

Bài 8: Memory layout

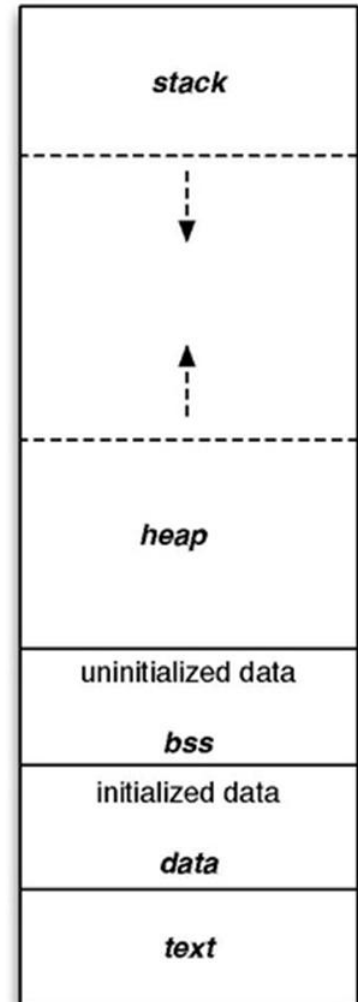
Chương trình main.exe (trên window), main.hex (nạp vào vi điều khiển) được lưu ở bộ nhớ **SSD** hoặc **FLASH**. Khi nhấn run chương trình trên window (cấp nguồn cho vi điều khiển) thì những chương trình này sẽ được copy vào bộ nhớ RAM để thực thi.

Tổng cộng có 5 vùng nhớ:

- Text Segment
- Data Segment(DS và BSS)
- Heap Segment
- Stack Segment

Sự khác nhau ở các phần vùng là

- quyền đọc, ghi
- Địa chỉ, vùng nhớ cấp phát sẽ bị thu hồi khi nào



8.1.Text segment (Code segment)

- Mã máy: chứa tập hợp các lệnh thực thi.
- Quyền truy cập: Text Segment thường có quyền đọc và thực thi, nhưng *không có quyền ghi*.
- Lưu hằng số toàn cục (const), chuỗi hằng - string literal (Clang – macOS, windows - mingW)
- Tất cả các biến lưu ở phần vùng Text đều không thể thay đổi giá trị mà chỉ được đọc.

```
//Text Segment : Chỉ Đọc, thực thi, không được ghi vào
#include <stdio.h>

const int a = 10;      // phân vùng text segment
char arr[] = "Hello";  // text segment
char *arr1 = "Hello";  // text segment
int b = 0;
int *ptr = &b;

int main() {
    // Tất Các câu lệnh thực thi trong main sẽ phân vào text segment.
    // Con trỏ PC sẽ tro đến lần lượt từng câu lệnh để thực thi

    printf("a: %d\n", a);

    arr[3] = 'W';
    printf("arr: %s", arr);

    arr1[3] = 'E';
    printf("arr1: %s", arr1);

    return 0;
}
```

8.2 Data segment

Data segment có 2 phần vùng DS(Initialized) , BSS (unInitialized)

8.2.1.Initialized Data Segment (Chứa Dữ liệu Đã Khởi Tạo):

- Chứa các biến toàn cục được khởi tạo với giá trị khác 0.
- Chứa các biến static (global + local) được khởi tạo với giá trị khác 0.
- Quyền truy cập là đọc và ghi, tức là có thể đọc và thay đổi giá trị của biến .
- Tất cả các biến sẽ được thu hồi sau khi chương trình kết thúc.

8.2.2 BSS

- Uninitialized Data Segment (Dữ liệu Chưa Khởi Tạo hoặc khởi tạo =0):
- Chứa các biến toàn cục khởi tạo với giá trị bằng 0 hoặc không gán giá trị.

- Chứa các biến static với giá trị khởi tạo bằng 0 hoặc không gán giá trị.
- Quyền truy cập là đọc và ghi, tức là có thể đọc và thay đổi giá trị của biến .
- Tất cả các biến sẽ được thu hồi sau khi chương trình kết thúc.

```
#include <stdio.h>
typedef struct
{
    int x;
    int y;
} Point_Data;

Point_Data p1 = {5,0}; //p1, x, y : data
Point_Data p2 = {0,0}; //p2,x, y: BSS
Point_Data p3;          //p3,x, y: BSS
int a = 0; // BSS
int b;      //bss
int *ptr; // bss
char *str = "hello world";
// str: Data
// "hello world" : text (rdata): chuỗi hàng

static int global = 0; //BSS
static int global_2; //BSS
static Point_Data p4 = {0,0}; //p4, x, y :BSS
const int e = 0; // text

void test()
{
    int a = 10; // bss , do đã được khởi tạo ở biến toàn cục
    static int local = 0; // BSS
    static int local_2; // BSS
}

int main() {

    printf("a: %d\n", a);
    printf("global: %d\n", global);
    //str[1] = 'a';
    printf("chuoi: %s\n", str);
    return 0;
}
```

Muốn kiểm tra các biến nằm phân vùng nào biên dịch ra file assembly để xem phân vùng.

The image shows a code editor with two tabs: 'BSS.c' and 'BSS.s'. The 'BSS.c' tab contains C code defining a struct 'Point_Data' and several variables, including static and global ones. The 'BSS.s' tab contains the corresponding assembly code, using directives like '.file', '.text', '.globl', '.data', '.bss', and '.align' to initialize the BSS segment. The assembly code also includes instructions for setting up the stack frame and calling the 'test' function.

```

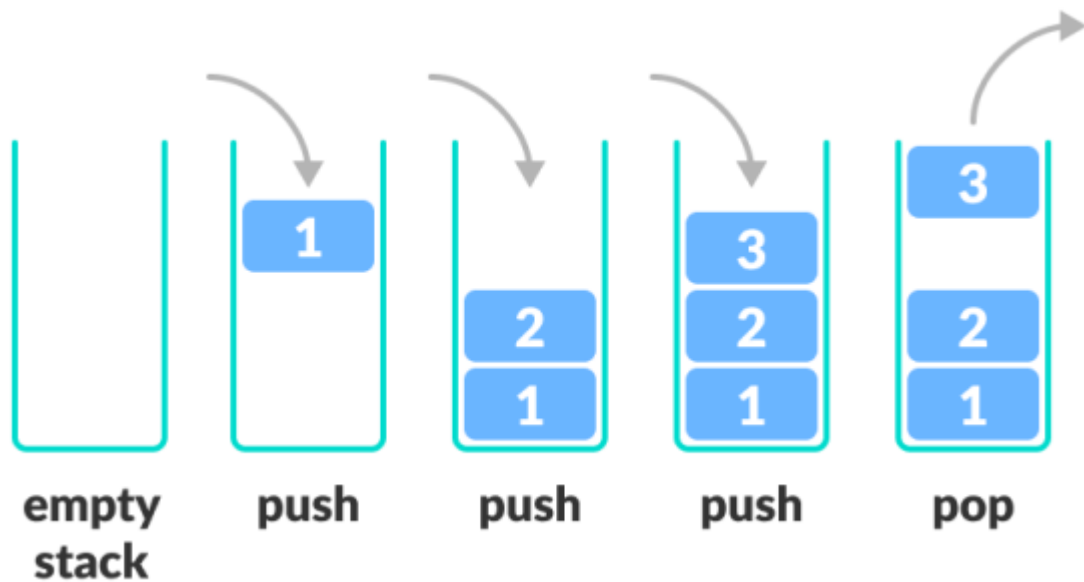
C BSS.c
1 #include <stdio.h>
2 typedef struct
3 {
4     int x;
5     int y;
6 } Point_Data;
7
8 Point_Data p1 = {5,0}; //p1, x, y : data
9 Point_Data p2 = {0,0}; //p2,x, y: BSS
10 Point_Data p3; //p3,x, y: BSS
11 int a = 0; // BSS
12 int b; //bss
13 int *ptr; // bss
14 char *str = "hello world";
15 // str: Data
16
17 static int global = 0; //BSS
18 static int global_2; //BSS
19 static Point_Data p4 = {0,0}; //p4, x, y :BSS
20 const int e = 0; // text
21
22 void test()
23 {
24     int a = 10; // bss , do đã được khởi tạo ở biến to
25     static int local = 0; // BSS
26     static int local_2; // BSS
27 }
28 int main() {
29
30     printf("a: %d\n", a);
31     printf("global: %d\n", global);
32     //str[1] = 'a';
33     printf("chuoii: %s\n", str);
34     return 0;
35 }
36
37
38
39

ASM BSS.s
1 .file "BSS.c"
2 .text
3 .globl p1 //p1 bien toan cu
4 .data // p1 phan vung data
5 .align 8
6 p1:
7 .long 5
8 .long 0
9 .globl p2 // p2 bien toan cuc
10 .bss // p2 o phan vung bss
11 .align 8
12 p2:
13 .space 8
14 .globl p3
15 .align 8
16 p3:
17 .space 8
18 .globl a
19 .align 4
20 a:
21 .space 4
22 .globl b
23 .align 4
24 b:
25 .space 4
26 .globl ptr
27 .align 8
28 ptr:
29 .space 8
30 .globl str
31 .section .rdata,"dr"
32 .LC0:
33 .ascii "hello world\0"
34 .data
35 .align 8
36 str:
37 .quad .LC0
38 .lcomm global,4,4
39 .lcomm global_2,4,4
40 .lcomm p4,8,8

```

8.3. Stack

- Chứa các biến cục bộ (trừ static cục bộ), tham số truyền vào. Biến cục bộ là biến nằm trong các hàm.
- Hằng số cục bộ, có thể thay đổi thông qua con trỏ.
- Quyền truy cập: đọc và ghi, nghĩa là có thể đọc và thay đổi giá trị của biến trong suốt thời gian chương trình chạy.
- Sau khi ra khỏi hàm, tự động thu hồi vùng nhớ ngay. Để không muốn biến bị thu hồi địa chỉ vùng nhớ và vẫn lưu giá trị thì phải khai báo thêm từ khóa “static”.



// Vùng nhớ stack lưu biến cục bộ, và tham số truyền vào hàm. hoạt động theo nguyên tắc LIFO.

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

```
void test()
```

```
{
```

```
    int test = 0; //stack
```

```
    test = 5; // stack
```

```
    printf("test: %d\n",test);
```

```
}
```

```
int sum(int a, int b) // stack : a,b,c
```

```
{
```

```
    int c = a + b;
```

```
    printf("sum: %d\n",c);
```

```
    return c;
```

```
}
```

```
int main() {
```

```
    sum(3,5);
```

```
    /*
```

```
        0x01
```

```
        0x02
```

```
        0x02
```

```
        0x03
```

```
    */
```

```
    test();
```

```
    /*
```

```
        int test = 0; // 0x01
```

```
    */
```

```
    return 0;
}
```

8.4. Heap

Đặt vấn đề:

Viết chương trình yêu cầu người dùng nhập tên, sau đó hiển thị tên vừa nhập. cần phải cấp phát động bộ nhớ:

- Heap được sử dụng để cấp phát bộ nhớ động trong quá trình thực thi của chương trình.
- Điều này cho phép chương trình tạo ra và giải phóng bộ nhớ theo nhu cầu, thích ứng với sự biến đổi của dữ liệu trong quá trình chạy.
- Các hàm như malloc(), calloc(), realloc(), và free() được sử dụng để cấp phát và giải phóng bộ nhớ trên heap.

8.4.1. malloc():

Tham số truyền vào: kích thước mong muốn (byte), số lần của 1 kiểu dữ liệu nào đó (int, double, float, uint8_t...)

Giá trị trả về: con trỏ void, đặc điểm kiểu void là đọc được địa chỉ, nhưng chưa đọc được giá trị tại vùng nhớ mà con trỏ này cấp phát ra. Bởi vậy phải ép kiểu để đọc được giá trị tại vùng nhớ

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

int main() {
    int *arr_malloc;
    int size = 6;

    // Sử dụng malloc
    //malloc cấp phát cho con trỏ ptr vùng nhớ 0x01, 0x02, 0x03... có kích
    //thước = 5 * sizeof(uint16_t)
    //Do hàm malloc trả về kiểu void nên phải ép kiểu uint16_t* để đọc được
    //giá trị lưu trong vùng nhớ đó.
    //con trỏ ptr nằm phân vùng data, bss hoặc stack tùy cách khai báo. nhưng vùng
    //nhớ nó trỏ đến nằm phân vùng heap

    uint16_t *ptr = (uint16_t*)malloc(size * sizeof(uint16_t)); //0x01, 0x02...
    /heap
    ptr[2] = 30;
```

```

for(size_t i=0; i< size; i++){

    printf("dia chi ptr %d : %p, value %d \n",i, ptr+i, *(ptr+i));

}

// Sử dụng calloc
arr_malloc = (int*)calloc(size, sizeof(int));

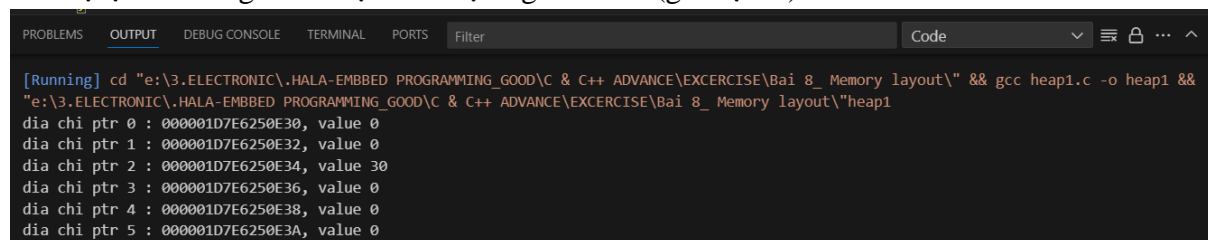
// Giải phóng bộ nhớ
free(ptr); // sau khi sử dụng vùng nhớ xong phải thu hồi lại vùng nhớ
tránh để đó sẽ bị leak

free(arr_malloc);

return 0;
}

```

Kết quả chạy: do uint16_t : kích thước 2 byte nên địa chỉ cũng cách nhau 2 byte.
 Giá trị tại các vùng nhớ được khởi tạo ngẫu nhiên (giá trị rác).



```

[Running] cd "e:\3.ELECTRONIC\HALA-EMBEDDED PROGRAMMING_GOOD\c & c++ ADVANCE\EXERCISE\Bai 8_ Memory layout\" && gcc heap1.c -o heap1 &&
"e:\3.ELECTRONIC\HALA-EMBEDDED PROGRAMMING_GOOD\c & c++ ADVANCE\EXERCISE\Bai 8_ Memory layout\"heap1
dia chi ptr 0 : 000001D7E6250E30, value 0
dia chi ptr 1 : 000001D7E6250E32, value 0
dia chi ptr 2 : 000001D7E6250E34, value 30
dia chi ptr 3 : 000001D7E6250E36, value 0
dia chi ptr 4 : 000001D7E6250E38, value 0
dia chi ptr 5 : 000001D7E6250E3A, value 0

```

8.4.2.Hàm realloc()

Void realloc(void *_memory, size_t _NewSize):

Truyền vào 2 tham số:

- Tham số 1: con trỏ trỏ đến vùng nhớ ban đầu,
- Tham số 2: kích thước mới

Thay đổi vùng nhớ đã được cấp phát cho malloc() hoặc calloc() (tăng hoặc giảm).

Ví dụ vùng nhớ trong heap đang được cấp phát 10 byte, muốn tăng lên 20 byte, 30 byte thì dùng hàm realloc().

Tại sao không dùng hàm malloc() thêm một lần nữa. do nếu khai báo malloc() thêm 1 lần nó sẽ cấp phát ra thêm 1 vùng nhớ khác, vùng cũ vẫn chưa mất đi. Tránh rò rỉ bộ nhớ, lãng phí tài nguyên

```

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

int main() {

```

```

    int *arr_calloc;
    int size = 6;
    //Note kiểu khai báo: *ptr: dấu * đứng trước => nghĩa là ptr là con trỏ.
    //Còn khai báo : datatype* : dấu * đằng trước kiểu dữ liệu => ép kiểu dữ
    liệu này cho con trỏ void.
    // Dùng malloc() cấp phát vùng nhớ.
    uint16_t *ptr = (uint16_t*)malloc(size * sizeof(uint16_t)); //0x01, 0x02...
/heap
    ptr[2] = 30;

    for(size_t i=0; i< size; i++){

        printf("dia chi ptr %d : %p, value %d \n",i, ptr+i, *(ptr+i));

    }
    //realloc() thay đổi kích thước vùng nhớ mà malloc() đã cấp phát mà không
    thay đổi giá trị trong vùng nhớ cũ đó.
    // phải ép kiểu uint16_t cùng kiểu dữ liệu trong vùng nhớ cũ.
    ptr = (uint16_t*)realloc(ptr,10*sizeof(uint16_t));
    printf("Vùng nhớ mới cấp phát lên 10* sizeof(uint16_t)\n");

    for(size_t i=0; i< 10; i++){
        printf("dia chi ptr %d : %p, value %d \n",i, ptr+i, *(ptr+i));
    }

    // Sử dụng calloc
    arr_calloc = (int*)calloc(size, sizeof(int));

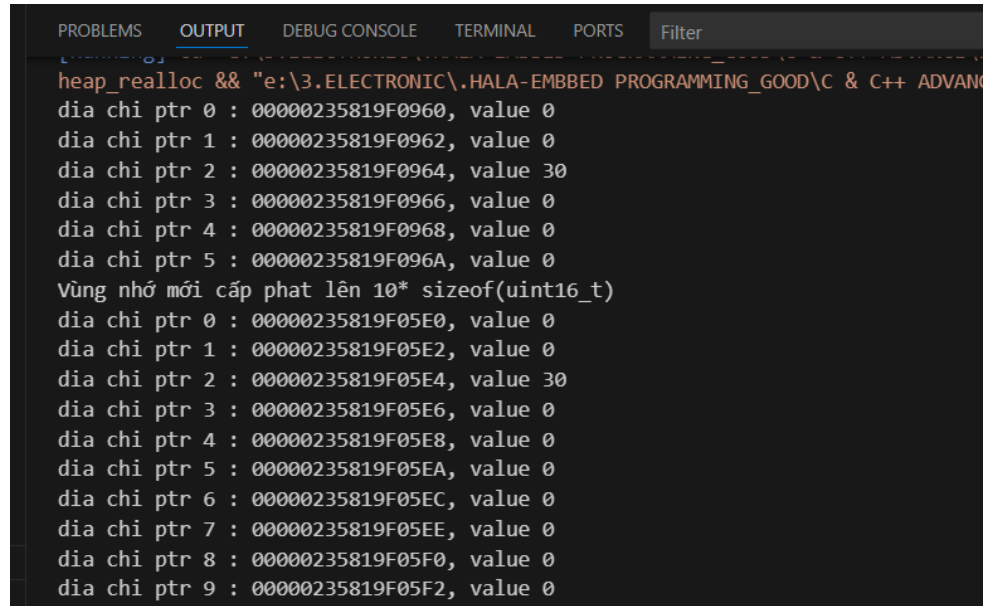
    // Giải phóng bộ nhớ
    free(ptr); // sau khi sử dụng vùng nhớ xong phải thu hồi lại vùng nhớ
    tránh để đó sẽ bị leak

    free(arr_calloc);

    return 0;
}

```


Kết quả



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Filter
heap_realloc && "e:\3.ELECTRONIC\HALA-EMBEDD PROGRAMMING_GOOD\C & C++ ADVANC
dia chi ptr 0 : 00000235819F0960, value 0
dia chi ptr 1 : 00000235819F0962, value 0
dia chi ptr 2 : 00000235819F0964, value 30
dia chi ptr 3 : 00000235819F0966, value 0
dia chi ptr 4 : 00000235819F0968, value 0
dia chi ptr 5 : 00000235819F096A, value 0
Vùng nhớ mới cấp phát lên 10* sizeof(uint16_t)
dia chi ptr 0 : 00000235819F05E0, value 0
dia chi ptr 1 : 00000235819F05E2, value 0
dia chi ptr 2 : 00000235819F05E4, value 30
dia chi ptr 3 : 00000235819F05E6, value 0
dia chi ptr 4 : 00000235819F05E8, value 0
dia chi ptr 5 : 00000235819F05EA, value 0
dia chi ptr 6 : 00000235819F05EC, value 0
dia chi ptr 7 : 00000235819F05EE, value 0
dia chi ptr 8 : 00000235819F05F0, value 0
dia chi ptr 9 : 00000235819F05F2, value 0
```

Chương trình nhập tên

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

int main(int argc, char const *argv[])
{
    int soluongkytu = 0;

    char* ten = (char*) malloc(sizeof(char) * soluongkytu);

    for (int i = 0; i < 3; i++)
    {
        printf("Nhap so luong ky tu trong ten: \n");
        scanf("%d", &soluongkytu);
        ten = realloc(ten, sizeof(char) * soluongkytu);
        printf("Nhap ten cua ban: \n");
        scanf("%s", ten);

        printf("Hello %s\n", ten);
    }

    return 0;
}
```

- Stack: vùng nhớ Stack được quản lý bởi hệ điều hành, dữ liệu được lưu trong Stack sẽ tự động giải phóng khi hàm thực hiện xong công việc của mình.
- Heap: Vùng nhớ Heap được quản lý bởi lập trình viên (trong C hoặc C++), dữ liệu trong Heap sẽ không bị hủy khi hàm thực hiện xong, điều đó có nghĩa bạn phải tự tay

giải phóng vùng nhớ bằng câu lệnh free (trong C), và delete hoặc delete [] (trong C++), nếu không sẽ xảy ra hiện tượng rò rỉ bộ nhớ.

```
- #include <stdio.h>
- #include <stdlib.h>
-
- void test1(){
-     int array[3];
-     for (int i = 0; i < 3; i++){
-         printf("address of array[%d]: %p\n", i, (array+i));
-     }
-     printf("-----\n");
- }
-
- void test2(){
-     int *array = (int*)malloc(3*sizeof(int));
-     for (int i = 0; i < 3; i++){
-         printf("address of array[%d]: %p\n", i, (array+i));
-     }
-     printf("-----\n");
-     free(array);
- }
-
- int main(int argc, char const *argv[]){
-     test1();
-     test1();
-     test2();
-     test2();
-     return 0;
- }
```

- Stack: bởi vì bộ nhớ Stack cố định nên nếu chương trình bạn sử dụng quá nhiều bộ nhớ vượt quá khả năng lưu trữ của Stack chắc chắn sẽ xảy ra tình trạng tràn bộ nhớ Stack (Stack overflow), các trường hợp xảy ra như bạn khởi tạo quá nhiều biến cục bộ, hàm đệ quy vô hạn,...

```
int foo(int x){
    printf("De quy khong gioi han\n");
    return foo(x);
}
```

- Heap: Nếu bạn liên tục cấp phát vùng nhớ mà không giải phóng thì sẽ bị lỗi tràn vùng nhớ Heap (Heap overflow). Nếu bạn khởi tạo một vùng nhớ quá lớn mà vùng nhớ Heap không thể lưu trữ một lần được sẽ bị lỗi khởi tạo vùng nhớ Heap thất bại.

```
int *A = (int *)malloc(18446744073709551615);
```