Chia chức năng cho bài tập.

- Chương trình sẽ gọi các function đấy để chạy chương trình
- Ko nên chia quá nhiều chức năng nhỏ ko cần thiết hoặc ko chia chức năng
- Chia file, trong 1 chương trình phải chia thành nhiều file khác nhau, phân ra làm file cho thư viện và file cho ứng dụng.

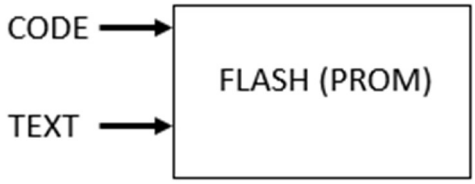
Những file cho thư viện thì phục vụ cho yêu cầu của bài tập, những file ứng dụng là sử dụng để chứng minh được những yêu cầu bài tập chạy đúng hay là chạy sai

Tuân thủ viết code theo coding convention.

Bộ nhớ ROM

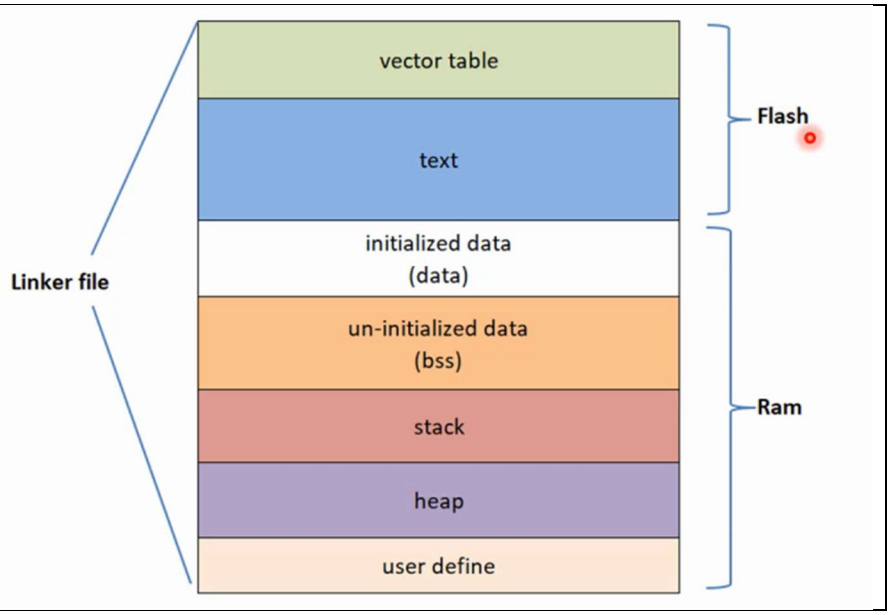
Bộ nhớ **ROM** trong nhúng thì ko đúng, mặc dù là **read only memory** nhưng khi dùng các module có thể ghi được vào bộ nhớ đó dẫn đến bộ nhớ đó ko chỉ là read only cho nên dùng khái niệm ROM sẽ bị sai và ngta dùng khái niệm **flash** hay **prom** còn vùng RAM thì vẫn là như vậy.

Khi build chương trình ra thì phần **code** và phần **text** được đẩy vào trong vùng này.



Bộ nhớ RAM

Vùng **RAM** chia làm 4 vùng cơ bản và 1 vùng khác user define do người sử dụng tự define ra vùng này (có thể là 1 vùng hay nhiều vùng tuỳ người sử dụng).



Chia các vùng bộ nhớ như thế này quan trọng trong quá trình làm việc, quá trình debug, ví dụ khi chạy chương trình sai thì sẽ dựa vào việc phân chia bộ nhớ để phán đoán xem chương trình chạy sai nguyên nhân do đâu.

Ví dụ giả sử chương trình lần đầu tiên chạy đúng, lần thứ 2 chạy sai thì nguyên nhân sẽ dẫn đến vùng stack có thể thiếu, …

Dẫn đến việc chia bộ nhớ rất quan trọng trong quá tình debug.

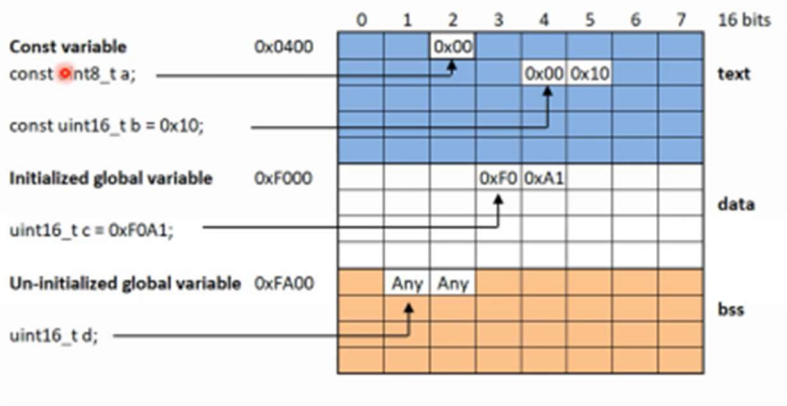
Linker file

Bình thường nsx đưa 1 con chip thì phần **datasheet** ngta chỉ quy định vùng **rom/flash** và ram, những vùng nhỏ kia sẽ được quy định trong 1 file được gọi là **linker file**.

Ví dụ vùng **initialized data** kia bắt đầu từ địa chỉ ô nhớ nào, vùng **bss** sẽ bắt đầu từ ô nhớ đến ô nhớ nào. Về mặt vật lý sẽ ko có những vùng như thế này mà sẽ phải chia bằng linker file.

Vùng user define thì người dùng có thể chọn vùng nhớ từ đâu tới đâu bằng cách sửa linker file để tăng giảm bộ nhớ các vùng khác đi hoặc có thể tăng vùng nhớ của các vùng nhớ cơ bản.

Variable and memory location



Biến hằng số (**const**) được lưu ở trong vùng text (no-volatile memory) (thuộc **Flash**).

Hằng ở bất kỳ đâu đều lưu trong vùng text.

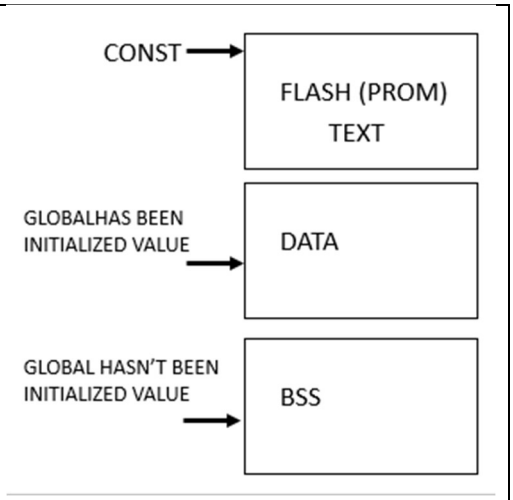
Volatile: có khả năng thay đổi nhanh chóng và không thể đoán trước, đặc biệt là đối với những điều tồi tệ hơn.

- Vùng data chứa các biến toàn cục và có giá trị khởi tạo.

- Nếu ko khởi tạo thì giá trị ban đầu là 0, tuy nhiên 1 số cái trình biên dịch khác nhau thì ko cho giá trị ban đầu hoặc cho bằng 0 hoặc bất kỳ (ko được phép mặc định là khi ko khởi tạo thì auto bằng 0);

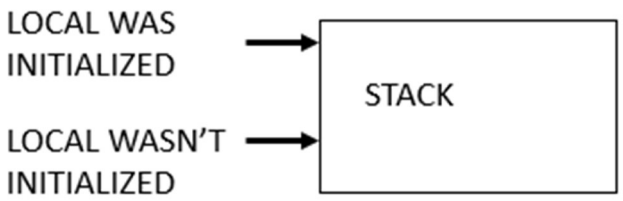
→Khi tạo ra 1 biến nên có giá trị khởi tạo.

- ✓ Khi biến toàn cục được khởi tạo giá trị bằng 0 thì nó sẽ được đẩy vào vùng bss, khác 0 thì vào vùng data.
- ✓ Ko khởi tạo thì nó sẽ vào vùng bss nhưng nó có giá trị bất kỳ.
- ✓ Khi khởi tạo ko có giá trị thì nên gán về 0.



Giả sử tạo ra 2 biến toàn cục liên tiếp nhau thì thông thường **liên tiếp** nhau trong vùng nhớ nhưng về mặt khái niệm thì khi khai báo 2 biến liên tiếp nhau thì nó chưa chắc nằm liên tiếp trong vùng nhớ. (do kinh nghiệm).

Vùng stack học ở trường là lưu trữ các biến local, bên trong thân hàm, có giá trị khởi tạo hay ko cũng lưu trong stack.



Biến tham số được lưu trữ ở đâu?

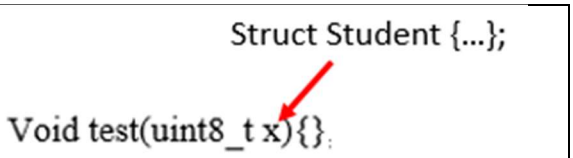
Void test(uint8_t **x**){},

x là biến **tham số**, nếu có giá trị truyền vào thì nó được lưu trữ ở đâu đó.

Trong hệ thống nhúng mà các core arm, core text, các tham số này được lưu trong thanh ghi **R0 đến R3** để lưu trữ các tham số đầu vào và các giá trị trả về.

Ví dụ 1 hàm có 4 tham số thì tham số đầu tiên được lưu trữ trong **thanh ghi R0, R1, R2, R3**. Có nhiều hơn 4 tham số thì tham số thứ 5 thứ 6 được lưu trữ trong stack.
Tức là quy định chỉ có 4 **thanh ghi từ R0 đến R3** để lưu trữ cho cái tham số đầu vào và giá trị trả về, nhiều hơn 4 tham số thì lưu trong stack cho nên chương trình quy định số tham số đầu vào hàm ko được quá nhiều

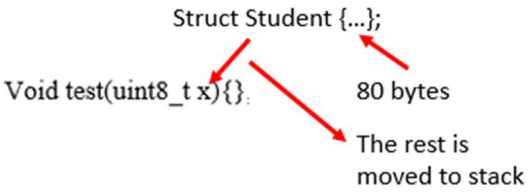
Thông thường chỉ được 4 trở xuống, nhiều hơn 4 thì hàm ta viết chưa thực sự tốt, tham số nhiều hơn 4 thì việc tính toán bị chậm hơn, tham số thứ 5 được lưu trữ trong stack và khi truy xuất đến tham số thứ 5 đấy thì giá trị của tham số thứ 5 đấy phải được tải lên thanh ghi mục đích chung để tính toán.
Thanh ghi R4 đến R12 là các thanh ghi mục đích chung.
Khi tính toán thì các thanh ghi mục đích chung sẽ được sử dụng

Câu hỏi: Tham số truyền vào là struct thì khi truyền vào thanh ghi R thì thế nào? (nếu dung lượng của tham số đấy lớn hơn dung lượng của thanh ghi). **Trả lời:**Trong tất cả các trường hợp thì sẽ bị đẩy vào stack	

Nếu số lượng tham số lớn quá thì nên đẩy vào **struct** thì trong trường hợp đó nên giải quyết như thế nào?

Với **struct** thì dung lượng của nó quá lớn sẽ ko đủ vào thanh ghi đẩy.

- Ko phải dung lượng mà size của kiểu data sẽ lớn.
- Mỗi 1 thanh ghi lưu trữ tối đa (với chip 32 bit) thì thanh ghi tối đa lưu trữ được 32 bit.
- Nếu struct đó vượt quá 32 bit thì ko đủ lưu trong thanh ghi đẩy và nó sẽ bị chuyển xuống lưu trong stack.
- Để giải quyết vấn đề size của struct là 10bytes chẳng hạn nhân với 8 bit là 80 bit.



Để giải quyết vấn đề ko phù hợp với thanh ghi thì chuyển struct đấy thành con trỏ và khi con trỏ sẽ chỉ lưu địa chỉ thôi và khi đấy nó sẽ chỉ cần 32bit lưu địa chỉ của struct là xong (đây là cách giải quyết vấn đề).

Khi truyền struct thì ko truyền trực tiếp giá trị vào**, ko truyền theo kiểu tham trị (sẽ như hình trên) mà truyền theo kiểu tham chiếu.**

Nếu dùng struct thì các kiểu dữ liệu thông thường được truyền theo kiểu thông thường thì sẽ truyền theo kiểu giá trị, nếu nhiều quá thì struct thì nên truyền theo kiểu địa chỉ

Tham số kiểu tham chiếu cũng được lưu trong stack:

```
Uint8_t test1(uint8_t z)
{
        Z = a+5;
        ….
        Return z;
}
```

- z được lưu trong **stack** do **giá trị của tham số được thay đổi trong thân của hàm** đấy thì thông thường tham số đấy được lưu trữ trong vùng **stack**.
- Tuy nhiên mặc dù là như vậy như **khi optimize** thì tham số đấy vẫn được lưu trong thanh ghi.
  z ở đây truyền vào z = 10 thì giá trị bằng 10 đó được lưu trong thanh ghi r0 hay r1 gì đó.
-

Truyền **z** ở đây thì có 2 trường hợp:

Nếu truyền giá trị bằng 10 thì người ta tạo 1 giá trị **z** và lưu trữ trong **stack**. Biến z trong **stack** và được khởi tạo giá trị bằng 10, sau đấy **z = a + 5.** Giả sử **a** là 1 biến toàn cục, **a** lấy giá trị bằng 7 chẳng hạn, thì khi a + 5 thì nó sẽ bằng 12, thì khi đó **z** ở đây sẽ bằng 12 và nằm trong **stack**.	Trường hợp 2 là **z** ko lưu trữ trong **stack** mà nó lưu trữ trực tiếp vào trong **thanh ghi R1** chẳng hạn. Khi **z = a+5** vậy giá trị 7+5 bằng 12 thì nó sẽ lưu trực tiếp giá trị 12 vào **thanh ghi R1** đấy luôn. Bất cứ chỗ nào gặp **z** thì nó sẽ lưu chính giá trị **thanh ghi R1** đấy luôn. Nó sẽ lưu giá trị tức thời của **z** vào trong **thanh ghi R1** và nó sẽ **ko quan tâm** việc biến đổi **z** nữa cả.

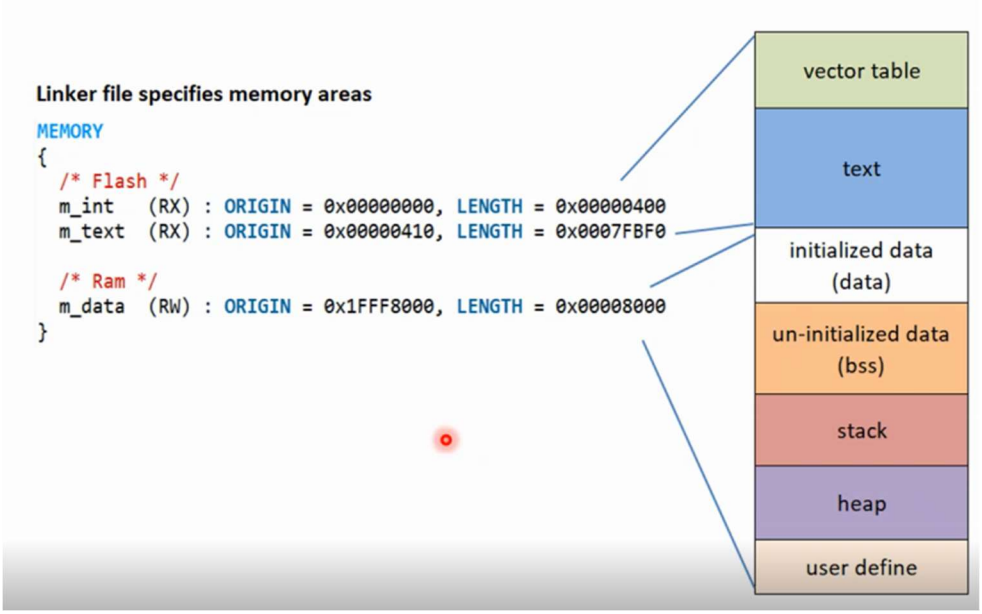
Tức là z bị thay đổi trong thân hàm thì sẽ xảy ra 1 trong 2 tình huống: 1 là z được lưu trong **stack**, tình huống thứ 2 là z được lưu trong **thanh ghi**, tức là nó sẽ ko cố định, cả 2 tình huống đều có khả năng xảy ra.

Còn

```
Void test(uint8_t x)
{
        Uint8_t y =0;
        …
}
```

- x được lưu trong R0 hoặc R1, nhiều tham số hơn (tối đa thêm 3 tham số nữa) thì được lưu đến tận R3 nhưng nhiều hơn nữa thì được lưu trong stack.
- Nếu x ko được thay đổi trong thân hàm thì sao thì nó sẽ luôn luôn được lưu trong thanh ghi r0 đến r3.
- Biến y luôn luôn được lưu trong stack bởi vì nó là biến cục bộ còn biến tham số thì có tình huống 1 là nằm trong stack và 2 là nằm trong thanh ghi.
- Việc optimize được quyết định bởi trình biên dịch, các trình biên dịch khác nhau thì việc optimize khác nhau.

Linker file



Vùng heap liên quan đến pointer

Nhà sản xuất sẽ mô tả cho ta là có vùng **flash**, vùng **ram** và để phân biệt từng vùng trong vùng **flash** hay **ram** thì sẽ được mô tả bởi **linker file**

Linker file defines output sections, below is example:

```
.stack __StackLimit :
{
  . = ALIGN(8);
  __stack_start__ = .;
  . += STACK_SIZE;
  __stack_end__ = .;
} > m_data
```

"." means begin at ".stack __StackLimit:"
-> Stack section begins at __StackLimit address.

```
.mySection :
{
  __mySection_start__ = .;
  . += MY_SECTION_SIZE;
  __mySection_end__ = .;
} > m_data
```

".mySection" is user defined section.
-> ".mySection :" means "user defined section begins after stack section

Ví dụ khai báo các vùng liên tiếp nhau thì mỗi dấu "." sẽ bắt đầu địa chỉ tiếp theo đấy, dấu ">" là đẩy vào vùng nào đó

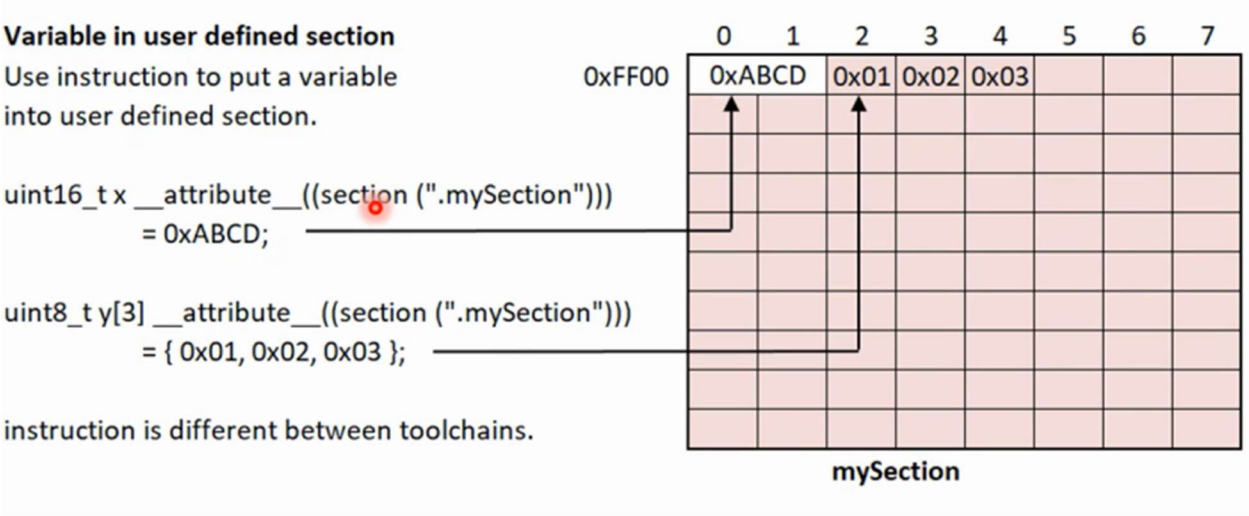
User defined Section

Khi tạo 1 biến nằm trong vùng **user defined thì sao?**

Thông thường khi khai báo **1 biến có giá trị khởi tạo (biến toàn cục, còn local dù có hay không thì cũng trong stack)** thì biến đó nằm trong vùng **data**, 1 biến toàn cục ko có giá trị khởi tạo hoặc bằng 0 thì nằm trong **bss**, biến hằng số nằm trong vùng **text**, biến cục bộ thì nằm trong vùng stack

Vùng user defined thì khi khai báo 1 biến thì sử dụng các **instruction** để đẩy biến đó vào vùng user defined

Ví dụ



Instruction attribute sau đó là đến từ khoá **section .mySection** và giá trị khởi tạo là abcd thì khi đó tạo ra 1 biến x nằm trong vùng user defined ấy và giá trị khởi tạo là abcd. Khi define 1 biến thì sẽ cần **attribute**, khi define xong thì sử dụng các biến này như các biến thông thường

Tương tự như vậy khi khai báo 1 mảng thì nó sẽ vẫn là từ khoá attribute section và tên section của ta, và giá trị khởi tạo là 123 (mảng 3 phần tử 8 bit, giá trị 1 2 3 nằm trong vùng mySection và attribute là instruction để đẩy biến đấy vào vùng mySection user defined.

Khi muốn đẩy biến vào vùng user defined thì ko đẩy trực tiếp được mà đẩy qua các instruction này.

Vùng cơ bản thì cứ khai báo đúng quy định là nó tự đẩy vào, ko cần instruction nào.

Ví dụ khi làm về USB, trong USB có vùng đặc biệt để làm bảng mô tả USB, bảng đấy là 512bytes, nó cần phải allign 512bytes luôn và ko thay đổi giá trị gì thì cần bảng đó thì

phải tạo ra vùng nhớ user defined và sau đấy đẩy biến usb definator vào trong vùng nhớ user defined đấy.

RAM extension

Với nhu cầu lưu trữ thông tin trên các thiết bị di động ngày càng lớn thì **RAM expansion** ra đời nhằm làm tăng dung lượng RAM của máy nhờ thiết lập **RAM ảo**. Nhờ đó RAM được nâng cấp thêm nhiều khoảng trống lưu trữ hơn.

Bộ nhớ RAM cao sẽ nâng tầm trải nghiệm của bạn với những tính năng sau:

- Tăng cường hiệu suất

Giúp tốc độ xử lý của máy nhanh hơn. Đặc biệt là khi người dùng sử dụng các phần mềm chỉnh sửa ảnh, video; ứng dụng game có đồ họa nặng,...

- Đa nhiệm tốt hơn

Dù là cả khi bạn cho chạy đồng thời nhiều ứng dụng trên màn hình thì với dung lượng RAM lớn bạn vẫn có thể thoải mái sử dụng và thực hiện nhiều thao tác cùng lúc.

- Thao tác trên các ứng dụng nhanh hơn

Chính vì bộ nhớ RAM lớn nên các ứng dụng sẽ có nhiều khoảng trống để chạy các thiết lập và phản hồi các yêu cầu của người dùng nhanh hơn.

- Chơi game mượt mà hơn

RAM lớn luôn là tiêu chí hàng đầu của những người muốn sắm điện thoại để phục vụ cho nhu cầu chơi game. Vì vậy với RAM expansion bạn sẽ không cần lo lắng với tốc độ xử lý dữ liệu của máy.

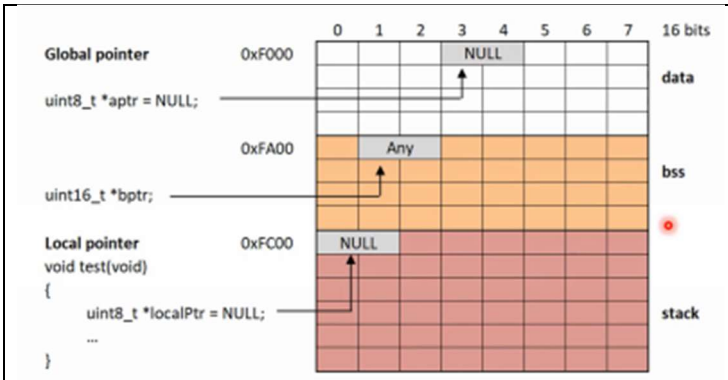
RAM retention là gì, dính dáng tới việc đẩy 1 biến vào user defined

Mọi người có trí nhớ ngắn hạn, kéo dài khoảng 20 phút và trí nhớ dài hạn, vĩnh viễn hơn. Không phải tất cả những ký ức ngắn hạn đều được giữ lại trong dài hạn. Những người mắc chứng hay quên anterograde không thể giữ lại những ký ức dài hạn mới.

Tức là:

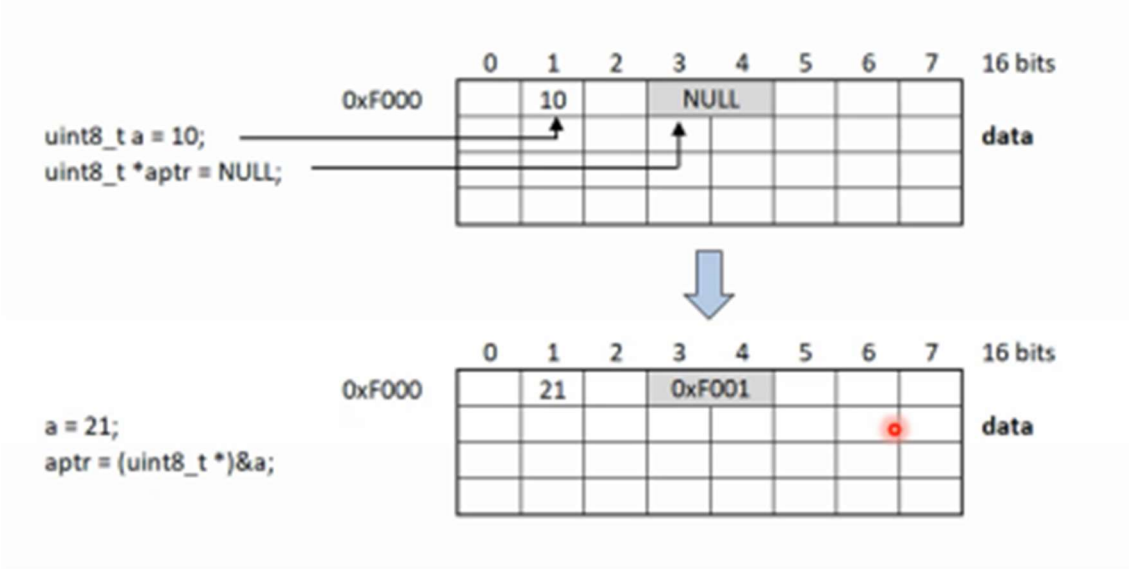
Life time của biến đó tương tự như việc nằm trong vùng RAM, trong quá trình RUN thì sẽ tồn tại và khi RESET thì sẽ bị mất giá trị.

Pointer



- giả định học trên con chip 16bit chứ ko phải 32 bit vì nó ít ô nhớ và dễ hình dung hơn chứ 8 bit thì nó sẽ ko đủ tính bao quát
- Chip 16 bit thì cho 2 ô nhớ, chip 32 bit thì 4 ô nhớ, chip 64 thì 8 ô nhớ để lưu trữ cho biến con trỏ
- Tương tự như các biến thông thường

Gán giá trị cho 1 biến con trỏ



Khi gán giá trị a = 21 (0x15) thì tại ô nhớ lưu trữ giá trị biến a sẽ thay đổi thành 21, tương tự khi gán aprt = (uint8_t*)&a;

Ta thấy a ở ô nhớ thứ 1 tại vùng nhớ bắt đầu từ F000 -> a có địa chỉ tại F001, khi gán aptr bằng địa chỉ của a thì giá trị của con trỏ aptr = F001. So sánh như thế này để nói rằng biến aptr ko khác gì biến ko phải con trỏ cả cũng hoàn toàn giống so với biến a ở đây.

Tức là 1 biến con trỏ thì vẫn là 1 biến thông thường có thể gán giá trị cho nó, khi mà gán giá trị thì ô nhớ lưu trữ giá trị đó thay đổi giá trị. Vậy thôi.

Hoàn toàn giống với 1 biến thông thường.

Khi ko ép kiểu (uint8_t *aptr = NULL; aptr = (uint8_t*)&a;) thì sẽ có warning ko đồng bộ về kiểu dữ liệu

Khi lấy địa chỉ của a thì thành 16bit nhưng với con trỏ thì nó vẫn cần 2 ô nhớ tức là 16 bit để lưu trữ cho con trỏ này (kiểu uint8_t* chứ ko phải kiểu uint8_t), ép kiểu là ép kiểu uint8_t* chứ ko phải ép kiểu uint8_t.

Với uint8_t* thì độ rộng của nó là 16bit với chip 16bit, độ rộng của uint8_t* với chip 32 bit sẽ là 32bit thì khi đẩy địa chỉ của a với chip 16 bit thì địa chỉ của a sẽ là 16 bit và khi ép kiểu về uint8_t thì nó sẽ ép kiểu thành kiểu 16bit và nó hoàn toàn đồng bộ. (cho nên địa chỉ của con trỏ aptr mới sử dụng 2 ô nhớ F003 và F004).

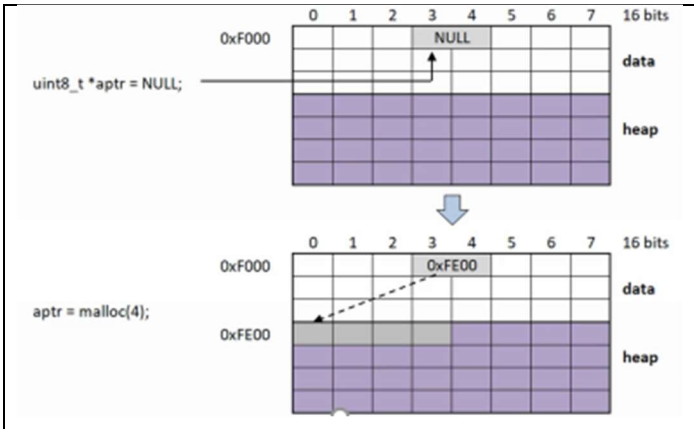
Câu hỏi: Giả sử trong hàm có biến cục bộ có địa chỉ của biến khi mà khai báo bên trong mà mình có một biến con trỏ global thì khi mà mình trỏ vào thì sau khi ra khỏi hàm thì có phải là biến cục bộ sẽ bị giải phóng ko? Lúc đấy mình vẫn trỏ vào và cố tình đọc thì có bị làm sao ko?	**Trả lời:** Ko sao cả, vẫn đọc ra giá trị nhưng giá trị đó ko kiểm soát được, thậm chí ghi được vào đó luôn vì vùng stack vẫn nằm trong Ram và nó vẫn cho phép ta ghi.

Vấn đề là khi ghi vào đó thì sao thì khi chạy chương trình con thì nó sẽ khởi tạo lại biến và nó vẫn nằm cùng giá trị đấy và nó vẫn sẽ được update bằng giá trị chạy trước khi chạy chương trình con thôi và sẽ bị mất, tức là sẽ ko kiểm soát được giá trị nằm trong ô nhớ tại stack đấy, và có thể đọc và ghi hoàn toàn vào đấy luôn, còn chạy đúng hay sai thì ko khẳng định. ở đây khẳng định là trỏ vào con trỏ nằm trong vùng stack, đọc và ghi giá trị vào vùng stack đấy, vấn đề là mục đích là gì và chạy chương trình như thế nào thì ko khẳng định…..

Có 2 khái niệm cần lưu ý ở đây:

- Tham số được lưu trong thanh ghi.
- Tham số còn có thể được lưu trong stack để trong quá trình debug sẽ ko bị quá ngạc nhiên tại sao giá trị của tham số này ko được lưu trong vùng stack.

Cấp phát động cho 1 con trỏ

	Biến aptr nằm trong vùng data, còn chương trình khi cấp phát thì sẽ tìm trong vùng heap những ô nhớ có độ rộng bằng phần mà mình yêu cầu, ví dụ ở đây yêu cầu cấp phát là 4 bytes thì chương trình sẽ tìm trong vùng heap 4 bytes ô nhớ mà ko được cấp phát cho bất kỳ 1 con trỏ nào và nó sẽ trả về giá trị là địa chỉ của cái ô nhớ đầu tiên của vùng nhớ đấy cho con trỏ được cấp phát.

Ta thấy 4 bytes nhớ đầu tiên chưa được cấp phát cho bất kỳ 1 con trỏ nào thì nó sẽ đánh dấu là 4 bytes nhớ này được cấp phát và trả về địa chỉ là FE00 – địa chỉ ô nhớ đầu tiên cho con trỏ aptr.

Khi cấp phát xong thì giá trị tại ô nhớ lưu trữ con trỏ aptr sẽ đặt thẳng giá trị trả về của hàm cấp phát. Hàm cấp phát ở đây đang trả về giá trị FE00.

Khi đấy aptr khi đó sẽ bằng FE00 hoàn toàn giống việc bên trên khi gán aptr bằng địa chỉ a thì giá trị của nó bằng F001 thì ở dưới thay vì việc gán thì cái địa chỉ của a là F001 thì hàm cấp phát động ở đây là hàm malloc sẽ trả về địa chỉ là FE00 thì nó tương đương với việc gán aptr bằng FE00 thôi.

Việc cấp phát và giá trị lưu trữ của con trỏ hoàn toán giống như là gán.

Khi cấp phát động cho con trỏ thì vùng nhớ sẽ luôn cấp phát ở trong heap tại vì vùng heap là để cấp phát động.

Con trỏ NULL

Có 2 chiều hướng:

✓ Thông thường giá trị NULL là giá trị 0.

NULL = (void*)0x00000000.

Thông thường **con trỏ NULL** được ép kiểu thành **con trỏ void của địa chỉ 0**.

Khi gán giá trị như thế thì giá trị ban đầu được khởi tạo thành 0 và có thể nằm trong vùng bss, tuy nhiên 1 số complier ko tạo giá trị kia bằng NULL,

✓ Chiều hướng thứ 2:

NULL = (void*)0xFFFFFFFF được định nghĩa là giá trị invalid là giá trị FFFFFFFF thì khi đấy giá trị khởi tạo đất ko nằm trong vùng bss nữa mà nằm trong vùng data

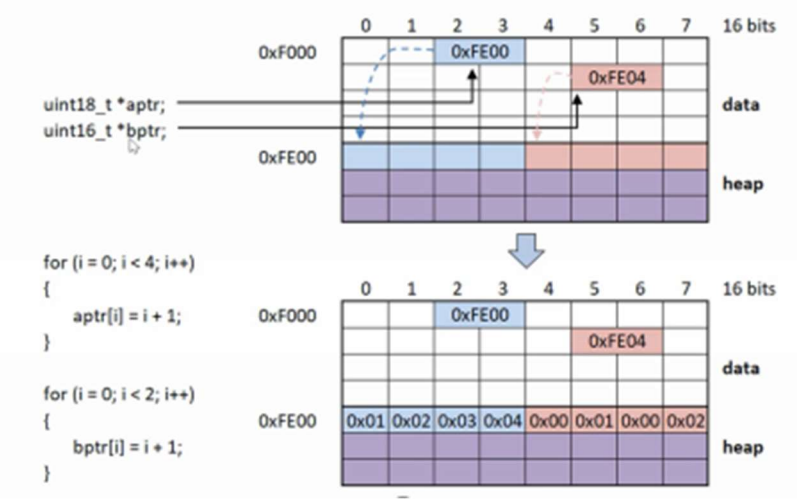
→Tuỳ vào cách defined giá trị NULL như thế nào thì biến con trỏ khi khai báo giá trị NULL sẽ được nằm ở các vùng khác nhau.

ở đây mình có giá trị bằng NULL tức là có giá trị khởi tạo thì mình sẽ hiểu là nó nằm trong vùng data.

ở đây tiếp tục cấp phát cho con trỏ khác thì nó sẽ lấy giá trị tiếp theo hay ko? Lấy địa chỉ tiếp theo hay ko thì có 2 câu trả lời:

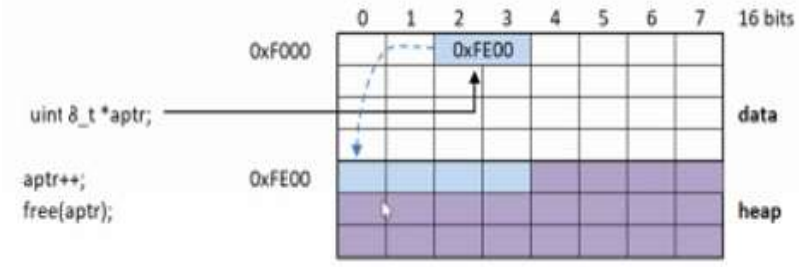
về mặt thực tế khi cấp phát 2 con trỏ liên tiếp nhau thì bộ nhớ được cấp phát liên tiếp nhau, tuy nhiên về mặt lý thuyết thì ko có bất kỳ lý thuyết nào mô tả khi cấp phát thì nó phải liên tiếp nhau cả giống biến bình thường thì khi debug thì khi cấp phát cho con trỏ đầu tiên này thì con trỏ tiếp theo ko nằm kế nó thì cũng là điều hoàn toàn bình thường.

thực tế là liên tiếp nhau nhưng ko có bất kỳ tài liệu nào bảo vậy cả nên sẽ ko khẳng định



Kiểu uint8_t hay uint16_t thể hiện kiểu data tại địa chỉ mà con trỏ trỏ tới, ví dụ con trỏ aptr trỏ tới địa chỉ FE00 thì kiểu data tại ô nhớ FE00 đến hết vùng size 4 byte sẽ là kiểu uint8_t.

Con trỏ uint16_t thì giá trị tại data, ô nhớ trỏ về con trỏ bptr sẽ là kiểu 16 bit cho nên giá trị 16 bit sẽ là 0x0001 và giá trị thứ 2 là 0x0002; kiểu 8 bit hay 16 bit đều chứa 2 ô nhớ để lưu địa chỉ của con trỏ, vì vậy ý nghĩa **của 8bit và 16 bit thể hiện data tại địa chỉ mà con trỏ trỏ tới.**

Câu hỏi:	**Trả lời**
 Giả sử có 1 con trỏ aptr, cấp phát động cho nó. Sau khi cấp phát động 4 ô thì nó sẽ trỏ đến địa chỉ FE00, sau đấy aptr ++ lên và free aptr, câu hỏi là có free được ko nếu free được thì free như thế nào.	Sau lệnh **aptr++** thì nó sẽ trỏ đến vùng nhớ **FE01.** Sau khi free thì vẫn free được và nó sẽ free cái vùng nhớ từ **FE01** đến **FE04**, có thể nó ko tác động đến giá trị nhưng nó vẫn free con trỏ →Câu trả lời trên sai, ko free được và free thì trả về giá trị lỗi.

Trả lời: Khi cấp phát như thế này thì ô nhớ địa chỉ **FE00** được cấp phát cho 1 con trỏ nào đó và với size bằng 4 byte.

Khi truyền free xuống **FE01** này, nó sẽ search các địa chỉ được cấp phát và nó sẽ bảo là FE01 chưa được cấp phát đi đâu cả dẫn đến hàm free bị sai đơn giản vì **FE00** được cấp phát còn Fe01 thì ko, dẫn đến hàm free ko free được và trả về giá trị lỗi.

Nó chỉ quan tâm là **cấp phát đến địa chỉ địa chỉ FE00 cho 1 con trỏ nào đấy chứ ko quan tâm đến biến nào cả, con trỏ nào cả, cấp phát ở đâu ko quan tâm**.

Quan tâm địa chỉ được cấp phát. Muốn free thì truyền vào địa chỉ **FE00** là được. ở đây truyền aptr vì đang lưu trữ giá trị của nó là **FE00**

Câu hỏi:

Cho con trỏ trỏ vào 1 biến cục bộ trong hàm, sau khi ra khỏi hàm đấy và lúc sau quay lại thì biến cục bộ vẫn dùng lại địa chỉ ấy, **cơ chế nào để nó sinh ra việc đó**?

Trả lời:

Khi bắt đầu build chương trình thì **con trỏ stack** sẽ được gán địa chỉ vào ô đầu tiên.

Sau khi kết thúc thì biến cục bộ đó được giải phóng, **con trỏ lùi lại**, sau đó gán giá trị cục bộ bằng 1 giá trị khác thì con trỏ stack trỏ vào ô kế tiếp, dẫn tới là lúc nào cũng thấy được địa chỉ đó (địa chỉ ko thay đổi).