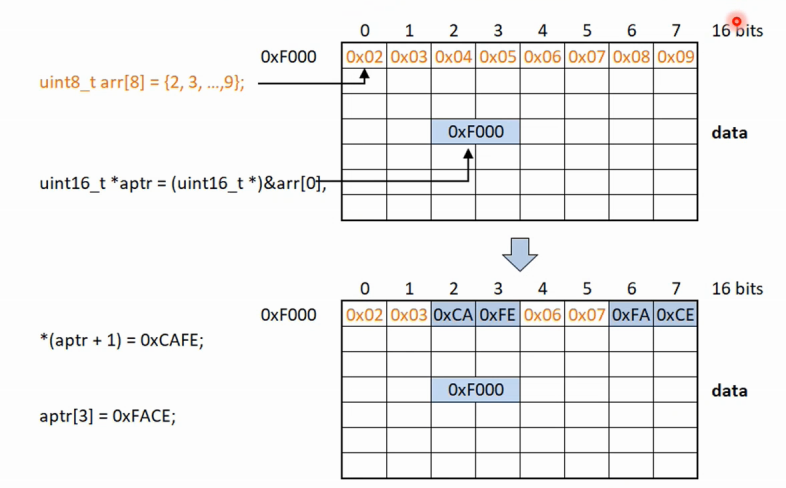
Pointer Advance

Việc gán 1 con trỏ tới 1 địa chỉ và chỉ ra 1 trường hợp hay gặp trong việc sử dụng sai con trỏ. Function pointer và việc ứng dụng vào trong việc xây dựng các call back function.

Việc gán 1 con trỏ cho 1 địa chỉ



Chip 16 bit và vùng data bắt đầu từ địa chỉ 0XF000

Trong ví dụ này là tạo 1 mảng array 8 phần tử có các giá trị khởi tạo từ 2 đén 9 và mảng array được lưu trữ từ ô nhớ 0xF000 đến ô nhớ 0XF007, sau đó gán 1 con trỏ 16bit uint16\_t\* cho địa chỉ của phần tử đầu tiên của mảng array này

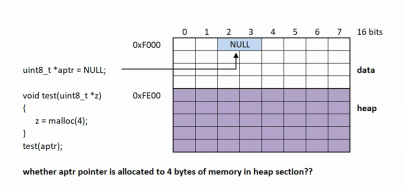
Biến con trỏ aptr là biến con trỏ thì với giả định ban đầu là đang thao tác với chip 16bit nên để lưu trữ biến con trỏ này sẽ cần tốn 2 nhớ như trên slide để lưu trữ cho biến con trỏ aptr này.

Sau phép gán thì giá trị của con trỏ aptr sẽ bằng giá trị 0xF000 và nội dung tại 2 ô nhớ lưu trữ cho biến aptr sẽ thay đổi giá trị bằng 0xF000.

Tiếp theo khi thao tác với con trỏ aptr cụ thể ở đây là kiểu uint16\_t thể hiện kiểu data tại địa chị con trỏ aptr trỏ tới sẽ có kiểu là uint16\_t

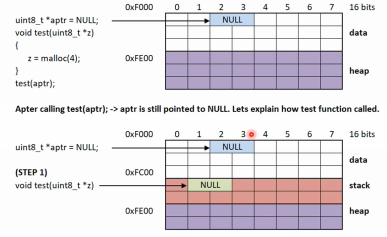
Chính vì vậy khi mà thao tác với việc aptr +1 thì hoặc aptr[3] ở đây nó sẽ offset với kích thước bằng kích thước của kiểu uint16\_t, cụ thể ở đây là 2. Vậy khi thao tác với aptr +1 tương đương với việc các bạn thao tác với ô nhớ 0XF002 và khi các bạn thao tác với ví trí 3 ở đây (aptr[3]) thì nó sẽ tương đương với ô nhớ 0xF006 và 0XF007 và giá trị tại các ô nhớ sẽ thay đổi bằng các giá trị mà mình thao tác, gán như thế này.

Tổng kết là thao tác gán giá trị cho 1 biến con trỏ thì không khác gì thao tác gán cho 1 biến thông thường chỉ có điều giá trị mà mình gán cho con trỏ thì nên hiểu nó là 1 địa chỉ. Thứ 2 là kiểu data của con trỏ, cụ thể là kiểu uint16\_t nó sẽ quy định kiểu dữ liệu tại địa chỉ mà con trỏ trỏ tới. chính vì vậy mà khi các bạn thao tác với ô địa chỉ mà con trỏ trỏ tới thì nó sẽ có offset là 16 bit tương ứng với 2 byte



Việc sử dụng sai 1 con trỏ thông qua 1 ví dụ. ở đây mình vẫn thao tác với con chip 16bit, vùng data bắt đầu từ địa chỉ 0xF000, ở đây mình thêm 1 vùng heap nữa để cấp phát động cho con trỏ, cụ thể ở đây là vùng heap bắt đầu từ địa chỉ 0XFE00.

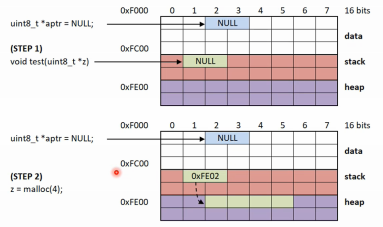
Ví dụ ở đây có 1 hàm Test với 1 tham số đầu vào là z có kiểu là uint8\_t\*, mục đích của hàm này là cấp phát động cho con trỏ thông qua việc gọi hàm test này và truyền aptr vào, câu hỏi đặt ra là sau khi gọi hàm test thì con trỏ aptr này có được cấp phát tới vùng nhớ có kích thước là 4 byte trong heap hay không?



Giả định thêm vùng nhớ stack bắt đầu từ địa chỉ 0xFC00 với những kiến thức mà chúng ta đã biết về con trỏ basis thì ở đây mình giả định 2 ô nhớ để lưu trữ biến con trỏ aptr là 0xF002 và F003. Size của con trỏ bằng size của thanh ghi con chip và giả định là chip 16 bit vậy là size của con trỏ 16 bit hay là size của bất kỳ con trỏ nào nó cũng có thể dùng 2 ô nhớ để lưu trữ cho biến của con trỏ đấy.

Phân tích các bước xét cho hàm Test gọi như thế nào, bước đầu tiên khi hàm test được gọi ở đây ta có 1 tham số z, chúng ta có thể hiểu tham số z là 1 biến tạm cần được lưu trữ ở đâu đó cụ thể ở đây là stack, do biến z là biến con trỏ nên chúng ta cần có 2 ô nhớ để lưu trữ cho biến z này.

Giá trị khởi tạo của biến z bằng NULL đơn giản là ta gọi hàm z nó truyền aptr vào và aptr đang bằng NULL tương đương với việc ta gọi hàm test để ta truyền NULL vào. Chính vì vậy, giá trị khởi tạo của biến con trỏ z bằng NULL

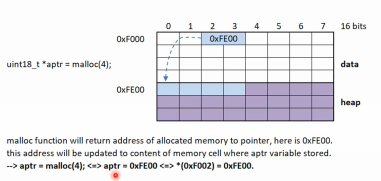


Bước tiếp theo sau khi khởi tạo biến con trỏ z rồi và chúng ta cấp phát động cho con trỏ z, cụ thể ở đây chương trình sẽ tìm trong vùng heap, vùng nhớ với kích thước bằng 4 bytes mà chưa được cấp phát cho bất kỳ con trỏ nào để cấp phát cho con trỏ z.

Trong ví dụ này mình tìm được vùng nhớ có 4 byte bắt đầu từ địa chỉ 0xFE02 và nó sẽ cấp phát cho con trỏ z. việc cấp phát cho con trỏ z sẽ gán giá trị z bằng 0xFE02 tương đương với việc nội dung tại ô nhớ lưu trữ biến z này sẽ được update thành giá trị 0xFE02.

Tiếp theo sau khi kết thúc gọi hàm thì biến z được giải phóng còn vùng nhớ được cấp phát cho con trỏ z vẫn được giữa nguyên ở đây và biến aptr vẫn có giá trị bằng NULL. Kết luận là sau khi gọi hàm z thì biến aptr ko thay đổi gì cả và ko cấp phát động được cho biến aptr và đây là 1 trường hợp hay gặp nhất trong việc sử dụng sai về con trỏ.

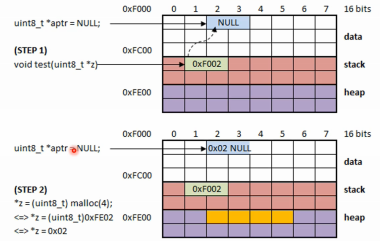
Nguyên nhân là chưa nắm rõ được bản chất của con trỏ nên việc vận dụng, cụ thể ở đây là việc cấp phát cho con trỏ đang bị sai.



Bản chất của con trỏ, ví dụ ở trên là việc cấp phát động cho con trỏ và sự thay đổi giá trị của con trỏ như thế nào. Giả định con trỏ aptr vẫn được lưu trữ ở địa chỉ 0XFE02 và 0xFE03. Sau khi cấp phát thì ta thấy giá trị 2 ô địa chỉ F002 và F003 sẽ được thay đổi bằng giá trị trả về của hàm cấp phát động, cụ thể ở đây là giá trị 0xFE00. Như vậy ta có thể hiểu là việc cấp phát động cho con trỏ aptr tương đương với việc thay đổi nội dung tại cái ô nhớ lưu trữ cho biến aptr, cụ thể ở đây là thay đổi nội dung tại cái ô nhớ 0XF002 và F003

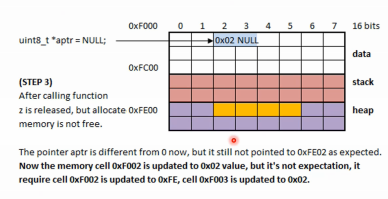
\*(0xF002) = 0xFE00. Vậy áp dụng nguyên tắc này vào bài toán đã làm sai lúc trước sẽ như thế nào. Việc mà chúng ta cần phải update nội dung F002 và F003 bằng giá trị FE00. Như vậy trong thân hàm của hàm Test này, chúng ta cần có phần code tương ứng với việc gán nội dung tại 0XFE02 bằng FE00, để làm được điều này chúng ta cần có thông tin của 0xF002. Thông tin này chúng ta lấy được bằng cách nào. Chúng ta sẽ lấy thông qua tham số z bằng cách chúng ta truyền cái giá trị 0xF002 cũng chính là địa chỉ của cái biến aptr thông qua hàm test thông qua việc gọi hàm test như thế này cụ thể ở đây là gọi hàm test và truyền địa chỉ test(&aptr) của aptr vào. Tiếp theo sau khi truyền địa chỉ của cái aptr vào thì chúng ta sẽ có cái giá trị của z bằng F002, theo nguyên tắc thì ta lấy \* của z bằng cấp phát động cho 4 byte. Tuy nhiên nhìn vào phần code ở đây ta thấy sẽ có 1 cái gì đó sai sai.

Chúng ta sẽ phân tích các bước mà hàm test được gọi để xem kết quả như thế nào

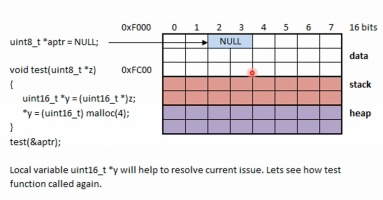


Bước 1 ta sẽ có z được cấp phát, được khởi tạo trong cái vùng stack kia với giá trị khởi tạo là F002, vì sao là F002 thì khi gọi hàm test chúng ta truyền địa chỉ của aptr vào tức là chúng ta truyền địa chỉ F002 vào chính vì vậy giá trị khởi tạo của biến ban đầu là z = F002. ở đây ta có thể hiểu là con trỏ z đang trỏ tới ô địa chỉ F002. Tiếp theo ta lấy \*z = malloc của 4. Chúng ta có con trỏ z ở đây là con trỏ 8 bit

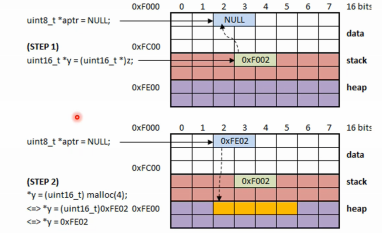
Uint8\_t có nghĩa là kiểu của data tại địa chỉ trỏ tới bởi con trỏ z sẽ có kiểu là uint8\_t. ở đây chúng ta đang thao tác là \*z = tức là thao tác với data tại địa chỉ trỏ tới bởi con trỏ z và data ở đây đang có kiểu là uint8\_t vậy nên cần ép kiểu uint8\_t cho hàm malloc này. ở đây giả định là hàm malloc đang trả về địa chỉ FE02. Việc ép kiểu FE02 về kiểu uint8\_t này thì sao đơn giản là **thông tin của byte cao sẽ bị mất đi** và thông tin của byte thấp sẽ được giữ lại và gán cho \*z ở đây. Vậy sao khi gọi \*z = ép kiểu uint8\_t, malloc 4 ở đây thì ô nhớ 0xF002 sẽ được cập nhật giá trị thành 0x02. Sau khi kết thúc lượt gọi hàm thì vẫn tương tự



Z được giải phóng, tuy nhiên kết quả đạt được ko phải là điều mà chúng ta mong muốn. cụ thể ở đây chỉ có ô nhớ 0xF002 được cập nhật giá trị thành 0x02 trong khi chúng ta mong muốn là ô nhớ 0xF002 được cập nhật thành 0xFE vào ô nhớ, 0xF003 được cập nhật thành 0x02. Vậy chúng ta cần phải làm gì?

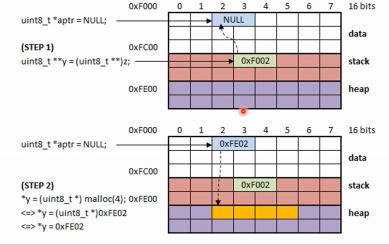


Giải pháp: chúng ta muốn update 2 ô nhớ, vậy đơn giản chúng ta sử dụng con trỏ 16 bit cụ thể ở đây mình tạo thêm 1 biến con trỏ y 16 bit và biến con trỏ y được gán cho cái giá trị của z. chúng ta sẽ phân tích từng bước hàm test được gọi thêm 1 lần nữa để xem issue được giải quyết như thế nào



Bước 1: ko nhắc lại biến z nữa, biến z có giá trị bằng 0xFE02, cụ thể ở đây nói đến biến y. biến y ở trong hàm, vậy biến y được lưu trữ trong stack. Y là 1 biến con trỏ vậy sẽ cần phải 2 ô nhớ để lưu trữ cho biến y này và giá trị khởi tạo của y bằng z và z bằng bn, z bằng 0xF002. Chính vì vậy giá trị khởi tạo cho biến y cũng bằng 0xF002, tức là chúng ta có thể hiểu là biến con trỏ 16bit y đang trỏ tới ô địa chỉ 0xF002. Tiếp theo chúng ta gán giá trị cho biến y và ép kiểu (uint16\_t)malloc(4) vì đơn giản hàm malloc(4) này sẽ trả về địa chỉ FE02 và chúng ta có y là con trỏ 16 bit, data của địa chỉ trỏ bởi con trỏ y sẽ có kiểu là 16 bit. Chính vì vậy \*y ở đây chính là data sẽ được cập nhật bằng giá trị FE02 và nhìn trên hình này sẽ có các giá trị của ô nhớ 0xF002 tức là địa chỉ trỏ bởi con trỏ y sẽ được cập nhật thành 0xFE và 0xF003 sẽ được cập nhật thành 0x02 nên ta có 2 ô nhớ này sẽ được cập nhật giá trị thành FE02.

Và tương tự, sau khi kết thúc 1 lời gọi hàm thì biến y được lưu trong stack sẽ được giải phóng và chúng ta có 2 ô nhớ F002 và 003 đã được cập nhật với giá trị FE02 tức là con trỏ aptr được cấp phát động và trỏ tới vùng nhớ bắt đầu từ địa chỉ 0xFE02. Tuy nhiên với giải pháp này nếu như ứng dụng cho chip 32 bit thì chúng ta gặp phải vấn đề ép kiểu và bị mất thông tin như ở trước, mất giá trị của 2 byte cao. Vậy giá trị của toàn diện cho bài toán này là như thế nào

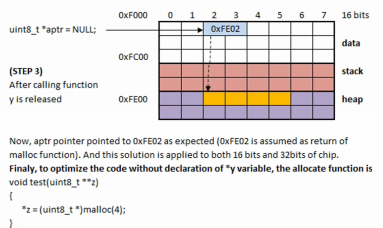


Thay vì sử dụng con trỏ 1\* 16 bit thì ta sử dụng con trỏ 2\* 8 bit, cụ thể ở đây thay vì tạo 1 biến trung gian là biến uint16\_t\*y thì mình tạo ra 1 biến uint8\_t\*\*y và y cũng được gán giá trị bằng z; cho dù con trỏ có là 1\* 2\* hay 3\* đi chăng nữa thì nó cũng là 1 con trỏ. Chương trình vẫn cần 2 ô nhớ với chip 16 bit để lưu trữ cho con trỏ y này và y này là 1 biến cục bộ dẫn đến y sẽ được lưu trong stack, giá trị khởi tạo của y là F002

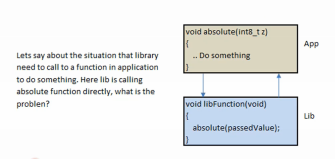
Tiép theo ta lấy \*y là nội dung tại địa chỉ mà con trỏ\*\* y trỏ tới nó sẽ có kiểu là con trỏ 1\* chính vì vậy khi lấy \*y ở đây tức là mình đang thao tác với nội dung tại địa chỉ mà con trỏ y\*\* trỏ tới, cụ thể địa chỉ ấy là địa chỉ 0xF002 và nội dung tại địa chỉ F002 này sẽ có kiểu 1\* uint8\_t\*.

Chính vì vậy khi mà gán \*y = (uint8\_t\*)malloc(4) thì có giá trị trả về của hàm malloc là FE02 và nó sẽ được update vào ô nhớ trỏ bởi con trỏ \*\*y ở đây.

Cụ thể ta có 2 ô nhớ F002 và F003 được cập nhật thành giá trị 0xFE02. Như vậy với việc sử dụng con trỏ \*\* ở đây, chúng ta đã đưa ra 1 giải pháp toàn diện

để giải quyết vấn đề gặp phải trong bài toán cấp phát cho con trỏ aptr và để ko có 1 bước tạo ra 1 biến tạm y thì thay vì việc sử dụng mẫu nguyên hàm với kiểu uint8\_t\* z thì sẽ gặp mẫu nguyên hàm là sử dụng kiểu uint8\_t\*\* z và bên trong thân hàm đấy chúng ta chỉ cần gán \*z = (uint8\_t\*)malloc(4); là được

Đây là mẫu nguyên hàm hay gặp cho hàm cấp phát động cho 1 con trỏ. Như vậy từ việc phân tích các cái lỗi gặp phải chúng ta hiểu được tại sao tham số z trong hàm cấp phát động phải là kiểu con trỏ \*\*. Chúng ta hiểu được bản chất

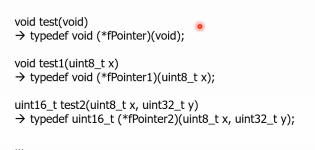


Con trỏ hàm

Bài toán: có 1 cái thư viện nào đó cần phải call đến hàm ứng dụng thực hiện 1 chức năng nào đó tuy nhiên nếu việc call trực tiếp hàm ứng dụng thông qua tên hàm này thì vấn đề gì sẽ xảy ra

Vấn đề là các bạn viết thư viện và ứng dụng sẽ được viết bởi 1 bên khác cho nên khi các bạn viết thư viện, các bạn sẽ không có phần hàm ứng dụng kia. Và ko có phần hàm ứng dụng kia, các bạn sẽ không thể build thư viện 1 cách độc lập được và lỗi xảy ra khi các bạn build thư viện mà ko có cái hàm absolute kia được định nghĩa ở đâu.

Giải pháp giả quyết bài toán này là sử dụng con trỏ hàm. Chúng ta tìm hiểu con trỏ hàm là gì. Đơn giản thì con trỏ hàm là 1 kiểu dữ liệu mới được định nghĩa bởi người sử dụng hoặc người lập trình. Một biến con trỏ hàm thì sẽ được lưu trữ địa chỉ của một hàm nào đó và với con trỏ hàm thì sẽ ko thể cấp phát động được dù nó là 1 con trỏ. Chỉ có thể gán biến của 1 con trỏ hàm cho 1 cái địa chỉ của 1 hàm nào đó thôi. Việc định nghĩa 1 con trỏ hàm như thế nào



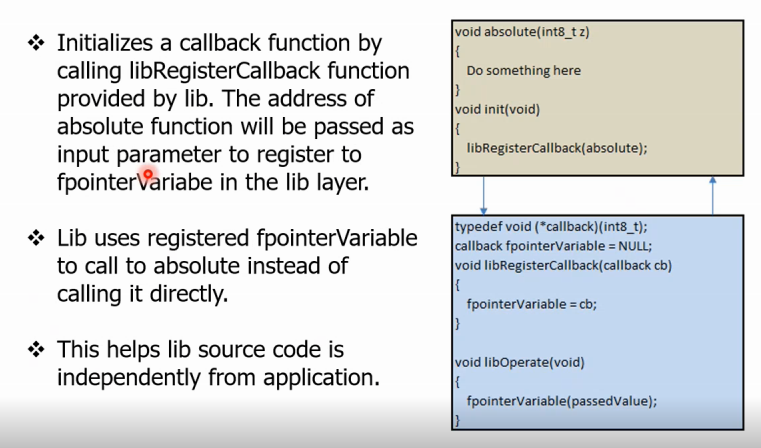
Ví dụ muốn con trỏ hàm trỏ tới hàm của mỗi nguyên hàm như thế nào thì kiểu con trỏ hàm sẽ được định nghĩa tương tự như thế

Trên là 3 ví dụ về 3 mẫu nguyên hàm với ví dụ đầu tiên là mẫu nguyên hàm ko có giá trị trả về và ko có tham số

Với mẫu nguyên hàm thứ hai là hàm ko có giá trị trả về và có 1 tham số

Thứ 3 là có 2 tham số và có giá trị trả về.

Đơn giản để định nghĩa 1 kiểu dữ liệu mới, đơn giản là bạn dùng typedef thôi, với con trỏ hàm thì các bạn định nghĩa tương tự với 1 mẫu nguyên hàm mà con trỏ hàm đó trỏ tới, cụ thể ở đây là thay đổi tên hàm bằng tên kiểu con trỏ hàm là xong. Tất cả các thông tin về giá trị trả về hay tham số giữ nguyên. Với cách định nghĩa như vậy ta có 3 kiểu con trỏ hàm và cái biến khai báo với cái kiểu con trỏ hàm này sẽ được trỏ tới 3 mẫu nguyên hàm khác nhau



Sau khi hiểu về con trỏ hàm rồi thì mình biết tới ứng dụng thông dụng nhất của con trỏ hàm trong việc xây dựng các call back function. ở đây mình đưa ra việc call back function sẽ giải quyết được bài toán lúc trước.

Cụ thể hàm thư viện sẽ cung cấp 1 cái hàm đăng ký call back và ứng dụng trước khi chạy sẽ phải gọi cái hàm đăng ký call back này để đăng ký cái hàm call back mà thư viện muốn gọi. đơn giản, nội dung của hàm call back sẽ gán cái con trỏ hàm bằng địa chị hàm được đăng ký bởi ứng dụng và sau đó thì trong hoạt động của thư viện sẽ gọi đến cái hàm của ứng dụng thông qua tên con trỏ hàm và ko thông qua tên hàm nữa. việc gọi thông qua con trỏ hàm như thế này đảm bảo tính độc lập của thư viện với ứng dụng và chúng ta có thể build thư viện này ko cần có code của ứng dụng, build 1 cách độc lập.