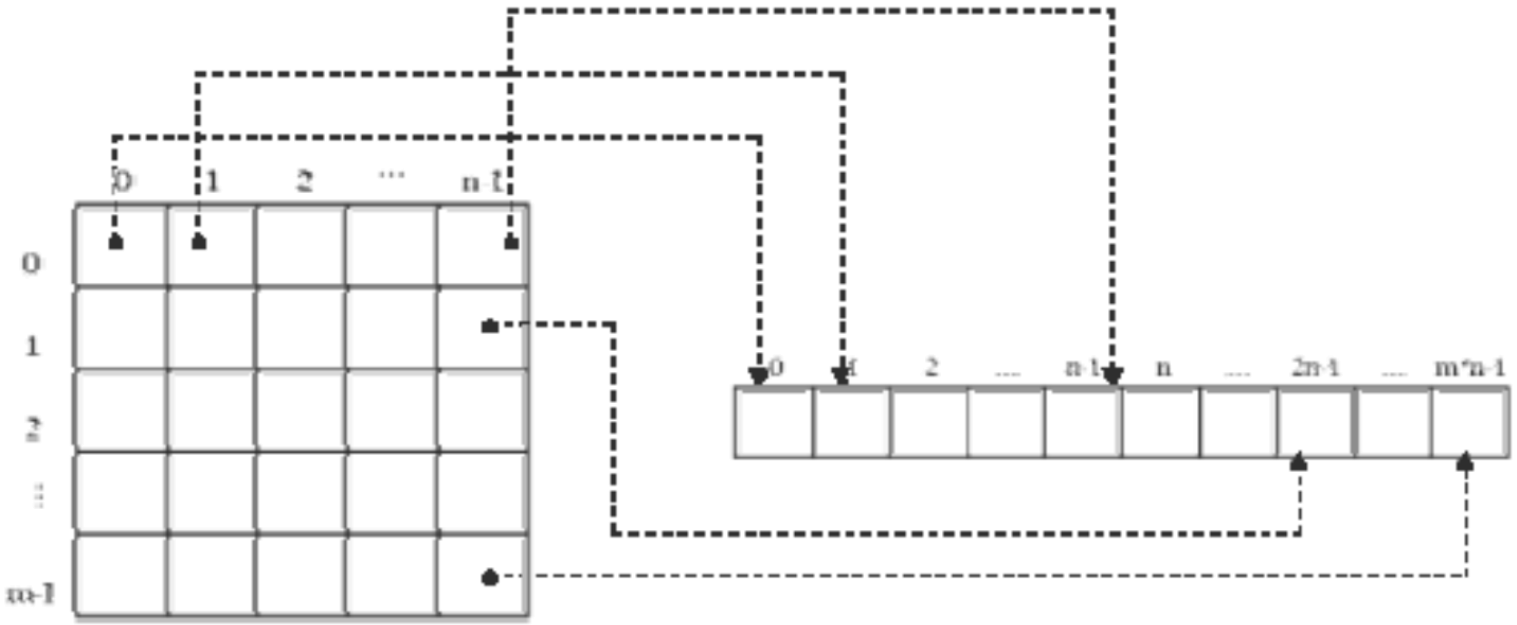
```
int *pds;
pds = new int [200];
if (pds == NULL) { // thông báo lỗi và xử lý
```

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;
    }
```

Bộ nhớ động cho mảng 2 chiều

- C1: Coi một mảng 2 chiều là 1 **mảng 1 chiều**



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]

Bộ nhớ động cho mảng 2 chiều

- C2. Dùng **con trỏ của con trỏ**
- *Ví dụ:* 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;
```

Ví dụ

```
// Khởi tạo ma trận với
// R hàng và C cột
 float  ** M = new float *[R];
 for (i=0; i<R;i++)
    M[i] = new float[C];
// Dùng M[i][j] cho
// các phần tử của ma trận
```

```
// Giải phóng
 for(i=0; i<R;i++)
    // Giải phóng các hàng
    delete []M[i];
 delete []M;
```

3.
Hàm và tham số

Function

Hàm và truyền tham số

- Chương trình *C* được cấu trúc thông qua các **hàm**. Mỗi hàm là một module 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++*: 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

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);
```

Phép
tham chiếu

{C++}

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);
```

Phép 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.h>
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;
}
```

Hàm với
tham số ngầm định

{C++}

- 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

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

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); // ✓
```

4.
Đa năng hóa

Overloading

Đa năng hóa
hàm

{C++}

- 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);
```

Đa năng hóa
toán tử

{C++}

- 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

Ví dụ cộng, trừ số phức trong *C*

```
#include <stdio.h>
// Định nghĩa số phức
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;
}
```

Ví dụ cộng, trừ số phức trong *C*

```
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; }
```

Đa năng hóa
toán tử

{C++}

- *Cú pháp*

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

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

Ví dụ cộng, trừ số phức trong *C++*

```
#include <iostream.h>
// Định nghĩa số phức
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;
}
```

Ví dụ cộng, trừ số phức trong *C++*

```
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);
}
```

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 đa năng hóa được*
    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. **C**ác ký hiệu tiền xử lý

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.

Thanks!

Any questions?

Email me at trungtt@soict.hust.edu.vn

Presentation template by SlidesCarnival