Chương 3
HÀM VÀ CHƯƠNG TRÌNH

Nội dung

1. Hàm và tổ chức chương trình
2. Tham số cho chương trình con
3. Truyền dữ liệu sang hàm qua đối số.
4. Hàm với biến tham chiếu.
5. Biến cục bộ và biến toàn cục.
6. Hàm đệ quy.
7. Hàm với con trỏ.
8. Con trỏ hàm.
9. Một số bài toán đệ quy phổ biến

Hàm và tổ chức chương trình

- Khái niệm
  - Hàm là một khối lệnh thực hiện một công việc hoàn chỉnh (module), được đặt tên và được gọi thực thi nhiều lần tại nhiều vị trí trong chương trình.
  - Hàm còn gọi là chương trình con (subroutine)

Hàm và tổ chức chương trình

- Khái niệm (tt):
  - Hàm có thể được gọi từ chương trình chính (hàm main) hoặc từ 1 hàm khác.
  - Hàm có giá trị trả về hoặc không. Nếu hàm không có giá trị trả về gọi là thủ tục *(procedure)*

Khái niệm và khai báo hàm

- Khái niệm (tt)
  - Có hai lọai hàm:
    - Hàm thư viện: là những hàm đã được xây dựng sẵn. Muốn sử dụng các hàm thư viện phải khai báo thư viện chứa nó trong phần khai báo #include.
    - Hàm do người dùng định nghĩa.

Khái niệm và khai báo hàm

- Dạng tổng quát của hàm do người dùng định nghĩa:

```
returnType functionName(parameterList)
{
   //Thân hàm
}
```

Khái niệm và khai báo hàm

```
#include<iostream.h>
#include<conio.h>
int Tonghaiso(int a,int b);
void main()
{
    int c,d,kq;
    cout<<"Nhap c = ";
    cin>>c;
    cout<<"Nhap d = ";
    cin>>d;
    kq=Tonghaiso(c,d);
    cout<<"Tong la: "<<kq;
}
int Tonghaiso(int a,int b)
{
    return a+b;
}
```

Khái niệm và khai báo hàm

```
#include<iostream.h>
int conghaiso(int a,int b);
void main()
{

}
int conghaiso(int a,int b)
{
    return a+b;
}

void ham1(int x,int y)
{
    return x+y;
}
```

SAI

Khái niệm và khai báo hàm

- Một hàm khi đã định nghĩa nhưng chúng vẫn chưa được thực thi, hàm chỉ được thực thi khi trong chương trình có một lời gọi đến hàm đó.
- Cú pháp gọi hàm:

<Tên hàm>([Danh sách các tham số])

Khái niệm và khai báo hàm

```
void main()
{
    int a, b, USC;
    cout<<“Nhap a,b: ”;
    cin>>a>>b;
    USC = uscln(a,b);
    cout<<“Uoc chung
    lon nhat la: ”<<USC;
}
```

```
int uscln(int a, int b)
{
    a=abs(a);
    b=abs(b);
    while(a!=b)
    {
        if(a>b) a-=b;
        else    b-=a;
    }
    return a;
}
```

Truyền dữ liệu sang hàm qua đối số

- Tham số hình thức: Khi hàm cần nhận đối số (*arguments*) để thực thi thì khi khai báo hàm cần khai báo danh sách các đối số để nhận giá trị từ chương trình gọi. Các tham số này được gọi là.

Ví dụ:

```
int min(int a, int b)
{
        if(a<b)
                return a;
        else
                return b;
}
```

Truyền dữ liệu sang hàm qua đối số

- Khi gọi hàm, ta cung cấp các giá trị thật, các giá trị này sẽ được sao chép vào các tham số hình thức và các giá trị thật được gọi là tham số thực.

  Ví dụ: Để tìm giá trị nhỏ nhất của 2 số 5 và 6 ta gọi hàm **min(5, 6)**

Truyền dữ liệu sang hàm qua đối số

- Có hai cách truyền đối số vào tham số hình thức:
  - Truyền ***tham trị:***

    Sau khi thoát khỏi hàm nó vẫn giữ giá trị gốc
  - Truyền ***tham biến:***

    Sau khi thoát khỏi hàm, nó sẽ lấy giá trị bị thay đổi trong hàm

Truyền dữ liệu sang hàm qua đối số

- Truyền tham trị (call by value)
  - Sao chép giá trị của ***đối số*** vào *tham số hình thức* của hàm.
  - Những thay đổi của tham số không ảnh hưởng đến giá trị của đối số.

Truyền dữ liệu sang hàm qua đối số

Ví dụ:

```
void hamVidu(int a)
{
    a = a*2;
    cout << “gia tri cua a
    trong hamVidu:“<< a;
}
```

```
void main()
{
    int a=40;
    hamVidu (a);
    cout << “\n Gia tri cua a
    trong ham main: ”;
    cout << “a = ” << a <<
    endl;
}
```

Gia tri cua a trong ham **hamVidu**: 80
Gia tri cua a trong ham main: 40

Hàm với biến tham chiếu

- Truyền tham chiếu *(call by reference)*
  - Sao chép địa chỉ của đối số vào tham số hình thức. Do đó, những thay đổi đối với tham số sẽ có tác dụng trên đối số.

Ví dụ: Khi gọi hàm hamVidu (&a);

Địa chỉ của a truyền vào cho tham số hình thức của hàm: hamVidu **(int &b**)

Hàm với biến tham chiếu

```
void main()
{
  int a=40;
  hamgido (a);
  cout << “\Trong ham
    main : a = “ << a ;
}
```

```
void hamgido ( int &b)
{
  b*= 2;
  cout << “Trong hàm
    double a = “ << b;
}
```

```
Trong hàm hamVidu a = 80
Trong hàm main a = 80
```

Hàm với biến tham chiếu

```
#include<iostream.h>
void ham1(int x);//truyen tham tri
void ham2(int &x);//truyen tham bien
void main()
{
    int x=5;
    cout<<"x ban dau = "<<x;
    cout<<"\n";
    ham1(x);
    cout<<"x sau khi goi ham1 = "<<x;
    cout<<"\n";
}
void ham1(int x)
{
    x=10;
    cout<<"x trong ham1 ="<< x <<"\n";
}

void ham2(int &x)
{
    x=10;
    cout<<x <<"\n";
}
```

Gọi hàm truyền tham trị

```
#include<iostream.h>
void ham1(int x);//truyen tham tri
void ham2(int &x);//truyen tham bien
void main()
{
    int x=5;
    cout<<"x ban dau = "<<x;
    cout<<"\n";
    ham2(x);
    cout<<"x sau khi goi ham2 = "<<x;
    cout<<"\n";
}
void ham1(int x)
{
    x=10;
    cout<<"x trong ham1 ="<< x <<"\n";
}

void ham2(int &x)
{
    x=10;
    cout<<x <<"\n";
}
```

Gọi hàm truyền tham biến

Prototype (nguyên mẫu) của hàm

- Chương trình bắt buộc phải có prototype của hàm hoặc phải bắt buộc viết định nghĩa của hàm trước khi gọi.
- Sau khi đã sử dụng prototype của hàm, ta có thể viết định nghĩa chi tiết hàm ở bất kỳ vị trí nào trong chương trình.

Prototype (nguyên mẫu) của hàm

```
#include <iostream.h>// Khai báo thư viện iostream.h
int max(int x, int y);// khai báo nguyên mẫu hàm max
void main()//hàm main (sẽ gọi các hàm thực hiện)
{
    int a, b;// khai báo biến
    cout<<" Nhap vao 2 so a, b ";
    cin>>a>>b;
    cout<<"so lon nhat la:"<< max(a,b);
}
int max(int x, int y)// Định nghĩa hàm max(a,b)
{
    return (x>y) ? x:y;
}
```

Đệ quy

- Một hàm được gọi là đệ qui nếu một lệnh trong thân hàm gọi đến chính hàm đó.
- Đệ qui giúp giải quyết bài toán theo cách nghĩ thông thường một cách tự nhiên.
- Đệ qui phải xác định được điểm dừng. Nếu không xác định chính xác thì làm bài toán bị sai và có thể bị lặp vĩnh cửu (Stack Overhead)

Đệ quy

- Ví dụ: Định nghĩa giai thừa của một số nguyên dương n như sau:
- 5!=5*4!
- 4!=4*3!
- Tức là nếu ta biết được (n-1) giai thừa thì ta sẽ tính được n giai thừa, vì n!=n*(n-1)!
- Thấy n=0 hoặc n=1 thì giai thừa luôn = 1 ➔ chính là điểm dừng
  - n!=1* 2 * 3 *…* (n-1) *n = (n-1)! *n (với 0!=1)

Đệ quy

```
int giaiThua(int n)
{
    if(n<=1)
        return(1);
    return n*giaiThua(n-1); // goi de qui
}
```

Đệ quy

- **Phân loại đệ qui**
  - Đệ qui tuyến tính.
  - Đệ qui nhị phân.
  - Đệ qui phi tuyến.
  - Đệ qui hỗ tương.

Đệ quy tuyến tính

Trong thân hàm có duy nhất một lời gọi hàm gọi lại chính nó một cách tường minh.

<Kiểu dữ liệu hàm> **TenHam** (<danh sách tham số>)

```
{
    if (điều kiện dừng)
    {
        ...
        //Trả về giá trị hay kết thúc công việc
    }
    //Thực hiện một số công việc (nếu có)
    . . . TenHam (<danh sách tham số>);
    //Thực hiện một số công việc (nếu có)
}
```

Đệ quy tuyến tính

Ví dụ:

Tính *S* (*n*) = 1 + 2 + 3 + L + *n*

- Điều kiện dừng: S(0) = 0.

- Qui tắc (công thức) tính: S(n) = S(n-1) + n.

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

Đệ quy nhị phân

Trong thân của hàm có hai lời gọi hàm gọi lại chính nó một cách tường minh.

```
<Kiểu dữ liệu hàm> TenHam (<danh sách tham số>)
{
    if (điều kiện dừng)
    {
        ...
        //Trả về giá trị hay kết thúc công việc
    }
    //Thực hiện một số công việc (nếu có)
    . . .TenHam (<danh sách tham số>); //Giải quyết vấn đề nhỏ hơn
    //Thực hiện một số công việc (nếu có)
    . . . TenHam (<danh sách tham số>); //Giải quyết vấn đề còn lại
    //Thực hiện một số công việc (nếu có)
}
```

Đệ quy nhị phân

Ví dụ: Tính số hạng thứ n của dãy Fibonaci được định nghĩa như sau:

$f_1 = f_0 = 1$;

$f_n = f_{n-1} + f_{n-2}$;

Điều kiện dừng: f(0) = f(1) = 1.

```
long Fibonaci (int n)
{
    if(n==0 || n==1)
    return 1;
    return Fibonaci(n-1) + Fibonaci(n-2);
}
```

Đệ quy phi tuyến

Trong thân của hàm có lời gọi hàm gọi lại chính nó được đặt bên trong vòng lặp.

```
<Kiểu dữ liệu hàm> TenHam (<danh sách tham số>)
{
    for (int i = 1; i<=n; i++)
    {
        //Thực hiện một số công việc (nếu có)
        if (điều kiện dừng)
        { ...
        //Trả về giá trị hay kết thúc công việc
    }
    else
    { //Thực hiện một số công việc (nếu có)
        TenHam (<danh sách tham số>);
}}}
```

Đệ quy phi tuyến

Ví dụ: Tính số hạng thứ n của dãy {Xn} được định nghĩa như sau:

$X_0 = 1$;

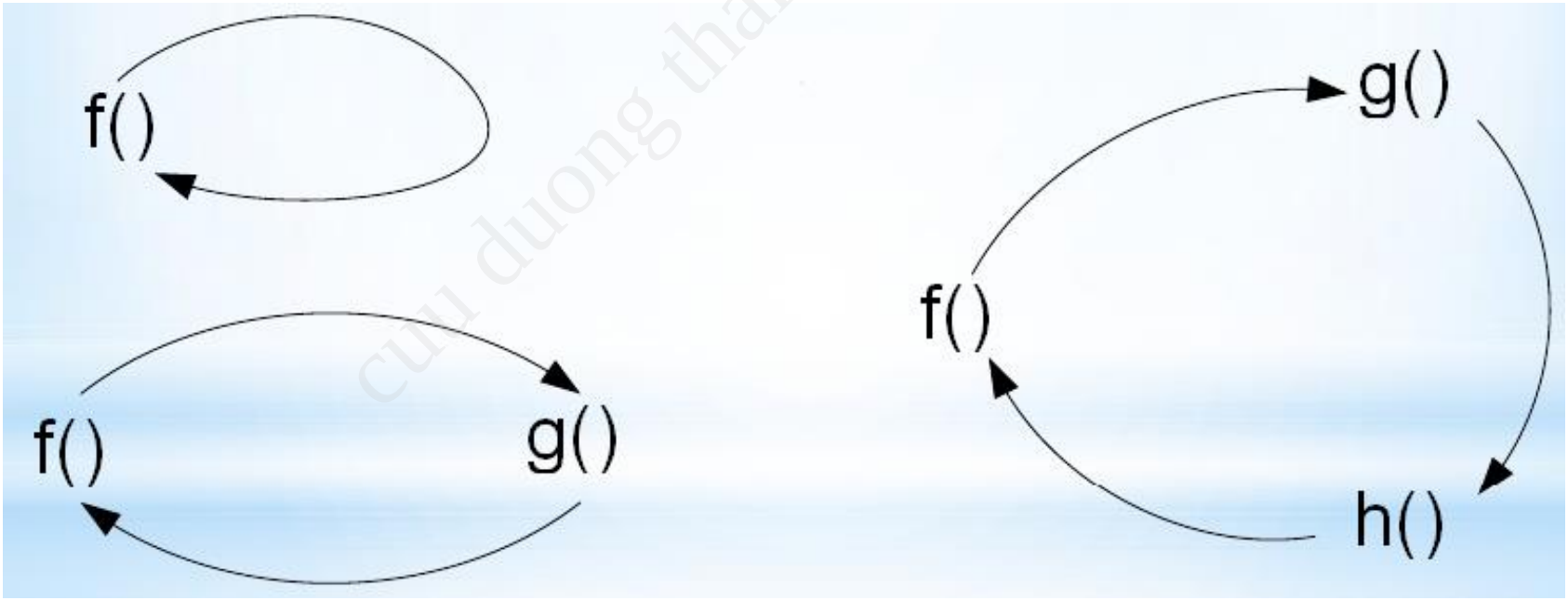
$Xn = n^2X_0 + (n-1)^2X_1 + \ldots + 1^2X_{n-1}$;

Điều kiện dừng:X(0) = 1.

```
long TinhXn (int n)
{
    if(n==0)
    return 1;
    long s = 0;
    for (int i=1; i<=n; i++)
    s = s + i * i * TinhXn(n-i);
    return s;
}
```

Đệ quy hỗ tương

Trong thân của hàm này có lời gọi hàm đến hàm kia và trong thân của hàm kia có lời gọi hàm tới hàm này.



Đệ quy hỗ tương

<Kiểu dữ liệu hàm> **TenHam2** (<danh sách tham số>);
<Kiểu dữ liệu hàm> **TenHam1** (<danh sách tham số>)
{
    //Thực hiện một số công việc (nếu có)
    …TenHam2 (<danh sách tham số>);
    //Thực hiện một số công việc (nếu có)
}
<Kiểu dữ liệu hàm> **TenHam2** (<danh sách tham số>)
{
    //Thực hiện một số công việc (nếu có)
    …TenHam1 (<danh sách tham số>);
    //Thực hiện một số công việc (nếu có)
}

Đệ quy hỗ tương

Ví dụ: Tính số hạng thứ n của hai dãy {Xn}, {Yn} được định nghĩa như sau:

$X_0 = Y_0 = 1$;

$X_n = X_{n-1} + Y_{n-1}$; (n>0)

$Y_n = n^2 X_{n-1} + Y_{n-1}$; (n>0)

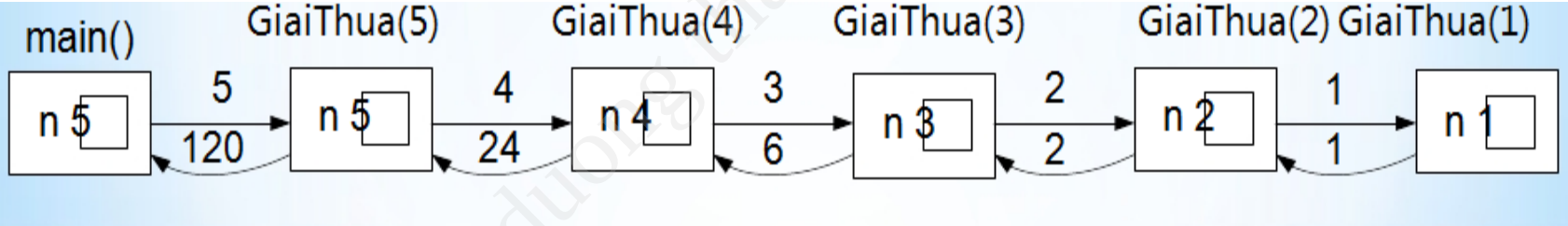
- *Điều kiện dừng*:*X(0) = Y(0) = 1.*

```
long TinhYn(int n);
long TinhXn (int n)
{
      if(n==0)
      return 1;
      return TinhXn(n-1) + TinhYn(n-1);
      }
long TinhYn (int n)
{
if(n==0)
return 1;
return n*n*TinhXn(n-1) + TinhYn(n-1);

}
```

Hoạt động đệ quy

Ví dụ tính n! với n=5



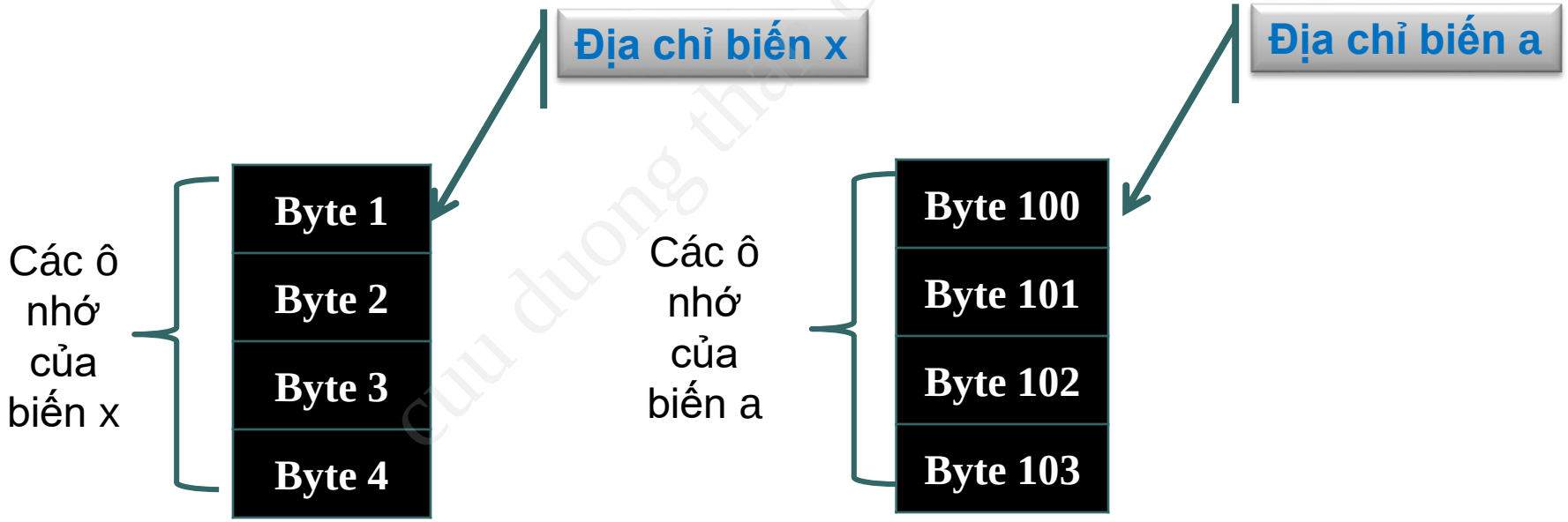
Con trỏ

- Thông tin của một biến bao gồm:
  - ✓ Tên biến
  - ✓ Kiểu dữ liệu của biến
  - ✓ Giá trị của biến
- Mỗi biến sẽ được lưu trữ tại một vị trí xác định trong ô nhớ, nếu kích thước của biến có nhiều byte thì máy tính sẽ cấp phát một dãy các byte liên tiếp nhau, địa chỉ của biến sẽ *lưu byte đầu tiên* trong dãy các byte này

Con trỏ

Ví dụ:

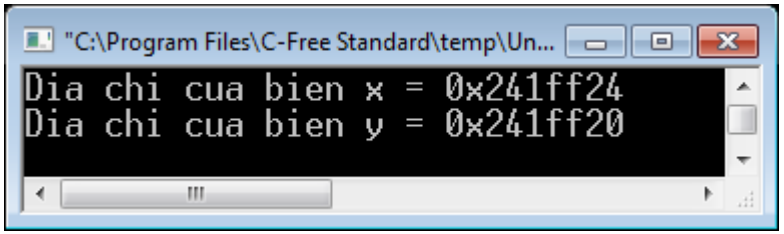
```
float x;
int a;
```



Con trỏ

- Địa chỉ của biến luôn luôn là một số nguyên (hệ thập lục phân) dù biến đó chứa giá trị là số nguyên, số thực hay ký tự, …
- Cách lấy địa chỉ của biến: &tênbiến
- Ví dụ:

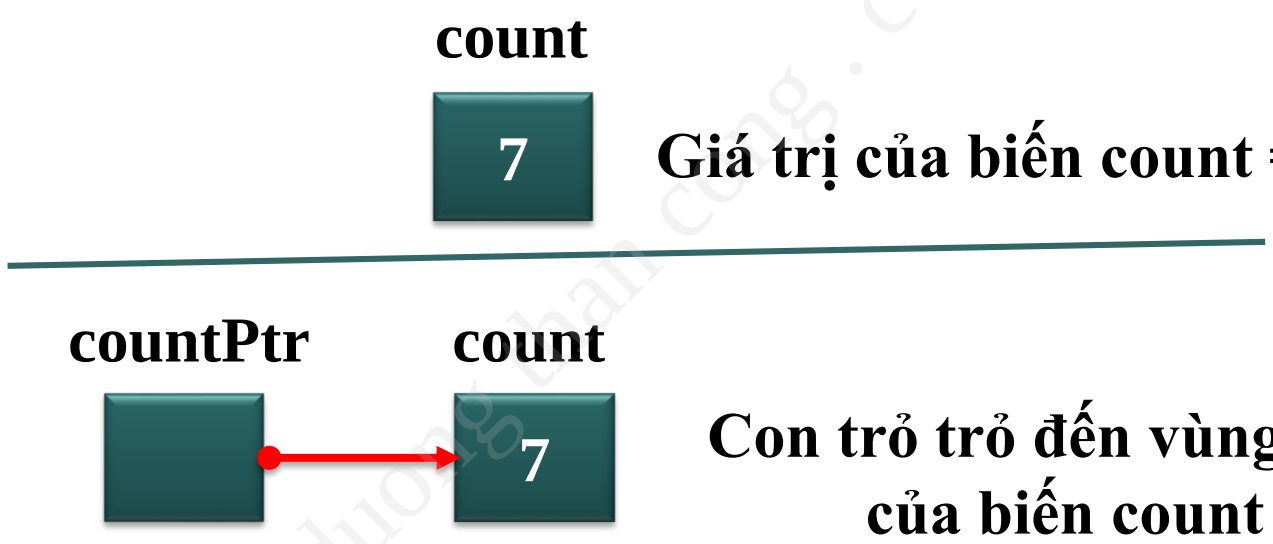
```
void main()
{
        int x=7;
        float y=10.5;
        cout<<"Dia chi cua bien x = "<<&x<<endl;
        cout<<"Dia chi cua bien y = "<<&y;
}
```



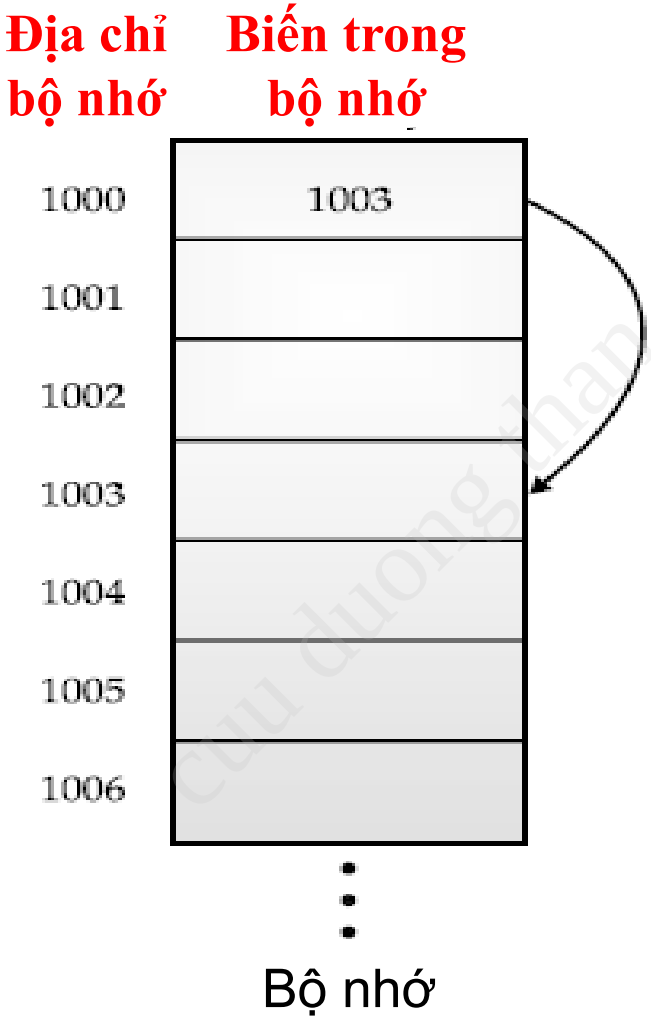
Con trỏ

- Con trỏ là 1 biến chứa một địa chỉ bộ nhớ. Địa chỉ này là vị trí của một đối tượng khác trong bộ nhớ.
- Nếu một biến chứa địa chỉ của một biến khác, biến thứ nhất được gọi là trỏ đến biến thứ hai.

Con trỏ



Con trỏ



Một biến được cấp phát ô nhớ tại địa chỉ 1000 có giá trị là địa chỉ (1003) của 1 biến khác. Biến thứ nhất được gọi là con trỏ.

Con trỏ

- Cú pháp:

type ***pointerVariable**;

type: xác định kiểu dữ liệu của biến mà con trỏ trỏ đến.

Ví dụ:

int *a;

a

Con trỏ

➢ Toán tử &:

Là toán tử 1 ngôi, trả về địa chỉ bộ nhớ của toán hạng của nó.

- Toán tử & dùng để gán địa chỉ của biến cho biến con trỏ

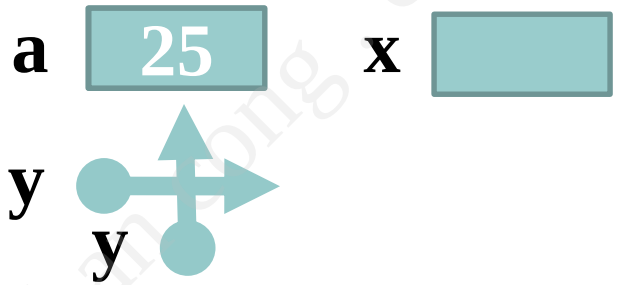
Cú pháp:

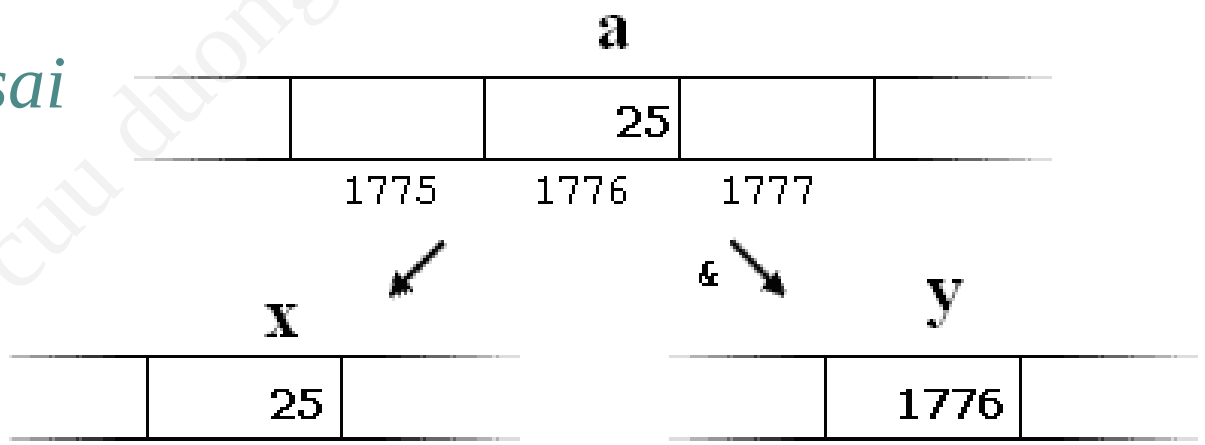
<Tên biến con trỏ>=&<Tên biến>

Con trỏ

Ví dụ:

```
int a=25, x;
int *y;
    x=a;
    y=&a;
    y=a;//sai
```





Con trỏ

- Toán tử * :

  Là toán tử một ngôi trả về giá trị tại địa chỉ con trỏ trỏ đến.

- Cú pháp:

  *<Tên biến con trỏ>

  Ví dụ: a=*p;

  a=p;*//sai*

Các thao tác trên con trỏ

Lệnh gán con trỏ

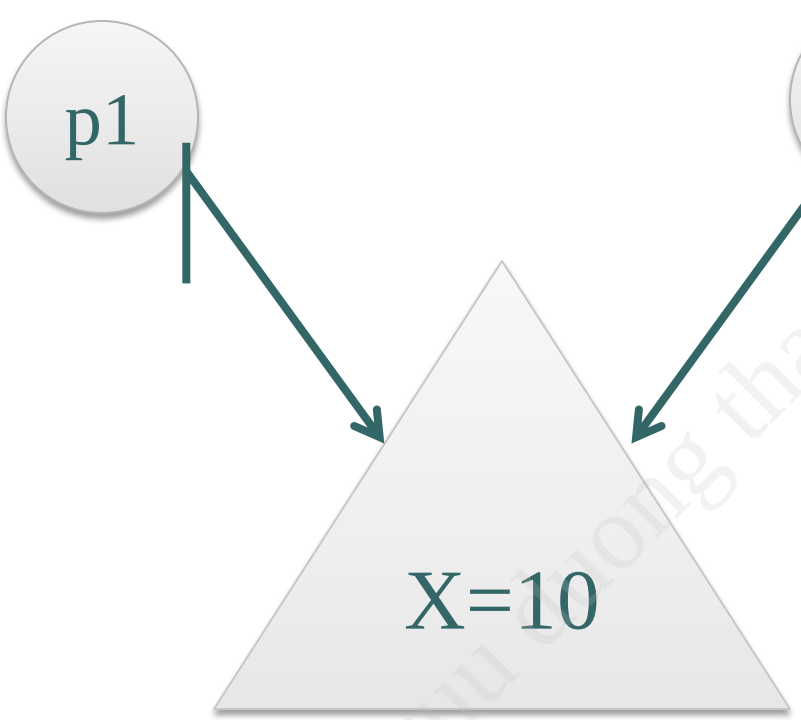
Có thể dùng phép gán để gán giá trị của một con trỏ cho một con trỏ khác có cùng kiểu

Ví dụ:

```
int x=10;
int *p1, *p2;
p1 = &x;
p2 = p1;
```

Sau khi đoạn lệnh trên được thực hiện, cả p1 và p2 cùng trỏ đến biến x.

Các thao tác trên con trỏ




```
#include <iostream.h>
int main(int argc, char *argv[])
{
    int x=10;
    int *p1, *p2;
    p1 = &x;
    p2 = p1;

    cout<<"x = "<<x<<"\n";
    cout<<"*p1 = "<<*p1<<"\n";
    cout<<"*p2 = "<<*p2<<"\n";
    *p1=0;
    cout<<"x = "<<x<<"\n";
    cout<<"*p2 = "<<*p2<<"\n";
    return 0;
}
```

Các thao tác trên con trỏ

- Phép toán số học trên con trỏ
  - Chỉ có 2 phép toán sử dụng trên con trỏ là phép cộng và trừ
  - Khi cộng (+) hoặc trừ (-) 1 con trỏ với 1 số nguyên N; kết quả trả về là 1 con trỏ. Con trỏ này chỉ đến vùng nhớ cách vùng nhớ của con trỏ hiện tại một số nguyên lần kích thước của kiểu dữ liệu của nó.

Các thao tác trên con trỏ

Ví dụ:

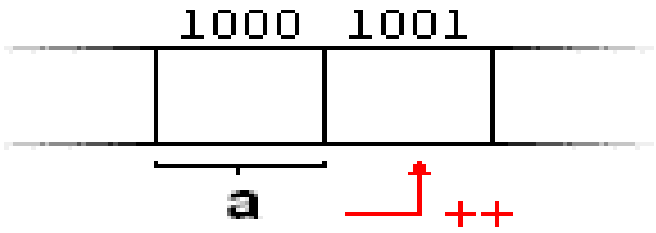
```
char *a;
short *b;
long *c;
```

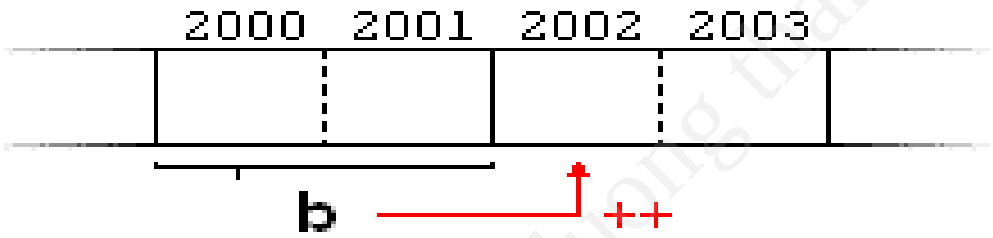
Các con trỏ a, b, c lần lượt trỏ tới ô nhớ **1000**, **2000** và **3000**.

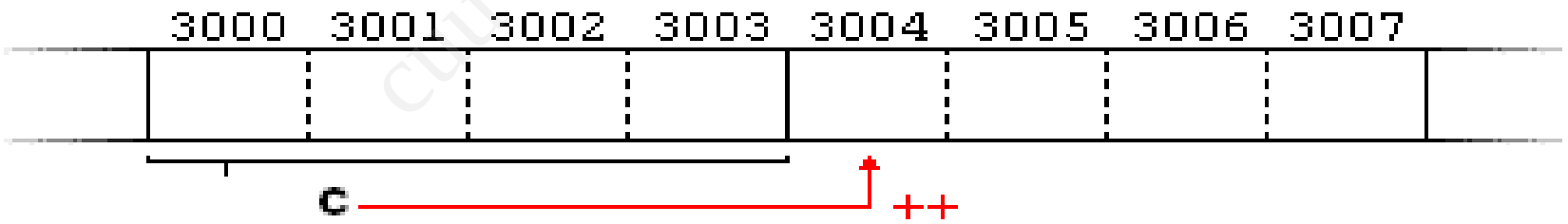
Cộng các con trỏ với một số nguyên:

```
a = a + 1;//con trỏ a dời đi 1 byte
b = b + 1;//con trỏ b dời đi 2 byte
c = c + 1; //con trỏ c dời đi 4 byte
```

Các thao tác trên con trỏ







Các thao tác trên con trỏ

❖ *Lưu ý*: cả hai toán tử tăng (++) và giảm (--) đều có quyền ưu tiên lớn hơn toán tử *****

Ví dụ: *p++;

Lệnh *p++ tương đương với *(p++) : thực hiện là tăng **p** (địa chỉ ô nhớ mà nó trỏ tới chứ không phải là giá trị trỏ tới).

Các thao tác trên con trỏ

Ví dụ:

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

Cả hai toán tử tăng (++) đều được thực hiện sau khi giá trị của *q được gán cho *p và sau đó cả q và p đều tăng lên 1. Lệnh này tương đương với:

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

Các thao tác trên con trỏ

```
#include <iostream.h>
#include<conio.h>
void main ()
{
    int a = 20, b = 15, *pa, *pb, temp;
    pa = &a; // con trỏ pa chứa địa chỉ của a
    pb = &b; // con trỏ pb chứa địa chỉ của b
        temp = *pa;
        *pa = *pb;
        *pb = temp;
    cout << "a = " << a << endl;
    cout << “b = ” << b;
}
```

Cấp phát bộ nhớ động

- Con trỏ cung cấp sự hỗ trợ cho cấp phát bộ nhớ động trong C/C++.
- Cấp phát động là phương pháp giúp chương trình có thể dành được thêm bộ nhớ trong khi đang thực thi, giải phóng bộ nhớ khi không cần thiết.
- C/C++ hỗ trợ hai hệ thống cấp phát động: một hệ thống được định nghĩa bởi C và một được định nghĩa bởi C++.

Cấp phát bộ nhớ động

- Các hàm cấp phát bộ nhớ động của C
  - Vùng nhớ Heap được sử dụng cho việc cấp phát động các khối bộ nhớ trong thời gian thực thi chương trình. Gọi là bộ nhớ động.
  - Hàm malloc() và free() dùng để cấp phát và thu hồi bộ nhớ, trong thư viện stdlib.h

Cấp phát bộ nhớ động

- Hàm malloc(): cấp phát bộ nhớ động.
  - Prototype của hàm có dạng:

    void *malloc(length)
  - length: là số byte muốn cấp phát.
  - Hàm malloc() trả về một con trỏ có kiểu void, do đó có thể gán nó cho con trỏ có kiểu bất kỳ.
  - Sau khi cấp phát thành công, hàm malloc() trả về địa chỉ của byte đầu tiên của vùng nhớ được cấp phát từ heap. Nếu không thành công (không có đủ vùng nhớ trống yêu cầu), hàm malloc() trả về null.

Cấp phát bộ nhớ động

Ví dụ 1:

```
char *p;
p = (char *) malloc(1000); //cấp phát 1000 byte
```

Vì hàm malloc() trả về con trỏ kiểu void, nên phải ép kiểu (casting) nó thành con trỏ char cho phù hợp với biến con trỏ p.

Ví dụ 2:

```
int *p;
p = (int *) malloc(50*sizeof(int));
```

Toán tử sizeof xác định kích thước kiểu dữ liệu int

Cấp phát bộ nhớ động

- Kích thước của heap không xác định nên cần kiểm tra giá trị trả về của hàm malloc() để biết việc cấp phát thành công hay không.

Ví dụ:

```
p = (int *)malloc(100);
if(p == NULL)
{
  cout << "Khong du bo nho";
  exit(1);
}
```

Cấp phát bộ nhớ động

- Hàm free(): Trả về vùng nhớ được cấp phát bởi hàm malloc().
- Cú pháp:

```
void free(void *p);
```

p là con trỏ đến vùng nhớ đã được cấp phát trước đó bởi hàm malloc().

Cấp phát bộ nhớ động

Các toán tử của C++

C++ cung cấp hai toán tử cấp phát bộ nhớ động: new và delete.

- Toán tử new cấp phát bộ nhớ và trả về một con trỏ đến byte đầu tiên của vùng nhớ được cấp phát.
- Toán tử delete thu hồi vùng nhớ được cấp phát trước đó bởi toán tử new.

Cấp phát bộ nhớ động

- Cú pháp:

```
p = new type;
delete p;
```

- p là một biến con trỏ nhận địa chỉ của vùng nhớ được cấp phát đủ lớn để chứa 1 đối tượng có kiểu là type

Cấp phát bộ nhớ động

```
#include <iostream.h>
void main(){
      int *p = new int; //allocate space for an int
      if(p==NULL){
         cout<<“loi cap phat”;
         exit(0);
      }
      *p = 100;
      cout << "At " << p << " ";
      cout << "is the value " << *p << "\n"; //Tranh Memory Leak
      if(p!=NULL)  {
            delete p;
            p=NULL;
         }
}
```

Cấp phát bộ nhớ động

- Con trỏ void là con trỏ đặc biệt có thể trỏ đến bất kỳ kiểu dữ liệu nào.
- Cú pháp:

```
void *pointerVariable;
```

Ví dụ:

```
void *p;
p = &a; //p trỏ đến biến nguyên a
p = &f; //p trỏ đến biến thực f
```

Cấp phát bộ nhớ động

- Kiểu dữ liệu khi khai báo biến con trỏ chính là kiểu dữ liệu của ô nhớ mà con trỏ có thể trỏ đến.
- Địa chỉ đặt vào biến con trỏ phải cùng kiểu với kiểu của con trỏ.

Ví dụ:

```
int a; float f;
int *pa; float *pf;
pa = &a; pf = &f; //hợp lệ
pa = &f; pf = &a; //không hợp lệ
```

Cấp phát bộ nhớ động

```
1  #include <iostream.h>
2  int main(int argc, char *argv[])
3  {
4      int a=10;;
5      float f=11.0;
6      int *pa;
7      float *pf;
8
9      void *pV;
10
11     pa=&a;
12     pf=&f;
13
14     pV=&f;
15     cout<<"float PV= "<<*((float*)pV);
16     pV=&a;
17     cout<<"\n int PV= "<<*((int*)pV);
18     return 0;
19 }
```

Cấp phát bộ nhớ động

- Cũng có thể ép kiểu con trỏ về đúng kiểu tương ứng khi dùng trong các biểu thức.

Ví dụ:

- Nếu p đang trỏ đến biến nguyên a, để tăng giá trị của biến a lên 10 ta phải dùng lệnh sau:

```
(int*)*p + 10;
```

- Nếu p đang trỏ đến biến thực f, để tăng giá trị của biến f lên 10 ta phải dùng lệnh sau:

```
(float*)*p + 10;
```

Cấp phát bộ nhớ động

- Một con trỏ không trỏ đến một địa chỉ bộ nhớ hợp lệ thì được gán giá trị NULL
- NULL được định nghĩa trong <cstdlib>

Ví dụ:

```
#include <iostream.h>
void main()
{
        int *p;
        cout <<"Gia tri con tro p tro den la: "<< *p;
}
```

Kết quả của chương trình trên là:

NULL POINTER ASSIGNMENT

Cấp phát bộ nhớ động

➢ Con trỏ và mảng

Giữa mảng và con trỏ có một sự liên hệ rất chặt chẽ:

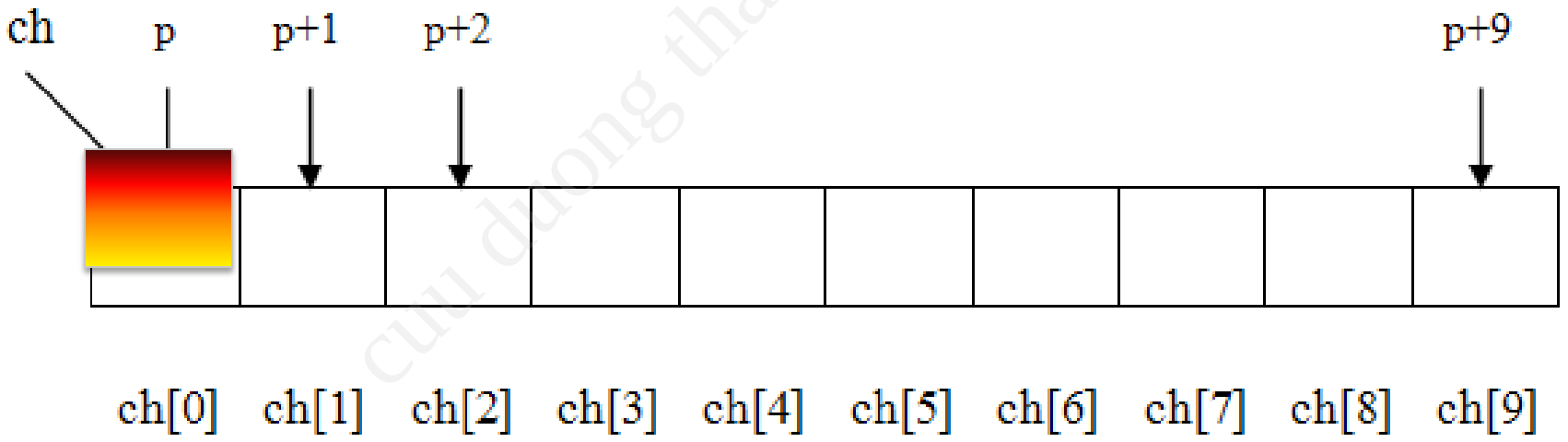
- Những phần tử của mảng được xác định bằng chỉ số trong mảng và cũng có thể được xác định qua biến con trỏ.
- Tên của một mảng tương đương với địa chỉ phần tử đầu tiên của nó, tương tự một con trỏ tương đương với địa chỉ của phần tử đầu tiên mà nó trỏ tới.

Cấp phát bộ nhớ động

- Con trỏ và mảng

Ví dụ:

*char ch[10], *p;*



Cấp phát bộ nhớ động

➢ Con trỏ và mảng

- p được gán địa chỉ của phần tử đầu tiên của mảng ch.

```
p = ch;
```

- Để tham chiếu phần tử thứ 3 trong mảng ch, ta dùng một trong 2 cách sau:

✓ch[2]

✓*(p+2).

Cấp phát bộ nhớ động

➢ Con trỏ và mảng

Truy cập các phần tử mảng bằng con trỏ

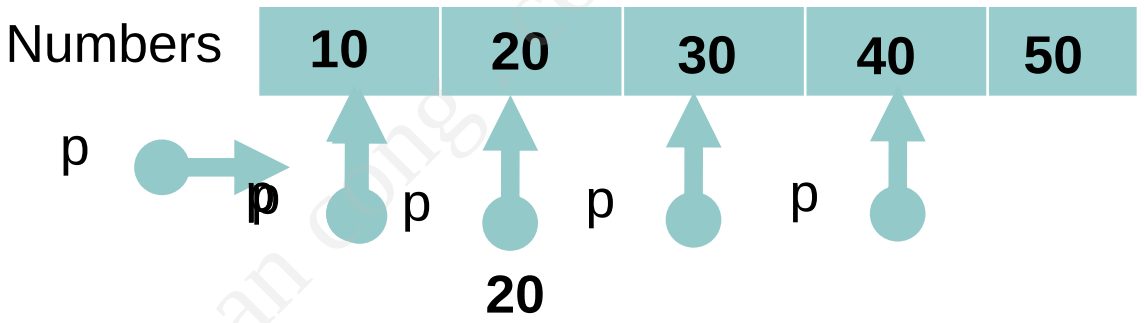
Kiểu mảng	*Kiểu con trỏ*
&<Tên mảng>[0]	<Tên con trỏ >
&<Tên mảng> [<Vị trí>]	<Tên con trỏ> + <Vị trí>
<Tên mảng>[<Vị trí>]	*(< Tên con trỏ > + <Vị trí>)

Cấp phát bộ nhớ động

```
#include <iostream.h>
#include <conio.h>
void main ()
{
    int numbers[5], * p;
    p = numbers;  *p = 10;
    p++;  *p = 20;
    p = &numbers[2];  *p = 30;
    p = numbers + 3;  *p = 40;
    p = numbers;  *(p+4) = 50;
    for (int n=0; n<5; n++)
          cout << numbers[n] << ", ";
}
```

Cấp phát bộ nhớ động

```
int  Numbers[5];
int  *p;
p= Numbers;
*p = 10;
p++;      *p = 20;
p = &numbers[2];     *p = 30;
p = Numbers + 3;     *p = 40;
p = Numbers;         *(p+4) = 50;
```



Cấp phát bộ nhớ động

Ví dụ: Xuất mảng sử dụng con trỏ

```
#include <iostream>
void main()
{
    int a[] = {113,4,7,0,5,6,3,7,8,9};
    int *p;
    p = a;
    for(int i=0 ; i<10 ; i++)
    {
        *(p+i) *= 10; //tuong duong a[i] = a[i]*10
        cout << “a[“ << i << “] = “ << *(p+i) << ”\n”;
    }
}
```

Cấp phát bộ nhớ động

- Mỗi biến con trỏ là một biến đơn. Ta có thể tạo mảng của các con trỏ với mỗi phần tử của mảng là một con trỏ.
- Cú pháp:

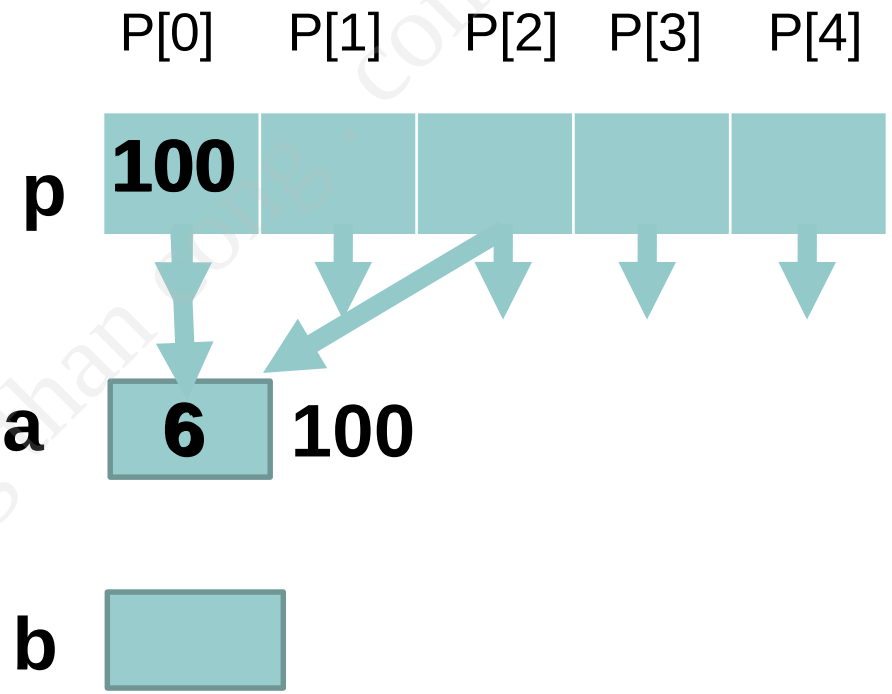
```
type *pointerArray[elements];
```

  - type: kiểu dữ liệu mà các con trỏ phần tử trỏ đến.
  - pointerArray: tên mảng con trỏ.
  - elements: số phần tử của mảng con trỏ.

Cấp phát bộ nhớ động

Ví dụ:

```
int *p[5];
int a=6;
p[0] = &a;
p[2] = p[0];
int b;
b = *p[0];
```



Bài tập

Nhóm tối đa 4 người.

Viết chương trình với các hàm thực hiện các chức năng sau:

a) Hiển thị tên các thành viên của nhóm.

b) Lấy tất cả phần ký số trong MSSV của các thành viên và tính tổng.

c) Xác định đặc tính của tổng này (chẵn/lẻ, nguyên tố, chia hết cho 3).