C Language & Affect*

Phạm Anh Tuấn

Posts and Telecommunications Institute of Technology

Km10, Đường Nguyễn Trãi, Q.Hà Đông, Hà Nội

2020
Feb

Tóm Tắt

Ngôn ngữ C lần đầu được thiết kế bởi Dennis Ritchie tại phòng thí nghiệm Bell Telephone năm 1972. Được phát triển dựa theo ngôn ngữ B nhằm mục đích chính là xây dựng hệ điều hành UNIX. Chính vì thế nó không đem lại sự tiện dụng cho người lập trình. Nhưng C là ngôn ngữ **mạnh** và **linh hoạt** nên nhanh chóng trở thành ngôn ngữ phổ biên được lập trình viên viết nhiều loại ứng dụng ở các mức độ khác nhau. Ngày nay có nhiều ngôn ngữ lập trình mạnh mẽ và tính ứng dụng cao hơn, tuy nhiên nhờ những đặc tính nổi bật của nó nên C là lựa chọn tốt trong việc học và sau này là lập trình hướng đối tượng nơi mà ngôn ngữ C++ hỗ trợ rất nhiều mà C làm nền tảng.

*Một bài phân tích, đánh giá và cảm nhận nhỏ trong quá trình tự học C

1 Các vấn đề sơ khai

1.1 Biến, Hằng và Các kiểu dữ liệu liên quan

Đối với những người mới bắt đầu thì lập trình trên Console Window có thể coi là bước nhập môn bắt buộc. Bỏ qua GUI[1], làm việc trên Console đưa cho ta góc nhìn về bản chất nhất. Và những "Nhân vật " chủ chốt phải kể đến là **Biến** .

Biến rất dễ hiểu như là "cái hộp" dùng để chứa những dữ liệu. Khác với biến x,y,z... trong toán học. Những chiếc hộp này được xác định ở một vị trí mà máy tính tự động chỉ thị. **Hằng** là một loại giống biến, chỉ khác dữ liệu & địa chỉ của nó được bạn khai báo và xác định trước. Nói cách khác, đây là một cái hộp không trống

Và trong lập trình, ta luôn tập trung đến việc "Làm ít, hiệu quả tối ưu". Môi trường lập trình mà ta làm việc không thể như vũ trụ luôn giãn nở được, để tránh lượng dư thừa không cần thiết thì những cái hộp chứa dữ liệu ấy không được quá lớn hay quá nhỏ so với dữ liệu mà nó phải chứa. Định nghĩa kiểu dữ liệu cho kích thước biến cũng hình thành ra

- Kiểu số nguyên

Type	Kích thước	Phạm vi giá trị
char	1 byte	-128 to 127 hoặc 0 to 255
unsigned char	1 byte	0 tới 255
signed char	1 byte	-128 tới 127
int	2 or 4 bytes	-32,768 tới 32,767 hoặc -2,147,483,648 tới 2,147,483,647
unsigned int	2 or 4 bytes	0 tới 65,535 hoặc 0 tới 4,294,967,295
short	2 bytes	-32,768 tới 32,767
unsigned short	2 bytes	0 tới 65,535
long	8 bytes	-9223372036854775808 tới 9223372036854775807
unsigned long	8 bytes	0 tới 18446744073709551615

[1]Giao diện đồ họa người dùng, được sử dụng khá rộng rãi trong ngành CNTT hiện nay. Đây như là một cách giao tiếp với máy tính bằng hình ảnh và chữ viết thay vì các dòng lệnh đơn thuần, giúp đưa ra những trải nghiệm trực quan

- Kiểu số nguyên

Type	Kích thước	Phạm vi giá trị	Độ chính xác
float	4 byte	1.2E-38 to 3.4E+38	6 chữ số thập phân
double	8 byte	2.3E-308 to 1.7E+308	15 chữ số thập phân
long double	10 byte	3.4E-4932 to 1.1E+4932	19 chữ số thập phân

- Kiểu ký tự

Type	Kích thước	Phạm vi giá trị
char or signed char	1 byte	-128 to 127
unsigned char	1 byte	0 to 255

1.2 Nhập & Xuất - Nơi khiến dữ liệu được tổ chức và trực quan hoá

- Hàm nhập 'scanf()':
  - Được dùng để đọc các kiểu dữ liệu mà người dùng nhập từ bàn phím.
  - Hàm 'scanf()' nhận vào tham số là **địa chỉ** của biến chứ không đơn thuần là giá trị của biến. Thế nên có hai thành phần mà ta nên chú ý là địa chỉ và giá trị khi dùng hàm 'scanf()
  - Cấu trúc cơ bản: scanf("<Kiểu giả trị biến>", &<Tên biến>)[2];
- Hàm xuất 'printf()':
  - Dùng để in ra các ký tự, chuỗi ... hiển thị lên màn hình console.
  - Cấu trúc cơ bản của hàm printf("Tên bạn muốn hiển thị trên cửa sổ + %<kiểu dự liệu bạn muốn trích ra từ biến>",<tên gọi các biến được truyền vào> ....)
  - Các ký tự biểu thị các định dạng kiểu dữ liệu:

[2]Dấu '&' trong hàm scanf() dùng để chỉ địa chỉ của biến

Kiểu dữ liệu	Định dạng
`int`	`%d`
`char`	`%c`
`float`	`%f`
`double`	`%lf`
`short int`	`%hd`
`unsigned int`	`%u`
`long int`	`%li`
`long long int`	`%lli`
`unsigned long int`	`%lu`
`unsigned long long int`	`%llu`
`signed char`	`%c`
`unsigned char`	`%c`
`long double`	`%Lf`

1.3 Toán tử

Một loạt các phép tính mà máy tính dùng để tác động **trực tiếp** lên các biến. Nó khá giống các phép toán thông thường nhưng được bổ xung thêm một số toán tử đặc trưng trong lập trình

1.3.1 Toán tử số học (Arithmetic Operators)

Còn được gọi là **toán tử 2 ngôi** khi nó được dùng để tham gia các phép toán với sự tham gia của 2 giá trị

Toán tử	Ý nghĩa
+	phép toán cộng
-	phép toán trừ
*	phép toán nhân
/	phép toán chia
%	phép toán lấy số dư(chỉ áp dụng cho số nguyên)

1.3.2 Toán tử tăng giảm

- Toán tử '++': Tăng giá trị biến lên 1 đơn vị
- Toán tử '–': Giảm giá trị biến đi 1 đơn vị

1.3.3 Toán tử gán (Assignment Operators)

Để gán giá trị của 1 biến trong lập trình

Toán tử	Ý nghĩa
+	phép toán cộng
–	phép toán trừ
*	phép toán nhân
/	phép toán chia
%	phép toán lấy số dư(chỉ áp dụng cho số nguyên)

1.3.4 Toán tử quan hệ (Relational Operators)

Dùng để **kiểm tra** mối quan hệ giữa các toán hạng từ đó trả về giá trị 'TRUE' hoặc 'FALSE'

Toán tử	Ý nghĩa	Ví dụ
==	so sánh bằng	`7 == 3` cho kết quả là 0
>	so sánh lớn hơn	`5 > 1` cho kết quả là 1
<	so sánh nhỏ hơn	`5 < 2` cho kết quả là 0
!=	so sánh khác	`5 != 4` cho kết quả là 1
>=	lớn hơn hoặc bằng	`8 >= 3` cho kết quả là 1
<=	nhỏ hơn hoặc bằng	`5 <= 0` cho kết quả là 0

1.3.5 Toán tử Logic (Logical Operators)

Gồm những toán tử với các công dụng sau:

- Toán tử '&&': là toán tử AND, trả về 'TRUE' khi và chỉ khi tất cả các toán hạng đều đúng.
- Toán tử '||': là toán tử OR, trả về 'TRUE' khi có ít nhất 1 toán hạng đúng.
- Toán tử '!': là toán tử NOT, phủ định giá trị của toán hạng.

1.3.6 Các toán tử khác

- Toán tử Bit
- Toán tử Comma Operator
- Toán tử sizeof Operator

2 Cấu trúc điều khuyển

Cấu trúc điều khuyển hiểu đơn giản là tập các câu lệnh giúp điều phối luồng chạy dữ liệu và xử lý dữ liệu biến trong lập trình bên cạnh các yếu tố then chốt khác như: Hàm, Mảng, Chuỗi và Con trỏ ... Để dễ hình dung và đỡ lẫn lộn giữa cấu trúc điều khuyển nói chung hay các câu lệnh nói riêng với toán tử là "**Câu Lệnh** có thể biểu diễn dưới dạng lưu đồ thuật toán còn **Toán Tử** là cách xử lý có trong lưu đồ thuật toán."

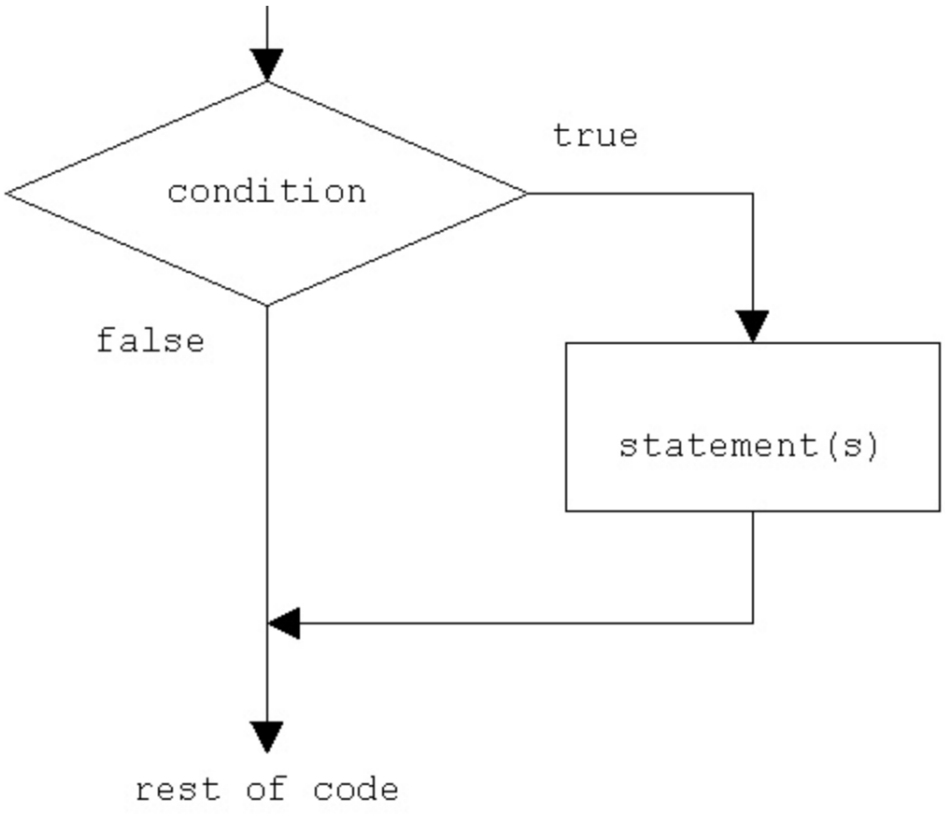
Rẽ nhánh

2.1 Cấu lệnh điều khuyển If

Là thành phần được sử dụng gần như trong mọi chương trình phần mềm, thế nên việc nắm rõ được khối điều khuyển này sẽ giúp bạn làm chủ chương trình.

Câu lệnh có cấu trúc và lưu đồ thuật toán được biểu diễn như sau:

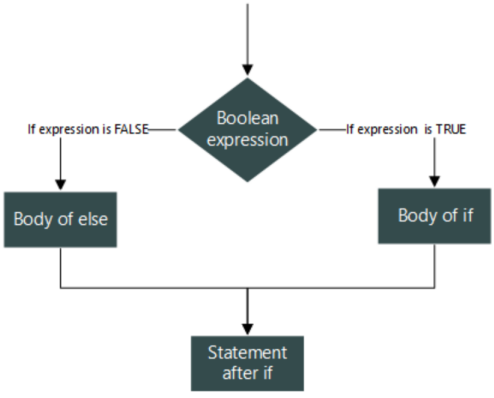
```
if (<Dieu kien>){
    // Khoi lenh se duoc thuc hien neu <Dieu Kien> dung
}
```



2.2 Câu lệnh điều khuyển If ... Else

Câu lệnh và lưu đồ thuật toán[3] của khối lệnh này

```
if (<Dieu kien>){
   <khoi lenh se thuc hien dieu kien dung>
}else {
   <Khoi lenh se thuc hien ney dieu kien sai>
}
```

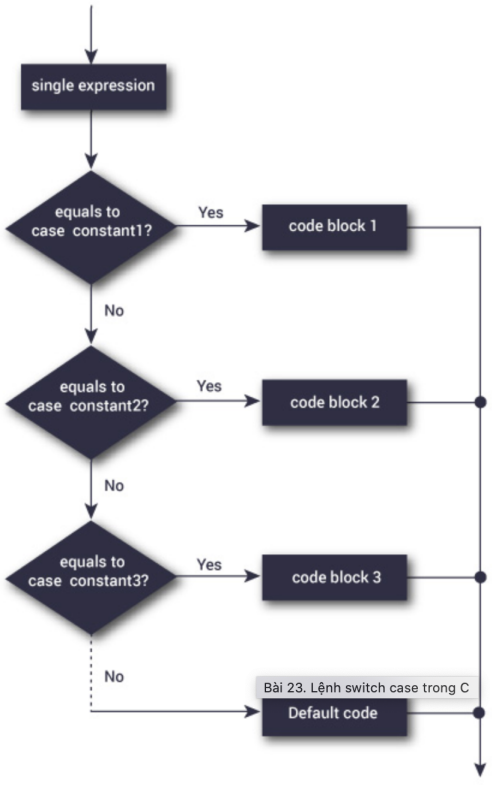


[3]Lưu đồ thuật toán (Flow Chart): là một sơ đồ có các khối và các đường dẫn chỉ định

2.3 Cấu lệnh điều khuyển Switch - Case

Tương tự về cấu trúc như If-else tuy nhiên việc sử dụng Switch-case đem lại khả năng dễ viết và đọc hơn. Nếu để ý kĩ hơn một chút, cấu trúc If-else mang xu hướng "Lọc" dữ liệu lớn đầu vào trong hầu hết các bài toán còn cấu trúc **Switch** mang tính lựa chọn, dựa vào yêu cầu người dùng nhập vào nhiều hơn từ đó xử lý bài toán theo các hàm mà người dùng đã định nghĩa trước (giống như việc ta sử dụng máy tính cầm tay để nó tính toán các số vậy).

```
switch (expression){
    case constant1:
      // statements
      break;
    case constant2:
      // statements
      break;
    . . .
    default:
      // default statements
}
```



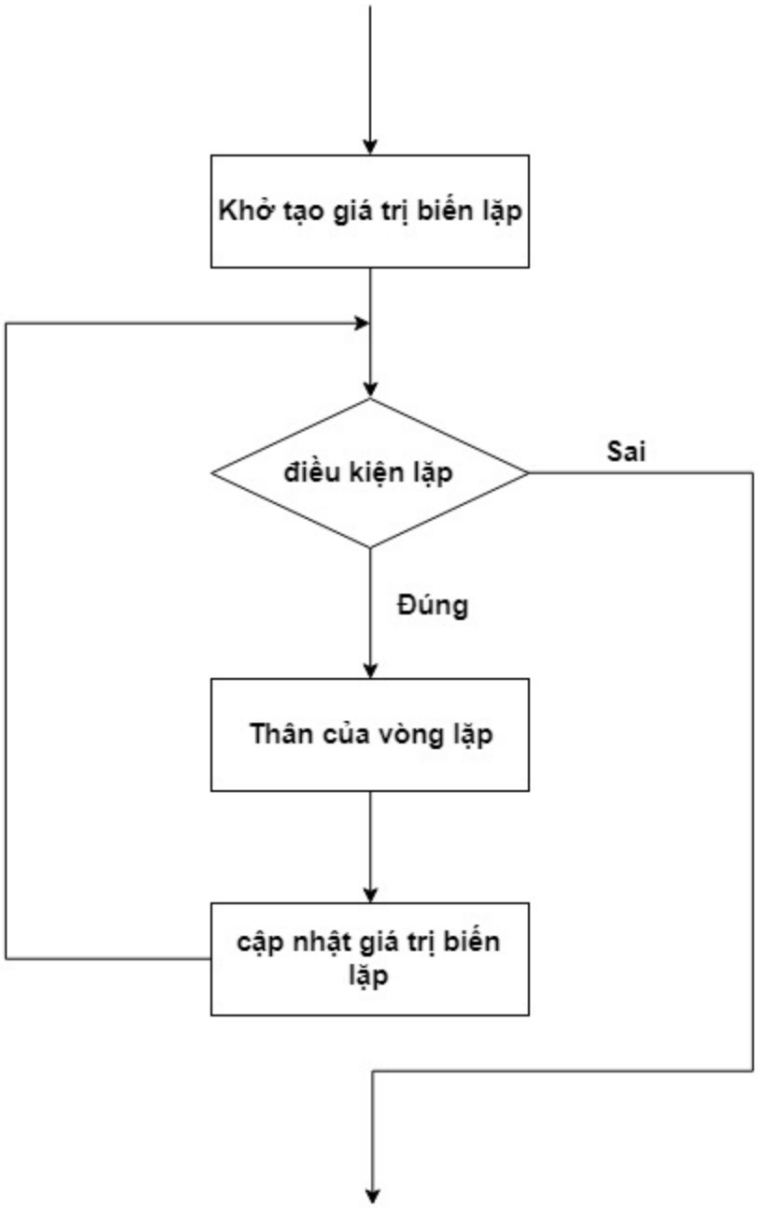
Vòng lặp

Tương tự như cấu trúc rẽ nhánh, vòng lặp được sử dụng khá nhiều trong việc xây dựng và phát triển phần mềm. Đặc trưng của **vòng lặp** là giúp chúng ta giải quyết các công việc lặp lại chỉ bằng những dòng code rất gọn

2.4 Vòng lặp For

Câu lệnh và lưu đồ thuật toán

```
for(<khoi tao gia tri bien lap>;<Dieu kien lap>;<cap nhat bien>){
   <khoi lenh>
}
```

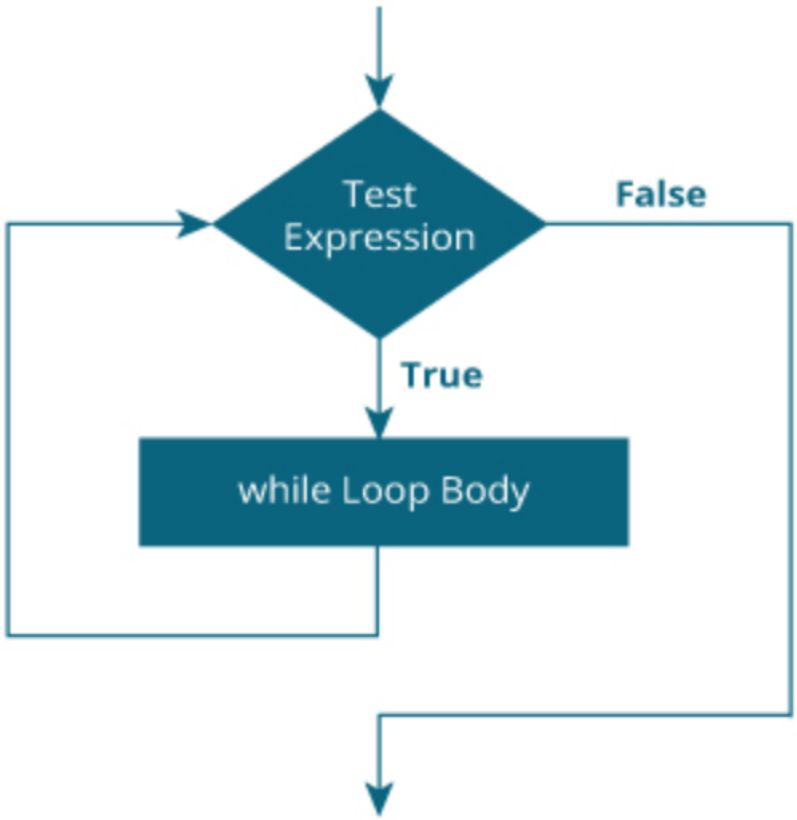


2.5 Vòng lặp While / Do ... While

Tương tự như vòng lặp For về điều hướng cấu trúc lệnh, tuy nhiên vòng lặp While/Do..While rất thích hợp cho việc lặp mà số lần không biết trước hay không xác định được. Điểm này thật sự cần lưu ý bởi nếu không cẩn thận sẽ dễ bị vòng lặp vô tận nếu điều kiện bên trong câu lệnh lặp không hợp lý

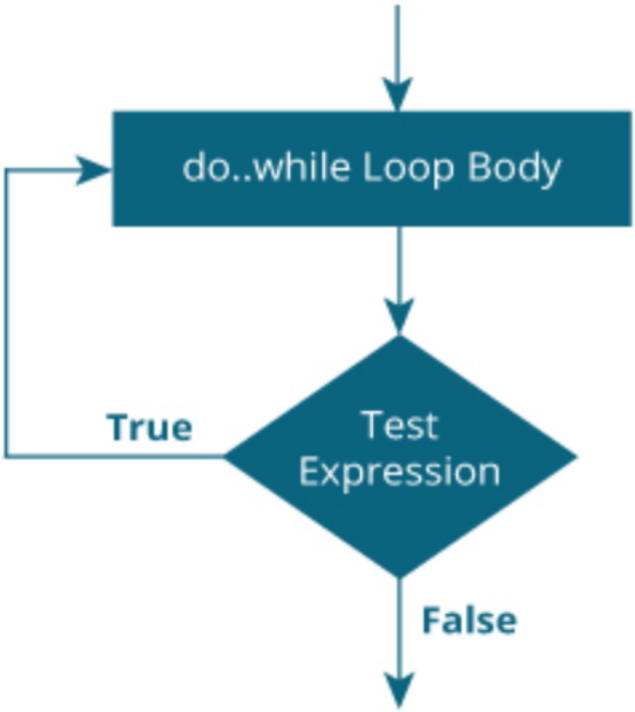
- Vòng lặp **While**:
  Đặc điểm của vòng lặp này là điều kiện được kiểm tra trước rồi mới khởi chạy câu lệnh

```
while (testExpression){
    // statements inside the body of the loop
}
```



- Vòng lặp **Do..While**
  Đặc điểm của vòng lặp này là câu lệnh được thực thi trước khi điều kiện được kiểm tra.

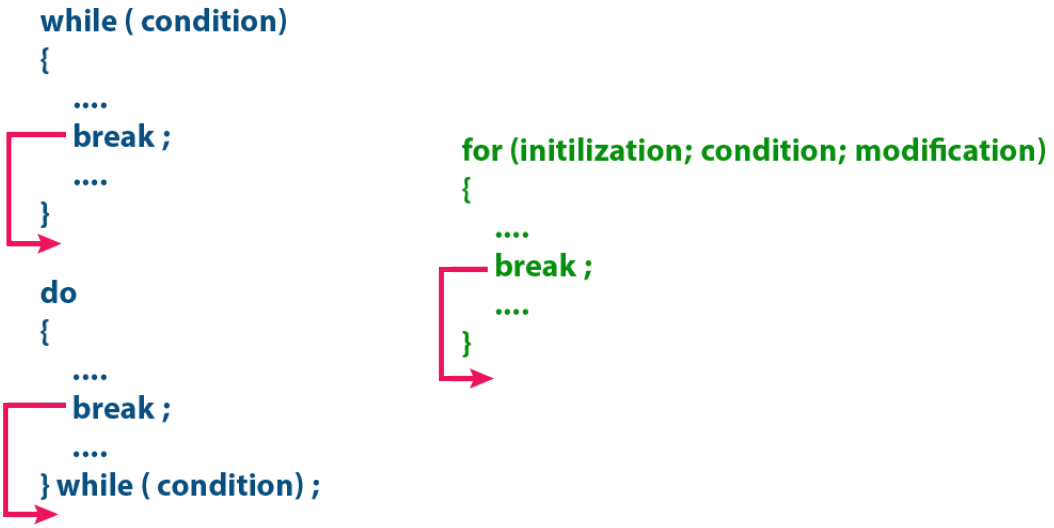
```
do{
... <khoi lenh>
}whlie(<dieu kien vong lap>)
```



Lệnh Break & Continue [4]

Việc sử dung những lệnh này cho phép chúng ta quản lý và làm việc với vòng lặp hiệu quả hơn. Đặc điểm của những câu lệnh này là nó thường xuất hiện khi được "bao bọc" bởi khối lệnh "if" bởi nếu không có lệnh "if" thì vòng lặp trở nên vô dụng

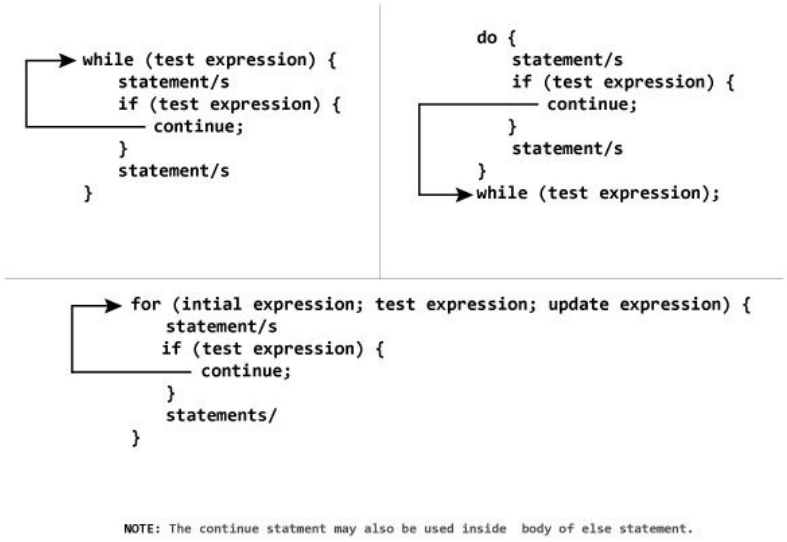
- Lệnh Break:
  Về cơ bản một vòng lặp khi thực thi nếu gặp lệnh break sẽ thoát vòng lặp ngay lập tức. Sơ đồ mô tả hoạt động



[4] Đây là những lệnh kiểm soát vòng lặp

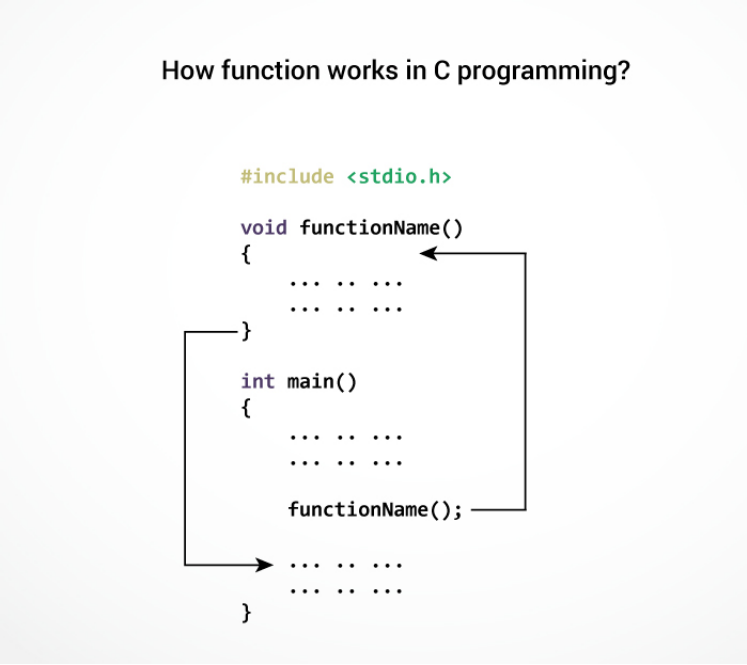
```
#include <stdio.h>

int main(){
    int number;
    printf("\nNhap number = ");
    scanf("%d", &number);

    bool allEven = true; // gia su ban dau la dung
    int last;
    while(number > 0){
        last = number % 10; // Lay chu so cuoi cua number

        if(last % 2 == 1){
            allEven = false;
            break; // thoat vong lap
        }
        number /= 10; // bo chu so cuoi cua Number
    }

    if(allEven){
        printf("\nToan chu so chan!");
    }else{
        printf("\nCo chu so le!");
    }

}
```

- Lệnh Continue:
  Một vòng lặp đang chạy mà gặp lệnh **"continue"**, tất cả các lệnh trong thân vòng lặp nằm phía dưới lệnh **"continue"** sẽ bị bỏ qua ở lần lặp hiện tại chuyển qua điều kiện và tiếp tục lặp (Nếu điều kiện lặp còn thoả mãn). Sơ đồ mô tả hoạt động:



3 Hàm [5]

Vẻ đẹp cũng như tính **quản lý** là điều tuyệt vời trong lập trình. Điều này luôn được ưu tiên khi mà bạn luôn phải làm việc với hàng trăm hay hàng nghìn dòng Code nên rất khó để kiểm soát cũng như tìm và sửa lỗi nếu cứ gõ lệnh tràn lan. Hàm trong C được ra đời với mục đích giải quyết vấn đề đó, tương tự như trong công ty, luôn có những phân ban với những mục đích cụ thể, giúp toàn bộ hệ thống hoạt động trơn chu. Khái quát chung về cách hoạt động của hàm:



[5] Bản chất thư viện khi ta đưa vào cũng là loại hàm được xây dựng sẵn. Như thư viện stdio.h chứa các hàm vào ra chuẩn

Phân loại các loại hàm

3.1 Hàm không tham số đầu vào & Không có giá trị trả về (Void)

3.2 Hàm có tham số & Không có giá trị trả về (Void)

3.3 Hàm không tham số & Có giá trị trả về

3.4 Hàm có tham số & Có giá trị trả về

Hàm đệ quy

Một hàm khi bản thân nó tự gọi lại chính nó thì được gọi là hàm đệ quy. Ví dụ mẫu về cách hoạt động hàm đệ quy

```
void recurse(){
      . . .
      recurse();
      . . .
}
int main()
```

Xem xét qua câu lệnh trên ta có thể thấy hàm đệ quy mang tính lặp như các cấu trúc for / while đã đề cập trước đó. Điều đó đồng nghĩa với việc phải để ý đến điều kiện để thoát vòng lặp, nếu không vòng lặp vô tận sẽ dễ xảy ra và bạn chắc chắn không muốn điều này

Phạm vi của biến

Cũng là một phần nội dung con liên quan đến **hàm**. Như đã đề cập từ nội dung trước, biến là một phần khá quan trọng trong việc giải các bài toán lập trình hay nói đúng hơn lập trình hầu như chỉ có ý nghĩa khi ta làm việc với biến. Các loại biến có giá trị và phạm vi sử dụng ở môi trường hay hàm khác nhau đối với từng loại ...

- Biến cục bộ (Global Variable)
  Biến cục bộ chỉ tồn tại và chỉ có thể sử dụng bên trong khối code đó trong khi khối

code đó đang thực thi đồng nghĩa là ta không dùng biến đó bên ngoài khối code được .
Khối code được hiểu là thân 1 hàm: Hàm main() hoặc hàm con, than vòng lặp, cấu trúc If .. Elsse ....

- Biến toàn cục (Local Variable)
  Trái ngược lại hoàn toàn với biến cục bộ là biến toàn cục. Chúng ta khai báo biến này ngoài tất cả các hàm và cũng sử dụng biến đó ở mọi nơi trong chương trình: Hàm main(), hàm con, câu lệnh rẽ nhánh ...

- Biến tĩnh (Static Variable)
  Đặc biệt và ít dùng hơi hai biến trên là biến tĩnh. Đó là giá trị của nó luôn giữ và tồn tại cho đến hết chương trình cho dù nó là biến cục bộ hay biến toàn cục. Ta sử dụng biến tĩnh bằng các khai báo từ khoá "static", VD: static: int i;

- Register
  Mang tính bên trong lõi hơn, Register được dùng đẻ khai báo các biến có tính chất cục bộ nhưng mà được lưu trong thanh ghi của CPU. Do được lưu trong thanh ghi nên việc truy xuất dữ liệu sẽ nhanh hơn so với biến được lưu trong bộ nhớ. Tuy nhiên việc khởi tạo và sử dụng biến này chỉ thật sự cần thiết nếu bạn làm việc với hệ thống nhúng hay tối ưu hoá hệ thống

Tham chiếu & Tham trị

Tham chiếu và tham trị là hai đặc tính hay trong lập trình C, để hiểu rõ hơn về cách thức hoạt động thì chúng ta sẽ đề cập đến ở phần con trỏ.

- Tham trị
  Khi ta gọi **tham trị**, việc tác động vào biến **tham trị** sẽ không ảnh hưởng tới biến ban đầu. Biến được tạo bởi tham trị là bản sao nhân đôi với biến ban đầu và hoàn toàn độc lập.

- Tham chiếu:
  Khi ta gọi tham chiều, đơn giản biến đó là bản **"shortcut"** của biến ban đầu. Mọi sự tương tác với biến **"shortcut"** này bản chất là tương tác trực tiếp với biến gốc

Return & Exit

- Lệnh return[6] Khi sử dụng lệnh này, chương trình sẽ thoát ra khi hàm gặp nó và tiếp tục trở lại thực thi những dòng Code sau lời gọi hàm đó (nếu có)
- Hàm Exit Bản chất là câu lệnh hệ thống và phải khai báo thư viện stdlib.h nếu muốn sử dụng. Khi sử dụng hàm Exit chương trình sẽ kết thúc ngay lập tức

4 Mảng

Hiểu về mảng & Cách gọi và sử dụng

- Lý thuyết:
  Hiểu một cách đơn giản, **Mảng** là tập hợp các phần tử có cùng kiểu dữ liệu và được lưu trữ trong một dãy **ô nhớ liên tục**. Việc gọi và truyền vào dữ liệu trong mảng là ta làm việc với chỉ số mảng hay bản chất là con trỏ. Mảng cũng là cấu trúc dữ liệu đầu tiên, đơn giản và phổ biến nhất khi bạn đến với thế giới lập trình. Các ô "đựng" dữ liệu sắp xếp liền nhau này được xác định bởi chỉ số mảng, bắt đầy bằng 0 và kết thúc tại n-1 với mảng có chiều dài n
- Khai báo, khởi tạo và sử dụng:
  - Khai báo:

```
<kieu du lieu> arr[kich thuoc mang];
```

  - Khởi tạo mảng:

```
arr[index] = gia tri muon truyen vao mang;
```

```
arr[<size>] = {(gia tri 1), (gia tri 2) ...};
```

  Bên cạnh đó ta thường dùng vòng lặp "for" để truyền vào từng giá trị từ bàn phím cũng như xuất ra màn hình

[6] Rất thích hợp để dùng cho hàm không trả về giá trị (Kiểu Void)

- Thuật toán sắp xếp nổi bọt (Bubble Sort)
  Thuật toán sắp xếp nổi bọt thực hiện sắp xếp dãy số bằng cách lặp lại công việc đổi chỗ 2 số liên tiếp nhau nếu chúng đứng sai thứ tự(số sau bé hơn số trước với trường hợp sắp xếp tăng dần) cho đến khi dãy số được sắp xếp. Code ví dụ về hoạt động của **Bubble Sort**:

```
#include <stdio.h>

void swap(int &x, int &y)
{
    int temp = x;
    x = y;
    y = temp;
}

void bubbleSort(int arr[], int n)
{
    int i, j;
    bool haveSwap = false;
    for (i = 0; i < n-1; i++){
        haveSwap = false;
        for (j = 0; j < n-i-1; j++){
            if (arr[j] > arr[j+1]){
                swap(arr[j], arr[j+1]);
                haveSwap = true;          }
        }
        if(haveSwap == false){
            break;
        }
    }
}

/* Array Output */
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", arr[i]);
```

```
}

// Driver program to test above functions
int main()
{
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);
    bubbleSort(arr, n);
    printf("Sorted array: n");
    printArray(arr, n);
    return 0;
}
```

- Thuật toán sắp xếp chọn (Selection Sort)
  Giả sử với sắp xếp mảng giảm dần, chương trình sẽ đi tìm giá trị lớn nhất bằng cách giả định a[0] (vị trí đầu tiên của mảng) là Max và rồi đối chiếu với các phần tử còn lại. Cứ thế những vị trí liền kề ngay sau đó trong mảng lần lượt đối chiếu với các phần tử còn lại để xác định các phần tử lớn nhất theo thứ tự giảm dần, ví dụ về Code sắp xếp mảng tăng dần:

```
#include <stdio.h>

void swap(int &xp, int &yp)
{
    int temp = xp;
    xp = yp;
    yp = temp;
}

void selectionSort(int arr[], int n)
{
    int i, j, min_idx;
    for (i = 0; i < n-1; i++)
    {
    min_idx = i;
    for (j = i+1; j < n; j++)
        if (arr[j] < arr[min_idx])
        min_idx = j;
```

```
        swap(arr[min_idx], arr[i]);
    }
}

void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", arr[i]);
    printf("n");
}

int main()
{
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr)/sizeof(arr[0]);
    selectionSort(arr, n);
    printf("Sorted array: n");
    printArray(arr, n);
    return 0;
}
```

- Thuật toán sắp xếp chèn (Insert Sort)
  Giả sử với một bài toán sắp xếp mảng theo thứ tự tăng dần. Thuật toán chèn hoạt động bằng cách, xét giá trị của chỉ số đầu tiên của mảng, với các giá trị của các mảng có chỉ số liền kề lớn hơn mảng ban đầu, để nguyên, nếu bé hơn, để len trước. Thuật toán được thể hiện qua đoạn code sau:

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


void insertionSort(int arr[], int n)
{
   int i, key, j;
   for (i = 1; i < n; i++)
   {
       key = arr[i];
       j = i-1;
```

```
        while (j >= 0 && arr[j] > key)
        {
            arr[j+1] = arr[j];
            j = j-1;
        }
        arr[j+1] = key;
    }
}

void printArray(int arr[], int n)
{
    int i;
    for (i=0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main()
{
    int arr[] = {12, 11, 13, 5, 6};
    int n = sizeof(arr)/sizeof(arr[0]);

    insertionSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);

    return 0;
}
```

Thuật toán tìm kiếm

Được sử dụng nhiều trong thực tế, khác với các phương thức tìm kiếm thông thường khi duyệt lần lượt các phần tử trong mảng đối chiếu với giá trị cần tìm kiếm (Độ phức tạp của thuật toán có thể rất lớn nếu phần tử cần truy xét nằm tận cuối trong mảng). Bằng cách sắp xếp các phần tử tăng dần (hoặc giảm dần), sử dụng thuật toán nhị phân đem lại khả năng tối ưu hơn, thuật toán được khai triển với ý tưởng (ở đây mảng sắp xếp giảm dần):

- Xét đoạn mảng arr[left .... right] cần tìm kiếm phần tử x. Ta so sánh x với phần tử ở vị trí giữa của mảng(mid = (left + right)/2). Nếu:
- Phần tử arr[mid] = x. Kết luận và thoát chương trình.
- Nếu arr[mid] < x. Thực hiện tìm kiếm trên đoạn arr[mid+1 ... right]
- Nếu arr[mid] > x. Thực hiện tìm kiếm trên đoạn arr[left ... mid-1]

```c
#include <stdio.h>
int binarySearch(int arr[], int n, int x) {
  int r = n - 1;
  int l = 0;
  while (r >= l) {
    int mid = l + (r - l) / 2; //Tranh bi tran so

    if (arr[mid] == x)
      return mid;

    if (arr[mid] > x)
      r = mid - 1;

    if (arr[mid] < x)
      l = mid + 1;
  }
  return -1;
}

int main(void) {
  int arr[] = {2,3,4,10,40} ;
  int n = sizeof(arr) / sizeof(arr[0]);
  int x = 10;
  int result = binarySearch(arr, n, x);
  if (result == -1)
    printf("%d xuat hien tai chi so %d", x, result);
  else
    printf("%d xuat hien tai chi so %d", x, result);
  return 0;
}
```

Mảng hai chiều

Tiếp đến với mảng một chiều là mảng hai chiều (hay còn gọi là ma trận). Mảng hai chiều bản chất cũng là mảng một chiều nhưng có thêm một trục không gian (hàng).

- Khai báo mảng hai chiều:
  Bạn phải chỉ định các tham số sau khi khai báo mảng: Số hàng, số cột và kiểu dữ liệu. Cú pháp

```
type arr[row_size][column_size] ={{Elements}, { Elements} .}
```

  Việc truyền dữ liệu vào mảng 2 chiều tương tự như mảng 1 chiều nhưng được nâng cấp hơn khi sử dùng hai vòng lặp lồng nhau.

```
type arr[row_size][column_size] = {{elements}, {elements} ... }
for i from 0 to row_size
    for j from 0 to column_size
        scanf arr[i][j]
```

  Việc xuất dữ liệu ra cũng tương tự như khi nhập

```
type arr[row_size][column_size] = {{elements}, {elements} ... }
for i from 0 to row_size
    for j from 0 to column_size
        print arr[i][j]
```

  Lưu ý, phải chỉ định kích cỡ cột của ma trận trong tham số hàm.

5 Chuỗi

Chuỗi là dạng đặc biệt của hàm với kiểu dữ liệu là kiểu ký tự **char**. Chính vì vậy việc khai báo khá tương tự khai báo mảng. Tuy nhiên, cần lưu ý khi khai báo kích thước của mảng bởi dấu cách và kết thức chuỗi cũng được tính là một ký tự.

Làm quen với thư viện string.h

- Hàm **strlen** (lấy chiều dài chuỗi)

```
int strlen ( const char * str );
```

- Hàm **strcmp** (so sánh các chuỗi với nhau).
  Trong ngôn ngữ C, bạn có thể so sánh độ lớn của các kiểu dữ liệu dạng số nhưng với dạng ký tự có trong chuỗi không thể sử dụng thế được mà phải dùng hàm strcmp.

```
int strcmp ( const char * str1, const char * str2 );
```

Kết quả của hàm trả ra dạng số thế nên kiểu dữ liệu là **int**. Ta có bảng sau:

Giá trị trả về	Giải thích
một số nguyên `< 0`	Khi ký tự đầu tiên của 2 chuỗi không giống nhau và ký tự này ở chuỗi str1 có giá trị nhỏ hơn ở chuỗi str2
giá trị `0`	hai chuỗi giống nhau
một số nguyên `> 0`	Khi ký tự đầu tiên của 2 chuỗi không giống nhau và ký tự này ở chuỗi str1 có giá trị lớn hơn ở chuỗi str2

Đoạn code ví dụ sử dụng hàm strcmp:

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

int main ()
{
  char key[] = "apple";
  char buffer[80];
  do {
     printf ("Hay doan loai qua toi thich? ");
     fflush (stdout);
     scanf ("%s",buffer);
  } while (strcmp (key,buffer) != 0);
  puts ("Ban doan dung roi!");
  return 0;
}
```

- Hàm **strcat** (hàm nối chuỗi):
  Giúp ghép chuỗi nguồn phía sau và chuỗi đích, ví dụ thể hiện qua mã nguồn như sau:

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

int main ()
{
  char str[80];
  strcpy (str,"Lap ");
  strcat (str,"trinh ");
  strcat (str,"khong ");
  strcat (str,"kho!");
  puts (str);
}
---------------------
KQ: Lap trinh khong kho!
```

- Hàm **strcpy** (Hàm copy chuỗi)
  Hàm có tác dụng Copy giá trị của chuỗi nguồn và lưu vào chuỗi đích. Bạn cần dùng hàm này khi muốn gán giá trị của chuỗi này cho chuỗi khác thay vì sử dụng dấu "="

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

int main ()
{
  char str1[]="Lap trinh khong kho!";
  char str2[40];
  char str3[40];
  strcpy (str2,str1);
  strcpy (str3,"Nguyen Van Hieu");
  printf ("str1: %s\nstr2: %s\nstr3: %s\n",str1,str2,str3);
  return 0;
}
------------
char * strcpy ( char * chuoi_dich, const char * chuoi_nguon);
```

- Hàm **strupr** (đưa chuỗi về dạng uppercase)

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

```
int main()
{
    char str[ ] = "Lap Trinh KHONG kho!";
    printf("%s\n",strupr (str));
}
```

- Hàm **strrev** (Hàm đảo ngược chuỗi)

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

int main()
{
    char name[30] = "Nguyen Van Hieu";

    printf("Truoc khi dao nguoc: %s\n", name);

    printf("Sau khi dao nguoc: %s", strrev(name));

    return 0;
}
```

Nhập xuất chuỗi

- Đối với chuỗi không chứa dấu trắng (dấu cách, dấu tab hay dấu xuống dòng), ta vẫn có thể nhập xuất chuỗi bình thường bằng lệnh scanf() và printf().

```
#include <stdio.h>
int main()
{
   char name[20];
   printf("Enter name: ");
   scanf("%s", name);
   printf("Your name is %s.", name);
   return 0;
}
-----
Enter name: Tuan Anh Pham
Your name is Tuan
```

như ta có thể thấy ở các câu lệnh trên, nó chỉ hiển thị chuỗi kí tự đầu trước phần có dấu cách, nếu bạn muốn xuất họ tên đầy đủ (có dấu cách, tab ...). Ta sẽ dùng **fgets()** để nhập và **puts()** để xuất chuỗi ra ngoài màn hình.

```
#include <stdio.h>

int main(){
    char name[20];
    printf("Enter name: ");
    fgets(name, sizeof(name), stdin);
    printf("Name: ");
    puts(name);   // display string
    return 0;
}
```

6 Con Trỏ

Hiểu về con trỏ & cách gọi, sử dụng

- Lý thuyết:
  Con trỏ bản chất cũng là **biến** (có thể khai báo, khởi tạo, lưu trữ giá trị và có địa chỉ của riêng nó) nhưng giá trị của con trỏ lại biểu thị địa chỉ của các **biến** thông thường khác. Như đã đề cập, địa chỉ của biến khá quan trọng, nó giúp máy tính quản lý dữ liệu bằng các địa chỉ các ô trống chứa biến. Điều này dễ thấy qua tập lệnh sau:

```
int number;
printf("\nNhap number = ");
scanf("%d", &number);
printf("\nnumber = %d", number);
```

  Rõ ràng khi dùng hàm Scanf chúng ta cần truyền vào number thế nhưng khi dùng hàm printf lại không cần dùng "&" trước tên biến. Đơn giản vì nếu muốn nhập giá trị cho biến, hàm **scanf** cần biết địa chỉ của biến đó trong bộ nhớ đó rồi mới nhập giá trị vào.

- Khai báo, khởi tạo và sử dụng [7]:

[7] Đặc biệt lưu ý khi khai báo con trỏ phải truyền ngay giá trị cho nó, nếu không giá trị của nó sẽ là giá trị rác, điều này sẽ cực kỳ nguy hiểm với chương trình của bạn

Một chút khác biệt so với khai báo biến bằng việc thêm "*" trước tên biến để trình biên dịch biết ta đang khai báo con trỏ.

```
<type> * <name of variable>;
   -------------- Vi du ve khai bao bien con tro --------------
int num = 5;
int *p = &num;
```

Lưu ý: Sau lần khai báo đầu tiên truyền địa chỉ của biến vào con trỏ, việc gán một giá trị bất kì nào đó vào con trỏ sau đó sẽ được hiểu là tác dụng lên giá trị của biến mà con trỏ đang trỏ tới

```
#include <stdio.h>
int main()
{
int value = 10;
 int *p = &value;
 printf("Gia tri cua bien: %d", value);
 printf("\nDia chi cua bien: %d", &value);
 printf("\nGia tri con tro: %d", p);
 printf("\nDia chi con tro: %d", &p);
 printf("\nGia tri bien noi con tro chi toi: %d", *p);
 printf("\n--- Cap nhat gia tri con tro ---");
 *p = 20;
 printf("\nGia tri cua bien khi cap nhat: %d", value);
 printf("\nGia tri bien ma con tro chi toi khi cap nhat: %d", *p);
 printf("\nDia chi con tro: %d\n", &p);
}
------------
KQ:
Gia tri cua bien: 10
Dia chi cua bien: -272632436
Gia tri con tro: -272632436
Dia chi con tro: -272632448
Gia tri bien noi con tro chi toi: 10
--- Cap nhat gia tri con tro ---
Gia tri cua bien khi cap nhat: 20
Gia tri bien ma con tro chi toi khi cap nhat: 20
Dia chi con tro: -272632448
Program ended with exit code: 0
```

Các mối liên hệ trong con trỏ

- Liên hệ giữa con trỏ với **mảng** Mảng như đã định nghĩa từ trước là một tập hợp tuần tự các phần tử có cùng kiểu dữ, và được lưu trong một dãy liên tục các bộ nhớ. Có thể hiểu rõ qua các câu lệnh sau:

```
#include <stdio.h>

int main(){
    int arr[] = {1, 2, 3, 4, 5};
    printf("Dia chi cua mang arr = %d\n", &arr);
    printf("Gia chi cua mang arr = %d\n", arr);
    for(int i = 0; i < sizeof (arr) / sizeof (int); i++){
        printf("Dia chi cua arr[%d] = %d\n", i, &arr[i]);
    }
}
-----------
KQ:
Dia chi cua mang arr = 6487552
Gia chi cua mang arr = 6487552
Dia chi cua arr[0] = 6487552
Dia chi cua arr[1] = 6487556
Dia chi cua arr[2] = 6487560
Dia chi cua arr[3] = 6487564
Dia chi cua arr[4] = 6487568
```

Qua trên ta có vài nhận xét:

  - Các phần tử liên tiếp có địa chỉ cách nhau 4 giá trị (bởi vì 1 phần tử kiểu nguyên có kích thước 4 bytes). Điều đó củng cố cho định nghĩa các phần tử trong mảng xếp cạnh nhau.
  - Như đã đề cập là khi truyền mảng vào hàm thì mặc định là truyền theo tham chiếu. Và trong ví dụ này bạn thấy đó, địa chỉ và giá trị của biến mảng chính là địa chỉ của phần tử đầu tiên của mảng.
  - Như vậy, &arr[0] tương đương &arr và tương đương arr. Điều đó có được là do biến arr trỏ tới phần tử đầu tiên của mảng.

- Liên hệ giữa con trỏ với **hàm** Để dễ hiểu, tìm hiểu mối quan hệ giữa con trỏ với hàm thực ra là tìm hiểu về tham chiếu trong C và Truyền con trỏ vào hàm

– Tham chiếu:

Xét qua ví dụ sau, có thể thấy hàm con không thay đổi giá trị của biến a và b trong hàm main(). Đó là bởi hàm của bạn đang truyền bởi tham trị – nghĩa là khi hàm swap() được gọi thì 2 tham số đó sẽ được hàm này sao chép sang 2 vùng nhớ mới, mọi thay đổi được thực hiện trên bản sao này.

```
#include <stdio.h>

void swap(int a, int b){
      printf("Ham con, truoc khi goi ham hoan vi, a = %d, b =
          %d\n", a , b);
      int tmp = a;
      a = b;
      b = tmp;
      printf("Ham con, sau khi goi ham hoan vi, a = %d, b = %d\n",
          a , b);
}

int main(){
      int a = 5, b = 7;
      printf("Ham main, truoc khi goi ham hoan vi, a = %d, b =
          %d\n", a , b);
      swap(a, b);
      printf("Ham main, sau khi goi ham hoan vi, a = %d, b =
          %d\n", a , b);
}
 ----------
 KQ:
   Ham main, truoc khi goi ham hoan vi, a = 5, b = 7
   Ham con, truoc khi goi ham hoan vi, a = 5, b = 7
   Ham con, sau khi goi ham hoan vi, a = 7, b = 5
   Ham main, sau khi goi ham hoan vi, a = 5, b = 7
```

Tương tự như bài toán này nhưng ta thử truyền tham chiếu bằng cách tham chiếu Và chứng minh cho việc khi ta có địa chỉ biến thì ta có thể thay đổi giá trị biến mà con trỏ đang truyền tới:

```
#include <stdio.h>

void swap(int *a, int *b){
    printf("Ham con, truoc khi goi ham hoan vi, a = %d, b = %d\n", *a
        , *b);
```

```
    int tmp = *a;
    *a = *b;
    *b = tmp;
    printf("Ham con, sau khi goi ham hoan vi, a = %d, b = %d\n", *a ,
        *b);
}

int main(){
    int a = 5, b = 7;
    printf("Ham main, truoc khi goi ham hoan vi, a = %d, b = %d\n", a
        , b);
    swap(&a, &b);
    printf("Ham main, sau khi goi ham hoan vi, a = %d, b = %d\n", a ,
        b);
}
--------
KQ:
Ham main, truoc khi goi ham hoan vi, a = 5, b = 7
Ham con, truoc khi goi ham hoan vi, a = 5, b = 7
Ham con, sau khi goi ham hoan vi, a = 7, b = 5
Ham main, sau khi goi ham hoan vi, a = 7, b = 5
```

– Truyền con trỏ vào hàm

Chúng ta đã quen với truyền giá trị vào hàm (truyền tham trị) ta cũng vừa làm quen với truyền tham chiếu qua đoạn code ở trên, qua phần này chúng ta sẽ hiểu rõ hơn về truyền con trỏ vào hàm trong C:

```
#include <stdio.h>

void addOne(int *ptr)
{
    (*ptr)++;
}
int main()
{
    int *p, i = 10;
    p = &i;
    addOne(p);
    printf("%d", *p); // Dap an: 11
    return 0;
}
```

Cấp phát bộ nhớ cho con trỏ

Một phần quan trọng khi làm quen và sử dụng con trỏ là cấp phát bộ nhớ động và giải phóng bộ nhớ bằng việc sử dụng các hàm malloc(), calloc(), free() và realloc() trong C

- Tại sao cần cấp phát bộ nhớ động, những định nghĩa ...
  Nói đến con trỏ không thể không nhắc tới các định nghĩa về **cấp phát bộ nhớ động** và **giải phóng bộ nhớ**. Chúng ta sẽ làm rõ vấn đề này về việc cấp phát bộ nhớ động sử dụng các hàm **malloc(), calloc(), free() và realloc()**
  Quay trở lại mảng, như ta đã biết và thường làm trước đó, ta thường chỉ định rõ kích thước tối đa của **mảng** và sau khi khai báo ta không thể thay đổi kích thước của mảng được nữa ⇒ Cấp phát tĩnh.
  Điều đó nảy sinh ra vấn đề là có thể kích thước giới hạn định sẵn đó không đủ dùng, để giải quyết vấn đề ta có cấp phát bộ nhớ theo cách thủ công trong thời gian chạy chương trình.
- Cấp phát và sử dụng bộ nhớ qua các hàm

  * Sử dụng hàm malloc (memory allocation)
    Hàm malloc() thực hiện cấp phát bộ nhớ bằng cách chỉ định số byte cần cấp và trả về kiểu **void** giúp ta có thể ép kiểu thành bất cứ kiểu dữ liệu nào. Cú pháp:

    ```
    ptr = (castType*) malloc(size);
    ----- Vi Du -----
    ptr = (int*) malloc(100 * sizeof(int));
    ```

    Ở ví dụ trên, ta thực hiện việc lưu trữ 100 số nguyên. Với **sizeof (int)** là 4, khi đó lệnh malloc() thực hiện cấp phát 400 bytes. Khi đó, con trỏ "ptr" có giá trị là địa chỉ của byte dữ liệu đầu tiên trong khối bộ nhớ.
  * Sử dụng hàm calloc (contiguous allocation)
    Đối với hàm malloc() khi cấp phát bộ nhớ thì vùng nhớ cấp phát đó không được khởi tạo giá trị ban đầu. Trong khi đó, hàm calloc() thực hiện cấp phát bộ nhớ và khởi tạo tất cả các ô nhớ có giá trị bằng 0/

    ```
    ptr = (castType*)calloc(n, size);
    ------ Vi du ------
    ptr = (int*) calloc(100, sizeof(int));
    ```

    Trong ví dụ trên, hàm calloc() thực hiện cấp phát 100 ô nhớ liên tiếp và mỗi ô nhớ có kích thước là số byte của kiểu int. Hàm này cũng trả về con

trỏ chứa giá trị là địa chỉ của byte đầu tiên trong khối bộ nhớ vừa cấp phát.

* Sử dụng hàm free()
  Việc cấp phát bộ nhớ động trong C dù sử dụng malloc() hay calloc() thì chúng cũng đều không thể tự giải phóng bộ nhớ. Bạn cần sử dụng hàm free() để **giải phóng vùng nhớ** không dùng tới để tối ưu dụng lượng trống hệ thống .

```
free(ptr);
```

Lệnh này sẽ giải phóng vùng nhớ mà con trỏ ptr đã được cấp phát. Giải phóng ở đây có nghĩa là trả lại vùng nhớ đó cho hệ điều hành và hệ điều hành có thể sử dụng vùng nhớ đó vào việc khác nếu cần.

Nếu bạn không giải phóng nó thì nó sẽ tồn tại cho tới khi chương trình kết thúc. Điều này sẽ rất nguy hiểm nếu chương trình của bạn liên tục cấp phát các vùng nhớ mới và sẽ gây ra hiện tượng tràn bộ nhớ mà mình đã nhắc tới ở trên. Thử code này xem sao (bảo đảm là máy bạn sẽ bị treo và chỉ còn cách ấn nút nguồn thôi, bạn có thể để cấp phát size nhỏ hơn và theo dõi thay đổi qua Task Manager)

* Sử dụng hàm malloc() và free():
  Trong ví dụ dưới đây, chúng ta sẽ sử dụng hàm **malloc()** để cấp phát động n * sizeof int byte và sử dụng xong sẽ dùng free() để giải phóng.

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

int main()
{
    int n, i, *ptr, sum = 0;
    printf("Nhap so luong phan tu: ");
    scanf("%d", &n);
    ptr = (int *)malloc(n * sizeof(int));

    if (ptr == NULL)
    {
        printf("Co loi! khong the cap phat bo nho.");
        exit(0);
    }
    printf("Nhap cac gia tri: ");
    for (i = 0; i < n; ++i)
```

```
    {
        scanf("%d", ptr + i);
        sum += *(ptr + i);
    }
    printf("Tong = %d", sum);

    // Giai phong bo nho con tro
    free(ptr);
    return 0;
}
```

---

* Sử dụng hàm **calloc()** và **free()**

  Trong ví dụ này, chúng ta sẽ dùng calloc() để cấp phát n ô nhớ liên tiếp và mỗi ô nhớ có kích thước là sizeof int. Lưu ý là hàm calloc() sẽ chậm hơn malloc() một chút do nó phải thêm bước khởi tạo các ô nhớ có giá trị bằng 0.

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

int main()
{
    int n, i, *ptr, sum = 0;
    printf("Nhap so luong phan tu: ");
    scanf("%d", &n);
    ptr = (int *)calloc(n, sizeof(int));

    if (ptr == NULL)
    {
        printf("Co loi! khong the cap phat bo nho.");
        exit(0);
    }
    printf("Nhap cac gia tri: ");
    for (i = 0; i < n; ++i)
    {
        scanf("%d", ptr + i);
        sum += *(ptr + i);
    }
    printf("Tong = %d", sum);

    free(ptr);
    return 0;
```

```
}
```

* Sử dụng hàm **realloc()**; Nếu việc cấp phát bộ nhớ động không đủ hoặc cần nhiều hơn mức đã cấp phát, bạn có thể thay đổi kích thước của bộ nhớ đã được cấp phát trước đó bằng cách sử dụng hàm realloc().
Chúng ta thực hiện cấp phát vùng nhớ mới cho con trỏ ptr, vùng nhớ mới sẽ có kích thước là n bytes qua cú pháp sau:

```
ptr = realloc(ptr, n);
```

Một ví dụ về việc sử dụng hàm realloc() để tái phân lại bộ nhớ :

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int *ptr, i , n1, n2;
    printf("Nhap so luong phan tu: ");
    scanf("%d", &n1);
    ptr = (int*) malloc(n1 * sizeof(int));
    printf("Dia chi cua vung nho vua cap phat: %u", ptr);

    printf("\nNhap lai so luong phan tu: ");
    scanf("%d", &n2);
    // phan bo lai vung nho
    ptr = (int*) realloc(ptr, n2 * sizeof(int));
    printf("Dia chi cua vung nho duoc cap phat lai: %u", ptr);
    // giai phong
    free(ptr);
    return 0;
}
```

7 Kiểu Struct trong C

Hiểu về Struct & cách gọi, sử dụng

– Tổng quan:
Như được làm quen với mảng trước đó, chúng cho phép bạn lưu trữ các giá trị của các biến có cùng kiểu dữ liệu. Làm quen với kiểu **Struct** lại là một loại tổ chức dữ liệu khác trong lập trình đó là cho phép kết hợp các kiểu dữ liệu khác

kiểu nhau (hơi ngược với kiểu lưu trữ mảng).

Struct thực sự hoạt động thế nào ? Một cách dễ hình dung, Struct tổ chức dữ liệu bạn dưới dạng bản ghi. Nghĩa là bạn giả sử muốn lưu trữ giá trị của một quyển sách trong thư viện của bạn. Bạn sẽ quan tâm đến những yếu tố sau của cuốn sách và đó là cách Struct giúp bạn quản lý và sử dụng:

* Tiêu đề
* Tác giả
* Chủ đề
* Book ID

– Cú pháp định nghĩa kiểu :

Trước khi chúng ta có thể khai báo biến với struct, bạn cần định nghĩa nó – Đó cũng là lý do tại sao struct được gọi là kiểu dữ liệu người dùng định nghĩa. Khi nào ta cần tự định nghĩa kiểu cấu trúc đó là khi cần lưu trữ một đối tượng có nhiều thược tính. Ví dụ, đối tượng SinhVien có các thuộc tính (MSV, họ, tên, giới tính, quê quán,...). Khi đó chúng ta dùng struct để quản lý chương trình. Cú pháp định nghĩa :

```
struct structureName
{
    dataType member1;
    dataType member2;
    ...
};
---------- Vi du ----------
struct SinhVien {
    int maSV;
    char ho[20];
    char ten[20];
    bool gioiTinh;
    char queQuan[100];
};
```

– Khai báo biến kiểu **Struct**:

Việc khai báo biến với **struct** cũng giống như cách khai báo biến thông thường, trong đó kiểu dữ liệu là kiểu struct trong C mà bạn vừa định nghĩa.

```
struct SinhVien
{
    int maSV;
```

```
    char ho[20];
    char ten[20];
    bool gioiTinh;
    char queQuan[100];
};

int main(){
   //khai bao 2 bien sv1 va sv2 co kieu SinhVien
    SinhVien sv1, sv2;

    // Nen them tu khoa Struct o tren dau
    // De phan biet du lieu tu dinh nghia
    struct SinhVien sv3, sv4;

    // Khai bao mang
    struct SinhVien sv[100];
}
```

– Truy xuất các thuộc tính cùa kiểu **Struct**:

Ta sử dụng 2 toán tử để truy xuất tới các biến thành viên của kiểu

* Sử dụng "." ⇒ Toán tử truy xuất tới thành viên khi khai báo biến bình thường
* Sử dụng "->" ⇒ Toán tử truy xuất tới thành viên khi biến là con trỏ.

  Giả sử bạn muốn truy suất **"gioiTinh"** của đối tượng Sinh Viên thì làm như sau:

```
SinhVien sv;
// to do
printf("Gioi tinh: %s", sv.gioiTinh);
```

– Từ khoá **typedef**:

Bạn có thể sử dụng từ khóa **typedef** để tạo ra một tên thay thế cho kiểu dữ liệu đã có. Nó thường được sử dụng kiểu **struct** để đơn giản hóa cú pháp khai báo biến. Nhưng nó cũng có thể sử dụng với các kiểu dữ liệu nguyên thủy nhé.

```
struct Distance{
    int feet;
    float inch;
};

int main() {
```

```
    structure Distance d1, d2;
}
 ---- Tuong Duong ----
 typedef struct SinhVien{
    int feet;
    float inch;
} distances;

int main() {
    distances d1, d2;
}
---- Hoac ----
struct PhanSo{
    int tu;
    int mau;
};
typedef struct PhanSo PS;
```

– Cấu trúc **Struct** lồng nhau:

Giả sử bạn muốn xây dựng kiểu dữ liệu để lưu trữ đối tượng Tam giác, khi đó chúng ta có thể xây dựng **struct** mô tả tọa độ của 1 điểm, khi đó đối tượng tam giác sẽ là 3 đối tượng điểm. Cụ thể:

```
struct Point{
    int x; // Hoanh do
    int y; // Tung do
};

struct Triangle{
    Point a; //dinh thu nhat
    Point b; //dinh thu hai
    Point c; //dinh thu ba
}

int main(){
    Triangle tg;

    // truy xuat hoanh do cua diem thu nhat
    tg.a.x = 5;
}
```