UIT
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

Data Structures & Algorithms

ÔN TẬP NHẬP MÔN LẬP TRÌNH

Nội dung

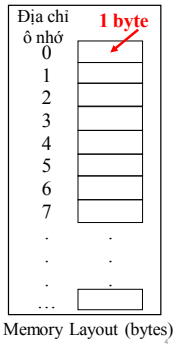
1. Biến
2. Hàm
3. Mảng
4. Cấu trúc
5. Con trỏ

Cấu trúc một chương trình C/C++


```
#include <iostream>
int main ()
{
    std::cout<<"Xin chao";
    return 0;
}
```


Ô nhớ & Vùng nhớ

- Bộ nhớ máy tính
  - Bộ nhớ RAM chứa rất **nhiều ô nhớ**, mỗi ô nhớ có **kích thước 1 byte**.
  - Mỗi ô nhớ **có địa chỉ duy nhất** và địa chỉ này **được đánh số từ 0 trở đi**.
  - RAM để lưu trữ mã **chương trình** và **dữ liệu** trong suốt quá trình thực thi.



Memory Layout (bytes)

1. Biến

- Biến **là một ô nhớ hoặc 1 vùng nhớ** dùng để chứa dữ liệu trong quá trình thực hiện chương trình
- Mỗi biến có một kiểu dữ liệu cụ thể**, kích thước của biến phụ thuộc vào kiểu dữ liệu.**
- Giá trị của biến có thể được thay đổi, **tất cả các bài phải khởi tạo trước khi sử dụng**
- Qui cách đặt tên biến:
  - Không trùng với các từ khóa, hoặc tên hàm.
  - Ký tự đầu tiên là chữ cái hoặc _
  - Không được sử dụng khoảng trắng ở giữa các ký tự
  - *Nên sử dụng tất cả chữ thường với dấu _ giữa các từ.*

```
int so_nguyen;
float so_thuc;
```


1. Biến

❑ Cách khai báo:

type variableNames;

- *type*: là một trong các kiểu dữ liệu hợp lệ.
- *variableNames*: tên của một hay nhiều biến phân cách nhau bởi dấu phẩy.

```
float mark1;
int x;
```

1. Biến

- Ngoài ra, ta có thể vừa khai báo vừa khởi tạo giá trị ban đầu cho biến:

type varName1=value, ... ,varName_n=value;

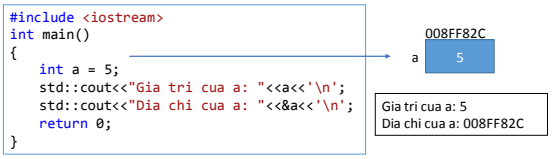
- Ví dụ:

```
float mark1, mark2, mark3, average = 0;
```

1. Biến - địa chỉ biến

- Mỗi 1 biến khi được khai báo sẽ được cấp 1 vùng nhớ với địa chỉ duy nhất để lưu trữ biến đó.
- Để truy cập vào địa chỉ của một biến ta sử dụng câu lệnh. &

```
#include <iostream>
int main()
{
    int a = 5;
    std::cout<<"Gia tri cua a: "<<a<<'\n';
    std::cout<<"Dia chi cua a: "<<&a<<'\n';
    return 0;
}
```




```
Gia tri cua a: 5
Dia chi cua a: 008FF82C
```

1. Biến - địa chỉ biến

Mọi biến đều có 2 thông tin: Giá trị và địa chỉ

int x=5;

cout<<"Giá trị của x="<<x;

cout<<"Địa chỉ của x="<<&x;

1. Biến – biến cục bộ

- Biến được **định nghĩa** trong một hàm hoặc 1 block được gọi là biến cục bộ
- Biến cục bộ chỉ được sử dụng bên trong hàm hoặc block
- Các hàm bên ngoài khác sẽ không truy cập được biến cục bộ

```
#include <iostream>
int main()
{
    int a = 2, b = 3;
    int c;
    c = a + b;
    std::cout<<c;
    return 0;
}
```

Biến a, b, c là các biến cục bộ bên trong hàm main

1. Biến – biến toàn cục

- Biến toàn cục được **định nghĩa bên ngoài các hàm**, và thường được định nghĩa ở phần đầu của source code file.
- Biến toàn cục sẽ giữ giá trị của biến xuyên suốt chương trình.
- **Tất cả các hàm đều có thể truy cập biến toàn cục**

Ví dụ 1

```
#include <iostream>
int g;
int main()
{
    int a = 2, b = 3;
    g = a + b;
    std::cout<<g;
    return 0;
}
```

g = ???

Ví dụ 2

```
#include <iostream>
int g = 20;
int main()
{
    int a = 2, b = 3;
    g = a + b;
    std::cout<<g;
    return 0;
}
```

g = ???

Biến cục bộ VS biến toàn cục

- Khi biến **cục bộ** được định nghĩa, **giá trị của biến sẽ không được khởi tạo.**

→ Ta phải gán giá trị cho biến cục bộ để khởi tạo

- Biến toàn cục sẽ được **tự động khởi tạo** giá trị

Kiểu dữ liệu	Giá trị khởi tạo
int	0
char	'\0'
float	0
double	0
pointer	NULL

1. Biến – Hằng

Cách định nghĩa hằng trong C++: Có 2 cách
Cách 1: Sử dụng **#define**
Câu lệnh:
Ví dụ: `#define tên_hằng giá_trị` Lưu ý: Không có ký tự ;

```
#include <iostream>
#define PI 3.14
int main()
{
    int r = 2;
    std::cout<<2*r*PI;
    return 0;
}
```

6.28

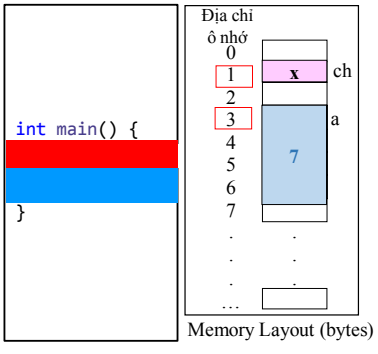
1. Biến – Hằng

Cách định nghĩa hằng trong C++: Có 2 cách
Cách 2: Sử dụng **const**
Câu lệnh:
Ví dụ: `const kiểu_giá_trị tên_hằng = giá_trị_hằng;`

```
#include <iostream>
int main()
{
    const int PI = 3.14;
    int r = 2;
    std::cout<<r*PI;
    getch();
    return 0;
}
```

6.28

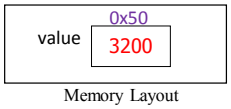
1. Biến – Và Vùng Nhớ



1. Toán tử * và & với Biến - Vùng Nhớ

- Toán tử & (**Address-of Operator**) **đặt trước tên biến** và cho biết địa chỉ của vùng nhớ của biến.
- Toán tử * (**Dereferencing Operator hay Indirection Operator**) **đặt trước một địa chỉ** và cho biết giá trị lưu trữ tại địa chỉ đó.
- Ví dụ:

```
int value;
value = 3200;
```



Memory Layout

```
cout << " value = " << value;
=> value = 3200;

cout << " &value = " << &value;
=> &value = 0x50;

cout << " *(&value) = " << *(&value);
=> *(&value) = 3200;
```

2. Hàm

- 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 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)*

2. Hàm

- Một đoạn chương trình **có tên, đầu vào và đầu ra.**
- Có chức năng **giải quyết một số vấn đề chuyên biệt cho chương trình chính**.
- **Được gọi nhiều lần** với các tham số khác nhau.
- Được sử dụng khi có nhu cầu:
  - *Tái sử dụng.*
  - *Sửa lỗi và cải tiến.*

2. Hàm

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

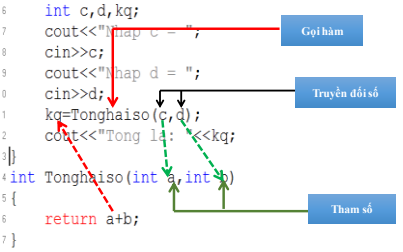
```
returnType functionName(parameterList)
{
   body of the function
   return value
}
```

Tham số và đối số

- **Parameter**
  - tạm dịch: *Tham số* hoặc *tham số hình thức*
  - Là các thông số mà **hàm nhận vào**
  - Xác định khi khai báo hàm
- **Argument**
  - Tạm dịch: *Đối số* hoặc *Tham số thực sự*
  - Là các thông số được **đưa vào hàm** khi tiến hành gọi hàm
- Hai thuật ngữ này đôi khi dùng lẫn lộn và gọi chung là *Tham số*

2. 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;
}
```



2. Hàm

- Chức năng của hàm
- Số lượng tham số, kiểu dữ liệu
- Một số hàm không có đầu vào
- Kiểu dữ liệu đầu ra
- Một số hàm không có đầu ra (void)
- Các lệnh cần thiết để hàm thực hiện công việc

Ví dụ

- Hàm có đầu ra, không có đầu vào:
  - *Tên hàm*: nhap_so_duong , Hàm yêu cầu người dùng nhập vào một số nguyên dương. Nếu không phải số dương yêu cầu nhập lại.
  - Đầu vào: Không có
  - Đầu ra: số nguyên dương.

```
int nhap_so_duong(){
    int n;
    do {
        cout << "Nhap mot so nguyen duong";
        cin >> n;
    } while (n <= 0);
    return n;
}
```

Ví dụ

- Hàm có đầu vào, không có đầu ra:
  - *Tên hàm*: xuat_so_lon, Xuất ra màn hình số lớn hơn trong 02 số.
  - Đầu vào: Hai số nguyên. Đặt tên là a và b
  - Đầu ra: Không có

```
void xuat_so_lon(int a, int b){
    int m;
    if (a > b) m = a;
    else m = b;
    }
    cout << "so lon nhat giua "
         << a << " va " << b << " la " << m;
}
```

Ví dụ

- Hàm không có đầu vào lẫn đầu ra
  - *Tên hàm*: nhap_xuat_so_lon, Yêu cầu nhập vào 02 số nguyên và xuất ra màn hình ước chung lớn nhất của 02 số đó.
  - Đầu vào: Không có
  - Đầu ra: Không có

```
void nhap_xuat_so_lon(){
    int m, n;
    cout << "Nhap so nguyen duong"; cin >> m;
    cout << "Nhap so nguyen duong"; cin >> n;
    cout << "So lon hon trong "
         << m << " va " << n << " la ";
    if (n > m) m = n;
    cout << m;
}
```

Ví dụ

- Hàm có cả đầu vào và đầu ra
  - *Tên hàm*: so_lon, Nhận vào 02 số nguyên dương và trả về số lớn hơn trong 02 số đó.
  - Đầu vào: Hai số nguyên dương, đặt tên m và n
  - Đầu ra: Số nguyên dương có giá trị lớn hơn trong m và n

```
int so_lon(int m, int n){
    if (n > m) m = n;
    return m;
}
```

Trả về giá trị

- Lệnh return dùng để trả về giá trị đầu ra của hàm
- Hàm chỉ trả về được **duy nhất 01 giá trị**. Lệnh return sẽ kết thúc quá trình thực thi của hàm

```
int so_lon(int m, int n){
    if (n > m) return n;
    return m;
}
```

28

Trả về giá trị

- Các hàm không có đầu ra sẽ có **kiểu trả về là void**
- **Không có biến kiểu void**
- Lệnh **return với các hàm không có đầu ra sẽ không kèm theo giá trị (nhưng vẫn sẽ kết thúc việc thực thi hàm)**

```
void xuat_so_lon(int a, int b){
    cout << "so lon nhat giua "
         << a << " va " << b << " la " ;
    if (a > b) {
        cout << a;
        return;
    }
    cout << b;
}
```

29

Thuật ngữ - truyền đối số

- Truyền đối số - **to pass argument** – là công việc **đưa các thông số cho hàm** hoạt động khi gọi hàm.
- Đối số phải được truyền tương ứng với cách tham số đã được khai báo.
- Có 02 cách truyền đối số chính
  - **Pass by value** – Truyền giá trị (truyền tham trị)
  - **Pass by reference** – Truyền tham chiếu

Truyền giá trị

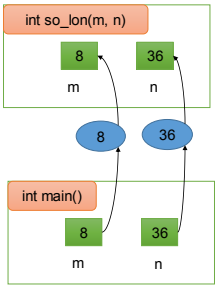
- Là cách mặc định của C/C++
- Tham số chứa bản sao giá trị của đối số. **Thay đổi tham số không ảnh hưởng đến đối số.**

```
int so_lon(int m, int n){
    if (n > m) m = n;
    return m;
}
int main()
{
    int m = 8, n = 36;
    int o = so_lon(m, n);
    cout << "UCLN cua " << m << " va " << n << " la " << o;
}
```

Truyền giá trị

```
int so_lon(int m, int n){
    if (n > m) m = n;
    return m;
}
int main()
{
    int m = 8, n = 36;
    int o = so_lon(m, n);
    cout << "UCLN cua " << m
         << " va " << n << " la " << o;
}
```

- Truyền **giá trị tạo ra bản sao của đối số và lưu vào trong vùng nhớ của tham số**



32

Truyền giá trị

- Có thể truyền **đối số là bất cứ cú pháp nào tính được thành giá trị (hằng, biến, biểu thức, lời gọi, v.v...)**

```
int so_lon(int m, int n){
    if (m > n) n = m;
    return n;
}
int nhap_so_duong(){
    int n; cout << "Nhap mot so nguyen "; cin >> n;
    return n;
}
int main()
{
      cout << "So lon hon la " << so_lon(9*4, nhap_so_duong());
}
```

Truyền giá trị

```
int so_lon(int m, int n){
    if (n > m) m = n;
    return m;
}
int nhap_so_duong(){
    int n;
    do {
        cout << "Nhap mot so nguyen duong";
        cin >> n;
    } while (n <= 0);
    return n;
}
int main()
{
      cout << "so lon nhat trong 04 so la "
           << so_lon(
                so_lon(nhap_so_duong(), nhap_so_duong())
              , so_lon(nhap_so_duong(), nhap_so_duong())
              );
}
```

- Giải quyết vấn đề đặt ra ở đầu bài.

Truyền tham chiếu

- Áp dụng cho các **tham số khi khai báo có dấu & phía sau kiểu dữ liệu**.
- Chỉ có thể truyền các **đối số là biến** (hoặc hằng nếu tham số khai báo là *const*)
- Các tham số là tham chiếu **không được cấp phát vùng nhớ**
  - Tham số được **truyền tham chiếu sẽ trỏ đến cùng địa chỉ vùng nhớ của đối số truyền cho nó**
  - Tham số sẽ trở thành **một ánh xạ đến đối số. Mọi thay đổi lên tham số sẽ thay đổi luôn đối số.**

Truyền tham chiếu

```
void hoan_vi(int& a, int& b){
    int c = a;
    a = b;
    b = c;
}
int main()
{
    int a, b;
    cin >> a >> b;
    hoan_vi(a, b);
    cout << a << " " << b;
}
```

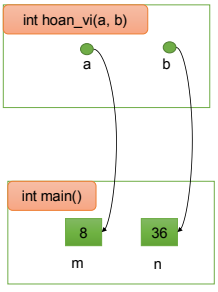
Output:

```
5 3
3 5
```

- Chương trình xuất ra hai số ngược với thứ tự chúng được nhập vào
- Đối số truyền vào bắt buộc phải là biến, không thể dùng hàm nhap_so_duong trong trường hợp này

Truyền tham chiếm

```
void hoan_vi(int& a, int& b){
    int c = a;
    a = b;
    b = c;
}
int main()
{
    int m, n;
    cin >> m >> n;
    hoan_vi(m, n);
    cout << m << " " << n;
}
```



- Truyền tham chiếu **liên kết tham số đến vùng nhớ của đối số**. Tham số **không có vùng nhớ**

Truyền tham chiếu

```
bool phep_chia(int x, int y, double& thuong){
    if (y != 0) {
        thuong = double(x)/y;
        return true;
    } else {
        return false;
    }
}
int main(){
    double thuong;
    if (phep_chia(5, 3, thuong)){
        cout << "Thuong so la " << thuong;
    } else {
        cout << "Khong the chia duoc ";
    }
}
```

- Dùng truyền tham chiếu như một cách trả về kết quả

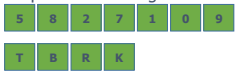
3. Mảng

- Biểu diễn **một dãy các phần tử có cùng kiểu** và mỗi phần tử trong mảng biểu diễn 1 giá trị.
- Kích thước mảng **được xác định ngay khi khai báo và không thay đổi**.
- **Một kiểu dữ liệu có cấu trúc** do người lập trình định nghĩa.
- Ngôn ngữ lập trình C/C++ luôn chỉ định một **khối nhớ liên tục** cho một biến kiểu mảng.

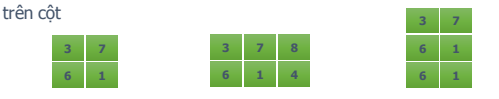
Ví dụ: dãy các số nguyên, dãy các ký tự…

3. Mảng

- Mảng 1 chiều gồm 1 dãy các phần tử có cùng kiểu dữ liệu (int, float, char …)



- Mảng 2 chiều (Ma trận) gồm các phần tử trên dòng và các phần tử trên cột



3. Mảng một chiều

- Cú pháp:

<Kiểu dữ liệu> <Tên biến mảng>[<Số phần tử mảng>];

Trong đó:

Kiểu dữ liệu: int, float, char

Tên biến mảng: 1 ký tự hoặc 1 dãy ký tự viết liền nhau và không có khoảng trắng

Số phần tử mảng: số lượng các phần tử của mảng 1 chiều

char A[10]	**int Mang1Chieu[30]**
Kiểu dữ liệu: char	Kiểu dữ liệu: int
Tên biến mảng: A	Tên biến mảng: Mang1Chieu
Số phần tử mảng: 10 phần tử	Số phần tử mảng: 30 phần tử

- Phải **xác định cụ thể <số phần tử mảng> ngay lúc khai báo**, không được sử dụng biến hoặc hằng thường.

```
int n1 = 10; int a[n1];
const int n2 = 20; int b[n2];
```

- **Nên sử dụng chỉ thị tiền xử lý #define** để định nghĩa số phần tử mảng

```
#define n1 10
#define n2 20
int a[n1];          // int a[10];
int b[n1][n2];      // int b[10][20];
```

- Khởi tạo giá trị cho mọi phần tử của mảng

int A[4] = {29, 137, 50, 4};

0	1	2	3
29	137	50	4

- Khởi tạo giá trị cho một số phần tử đầu mảng

int B[4] = {91, 106};

0	1	2	3
91	106		

- Khởi tạo giá trị 0 cho mọi phần tử của mảng

int a[4] = {0};

0	1	2	3
0	0	0	0

- Tự động xác định số lượng phần tử

int a[] = {22, 16, 56, 19};

0	1	2	3
22	16	56	19

43

- **Chỉ số mảng** (vị trí trong mảng) là một giá trị số nguyên int.
- Chỉ số **bắt đầu là 0** và không vượt quá số lượng phần tử tối đa trong mảng.
- Số lượng các chỉ số mảng = số lượng phần tử tối đa trong mảng

int A[5];

0	1	2	3	4
99	17	50	43	72

Tên mảng: A

Kiểu dữ liệu của từng phần tử trong mảng: int

Số phần tử tối đa trong mảng: 5 phần tử

Các chỉ số được đánh số: 0 → 4 (0, 1, 2, 3, 4)

44

- Truy xuất phần tử mảng thông qua chỉ số

<Tên biến mảng>[<chỉ số mảng>]

- Các phần tử mảng là 1 dãy liên tục có chỉ số từ 0 đến <Số phần tử mảng>-1

int A[4]

0	1	2	3
29	137	50	4

Các truy xuất hợp lệ: A[0], A[1], A[2], A[3]

Các truy xuất không hợp lệ: A[-1], A[4], A[5]

Giá trị các phần tử mảng A[0]=29, A[1]=137, A[2]=50, A[3]=4

45

3. Mảng một chiều

- Cú pháp:

int A[4]

0	1	2	3
29	137	50	4

Địa chỉ các phần tử mạng:

Địa chỉ phần tử thứ 0: &A[0]

Địa chỉ phần tử thứ 1: &A[1]

Địa chỉ phần tử thứ 2: &A[2]

Địa chỉ phần tử thứ 3: &A[3]

46

3. Mảng hai chiều

- Cú pháp:

<Kiểu dữ liệu> <Tên biến mảng>[<Số Dòng>][<Số Cột>];

Trong đó:

Kiểu dữ liệu: int, float, char

Tên biến mảng: 1 ký tự hoặc 1 dãy ký tự viết liền nhau và không có khoảng trắng

Dòng, Cột: số lượng các phần tử mỗi chiều của mảng

char A[10][20]

Kiểu dữ liệu: char

Tên biến mảng: A

Mảng có 10 dòng và 20 cột

int Mang2Chieu[3][5]

Kiểu dữ liệu: int

Tên biến mảng: Mang2Chieu

Mảng có 3 dòng và 5 cột

47

int A[2][4]

	0	1	2	3
0	29	137	50	4
1	5	32	657	97

int B[2][2]

	0	1
0	29	137
1	5	32

int C[2][1]

	0
0	29
1	5

48

- Chỉ số mảng là một giá trị số nguyên int.
- Chỉ số trong mảng 2 chiều gồm chỉ số dòng và chỉ số cột.
  - 0 ≤ chỉ số dòng ≤ số dòng của mảng - 1
  - 0 ≤ chỉ số cột ≤ số cột của mảng - 1

int A[2][3];

	0	1	2
0	2	45	7
1	73	11	187

Tên mảng: A

Kiểu dữ liệu của từng phần tử trong mảng: int

Số phần tử tối đa trong mảng: 2*3=6 phần tử

Các chỉ số được đánh số: Chỉ số dòng: 0, 1

Chỉ số Cột: 0, 1, 2

- Truy xuất phần tử mảng thông qua chỉ số

<Tên biến mảng>[<Chỉ số dòng>][<Chỉ số cột>]

int A[2][3]

	0	1	2
0	29	137	50
1	3	78	943

Các truy xuất hợp lệ: A[0][0], A[0][1],..., A[1][2], A[1][3]

Các truy xuất không hợp lệ: A[-1][0], A[1][4], A[2][0]

Giá trị các phần tử mảng:

A[0][0]=29, A[0][1]=137, A[0][2]=50

A[1][0]=3, A[1][1]=78, A[1][2]=943

4. Struct

- Cú pháp tường minh

```
struct <tên kiểu cấu trúc>
{
      <kiểu dữ liệu> <tên thành phần 1>;
      …
      <kiểu dữ liệu> <tên thành phần n>;
} <tên biến 1>, <tên biến 2>;
```

- Ví dụ

```
struct DIEM
{
      int x;
      int y;
} diem1, diem2;
```

4. Struct – Khai báo biến cấu trúc

- Cú pháp không tường minh

```
struct <tên kiểu cấu trúc>
{
      <kiểu dữ liệu> <tên thành phần 1>;
      …
      <kiểu dữ liệu> <tên thành phần n>;
};
struct <tên kiểu cấu trúc> <tên biến>;
```

- Ví dụ

4. Struct – Khai báo biến cấu trúc typedef

- Cú pháp

```
typedef struct
{
      <kiểu dữ liệu> <tên thành phần 1>;
      …
      <kiểu dữ liệu> <tên thành phần n>;
} <tên kiểu cấu trúc>;
<tên kiểu cấu trúc> <tên biến>;
```

- Ví dụ

Khởi tạo cho biến cấu trúc

- Cú pháp tường minh

```
struct <tên kiểu cấu trúc>
{
      <kiểu dữ liệu> <tên thành phần 1>;
      …
      <kiểu dữ liệu> <tên thành phần n>;
} <tên biến> = {<giá trị 1>,…,<giá trị n>};
```

- Ví dụ

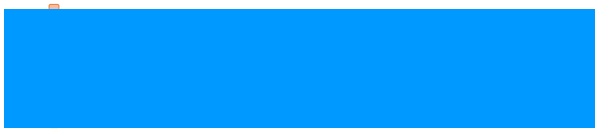
```
struct DIEM
{
      int x;
      int y;
}                    , diem2;
```

Truy xuất dữ liệu kiểu cấu trúc

- Đặc điểm
  - Không thể truy xuất trực tiếp
  - Thông qua toán tử thành phần cấu trúc . hay còn gọi là toán tử chấm (dot operation)

```
<tên biến cấu trúc>.<tên thành phần>
```

Ví dụ



Gán dữ liệu kiểu cấu trúc

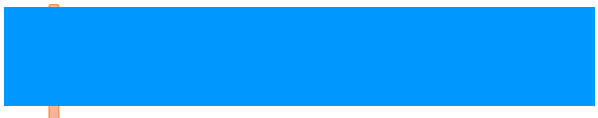
- Có 2 cách

- Ví dụ

```
struct DIEM
{
      int x, y;
} diem1 = {2912, 1706}, diem2;
…
```

Cấu trúc phức tạp

- Thành phần của cấu trúc là cấu trúc khác

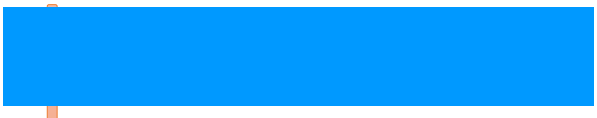


```
struct HINHCHUNHAT
{
```

```
} hcn1;
…
hcn1.traitren.x = 2912;
hcn1.traitren.y = 1706;
```

Cấu trúc phức tạp

- Thành phần của cấu trúc là mảng



```
…
strcpy(sv1.hoten, "Nguyen Van A");
sv1.toan = 10;
sv1.ly = 6.5;
sv1.hoa = 9;
```

Cấu trúc phức tạp

- Cấu trúc đệ quy (tự trỏ)

5. Con trỏ

- **Khái niệm:**

  Con trỏ (**Pointer**) là một **biến lưu trữ địa chỉ** của một **địa chỉ bộ nhớ**. Địa chỉ này thường là địa chỉ của một biến khác.

  VD: Biến x **chứa địa chỉ** của biến y. Vậy ta nói biến x **"trỏ tới"** y.

- Phân loại con trỏ:

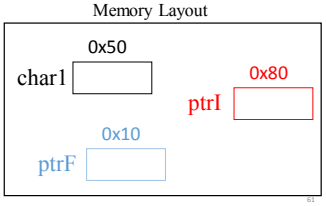
  Con trỏ kiểu int dùng để chứa địa chỉ của các biến kiểu int. Tương tự ta có con trỏ kiểu float, double, …

5. Con trỏ

- Khai báo `<kiểu dữ liệu> *<tên biến con trỏ>;`
  - Giống như mọi biến khác, biến con trỏ muốn sử dụng cũng cần phải được khai báo.

- Ví dụ

```
char char1;
int *ptrI;
float *ptrF;
```



63

5. Con trỏ

Khai báo:

DataType * PointerVariable;

Cấp phát:

PointerVariable = new DataType;

Hủy bộ nhớ:

delete PointerVariable ;

62

5. Con trỏ

```
int *p;
p = new int; // allocate space for an int
*p = 100;
cout << "At " << p << " ";
cout << "is the value " << *p << "\n";
delete p;
```

63

5. Con trỏ

- Có thể **gán biến con trỏ**:

```
int *p1, *p2;
p2 = p1;
```

⇒ Gán một con trỏ cho con trỏ khác

⇒ "Chỉ định p2 trỏ tới nơi mà p1 đang trỏ tới"

Dễ bị lẫn với: *p2 = *p1;

⇒ Gán "giá trị trỏ bởi p1" cho "giá trị trỏ bởi p2"

64

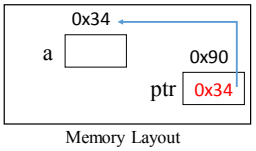
5. Con trỏ và toán tử * ,&

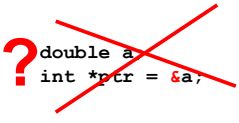
- Toán tử **&** dùng trong khởi tạo giá trị cho con trỏ

```
<kiểu dữ liệu> *<tên biến con trỏ> = & <tên biến>;
```

- Ví dụ:

```
int a;
int *ptr = &a;
```





65

5. Con trỏ NULL

- Khái niệm
  - Con trỏ NULL là con trỏ không trỏ vào đâu cả.
  - Khác với con trỏ chưa được khởi tạo.

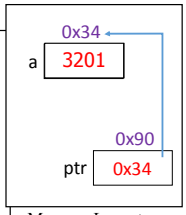
```
int n;
int *p1 = &n;
int *p2;  // unreferenced local variable
int *p3 = NULL;
```

66

5. Con trỏ và toán tử * ,&

- **Toán tử * đặt trước biến** con trỏ cho phép **truy xuất đến giá trị ô nhớ mà con trỏ trỏ đến**.
- Ví dụ

```
int a = 1000;
int *ptr = &a;
cout << ptr << " " << *ptr;
// a = 3200
*ptr = 3200;
cout << *ptr;
(*ptr) ++;
```



Memory Layout

67

5. Con trỏ và các lưu ý

- Con trỏ là **khái niệm quan trọng và khó nhất** trong C++. Mức độ thành thạo C++ được đánh giá qua mức độ sử dụng con trỏ.
- **Nắm rõ quy tắc sau**, ví dụ int a, int *pa = &a;
  - *pa và a đều chỉ nội dung của biến a.
  - pa và &a đều chỉ địa chỉ của biến a.
- Không nên sử dụng con trỏ khi chưa được khởi tạo. Kết quả sẽ không lường trước được.

  int *pa; *pa = 1904; => sai

68

5. Con trỏ và cấp phát động

- Vì **con trỏ có thể tham chiếu tới biến nhưng không thực sự cần phải có định danh cho biến đó.**
- Có thể cấp phát động cho biến con trỏ bằng toán tử new. Toán tử new sẽ tạo ra biến "không tên" cho con trỏ trỏ tới.
- Cú pháp: `<type> *<pointerName> = new <type>`

Ví dụ: `int *ptr = new int;`

  - Tạo ra một biến "không tên" và gán ptr trỏ tới nó
  - Có thể làm việc với biến "không tên" thông qua *ptr

69

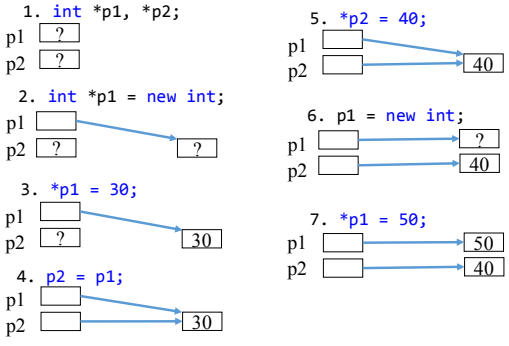
5. Con trỏ và cấp phát động

- Cú pháp khởi tạo giá trị con trỏ: `<type> pointer = new <type> (value)`
- Ví dụ:

```
#include <iostream>
using namespace std;

int main() {
    int *p;
    p = new int(99); // initialize with 99
    cout << *p; // displays 99
    return 0;
}
```

70



71

5. Con trỏ và cấp phát động

- Toán tử delete dùng để giải phóng vùng nhớ trong HEAP do con trỏ trỏ tới (con trỏ được cấp pháp bằng toán tử new). Cú pháp:

  delete <pointerName>;

- Ghi chú: Sau khi gọi toán tử delete thì con trỏ vẫn trỏ tới vùng nhớ trước khi gọi hàm delete. Ta gọi là "con trỏ lạc". Ta vẫn có thể gọi tham chiếu trên con trỏ, tuy nhiên:
  - Kết quả không lường trước được
  - Thường là nguy hiểm

  ⇒ Hãy tránh con trỏ lạc bằng cách gán con trỏ bằng NULL sau khi delete.
- Ví dụ:

```
delete pointer;
pointer = NULL;
```

72

5. Định nghĩa kiểu dữ liệu con trỏ

- Có thể **đặt tên cho kiểu dữ liệu con trỏ**
- Để có thể khai báo biến con trỏ như các biến khác
  - Loại bỏ * trong khai báo con trỏ

  Ví dụ: `typedef int* IntPtr;`
  - Định nghĩa một tên khác cho kiểu dữ liệu con trỏ
  - Các khai báo sau tương đương:

```
IntPtr p;
int *p;
```

73

5. Con trỏ và hàm

- Con trỏ **là kiểu dữ liệu hoàn chỉnh có thể dùng nó như các kiểu khác**
- **Con trỏ có thể là tham số của hàm**
- Có thể là **kiểu trả về của hàm**

  Ví dụ: `int* findOtherPointer(int* p);`

  Hàm này khai báo:
  - Có tham số kiểu con trỏ trỏ tới int
  - Trả về biến con trỏ trỏ tới int

74

5. Con trỏ và cấp phát động

- Ví dụ:

```
int a[10];
typedef int* IntPtr;
IntPtr p;
```

⇒ a và p là các biến con trỏ.
- Phép gán hợp lệ `p = a;`
  - p bây giờ sẽ trỏ tới nơi a trỏ, tức là tới phần tử đầu tiên của mảng a
- Phép gán không hợp lệ `a = p;`
  - Bởi con trỏ mảng là con trỏ hằng.

75

5. Con trỏ và cấp phát động

- Hạn chế của mảng chuẩn
  - Bắt buộc phải biết trước cần bao nhiêu bộ nhớ lưu trữ => tốn bộ nhớ, không thay đổi được kích thước, ...

⇒ **Dùng Mảng động**

- Mảng động
  - **Kích thước không xác định ở thời điểm lập trình**
  - **Mà xác định khi chạy chương trình**

76

5. Con trỏ và cấp phát động

- Cấp phát động **cho biến con trỏ**
- Sau đó dùng con trỏ như mảng chuẩn
- Cú pháp:

```
<type> <pointer> = new <type> [<number_of_elements>]
```

Ví dụ:

```
typedef double * doublePtr;
doublePtr d;
d = new double[10];
```

⇒ Tạo biến mảng cấp phát động d có 10 phần tử, kiểu cơ sở là double.

77

Xóa mảng động

- Dùng toán tử `delete[]` để xóa mảng động.

Ví dụ:

```
double *d = new double[10];
//... Processing
delete[] d;
```

⇒ Giải phóng tất cả vùng nhớ của mảng động này

⇒ Cặp ngoặc vuông báo hiệu có mảng

⇒ Nhắc lại: d vẫn trỏ tới vùng nhớ đó. Vì vậy sau khi `delete`, cần gán `d = NULL;`

78

Hàm trả về kiểu mảng

- Ta không được phép trả về kiểu mảng trong hàm.

Ví dụ:

```
int[] someFunction(); // Không hợp lệ!
```

- Có thể thay bằng trả về con trỏ tới mảng có cùng kiểu cơ sở:

```
int* someFunction(); // Hợp lệ!
```

Bài tập

- Hãy viết HÀM tạo mảng 1 chiều có n phần tử bằng cấp phát động.
- Viết hàm xuất mảng 1 chiều đã tạo.
- Viết hàm đếm số phần tử âm trong mảng 1 chiều.

Lời giải

```
#include <iostream>
using namespace std;

int* Input(int n){
     int *p;
     p = new int[n];
     for (int i = 0; i < n; i++){
          cin >> p[i];
     }
     return p;
}

int main() {
     int *arr, n;
     cout << "Nhap n: ";
     cin >> n;
     arr = Input(n);
}
```


Lời giải

```
// Hàm xuất mảng
void Output(int *p, int n) {
     cout << "\n Xuat mang 1 chieu: ";
     for (int i = 0; i < n; i++) {
          cout << p[i] << " " ;
     }
}
```


5. Mảng động hai chiều

- Là mảng của mảng
- Sử dụng định nghĩa kiểu con trỏ giúp hiểu rõ hơn:

```
typedef int* IntArrayPtr;
IntArrayPtr *m = new IntArrayPtr[3];
```

⇒ Tạo ra mảng 3 con trỏ

⇒ Sau đó biến mỗi con trỏ này thành mảng 4 biến int

```
for (int i = 0; i < 3; i++)
     m[i] = new int[4];
```

⇒ Kết quả là mảng động 3 x 4

Bài tập

Tạo mảng 2 chiều bằng con trỏ.

Lời giải

```
int main() {
    int row=2, col=3;

    // Cấp phát vùng nhớ
    int **p = new int*[row];
    if (p == NULL) exit(1);
    for (int i = 0; i < row; i++) {
        p[i] = new int[col];
        if (p[i] == NULL) exit(1);
    }

    // Giải phóng vùng nhớ
    for (int i = 0; i < row; i++) {
         delete[] p[i];
    }
    delete[] p;
    return 0;
}
```

85

5. Con Trỏ và Hàm Số

- Tham số của hàm là 1 biến con trỏ
  - Trường hợp thay đổi giá trị của đối số

```
void Hoanvi(int *x, int *y)
{
        int z = *x;
        *x=*y;
        *y=z;
}
void main()
{
        int a=1,b=2;
        cout<<a<<b; // 1 2
        Hoanvi(&a,&b);
        cout<<a<<b; // 2 1
}
```

86

5. Con Trỏ và Hàm Số

- Tham số của hàm là 1 biến con trỏ
  - Trường hợp không thay đổi giá trị của đối số

```
void Capphat(int *a)
{       a = new int[5];
        for(int i=0; i<5; i++)
        {       a[i]=i+1;
                cout<< a[i];
        }
}
void main()
{       int n=5;
        int *b = &n;
        cout<<*b; // 5
        Capphat(b); // 1 2 3 4 5
        cout<<*b; // 5
}
```

87

5. Con Trỏ và Hàm Số

- Kiểu trả về của hàm là 1 con trỏ

```
int* GetArray()
{
        int* a = new int [5];
        for(int i=0; i<5; i++)
                a[i]=i+1;
        return a;
}
```

88

5. Con Trỏ và cấu trúc

- Truy xuất các các thuộc tính dùng con trỏ:

  tên_biến_con_trỏ → tên_thuộc_tính
- Ví dụ cấu trúc phân số

```
struct PhanSo
{       int TuSo;
        int MauSo;
};
PhanSo x;
x.Tuso=1; x.MauSo=2;
PhanSo *p, *q;
p = &x;
P→Tuso=3; p→ MauSo=4; // x(3,4)

q = new PhanSo();
q->TuSo = 1;  // giống (*q).TuSo=1
q->MauSo = 2;// giống (*q).MauSo=2
```

89

5. Con Trỏ và cấu trúc

- Cấu trúc đệ quy (tự trỏ)

```
struct [          ]
{
        char hoten[30];
        struct PERSON *father, *mother;
};

struct [          ]
{
        int value;
        struct NODE *pNext;
};
```

Bài tập

- Hãy tóm tắt lại các **kiến thức quan trọng mà em nghĩ là cần thiết** của môn NMLT về các thành phần sau
  - Biến.
  - Hàm.
  - Mảng.
  - Cấu trúc.
  - Con trỏ.