



Union

1. Union là gì ?

Union là dữ liệu đặc biệt trong ngôn ngữ C cho phép bạn dự trữ các kiểu dữ liệu khác nhau trong cùng một vùng nhớ. Bạn có thể định nghĩa Union với rất nhiều tham số, nhưng chỉ một thành phần chứa giá trị tại một thời điểm.

Khai báo tường minh

```
union unionName
{
    dataType member1;
    dataType member2;
    ...
};
```

Khai báo không tường minh

```
typedef union
{
    dataType member1;
    dataType member2;
    ...
} unionName;
```

Ví dụ:

```
union Hoclaptoptrinh {
    int i;
    float f;
    char chuoi[50];
};
```

Ví dụ:

```
typedef union {
    int i;
    float f;
    char chuoi[50];
} Hoclaptoptrinh;
```



Union

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

union Hoclptrinh
{
    int i;
    float f;
    char chuoit[50];
};

int main( )
{
    union Hoclptrinh tenbien;

    tenbien.i = 15;
    tenbien.f = 25.67;
    strcpy( tenbien.chuoit, "Hoc Lap trinh C tai HocLapTrinh" );
    printf( "tenbien.i : %d\n", tenbien.i );
    printf( "tenbien.f : %f\n", tenbien.f );
    printf( "tenbien.chuoit : %s\n", tenbien.chuoit );

    printf("\n=====\\n");
    printf("HocLapTrinh chuc cac ban hoc tot! \\n");

    return 0;
}

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

union Hoclptrinh
{
    int i;
    float f;
    char chuoit[50];
};

int main( )
{
    union Hoclptrinh tenbien;

    tenbien.i = 15;
    printf( "tenbien.i : %d\\n", tenbien.i );
    tenbien.f = 25.67;
    printf( "tenbien.f : %f\\n", tenbien.f );
    strcpy( tenbien.chuoit, "Hoc Lap trinh C tai HocLapTrinh" );
    printf( "tenbien.chuoit : %s\\n", tenbien.chuoit );

    printf("\\n=====\\n");
    printf("HocLapTrinh chuc cac ban hoc tot! \\n");

    return 0;
}
```

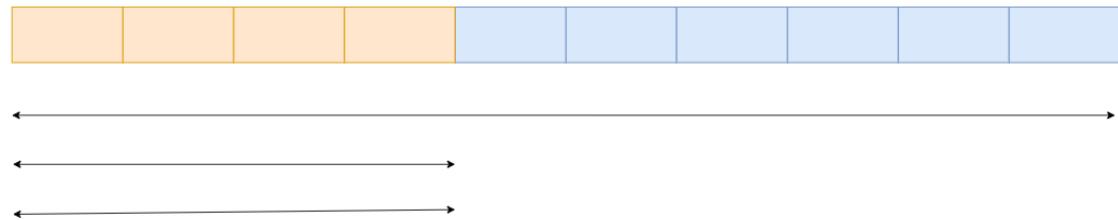


Union

```
#include<stdio.h>

typedef union{
    char x[10];
    int y;
    float z;
} sotunhien;

int main()
{
    sotunhien abc;
    abc.x[0] = 1;
    abc.x[1] = 2;
    abc.x[2] = 3;
    abc.x[3] = 4;
    abc.x[4] = 5;
    abc.x[5] = 6;
    abc.x[6] = 7;
    abc.y = 1000;
    abc.z = 1.3;
    for(int i = 0;i<10;i++)
    {
        printf("%d\n", abc.x[i]);
    }
    printf("%d\n", abc.y);
    printf("%f\n", abc.z);
}
```

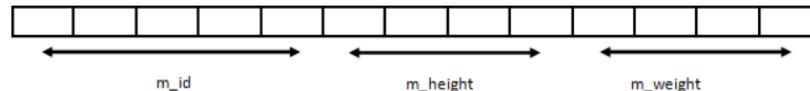


Union

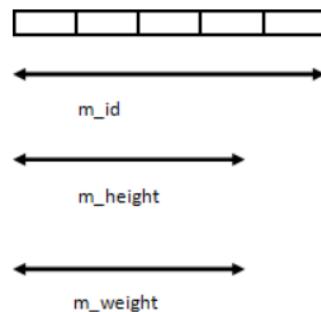
2. Sự khác nhau giữa Struct và Union

```
struct sSinhVien
{
    char    m_id[5];
    float   m_height;
    float   m_weight;
};
```

Struct



Union





Union

1. Viết 1 chương trình cho phép nhập thông tin của giáo viên hoặc thông tin của sinh viên. Sau đó in ra thông tin vừa nhập đó. Chạy liên tục. Sử dụng union và struct.
2. Sử dụng union để biểu diễn 1 số float theo kiểu mảng 4 bytes.



Bit Field

1. Khái niệm

Bit Fields là một kỹ thuật nhằm tối ưu bộ nhớ trong struct, trong một số trường hợp khi khai báo ta sử dụng kiểu dữ liệu có phạm vi giá trị lớn, trong khi giá trị thực tế nhỏ hơn và không bao giờ đạt đến những giá trị lớn đó.

Ví dụ 1: Khi mình khai báo biến `bool` (chỉ có 2 giá trị `true` hoặc `false`) tuy nhiên lại sử dụng mất 1 byte (8 bits) cho kiểu dữ liệu `bool`, trong khi ta chỉ cần sử dụng 1 bit trong 8 bit đó (giá trị 0 hoặc 1) là đủ để thể hiện 2 giá trị `true` và `false`. Vô hình dung chúng ta đã lãng phí mất 7 bit rồi.

Ví dụ 2: Khi chúng ta khai báo 1 biến `int tuoi_tac` (mất 4 byte) để chỉ thị số tuổi của một ai đó, nhưng thực tế chúng ta biết rằng nó không bao giờ sử dụng hết giá trị của cả 4 byte để chỉ thị điều đó.

→ **Vì vậy mà trường bit fields sẽ được sử dụng trong những trường hợp này.**



Bit Field

Giải thích kết quả ?

```
#include <stdio.h>

struct time
{
    unsigned int hours:5;
    unsigned int minutes:6;
    unsigned int seconds:6;
};

int main()
{
    struct time t = {
        .hours = 10,
        .minutes = 30,
        .seconds = 10
    }; // Here t is an object of the structure time

    printf("The time is %d : %d : %d\n", t.hours, t.minutes, t.seconds);
    printf("The size of time is %ld bytes.\n", sizeof(struct time));
    return 0;
}
```



Bit Field

```
#include <stdio.h>

struct time
{
    unsigned int hours;
    unsigned int minutes;
    unsigned int seconds;
};

int main()
{
    struct time t = {
        .hours = 10,
        .minutes = 30,
        .seconds = 10
    }; // Here t is an object of the structure time

    printf("The time is %d : %d : %d\n", t.hours, t.minutes, t.seconds);
    printf("The size of time is %ld bytes.\n", sizeof(struct time));
    return 0;
}
```

Giải thích kết quả ?



Bit Field

```
struct {
    int a : 1; // 1 bit - bit 0
    int b : 2; // 2 bits - bits 2 down to 1
    char c ;   // 8 bits - bits 15 down to 8
} reg1;
```

will be

1	2	3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1		
a b b x x x x c c c c c c c x x x x x x x x x x x x x x		
<----- int ----->		



Bit Field

```
struct {
    char a : 1; // 1 bit - bit 0
    char b : 2; // 2 bits - bits 2 down to 1
    char c ;    // 8 bits - bits 15 down to 8
} reg1;
```

will be

1	
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5	
a b b x x x x c c c c c c c	
<---- char ----><---- char ---->	



Bit Field

Dự đoán kết quả ?

```
struct x {
    short a : 2;
    short b : 15;
    char c ;
};

struct y {
    int a : 2;
    int b : 15;
    char c ;
};
```



CHỈ THỊ TIỀN XỬ LÝ

- Chỉ thị tiền xử lý là những chỉ thị cung cấp cho trình biên dịch để xử lý những thông tin trước khi bắt đầu quá trình biên dịch.
- Tất cả các chỉ thị tiền xử lý đều bắt đầu với # và các chỉ thị tiền xử lý không phải là lệnh C/C++ vì vậy không có dấu ; khi kết thúc.
- Để cho dễ chúng ta chia thành 3 nhóm chính đó là:
 - Chỉ thị bao hàm tệp (#include).
 - Chỉ thị định nghĩa cho tên (#define macro).
 - Chỉ thị biên dịch có điều kiện (#if, #else, #elif, #endif, ...).



CHỈ THỊ BAO HÀM TỆP

#include cho phép thêm nội dung một file khác vào file đang viết.

Chúng ta đặc biệt dùng **#include** để thêm vào **file.c** các nội dung từ những **file.h** của các thư viện (stdio.h, stdlib.h, string.h, math.h) và cũng có thể là từ những **file.h** của riêng bạn.

Để thêm nội dung những **file.h** có trong thư mục cài đặt IDE của bạn, bạn cần sử dụng những ngoặc nhọn

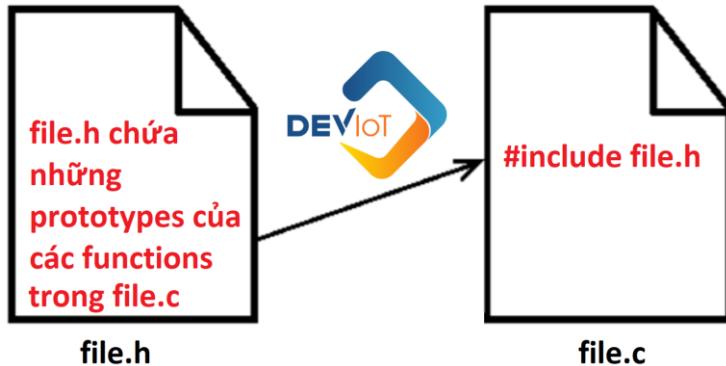
<>:

Ví dụ: #include <stdlib.h>

Khi sử dụng dấu "" bộ tiền xử lý sẽ tìm file name trong thư mục chứa project của bạn. Nếu không tìm thấy nó sẽ tiếp tục tìm trong các file có sẵn trong IDE:

Ví dụ: #include "yourfile.h"

CHỈ THỊ BAO HÀM TỆP



Tất cả nội dung của **file.h** sẽ được đặt vào trong **file.c**, ngay tại vị trí đặt chỉ thị tiền xử lý **#include file.h**.

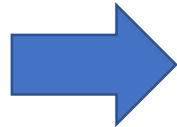


CHỈ THỊ BAO HÀM TỆP

Dưới đây là những gì ta có trong file.c:

```
//Code C
#include "file.h"
long myFunction(int x, double y)
{
    /* Source code of function */
}
void otherFunction(long value)
{
    /* Source code of function */
}
```

Tiền xử lý



```
//Code C
long myFunction(int x, double y);
void otherFunction(long value);

long myFunction(int x, double y)
{
    /* Source code of function */
}
void otherFunction(long value)
{
    /* Source code of function */
}
```

Và những gì có trong file.h:

```
long myFunction(int x, double y);
void otherFunction(long value);
```

CHỈ THỊ ĐỊNH NGHĨA CHO TÊN



#define là một từ khóa cho phép bạn tạo ra các Macro (Macro là 1 tên bất kỳ được lập trình viên đặt trả tới 1 khối lệnh thực hiện 1 chức năng nào đấy).

```
#define new_name_use name_want_swap
```

➡ Công việc của nó là copy `name_want_swap` paste vào chỗ của `new_name_use` trong code.

Có 2 loại Macro:

1. Macro giống như hàm.

```
#include <stdio.h>
#define nhan(x,y) x*y
int main() {
    printf("ket qua = %d", nhan(2+5, 2));
}
```

```
#include <stdio.h>
#define nhan(x,y) (x)*(y)
int main() {
    printf("ket qua = %d", nhan(2+5, 2));
}
```

CHỈ THỊ ĐỊNH NGHĨA CHO TÊN



2. Macro giống như đối tượng

- Các macro giống như một đối tượng với một văn bản thay thế

```
#define PI 3,14
#define U8 char
#define SLOGAN "Dao tao that - Hoc that - Lam that"
```

- Các macro giống như một đối tượng mà không có văn bản thay thế

```
#define USE_YEN
```

Các macro của biểu mẫu này hoạt động như sau: mọi sự xuất hiện tiếp theo của định danh macro này(USE_YEN) sẽ bị xóa và được thay thế bởi không có gì!

Điều này có vẻ khá vô dụng. Tuy nhiên, đó không phải là những gì lệnh này thường được sử dụng. Chúng ta sẽ thảo luận về việc sử dụng hình thức này ngay sau đây.

CHỈ THỊ ĐỊNH NGHĨA CHO TÊN



#**undef** khi ta cần định nghĩa lại 1 macro đã được định nghĩa từ trước đó, ta sử dụng #**undef** để hủy bỏ định nghĩa trước đó của nó.

Ví dụ

```
// Định nghĩa cho tên DEVIOT là "Hello DEVIOT"
#define DEVIOT "Hello DEVIOT"

#undef DEVIOT // Hủy bỏ định nghĩa cho tên DEVIOT

// Định nghĩa lại cho tên DEVIOT là "Welcome to DEVIOT"
#define DEVIOT "Welcome to DEVIOT"
```



CHỈ THỊ ĐỊNH NGHĨA CHO TÊN

Hãy xem xét chương trình sau:

```
1 #include <iostream>
2
3 #define MY_NAME "Alex"
4
5 int main()
6 {
7     std::cout << "My name is: " << MY_NAME;
8
9     return 0;
10 }
```

Bộ tiền xử lý chuyển đổi phần trên thành như sau:

```
1 // The contents of iostream are inserted here
2
3 int main()
4 {
5     std::cout << "My name is: " << "Alex" ;
6
7     return 0;
8 }
```

Mà khi chạy, in đầu ra **My name is: Alex**

PHẠM VI ĐỊNH NGHĨA



Phạm vi định nghĩa

Các chỉ thị sẽ được xử lý trước khi biên dịch, từ trên xuống dưới của từng file.

Hãy xem xét chương trình sau:

```
1 #include <iostream>
2
3 void foo()
4 {
5     #define MY_NAME "Alex"
6 }
7
8 int main()
9 {
10     std::cout << "My name is: " << MY_NAME;
11
12     return 0;
13 }
```

Mặc dù có vẻ như `#define MY_NAME "Alex"` Được định nghĩa bên trong hàm `foo`, bộ tiền xử lý đã được thông báo, vì nó không hiểu các khái niệm C++ như các hàm. Do đó, chương trình này hoạt động bình thường và nó không phụ thuộc vào vị trí định nghĩa của `#define MY_NAME "Alex"`. Khi đó, thì chúng ta có thể định nghĩa nó trước hoặc ngay sau hàm `foo` cũng được nhưng để dễ đọc, bạn thường muốn `#define` định danh bên ngoài các hàm.

Khi bộ tiền xử lý kết thúc, tất cả các định danh được định nghĩa từ file đó sẽ bị loại bỏ. Điều này có nghĩa là các chỉ thị chỉ có giá trị từ điểm định nghĩa đến cuối file mà chúng được định nghĩa. Các chỉ thị được định nghĩa trong một file code không có tác động đến các file code khác trong cùng một dự án.

Hãy xem xét ví dụ sau:

function.cpp:

```
1 #include <iostream>
2
3 void doSomething()
4 {
5     #ifdef PRINT
6         std::cout << "Printing!" ;
7     #endif
8     #ifndef PRINT
9         std::cout << "Not printing!" ;
10    #endif
11 }
```

main.cpp:

```
1 void doSomething(); // forward declaration for function doSomething()
2
3 #define PRINT
4
5 int main()
6 {
7     doSomething();
8
9     return 0;
10 }
```

Chương trình trên sẽ in:

```
1 Not printing!
```

CHỈ THỊ BIÊN DỊCH CÓ ĐIỀU KIỆN



Ở nhóm này gồm các chỉ thị `#if`, `#elif`, `#else`, `#ifdef`, `#ifndef`.

`#if, #elif, #else`

Được sử dụng tương tự như câu lệnh if else bình thường tuy nhiên nó được xử lý trong quá trình tiền xử lý và chỉ sử dụng với các constant-expression (các biểu thức không có biến trong đó).

Cú pháp:

```
#if constant-expression_1  
  
// Đoạn chương trình 1  
  
#elif constant-expression_2  
  
// Đoạn chương trình 2  
  
#else  
  
//Đoạn chương trình 3  
  
#endif
```



```
#include <stdio.h>  
  
#define ENG_US 1  
#define ENG_UK 2  
#define FRENCH 3  
  
#define LANGUAGE ENG_UK  
  
int main(){  
    #if (LANGUAGE == ENG_US)  
        printf("Selected language is: ENG_US\n");  
    #elif (LANGUAGE == ENG_UK)  
        printf("Selected language is: ENG_UK\n");  
    #else  
        printf("Selected language is: FRENCH\n");  
    #endif  
  
    return 0;  
}
```

CHỈ THỊ BIÊN DỊCH CÓ ĐIỀU KIỆN



```
#if 0
```

Một cách sử dụng phổ biến của biên dịch có điều kiện liên quan đến việc sử dụng `#if 0` để loại trừ một khối code khỏi được biên dịch (như thể nó nằm trong một khối comments):

```
1  #include <iostream>
2
3  int main()
4  {
5      std::cout << "Joe\n" ;
6
7      #if 0 // Don't compile anything starting here
8          std::cout << "Bob\n" ;
9          std::cout << "Steve\n" ;
10     #endif // until this point
11
12     return 0;
13 }
```

Đoạn code trên chỉ in ra Joe, bởi vì Bob và Steve đã ở trong một khối `#if 0` mà bô tiền xử lý sẽ loại trừ nó khỏi quá trình biên dịch.

Điều này cung cấp một cách thuận tiện để comments code có chứa các nhiều dòng.

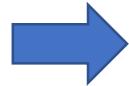
CHỈ THỊ BIÊN DỊCH CÓ ĐIỀU KIỆN



#ifdef: nếu đã được định nghĩa
#ifndef: nếu chưa được định nghĩa
Dùng để kiểm tra xem đoạn code nào có được chạy trong chương trình hay không.

Chỉ thị #ifdef.

```
#ifdef identifier
    //Đoạn chương trình 1
#else
    //Đoạn chương trình 2
#endif
```



```
#include<stdio.h>

int main()
{
    #ifdef PI           // neu da dinh nghia gia tri cua PI\r\n";
    printf("Da dinh nghia gia tri cua PI\r\n");
    #endif

    printf("Ket thuc\r\n");
    return 0;
}
```

Chỉ thị #ifndef

```
#ifndef identifier
    //Đoạn chương trình 1
#else
    //Đoạn chương trình 2
#endif
```



```
#include<stdio.h>

int main()
{
    #ifndef PI           // neu chua dinh nghia gia tri cua PI\r\n";
    printf("Chua dinh nghia gia tri cua PI\r\n");
    #endif

    printf("Ket thuc\r\n");
    return 0;
}
```

CHỈ THỊ ĐỊNH NGHĨA CHO TÊN



Hãy xem xét chương trình sau:

```
1 #include <iostream>
2
3 #define PRINT_JOE
4
5 int main()
6 {
7     #ifdef PRINT_JOE
8         std::cout << "Joe\n" ; // if PRINT_JOE is defined, compile this
9         code
10    #endif
11
12    #ifdef PRINT_BOB
13        std::cout << "Bob\n" ; // if PRINT_BOB is defined, compile this
14        code
15    #endif
16
17    return 0;
18 }
```

CHỈ THỊ BIÊN DỊCH CÓ ĐIỀU KIỆN



Bài tập:

1. Viết chương trình nhập liệu dành cho giáo viên (3 chức năng nhập điểm sinh viên và in điểm sinh viên và tìm điểm sinh viên theo tên) và học sinh (chức năng tra điểm) có sử dụng chỉ thị tiền xử lý.



CẤP PHÁT ĐỘNG

Phân biệt cấp phát động và cấp phát tĩnh

Cấp phát bộ nhớ tĩnh	Cấp phát bộ nhớ động
Bộ nhớ được cấp phát trước khi chạy chương trình (trong quá trình biên dịch)	Bộ nhớ được cấp phát trong quá trình chạy chương trình.
Không thể cấp phát hay phân bổ lại bộ nhớ trong khi chạy chương trình	Cho phép quản lý, phân bổ hay giải phóng bộ nhớ trong khi chạy chương trình
Vùng nhớ được cấp phát và tồn tại cho đến khi kết thúc chương trình	Chỉ cấp phát vùng nhớ khi cần sử dụng tới
Chương trình chạy nhanh hơn so với cấp phát động	Chương trình chạy chậm hơn so với cấp phát tĩnh
Tốn nhiều không gian bộ nhớ hơn	Tiết kiệm được không gian bộ nhớ sử dụng



CẤP PHÁT ĐỘNG

```
// Hàm cấp phát bộ nhớ động
void* malloc(size_t size);

// Hàm cấp phát bộ nhớ động
void* calloc(size_t num, size_t size);

// Hàm giải phóng bộ nhớ động
free(ptr); // ptr là con trỏ
```

Hãy viết 1 chương trình nhập vào
thông tin cá nhân của bạn sử dụng
struct và in ra console sử dụng cấp phát
động ?

```
// C program to demonstrate the use of malloc()
// and calloc()
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int* arr;

    // malloc() allocate the memory for 5 integers
    // containing garbage values
    arr = (int*)malloc(5 * sizeof(int)); // 5*4bytes = 20 bytes

    // Dealloates memory previously allocated by malloc() function
    free(arr);

    // calloc() allocate the memory for 5 integers and
    // set 0 to all of them
    arr = (int*)calloc(5, sizeof(int));

    // Dealloates memory previously allocated by calloc() function
    free(arr);

    return (0);
}
```



CẤP PHÁT ĐỘNG

Điểm khác nhau giữa hàm malloc() và calloc()

malloc()	calloc()
malloc viết tắt của memory allocation	calloc viết tắt của contiguous allocation
malloc nhận 1 tham số truyền vào là số byte của vùng nhớ cần cấp phát	calloc nhận 2 tham số truyền vào là số block và kích thước mỗi block (byte)
void *malloc(size_t n); Hàm trả về con trỏ trả tới vùng nhớ nếu cấp phát thành công, trả về NULL nếu cấp phát fail	void *calloc(size_t n, size_t size); Hàm trả về con trỏ trả tới vùng nhớ được cấp phát và vùng nhớ được khởi tạo bằng giá trị 0. Trả về NULL nếu cấp phát fail
Hàm malloc() nhanh hơn hàm calloc()	Hàm calloc() tốn thêm thời gian khởi tạo vùng nhớ. Tuy nhiên, sự khác biệt này không đáng kể.



CẤP PHÁT ĐỘNG

Xét ví dụ sau đây

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

void capphatbonho(int *pt)
{
    pt = calloc(10,sizeof(int));
}

void main()
{
    int *pt;
    capphatbonho(pt);
    pt[0] = 1;
    printf("%d", pt[0]);
}
```

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

int* capphatbonho(void)
{
    int *pt = calloc(10,sizeof(int));
    return pt;
}

void main()
{
    int *pt;
    pt = capphatbonho();
    pt[0] = 1;
    printf("%d", pt[0]);
}
```



CẤP PHÁT ĐỘNG

Viết 1 chương trình code cấp phát động 1 mảng byte đủ sài.

- Viết hàm nhập, xuất mảng
- Viết hàm thêm phần tử và xóa phần tử vào vị trí bất kì sử dụng cấp phát động để số phần tử sau khi chạy hàm vừa đủ sài.

Viết 1 chương trình quản lý tối đa

10 sinh viên bằng cấp phát động

- Các chức năng: thêm, sửa, xóa thông tin sinh viên
- Chọn chức năng qua menu

CON TRỎ CẤP 2



Con trả cấp 2 là gì ?

Pointer to pointer là một loại con trả dùng để lưu trữ địa chỉ của 1 biến con trả khác

Mình lấy ví dụ:

```
int value = 5;  
int *ptr = &value;
```

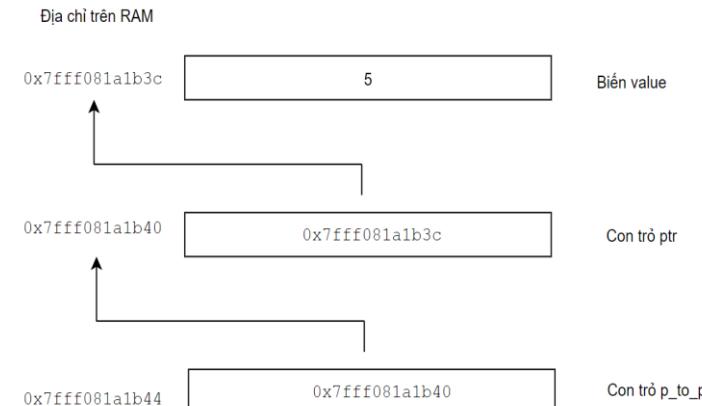
Ở đây mình có 1 con trả ptr, mình trả con trả ptr tới địa chỉ của biến value.

```
=> ptr = &value  
=> *ptr = 5
```

Vậy để một con trả cấp 2 (pointer to pointer) trả tới địa chỉ của một con trả khác (pointer) ta sẽ làm như sau:

```
int **p_to_p = &ptr; // trả con trả p_to_p tới địa chỉ của con trả ptr
```

Con trả p_to_p được gọi là một con trả cấp 2.





CON TRỎ CẤP 2

```
#include <stdio.h>

int main()  {

    int value = 100;
    int *ptr = &value;
    int **p_to_p = &ptr;

    printf("value = %d\r\n",value);
    printf("&value = %p\r\n",&value);

    printf("*ptr = %d\r\n",*ptr);
    printf("ptr = %p\r\n",ptr);
    printf("&ptr = %p\r\n",&ptr);

    printf("p_to_p = %p\r\n",p_to_p); //địa chỉ ô nhớ của ptr
    printf("*p_to_p = %p\r\n",*p_to_p); //địa chỉ ô nhớ mà ptr trỏ đến
    printf("**p_to_p = %d\r\n",**p_to_p); //giá trị tại địa chỉ ô nhớ ptr trỏ đến

    return 0;
}
```



CON TRỎ CẤP 2

Sau khi xem ví dụ chúng ta có thể rút ra các kết luận như sau:

- Bản chất của con trỏ cấp 2 vẫn là một con trỏ, nên khi truy xuất giá trị của **p_to_p** chúng ta lấy được địa chỉ mà nó trỏ đến (địa chỉ của biến **ptr**).
- **p_to_p** tương đương với **&ptr**: chính là địa chỉ mà con trỏ cấp 2 trỏ tới, hay chính là địa chỉ của con trỏ **ptr**.
- ***p_to_p** tương đương với **ptr**: chính là giá trị của con trỏ **ptr**, hay cũng chính là địa chỉ ô nhớ mà **ptr** trỏ tới, cũng chính là địa chỉ của biến **value**.
- ****p_to_p** tương đương với ***ptr** hay chính là giá trị ô nhớ mà con trỏ **ptr** trỏ tới, cũng chính là giá trị của biến **value**.



CON TRỎ CẤP 2

Tác dụng 1: Thay đổi địa chỉ trả của con trỏ 1 chiều

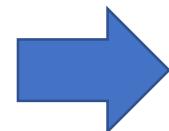
```
#include <stdio.h>

char *chuoi1 = "Chuoi so 1";
char *chuoi2 = "Chuoi so 2";

void thaydoiChuoi(char *pt)
{
    pt = chuoi2;          // trả con trỏ sang chuỗi số 2
}

int main()
{
    char *chuoi_hien_tai = chuoi1;
    printf("chuỗi hiện tại là: %s\n", chuoi_hien_tai);

    thaydoiChuoi(chuoi_hien_tai);
    printf("chuỗi hiện tại là: %s\n", chuoi_hien_tai);
    return 0;
}
```



Sai ở đâu, sửa lại cho đúng

CON TRỎ CẤP 2



Bài tập vận dụng: Viết hàm cấp phát động bộ nhớ cho 1 con trỏ cấp 1 thông qua con trỏ cấp 2.



MÃNG CON TRỎ

```
#include <stdio.h>

#define KICHTHUOC 3

void xuatmang(int *pt, int sophantu)
{
    for(int i=0;i<sophantu;i++)
    {
        printf("%d ", pt[i]);
    }
}

int main ()
{
    int pt1[5] = {1,2,3,4,5};
    int pt2[5] = {11,12,13,14,15};
    int pt3[5] = {21,22,23,24,25};

    int *contro[KICHTHUOC];

    contro[0] = pt1;
    contro[1] = pt2;
    contro[2] = pt3;

    //    contro[0][0] = 111;    // thay doi gia tri

    xuatmang(contro[0],5);

    return 0;
}
```

```
#include <stdio.h>

const int KICHCO = 4;

int main ()
{
    char *hotensv[] = {
        "Tran Hung Cuong",
        "Ho Ngoc Ha",
        "Nguyen Son Tung",
        "Dam Vinh Hung",
    };

    for (int i = 0; i < KICHCO; i++)
    {
        printf("Gia tri cua hotensv[%d] = %s\n", i, hotensv[i] );
    }

    return 0;
}
```

MÃNG CON TRỎ



```
#include<stdio.h>

int main()
{
    int x[5] = {1,2,3,4,5};
    int y[5] = {11,12,13,14,15};
    int *contro[2] = {x,y};
    printf("%d\n", contro[0][0]);
    printf("%d\n", contro[1][0]);
}
```

địa chỉ

```
char *hotensv[2] = {pt1,pt2};
```

```
char *hovatensv[2] = {{1,2,3,4,5},  
{11,12,13,14,15}};
```



CON TRỎ CẤP 2

Tác dụng 2: Cấp phát động 1 mảng con trả

```
#include <stdio.h>
void xuatmang (int *pt, int sophantu)
{
    for(int i=0;i<sophantu;i++)
        printf ("%d ", pt[i]);
}

int main ()
{
    int mang1[5] = {1,2,3,4,5};
    int mang2[5] = {11,12,13,14,15};
    int mang3[5] = {21,22,23,24,25};
    int mang4[5] = {31,32,33,34,35};
    int mang5[5] = {41,42,43,44,45};

    int **pt = (int **)calloc(5, sizeof(int *));
    pt[0] = mang1;

    xuatmang(pt[0],5);
    return 0;
}
```



CON TRỎ CẤP 2

Tác dụng 3: Cấp phát động mảng 2 chiều

```
#include <stdio.h>
#include <stdlib.h>
void NhapMaTran(int **a, int dong, int cot)
{
    int i, j;
    for (i = 0; i < dong; i++)
        for (j = 0; j < cot; j++)
        {
            printf("a[%d][%d] = ", i, j);
            scanf("%d", &a[i][j]);
        }
}
void XuatMaTran(int **a, int dong, int cot)
{
    int i, j;
    for (i = 0; i < dong; i++)
    {
        for (j = 0; j < cot; j++)
            printf("%5d", a[i][j]);
        printf("\n");
    }
}
```

```
int main()
{
    int **a = NULL, dong, cot;
    int i;
    printf("Nhập vào số dòng = ");
    scanf("%d", &dong);
    printf("Nhập vào số cột = ");
    scanf("%d", &cot);
    // Cấp phát mảng các con trỏ cấp 1
    a = (int **)malloc(dong * sizeof(int *));
    for (i = 0; i < dong; i++)
    {
        // Cấp phát cho từng con trỏ cấp 1
        a[i] = (int *)malloc(cot * sizeof(int));
    }
    NhapMaTran(a, dong, cot);
    XuatMaTran(a, dong, cot);

    // Giải phóng từng hàng
    for (i = 0; i < dong; i++)
    {
        free(a[i]);
    }
    // Giải phóng con trỏ quản lý các dòng
    free(a);
    return 0;
}
```

CON TRỎ CẤP 2



BT1: Sử dụng con trỏ cấp 2 cấp phát ra mảng 2 chiều gồm 5 hàng 5 cột.

- Viết hàm xuất nhập mảng 2 chiều đó.
- Viết hàm xóa bỏ 1 hàng bất kì và free bộ nhớ của hàng đã bị xóa
- Viết hàm cấp phát 1 mảng 2 chiều mới bao gồm nội dung của mảng 2 chiều cũ và thêm nội dung 1 hàng mới bất kỳ. Sau đó free mảng 2 chiều cũ.



CON TRỎ HÀM

Con trả hàm là gì ?

- Ta định nghĩa 1 hàm như bình thường, và khi chạy chương trình, hàm đó sẽ được load lên RAM. Và điều thú vị là khi đã load lên RAM thì nó sẽ có địa chỉ tương ứng. Mà phàm là địa chỉ thì con trả đều trả đến được, do đó ta có một loại con trả là con trả hàm.
- Thực chất khi gọi một hàm chính ta là ta yêu cầu hệ điều hành hãy thực thi đoạn lệnh được lưu tại địa chỉ tương ứng.
- Con trả hàm là một biến lưu trữ địa chỉ của một hàm, thông qua biến đó, ta có thể gọi hàm mà nó trả tới để thực thi hàm đó mà không cần gọi tên của hàm.
- Con trả hàm cho phép ta viết những đoạn code mang tính tổng quát.
- Con trả hàm thường được sử dụng để viết hàm callback trong chương trình.



CON TRỎ HÀM

Khai báo tường minh

```
//Cú pháp
<kiểu trả về> (*<tên con trỏ>) (<ds tham số>);
//con trỏ hàm được khai báo giống như hàm,
//nhưng tên đặt bên trong (*...)
```

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

int cong2so(int x, int y)
{
    return x+y;
}

int main()
{
    int (*func) (int, int);
    func = cong2so;
    int kq = func(1,2);
    printf ("%d", kq);
    return 0;
}
```



CON TRỎ HÀM

Khai báo không tường minh

```
//định nghĩa một kiểu mới có tên FuncPointer
typedef int (*FuncPointer)(int, int);

#include <stdio.h>
#include <string.h>
typedef int (*func) (int, int);

int cong2so(int x, int y)
{
    return x+y;
}

int main()
{
    func pt;
    pt = cong2so;
    int kq = pt(1,2);
    printf("%d", kq);
}
```



CON TRỎ HÀM

Sử dụng con trỏ hàm là tham số của một hàm

```
1 //Một số hàm tính đơn giản
2 int Cong(int a, int b) { return a + b; }
3 int Tru(int a, int b) { return a - b; }
4 int Nhan(int a, int b) { return a * b; }
5
6 //Định nghĩa kiểu con trỏ hàm PhepTinh
7 typedef int (*PhepTinh)(int, int);
8
9 //Định nghĩa hàm tính toán tổng quát
10 //với tham số là 2 số nguyên và con trỏ PhepTinh
11 int TinhToan(int a, int b, PhepTinh tinh)
12 {
13     //Gọi hàm thông qua con trỏ hàm
14     return tinh(a, b);
15 }
16
17 //Một số lời gọi hàm
18 int tong = TinhToan(5, 9, Cong);
19 int hieu = TinhToan(5, 9, Tru);
```

Ví dụ sử dụng con trỏ hàm làm callback



CON TRỎ HÀM

Phân tích các ví dụ trong video sau:

[Ví dụ về cách sử dụng con trỏ hàm](#)



Ứng dụng viết Callback

Callback là gì ?

Callback function là hàm của người dung định nghĩa được gọi trong một hàm khác trong tiến trình hoạt động.

Sử dụng con trỏ hàm như là một call back

Thứ tưởng tượng bạn viết 1 thư viện. Thư viện có chức năng nhận dữ liệu và cần xử lý dữ liệu đó. Nhưng bạn muốn người dùng tự định nghĩa ra hàm xử lý dữ liệu này. Khi đó bạn cần sử dụng con trỏ hàm để gọi hàm người dùng tự định nghĩa.

Ứng dụng viết Callback



Viết 1 chương trình mô phỏng cơ chế hoạt động của timer.

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

typedef void (*funcTimer) (void);

funcTimer ptr;

void delay(int number_of_seconds)
{
    // Converting time into milli_seconds
    int milli_seconds = 1000 * number_of_seconds;

    // Storing start time
    clock_t start_time = clock();

    // looping till required time is not achieved
    while (clock() < start_time + milli_seconds)
        ;
}

void timerstart(int seconds)
{
    while(1)
    {
        delay(seconds);
        ptr();
    }
}

void settimercallback(funcTimer cb)
{
    ptr = cb;
}
```

```
#include <stdio.h>

void timercallback(void)
{
    printf("deviot.vn");
}

int main()
{
    settimercallback(timercallback);
    timerstart(1);
}

#ifndef __TIMER_H
#define __TIMER_H
void timerstart(int seconds);
void settimercallback(funcTimer cb);
#endif // __TIMER_H
```



CON TRỎ HÀM

Bài tập:

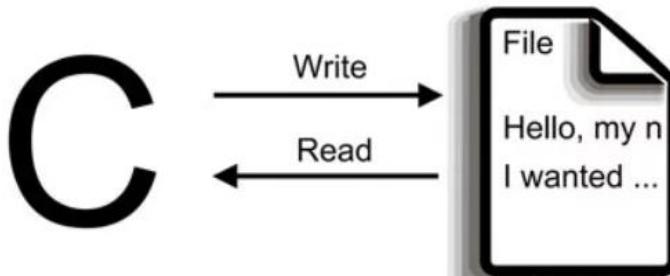
1. Viết 1 chương trình cho phép nhập vào 1 mảng và chọn 2 chức năng
 - sắp xếp mảng theo thứ tự tăng dần hoặc giảm dần sử dụng con trỏ hàm.

2. Viết 1 chương trình thực hiện các chức năng sau:
 - 1s in ra dòng chữ DEVIOT.VN 1 lần
 - 2s in ra dòng chữ GIANGDZ 1 lần
 - 3s in ra dòng chữ ABCD 1 lầnSử dụng 3 hàm callback

XỬ LÝ FILE

Tại sao chúng ta cần đến file?

- Dữ liệu được lưu ở biến của chương trình, và nó sẽ biến mất khi chương trình kết thúc. Sử dụng file để lưu trữ dữ liệu cần thiết để đảm bảo dữ liệu của chúng ta không bị mất ngay cả khi chương trình của chúng ta ngừng chạy.
- Nếu chương trình của bạn có đầu vào(input) là lớn, bạn sẽ rất vất vả nếu phải nhập mỗi khi chạy. Thay vào đó, hãy lưu vào file và chương trình của bạn sẽ tự đọc mỗi lần khởi chạy.
- Dễ dàng sao chép, di chuyển dữ liệu giữa các thiết bị với nhau





XỬ LÝ FILE

Có các loại file nào ?

1. File text

- File văn bản là file thường có đuôi là .txt. Những file này bạn có thể dễ dàng tạo ra bằng cách dùng các text editor thông dụng như Notepad, Notepad++, Sublime Text,...
- Khi bạn mở các file này bằng các text editor nói trên, bạn sẽ thấy được văn bản ngay và có thể dễ dàng thao tác sửa, xóa, thêm nội dung của file này.
- Kiểu file này thuận tiện cho chúng ta trong việc sử dụng hàng ngày, nhưng nó sẽ kém bảo mật và cần nhiều bộ nhớ để lưu trữ hơn.

2. File binary

- Các file này được lưu dưới dạng nhị phân, chỉ bao gồm các số 0 và 1.
- Loại file này giúp lưu trữ được dữ liệu với kích thước lớn hơn, không thể đọc bằng các text editor thông thường và thông tin lưu trữ ở loại file được bảo mật hơn so với file văn bản.

XỬ LÝ FILE



Thao tác 1: Mở file

- Khai báo con trỏ kiểu FILE

```
FILE *fptr;
```

- Tiến hành mở file

```
fptr = fopen("file direction", "mode")
```

Ví dụ:

```
fptr = fopen("D:\\CodeBlock\\program.txt", "w");
```

Mode	Ý nghĩa	Nếu file không tồn tại
r	Mở file chỉ cho phép đọc	Nếu file không tồn tại, fopen() trả về NULL.
rb	Mở file chỉ cho phép đọc dưới dạng nhị phân.	Nếu file không tồn tại, fopen() trả về NULL.
w	Mở file chỉ cho phép ghi.	Nếu file đã tồn tại, nội dung sẽ bị ghi đè. Nếu file không tồn tại, nó sẽ được tạo tự động.
wb	Open for writing in binary mode.	Nếu file đã tồn tại, nội dung sẽ bị ghi đè. Nếu file không tồn tại, nó sẽ được tạo tự động.
a	Mở file ở chế độ ghi "append". Tức là sẽ ghi vào cuối của nội dung đã có.	Nếu file không tồn tại, nó sẽ được tạo tự động.
ab	Mở file ở chế độ ghi nhị phân "append". Tức là sẽ ghi vào cuối của nội dung đã có.	Nếu file không tồn tại, nó sẽ được tạo tự động.
r+	Mở file cho phép cả đọc và ghi.	Nếu file không tồn tại, fopen() trả về NULL.
rb+	Mở file cho phép cả đọc và ghi ở dạng nhị phân.	Nếu file không tồn tại, fopen() trả về NULL.
w+	Mở file cho phép cả đọc và ghi.	Nếu file đã tồn tại, nội dung sẽ bị ghi đè. Nếu file không tồn tại, nó sẽ được tạo tự động.
wb+	Mở file cho phép cả đọc và ghi ở dạng nhị phân.	Nếu file đã tồn tại, nội dung sẽ bị ghi đè. Nếu file không tồn tại, nó sẽ được tạo tự động.
a+	Mở file cho phép đọc và ghi "append".	Nếu file không tồn tại, nó sẽ được tạo tự động.
ab+	Mở file cho phép đọc và ghi "append" ở dạng nhị phân.	Nếu file không tồn tại, nó sẽ được tạo tự động.



XỬ LÝ FILE

Thao tác 2: Đọc, ghi file văn bản

- Ghi file

```
int fprintf(FILE *f, const char *format, ...);  
// ghi theo định dạng vào File
```

```
int fputs(FILE *f, const char *format, ...);  
// ghi chuỗi vào File
```

- Đọc file

```
char* fgets(char *buf, int n, FILE *f); // Đọc ra (n-1) kí tự từ FILE đến khi gặp ký  
tự kết thúc hoặc xuống dòng
```

```
int fscanf(FILE *f, const char *format, ...); // Đọc dữ liệu theo format, phân biệt dấu  
cách, kí tự kết thúc, xuống dòng
```

*Note: đọc, ghi từ vị trí con trỏ trong file



XỬ LÝ FILE

Ghi file

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    char text[1000];
    FILE *fptr;

    fptr = fopen("D:\\CodeBlock\\program.txt", "w");
    if(fptr == NULL)
    {
        printf("Error!");
        exit(1);
    }

    printf("Enter a text:\n");
    gets(text);

    fprintf(fptr, "%s", text);
    fclose(fptr);

    return 0;
}
```

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

int main()
{
    char text[1000];
    FILE *fptr;

    fptr = fopen("D:\\CodeBlock\\C\\File\\File\\text.txt", "w");
    if(fptr == NULL)
    {
        printf("Error!");
        exit(1);
    }

    printf("Enter text:");
    gets(text);

    fputs(text, fptr);
    fclose(fptr);
    return 0;
}
```



XỬ LÝ FILE

Đọc file

```
#include <stdio.h>
#include <stdlib.h> // For exit() function
int main()
{
    char c[1000];
    FILE *fptr;

    if ((fptr = fopen("D:\\CodeBlock\\program.txt", "r")) == NULL)
    {
        printf("Error! opening file");
        // Program exits if file pointer returns NULL.
        exit(1);
    }

    // reads text until newline
    fscanf(fptr,"%s", c);

    printf("Data from the file:\n%s", c);
    fclose(fptr);

    return 0;
}
```



XỬ LÝ FILE

Thực hành:

1. Hãy viết 1 chương trình chạy có 2 hàm ghi file và đọc file và chạy nó trong hàm main. (Chú ý khi ghi File xong con trỏ sẽ ở cuối File nếu đọc luôn sẽ gây ra lỗi. Có 2 cách xử lý: Hoặc là đóng file mở lại để con trỏ về đầu File, hoặc là set con trỏ về đầu File bằng code).
2. Hãy ghi 1 File có nội dung như sau:

Toi yeu Viet Nam

Toi thich hoc lap trinh

Và đọc đầy đủ nội dung ra.



XỬ LÝ FILE

Đặt vị trí con trỏ trong FILE

```
int fseek(FILE *f, long int offset, int origin);
```

Trong đó:

- f là con trỏ trỏ đến đối tượng FILE đang mở.
- offset là số bytes được cộng thêm tính từ vị trí origin.
- origin là địa điểm đặt con trỏ trong file:

Constant	Description
SEEK_SET	Beginning of file
SEEK_CUR	Current position of the file pointer
SEEK_END	End of file

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    char w_text[1000];
    char r_text[1000];
    FILE *fptr;
    fptr = fopen("D:\\CodeBlock\\program.txt", "r+");
    if(fptr == NULL)
    {
        printf("Error!");
        return -1;
    }
    printf("Enter a text:\n");
    gets(w_text);
    fprintf(fptr,"%s\n", w_text);
    printf("Enter a text:\n");
    gets(w_text);
    fprintf(fptr,"%s\n", w_text);
    fseek(fptr, 0, SEEK_SET);
    printf("Get text from file:\n");
    fgets(r_text, 20, fptr);
    printf("%s", r_text);
    fgets(r_text, 20, fptr);
    printf("%s", r_text);
    fclose(fptr);
    return 0;
}
```



XỬ LÝ FILE

Thao tác 2: Đọc, ghi file binary

- Ghi file

```
size_t fwrite(const void *ptr, size_t size, size_t count, FILE *f);
```

- Đọc file

```
size_t fread(void *ptr, size_t size, size_t count, FILE *f);
```

Hàm này có bốn tham số:

1. Đường dẫn đến file cần ghi
2. Kích thước của dữ liệu
3. Số loại dữ liệu như vậy
4. Con trỏ đến file mà bạn muốn ghi



XỬ LÝ FILE

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

struct threeNum
{
    int n1, n2, n3;
};

int main()
{
    int n;
    struct threeNum num;
    FILE *fptr;

    if ((fptr = fopen("C:\\program.bin","wb")) == NULL){
        printf("Error! opening file");

        // Program exits if the file pointer returns NULL.
        exit(1);
    }

    for(n = 1; n < 5; ++n)
    {
        num.n1 = n;
        num.n2 = 5*n;
        num.n3 = 5*n + 1;
        fwrite(&num, sizeof(struct threeNum), 1, fptr);
    }
    fclose(fptr);

    return 0;
}
```

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

struct threeNum
{
    int n1, n2, n3;
};

int main()
{
    int n;
    struct threeNum num;
    FILE *fptr;

    if ((fptr = fopen("C:\\program.bin","rb")) == NULL){
        printf("Error! opening file");

        // Program exits if the file pointer returns NULL.
        exit(1);
    }

    for(n = 1; n < 5; ++n)
    {
        fread(&num, sizeof(struct threeNum), 1, fptr);
        printf("n1: %d\nn2: %d\nn3: %d", num.n1, num.n2, num.n3);
    }
    fclose(fptr);

    return 0;
}
```

XỬ LÝ FILE



Bài tập: Sinh viên gồm các thông tin: mã sinh viên là số nguyên 4 chữ số, họ và tên đầy đủ, tuổi, giới tính, điểm. Trong đó họ tên đầy đủ gồm: họ, đệm, tên. Điểm gồm: điểm toán, văn, anh và trung bình 3 môn này. Tạo struct phù hợp mô tả thông tin môn học, điểm, thông tin sinh viên. Viết hàm nhập vào thông tin đầy đủ cho 1 sinh viên và trả về sinh viên vừa nhập. Hãy tạo mảng 100 sinh viên và thực hiện:

- Tạo struct dành cho sinh viên
- Thêm mới sinh viên vào danh sách sinh viên.
- Hiển thị toàn bộ danh sách sinh viên hiện có. Thông tin của mỗi sinh viên phải hiển thị đầy đủ trên một dòng.
- Sắp xếp danh sách sinh viên theo tên a->z.
- Sắp xếp danh sách sinh viên theo điểm trung bình 3 môn tăng dần.
- Tìm sinh viên có họ và tên nhập từ bàn phím.
- Ghi thông tin sinh viên trong danh sách hiện có vào file SV.txt.
- Viết menu cho phép người dùng thực hiện các tùy chọn. Trong đó có chức năng thoát chương trình.
- Đọc dữ liệu ra từ File txt cung cấp sẵn, lưu vào mảng và hiển thị.



XỬ LÝ FILE

DANH SACH SINH VIEN HIEN THOI:

Ma SV	Ho	Dem	Ten	Tuoi	Gioi Tinh	Diem Toan	Diem Van	Diem Anh	Diem TBC
1003	nguyen	thi mai	linh	20	n	7.00	8.00	9.00	8.00
1004	Qua	Jaxiao	Hollowa	20	nu	6.00	8.00	5.00	6.33
1002	le	van	tam	20	nam	4.00	5.00	6.00	5.00
1000	tran	van	nam	20	nam	1.00	2.00	3.00	2.00
1005	tran	van	hung	20	nam	6.00	9.00	8.00	7.67
1006	le	hoai	ha	20	nu	9.00	9.00	8.00	8.67
1007	tran	thi thuong	thuong	20	nu	7.00	7.00	8.00	7.33

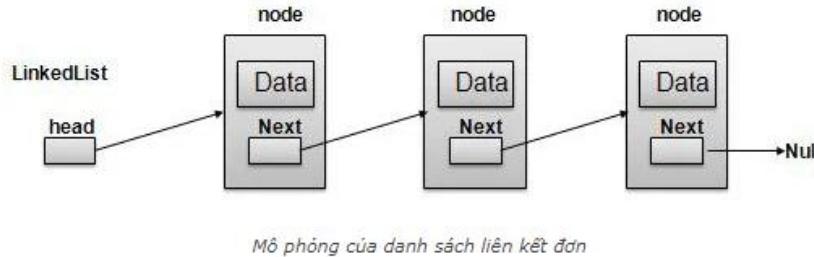
LINKED LIST (DANH SÁCH LIÊN KẾT)



Danh sách liên kết là gì?

Danh sách liên kết là một tập hợp các node được phân bổ động, được sắp xếp theo cách mà mỗi node sẽ chứa một giá trị và một con trỏ trỏ đến node tiếp theo nó. Nếu con trỏ là NULL, thì nó là nút cuối cùng trong danh sách.

Một danh sách được liên kết được giữ bằng cách sử dụng một biến con trỏ cục bộ trỏ đến mục đầu tiên của danh sách. Nếu con trỏ đó cũng là NULL, thì danh sách được coi là trống.



LINKED LIST



Về cơ bản, danh sách được liên kết hoạt động như một mảng có thể phát triển và thu nhỏ khi cần thiết, từ bất kỳ điểm nào trong mảng.

Danh sách được liên kết có một số lợi thế so với mảng:

1. Có thể thêm hoặc bớt các mục từ giữa danh sách
2. Không cần xác định kích thước ban đầu

Tuy nhiên, danh sách được liên kết cũng có một số nhược điểm:

1. Không có quyền truy cập "ngẫu nhiên" - không thể tiếp cận mục thứ n trong mảng mà không lặp lại lần đầu trên tất cả các mục cho đến khi mục đó. Điều này có nghĩa là chúng ta phải bắt đầu từ đầu danh sách và đếm số lần chúng ta tiến lên trong danh sách cho đến khi chúng ta đến được mục mong muốn.
2. Cấp phát bộ nhớ động và con trỏ là bắt buộc, điều này làm phức tạp mã code và tăng nguy cơ rò rỉ bộ nhớ và lỗi phân đoạn.
3. Danh sách được liên kết tiêu tốn bộ nhớ lớn hơn nhiều so với mảng, vì các node trong danh sách liên kết được cấp phát động (điều này kém hiệu quả hơn trong việc sử dụng bộ nhớ) và mỗi mục trong danh sách cũng phải lưu trữ một con trỏ bổ sung.

LINKED LIST



1. Khởi tạo cấu trúc của Node
 - Bao gồm biến lưu data
 - Con trỏ để trỏ đến Node tiếp theo trong List

Tại sao next lại là kiểu struct node của chính nó?

Bởi vì nó là con trỏ trỏ đến một thằng Node khác có cùng kiểu dữ liệu với nó.

```
typedef struct node{  
    int data;  
    struct node *next;  
} node_t;
```

LINKED LIST



1. Khởi tạo cấu trúc của Node

- Bao gồm biến lưu data
- Con trỏ để trỏ đến Node tiếp theo trong List

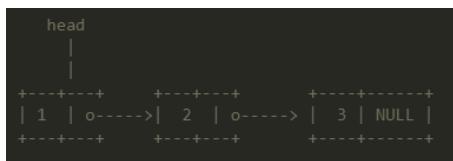
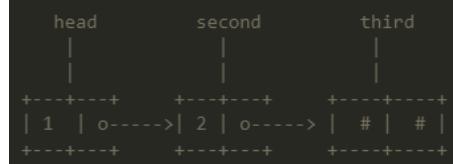
Tại sao next lại là kiểu struct node của chính nó?

Bởi vì nó là con trỏ trỏ đến một thằng Node khác có cùng kiểu dữ liệu với nó.

```
typedef struct node{
    int data;
    struct node *next;
} node_t;
```



Sau đây là ví dụ đơn giản về Linked List với 3 Node được khởi tạo.



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

typedef struct node{
    int data;
    struct node *next;
} node_t;

// Hàm in tất cả các Node trong List
void printList(node_t* head)
{
    node_t *pt = head;
    while (pt != NULL) {
        printf(" %d ", pt->data);
        pt = pt->next;
    }
}

int main()
{
    node_t* head = NULL;
    node_t* second = NULL;
    node_t* third = NULL;

    // allocate 3 nodes in the heap
    head = (node_t*)malloc(sizeof(node_t));
    second = (node_t*)malloc(sizeof(node_t));
    third = (node_t*)malloc(sizeof(node_t));

    head->data = 1; // assign data in first node
    head->next = second; // Link first node with second

    second->data = 2; // assign data to second node
    second->next = third;

    third->data = 3; // assign data to third node
    third->next = NULL;

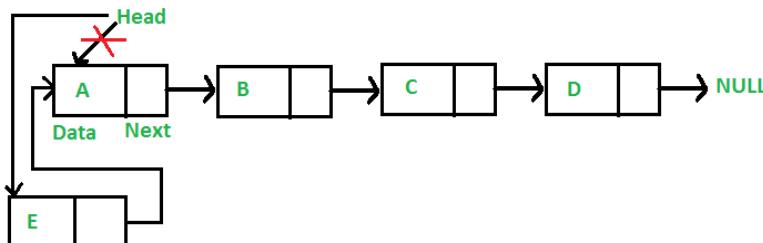
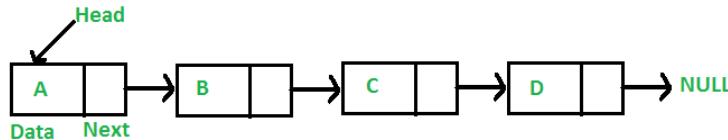
    printList(head);

    return 0;
}
```

LINKED LIST



2. Hàm thêm 1 Node mới đầu vị trí đầu của List



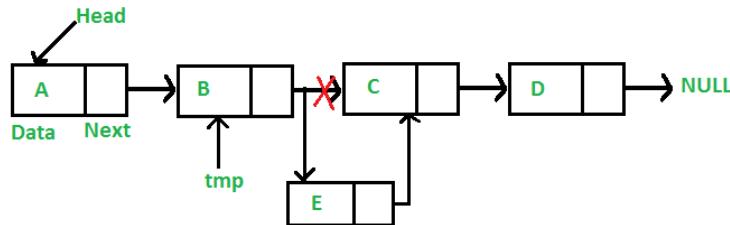
```
// khởi tạo mới 1 Node
static node_t *createNode(int val)
{
    node_t *temp = (node_t*)malloc(sizeof(node_t));
    temp->data = val;
    temp->next = NULL;
    return temp;
}

// thêm 1 Node mới vào đầu danh sách
void pushHead(node_t **head, int val)
{
    node_t *new_node = createNode(val);
    if (*head == NULL)
    {
        *head = new_node;
    }
    else{
        new_node->next = *head;
        *head = new_node;
    }
}
```

LINKED LIST



3. Thêm 1 Node mới vào vị trí bất kì của List



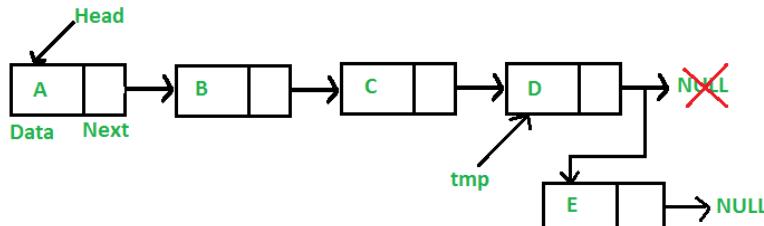
```
bool pushNodeAt(node_t *head, int val, int pos)
{
    int k = 0;
    node_t *pt = head;
    while(pt != NULL && k != pos-1)
    {
        pt = pt->next;
        k++;
    }

    if(k!=pos-1) {
        return false;
    }
    else{
        node_t *new_node = createNode(val);
        new_node->next = pt->next;
        pt->next = new_node;
    }
    return true;
}
```

LINKED LIST



4. Thêm 1 Node mới vào cuối của List



```
void pushTail(node_t *head, int val)
{
    node_t *pt = head;
    node_t *new_node = createNode(val);
    while(pt->next != NULL)
    {
        pt = pt->next;
    }
    pt->next = new_node;
}
```

LINKED LIST



5. Xóa 1 Node ở đầu List

```
void removeHead(node_t **head)
{
    node_t *new_head = (*head) ->next;
    free(*head);
    *head = new_head;
}
```

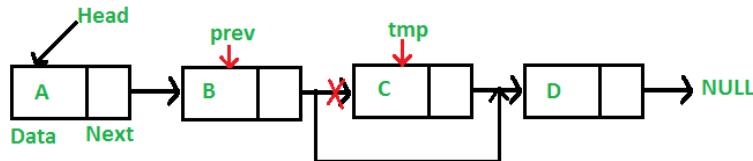
6. Xóa 1 Node ở cuối List

```
void removeTail(node_t *head)
{
    node_t *pt = head;
    while(pt->next->next != NULL)
    {
        pt = pt->next;
    }
    free(pt->next);
    pt->next = NULL;
}
```

LINKED LIST



7. Xóa 1 Node ở vị trí bất kì trong List



```
bool removeNodeAt(node_t *head, int pos)
{
    int k = 0;
    node_t *pt = head;
    while(pt != NULL && k != pos-1)
    {
        pt = pt->next;
        k++;
    }
    if(k != pos-1) {
        return false;
    }
    else{
        node_t *nodeToDelete = pt->next;
        pt->next = pt->next->next;
        free(nodeToDelete);
    }
    return true;
}
```

LINKED LIST



Bài tập

1. Viết hàm tính độ dài của List
2. Viết hàm lấy giá trị data của Node tại vị trí cho trước.
3. Viết hàm tìm vị trí của Node trong List có giá trị data cho trước.
4. Viết hàm in ra toàn bộ Node của List
5. Nhập 1 mảng linked list và tìm giá trị lớn nhất trong mảng
6. Viết hàm hoán đổi vị trí của 2 Node bất kì trong List
7. Nhập 1 mảng linked list và sắp xếp mảng theo thứ tự giá trị tăng dần
8. Viết hàm hoán đổi vị trí 2 Node trong Linked List
9. Giải quyết bài toán quản lý sinh viên bằng linked list

VARIADIC FUNCTION



1. Variadic function là gì ?

Là các hàm có thể truyền vào đa dạng các tham số mà không bị cố định số lượng tham số truyền vào.

2. Vì sao cần Variadic function ?

Giả sử bạn đang cần viết hàm cộng các số nguyên. Vậy nếu cộng 2 số nguyên ta có hàm sau:

```
int cong(int so1, int so2)
{
    return (so1+so2);
}
```

Nếu có 3 số nguyên ta có hàm sau:

```
int cong(int so1, int so2, int so3)
{
    return (so1+so2+so3);
}
```

Như vậy chúng ta thấy, mỗi lần chúng ta muốn thêm tham số truyền vào chúng ta lại phải định nghĩa ra một hàm mới. Vì vậy variadic function chính là giải pháp cho các hàm truyền vào số tham số thay đổi.

VARIADIC FUNCTION



3. Cách sử dụng

Đầu tiên chúng ta cần include thư viện ***stdarg.h***

Ngoài ra bạn cần nắm được một số phương thức trong thư viện này:

- ***va_list*** chứa các thông tin liên quan đến ***va_start*, *va_arg*, *va_end***.
- ***va_start*** và ***va_end*** dùng để bắt đầu và kết thúc quá trình lấy các tham số trong list tham số truyền vào.
- ***va_arg*** sẽ đại diện cho các tham số được thêm vào dấu ...

Methods	Description
<i>va_start(va_list ap, argN)</i>	This enables access to variadic function arguments.
<i>va_arg(va_list ap, type)</i>	This one accesses the next variadic function argument.
<i>va_copy(va_list dest, va_list src)</i>	This makes a copy of the variadic function arguments.
<i>va_end(va_list ap)</i>	This ends the traversal of the variadic function arguments.

VARIADIC FUNCTION



Cú pháp của một hàm Variadic sẽ bao gồm ít nhất một tham số (tham số này sẽ cho chúng ta biết có bao nhiêu tham số được truyền vào), và phía sau là dấu ... biểu thị cho việc có thể có nhiều tham số truyền vào phía sau.

```
int function_name(data_type variable_name, ...);
```

VARIADIC FUNCTION



Ta xét ví dụ sau:

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

// Khai báo 1 variadic function với n tham số truyền vào
int AddNumbers(int n, ...)
{
    int Sum = 0;
    // Định nghĩa 1 con trỏ trả về argument list
    va_list ptr;

    // Bắt đầu quá trình lấy tham số truyền vào từ list
    va_start(ptr, n);

    for (int i = 0; i < n; i++)
    {
        // Lấy tham số ra bằng phương thức va_arg
        // Cân truyền vào list và kiểu dữ liệu của tham số muốn lấy ra
        Sum = Sum + va_arg(ptr, int);
    }

    // Kết thúc quá trình lấy tham số
    va_end(ptr);

    return Sum;
}

int main()
{
    printf("\n\n Variadic functions: \n");

    // Variable number of arguments
    printf("\n 1 + 2 = %d ",
           AddNumbers(2, 1, 2));

    printf("\n 3 + 4 + 5 = %d ",
           AddNumbers(3, 3, 4, 5));

    printf("\n 6 + 7 + 8 + 9 = %d ",
           AddNumbers(4, 6, 7, 8, 9));

    printf("\n");
    return 0;
}
```

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

// Khai báo 1 variadic function với n tham số truyền vào
int AddNumbers(char *sophantu, ...)
{
    int Sum = 0;
    // Định nghĩa 1 con trỏ trả về argument list
    va_list ptr;

    int N = atoi(sophantu);

    // Bắt đầu quá trình lấy tham số truyền vào từ list
    va_start(ptr, N);

    for (int i = 0; i < N; i++)
    {
        // Lấy tham số ra bằng phương thức va_arg
        // Cân truyền vào list và kiểu dữ liệu của tham số muốn lấy ra
        Sum = Sum + va_arg(ptr, int);
    }

    // Kết thúc quá trình lấy tham số
    va_end(ptr);

    return Sum;
}

int main()
{
    printf("\n\n Variadic functions: \n");
    // Variable number of arguments
    printf("\n 1 + 2 = %d ",
           AddNumbers("2", 1, 2));
    printf("\n 3 + 4 + 5 = %d ",
           AddNumbers("3", 3, 4, 5));
    printf("\n 6 + 7 + 8 + 9 = %d ",
           AddNumbers("4", 6, 7, 8, 9));
    printf("\n");
    return 0;
}
```

VARIADIC FUNCTION



Bài tập

1. Viết hàm tìm max các số nguyên truyền vào
2. Viết hàm trả về giá trị trung bình của các tham số truyền vào là số float
3. Viết 1 hàm printf đơn giản theo ý của bạn. Truyền vào gì in ra đó.

VARIADIC MACRO



1. Variadic macro là gì ?

- Variadic macro giống như 1 macro có thể chứa được các tham số truyền vào 1 cách đa dạng.
- Thành phần quan trọng nhất của **Variadic Macro** là keyword **_VA_ARGS_**

Ví dụ:

```
9 #include <stdio.h>
10 #include <stdarg.h>
11
12 #define LOG(...) printf(__VA_ARGS__)
13
14 int main(void)
15 {
16     LOG("Hello World\n");
17     LOG("%d + %d = %d", 1, 2, 3);
18 }
```

A screenshot of a terminal window showing the output of the program. The output consists of two lines: "Hello World" followed by "1 + 2 = 3".

Dấu... được dùng để chỉ ra rằng đó là tham số không bắt buộc. Tất cả các tham số sẽ được truyền tới keyword **_VA_ARGS_**

VARIADIC MACRO



Ứng dụng viết hàm LOG

```
#define dbgprintf(...) printf("%s(%d) :%s", __FILE__, __LINE__, __VA_ARGS__)

int main()
{
    dbgprintf("Hello World");
    return 0;
}
```

→ printf("%s (%d) :%s", __FILE__, __LINE__, "Hello World");

```
#define dbgprintf(FormatLiteral, ...) printf("%s(%d) :" FormatLiteral, __FILE__, __LINE__, __VA_ARGS__)

int main()
{
    dbgprintf("Restart is %d", 12);
    return 0;
}
```

→ printf("%s(%d) :Restart is %d", __FILE__, __LINE__, 12);

Tuy nhiên khi viết `dbgprintf ("Restart");` sẽ gây ra lỗi.

Vì nó sẽ trở thành `printf ("%s(%d) :Restart", __FILE__, __LINE__,);`

Cách khắc phục: `#define dbgprintf(FormatLiteral, ...) printf("%s(%d) :" FormatLiteral, __FILE__, __LINE__, ##__VA_ARGS__)`
=> Giờ thì bạn có thể thoải mái truyền thêm tham số hoặc không

VARIADIC MACRO



Thực hành viết hàm LOG in theo level được định nghĩa

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

typedef enum{
    LOG_LEVEL_ERROR,
    LOG_LEVEL_WARNING,
    LOG_LEVEL_INFO,
    LOG_LEVEL_DEBUG
} log_level_t;

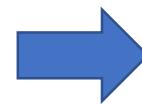
#define NRF_LOG_ENABLED 1
#define NRF_LOG_DEFAULT_LEVEL    LOG_LEVEL_DEBUG

#define END_OF_LINE_STRING "\r\n"
#define LOG_ERROR(...)          LOG_INTERNAL_ERROR( __VA_ARGS__ )
#define LOG_WARNING(...)         LOG_INTERNAL_WARNING( __VA_ARGS__ )
#define LOG_INFO(...)            LOG_INTERNAL_INFO( __VA_ARGS__ )
#define LOG_DEBUG(...)           LOG_INTERNAL_DEBUG( __VA_ARGS__ )

#define LOG_INTERNAL_ERROR(...) \
    LOG_INTERNAL_MODULE(LOG_LEVEL_ERROR, __VA_ARGS__)
#define LOG_INTERNAL_WARNING(...) \
    LOG_INTERNAL_MODULE(LOG_LEVEL_WARNING, __VA_ARGS__)
#define LOG_INTERNAL_INFO(...) \
    LOG_INTERNAL_MODULE(LOG_LEVEL_INFO, __VA_ARGS__)
#define LOG_INTERNAL_DEBUG(...) \
    LOG_INTERNAL_MODULE(LOG_LEVEL_DEBUG, __VA_ARGS__)

#define LOG_INTERNAL_MODULE(level,...)
    if (NRF_LOG_ENABLED && (level <= NRF_LOG_DEFAULT_LEVEL))
    {
        printf(__VA_ARGS__);
    }

int main()
{
    LOG_ERROR("Level of error is %d\n", LOG_LEVEL_ERROR);
    LOG_WARNING("Level of warning is %d\n", LOG_LEVEL_WARNING);
    LOG_INFO("Level of info is %d\n", LOG_LEVEL_INFO);
    LOG_DEBUG("Level of debug is %d\n", LOG_LEVEL_DEBUG);
    return 0;
}
```



Giải thích đoạn code

VARIADIC MACRO



BTVN

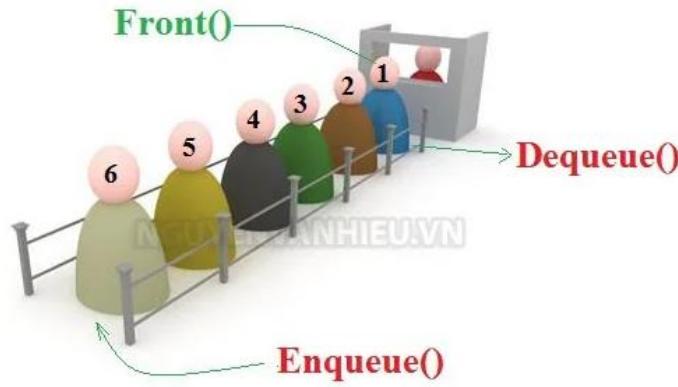
1. Xây dựng thư viện Log và sử dụng nó trong tối thiểu 2 File khác nhau trong 1 Project

QUEUE



Hàng đợi(tiếng anh: Queue) là một cấu trúc dữ liệu dùng để lưu giữ các đối tượng theo cơ chế **FIFO** (viết tắt từ tiếng Anh: *First In First Out*), nghĩa là “vào trước ra trước”.

Hình ảnh về hàng đợi rất hay gặp trong đời sống hàng ngày, hình ảnh việc xếp hàng dưới đây là một mô phỏng dễ hiểu nhất cho cấu trúc dữ liệu hàng đợi(queue): Người vào đầu tiên sẽ được tiến đón đầu tiên; Người mới vào bắt buộc phải xếp hàng ở phía cuối.



Cấu trúc dữ liệu hàng đợi

Trong cấu trúc hàng đợi(queue), ta chỉ có thể thêm các phần tử vào một đầu của queue(giả sử là cuối), và cũng chỉ có thể xóa phần tử ở đầu còn lại của queue(tạm gọi là đầu). Như vậy, ở một đầu không thể xảy ra hai hành động thêm và xóa đồng thời.

QUEUE



Với cấu trúc Hàng đợi(Queue), chúng ta có các chức năng sau:

- **EnQueue**: Thêm phần tử vào cuối(*rear*) của Queue.
- **DeQueue**: Xóa phần tử khỏi đầu(*front*) của Queue. Nếu Queue rỗng thì thông báo lỗi.
- **IsEmpty**: Kiểm tra Queue rỗng.
- **Front**: Lấy giá trị của phần tử ở đầu(*front*) của Queue. Lấy giá trị không làm thay đổi Queue.

Để cài đặt được Queue, chúng ta sẽ cần sử dụng các biến như sau:

- *queue[]*: Một mảng một chiều mô phỏng cho hàng đợi
- *arraySize*: Số lượng phần tử tối đa có thể lưu trữ trong *queue*
- *front*: Phần tử đầu của queue.
- *rear*: Phần tử cuối của queue.

Before Dequeue:

22	45	15	90	60	88	77	5
0	1	2	3	4	5	6	7

Front = 0 Rear = 7

After Dequeue:

	45	15	90	60	88	77	5
0	1	2	3	4	5	6	7

Front = 1 Rear = 7

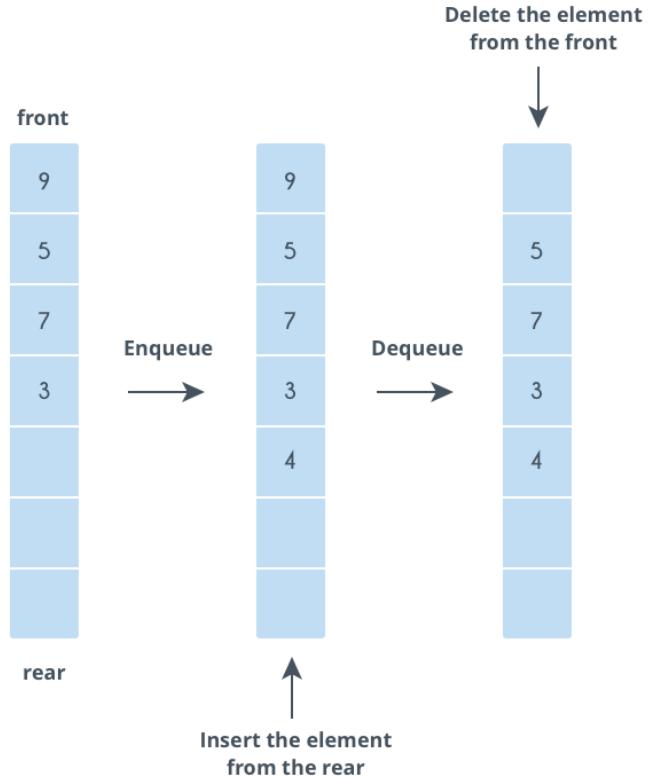
QUEUE



```
// Khai báo cấu trúc của 1 queue
typedef struct {
    int front;           // phần tử đầu queue
    int rear;            // phần tử cuối queue
    int size;             // số phần tử đang có trong queue
    unsigned capacity; // kích thước tối đa của queue
    int* array;          // mảng 1 chiều tương chung cho queue
} Queue;

// Tao queue
Queue* createQueue(unsigned capacity)
{
    Queue* queue = (Queue*)malloc(sizeof(Queue));
    queue->capacity = capacity;
    queue->front = queue->size = 0;
    queue->rear = -1;
    queue->array = (int*)malloc(queue->capacity * sizeof(int));
    return queue;
}
```

QUEUE



Cơ chế hoạt động của hàng đợi(chỉ số tăng theo chiều từ trên xuống dưới)

QUEUE



```
// check queue full
int isFull(Queue* queue)
{
    return (queue->size == queue->capacity);
}

// check queue rỗng
int isEmpty(Queue* queue)
{
    return (queue->size == 0);
}

// Add 1 item vào cuối của queue
void enqueue(Queue* queue, int item)
{
    if (isFull(queue)) // nếu queue full thì không cho add
        return;
    queue->rear = (queue->rear + |1);
    queue->array[queue->rear] = item;
    queue->size = queue->size + 1;
}
```

QUEUE



```
// Remove item ra khỏi đầu của queue
void dequeue(Queue* queue)
{
    if (isEmpty(queue)) // nếu queue rỗng thì không cho remove
        return;
    int item = queue->array[queue->front];
    queue->front = (queue->front + 1);
    queue->size = queue->size - 1;
}

// Lấy giá trị item đầu của queue
int front(Queue* queue)
{
    return queue->array[queue->front];
}

// Lấy giá trị item cuối của queue
int rear(Queue* queue)
{
    return queue->array[queue->rear];
}
```

QUEUE



```
int main()
{
    Queue* queue = createQueue(100);

    enqueue(queue, 10);
    enqueue(queue, 20);
    enqueue(queue, 30);
    enqueue(queue, 40);

    printf("Front item is %d\n", front(queue));
    printf("Rear item is %d\n", rear(queue));
    return 0;
}
```

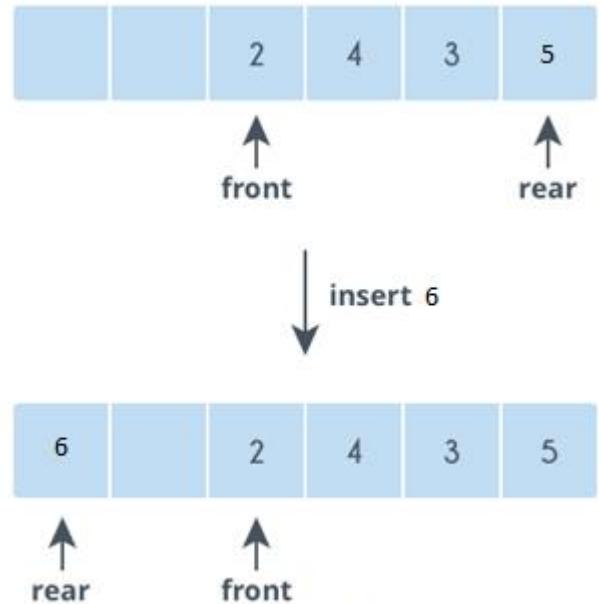
CIRCULAR QUEUE



Hàng đợi vòng là một cải tiến của hàng đợi tiêu chuẩn. Trong hàng đợi tiêu chuẩn, khi một phần tử bị xóa khỏi hàng đợi, các vị trí bị xóa đó sẽ không được tái sử dụng. Hàng đợi vòng sinh ra để khắc phục sự lãng phí này.

```
// Add 1 item vào cuối của queue
void enqueue (Queue* queue, int item)
{
    if (isFull(queue)) // nếu queue full thì không cho add
        return;
    queue->rear = (queue->rear + 1) % queue->capacity;;
    queue->array[queue->rear] = item;
    queue->size = queue->size + 1;
}

// Remove item ra khỏi đầu của queue
void dequeue (Queue* queue)
{
    if (isEmpty(queue)) // nếu queue rỗng thì không cho remove
        return;
    int item = queue->array[queue->front];
    queue->front = (queue->front + 1) % queue->capacity;;
    queue->size = queue->size - 1;
}
```



Hàng đợi vòng sử dụng lại các vị trí đã bị xóa ở đầu mảng

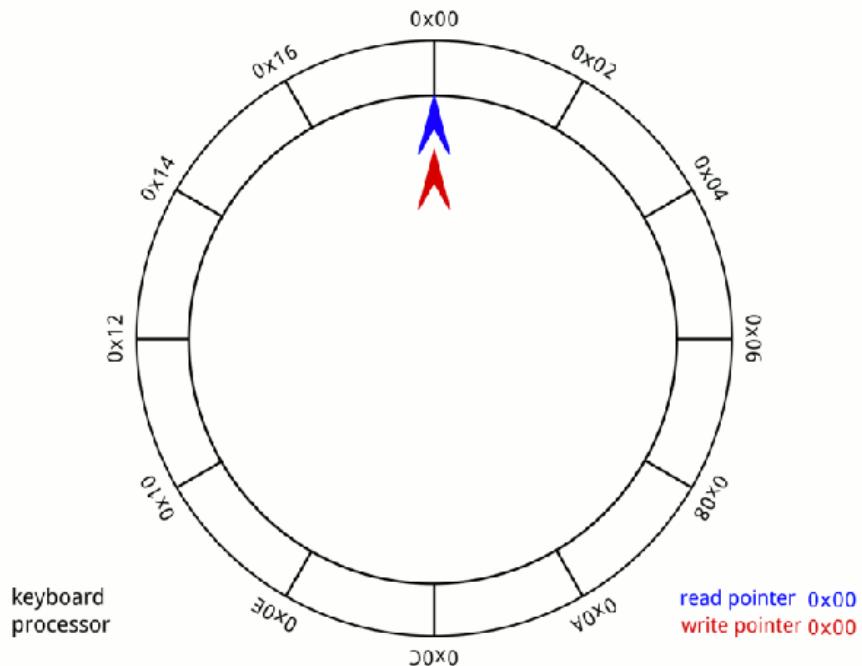
CIRCULAR QUEUE



BT

1. Thiết kế 1 chương trình 1s đẩy 1 dữ liệu vào queue. Sau đó đọc dữ liệu từ queue ra và đẩy vào 1 mảng gồm 20 phần tử. Khi đầy 20 phần tử mảng thì sắp xếp mảng theo thứ tự tăng dần và in ra màn hình. Tiếp tục làm thế với các dữ liệu tiếp theo.

RING BUFFER



RING BUFFER



Ringbuffer là một dạng của hàng đợi vòng.

1. Khai báo kiểu dữ liệu của một Ring buffer

```
typedef struct
{
    volatile uint32_t head;           // phần tử đầu
    volatile uint32_t tail;          // phần tử cuối
    volatile uint32_t size;          // max size
    volatile uint8_t *pt;            // con trỏ tới buffer của ring
} RINGBUF;
```

RING BUFFER



2. Khai báo Ring buffer

```
/**\n * \brief init a RINGBUF object\n * \param r pointer to a RINGBUF object\n * \param buf pointer to a byte array\n * \param size size of buf\n * \return 0 if successfull, otherwise failed\n */\nint32_t RINGBUF_Init(RINGBUF *r, uint8_t* buf, uint32_t size)\n{\n    if(r == NULL || buf == NULL || size < 2) return -1;\n\n    r->pt = buf;\n    r->head = 0;\n    r->tail = 0;\n    r->size = size;\n\n    return 0;\n}
```

RING BUFFER



3. Put 1 phần tử vào ring

```
/**  
 * \brief put a character into ring buffer  
 * \param r pointer to a ringbuf object  
 * \param c character to be put  
 * \return 0 if successfull, otherwise failed  
 */  
int32_t RINGBUF_Put(RINGBUF *r, uint8_t c)  
{  
    uint32_t temp;  
    temp = r->head;  
    temp++;  
    if(temp >= r->size)  
    {  
        temp = 0;  
    }  
    if(temp == r->tail)  
    {  
        return -1;      // ring buffer is full  
    }  
  
    r->pt[r->head] = c;  
    r->head = temp;  
  
    return 0;  
}
```

RING BUFFER



4. Get 1 phần tử từ ring

```
/**\n * \brief get a character from ring buffer\n * \param r pointer to a ringbuf object\n * \param c read character\n * \return 0 if successfull, otherwise failed\n */\nint32_t RINGBUF_Get(RINGBUF *r, uint8_t* c)\n{\n    if(r->tail == r->head)\n    {\n        return -1;           // ring buffer is empty, this should be atomic operation\n    }\n    *c = r->pt[r->tail];\n    r->tail++;\n    if(r->tail >= r->size)\n    {\n        r->tail = 0;\n    }\n    return 0;\n}
```

RING BUFFER



BT

1. Thiết kế 1 chương trình 1s đẩy 1 dữ liệu vào Ring. Sau đó đọc dữ liệu từ ring ra và đẩy vào 1 mảng gồm 20 phần tử. Khi đầy 20 phần tử mảng thì sắp xếp mảng theo thứ tự tăng dần và in ra màn hình. Tiếp tục làm thế với các dữ liệu tiếp theo.

STATE MACHINE



1. State machine là gì?

State Machine là một mô hình tính toán dựa trên một máy giả định được tạo thành từ một hoặc nhiều trạng thái. Nó chuyển từ trạng thái này sang trạng thái khác để thực hiện các hành động khác nhau.

2. Khi nào ta nên sử dụng State machine ?

Khi chúng ta muốn chia một nhiệm vụ lớn và phức tạp thành những đơn vị nhỏ độc lập.

3. Lợi ích của việc dùng state machine ?

- * dễ dàng theo dõi quá trình chuyển đổi / dữ liệu / sự kiện nào gây ra trạng thái hiện tại của yêu cầu.

- * ngăn chặn các hoạt động câu lệnh trái phép.

- * Bảo trì dễ dàng, quá trình chuyển đổi trạng thái độc lập nhau.

- * Ôn định và dễ dàng thay đổi.

STATE MACHINE



Phân tích bài toán 1: Ta có 1 thiết bị có 3 chế độ hoạt động: Tắt, Phun Tia Nước, Phun Sương. Và thiết bị chỉ có duy nhất 1 nút bấm để chọn giữa các mode chạy. Xây dựng chương trình code sử dụng State Machine.

```
#include <stdio.h>

typedef enum
{
    MODE_1,
    MODE_2,
    MODE_3,
    MODE_NUM
} mode_machine_t;

mode_machine_t mode_machine = MODE_1; // default tắt máy
bool is_press_button = false;

void function_on_mode_1(void)
{
    // tắt máy
}

void function_on_mode_2(void)
{
    // phun tia nước
}

void function_on_mode_3(void)
{
    // phun sương
}

void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) // ham xay ra ngat khi bam nút
{
    mode_machine = (mode_machine+1)%MODE_NUM;
    is_press_button = true;
}
```

STATE MACHINE



Bài tập thực hành: Viết chương trình quản lý sinh viên gồm 4 chức năng bất kì sử dụng State Machine.

STATE MACHINE



Phân tích MIN PROTOCOL



THANKS!



Do you have any questions?

truonggiangbkak58@gmail.com
0969 666 522
deviot.vn