**Bài 2: Từ khóa Volatile**

Optimization : +Compiler có nhiều option để biên dịch chương trình, mỗi option phù hợp với 1 nhu cầu của code (nhanh, dung lượng thấp,..),

+Các level của optimization giúp tối ưu hóa code, giảm dung lượng của code (code size)

+VD: các câu lệnh chung nhiệm vụ sẽ chỉ được thực thi 1 lần

Hoặc các biến không bị thay đổi trong code sẽ không được kiểm tra lại…

Từ khóa Volatile: có chức năng để compiler không optimize ( tối ưu) biến được gán

+ sử dụng ở các biến trong ngắt, hoặc các biến bị thay đổi đột ngột không do phần mềm

Vd: volatile int u32Check;

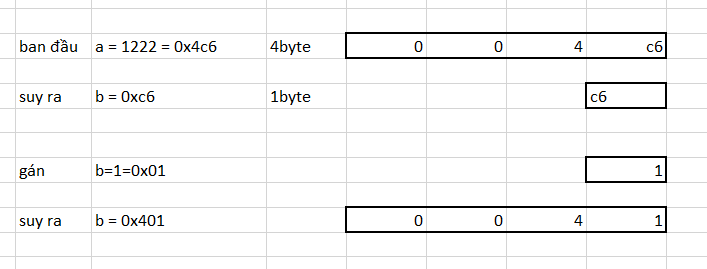
* Tối ưu hóa code, giảm code size nhưng chương trình vẫn chạy ngon

**Bài 3: Từ khóa Union**

Là 1 kiểu dữ liệu có thể tập hợp các loại dữ liệu khác 1 cách dễ dàng, gần giống như struct

Size của union sẽ lấy theo size của biến lớn nhất trong nó

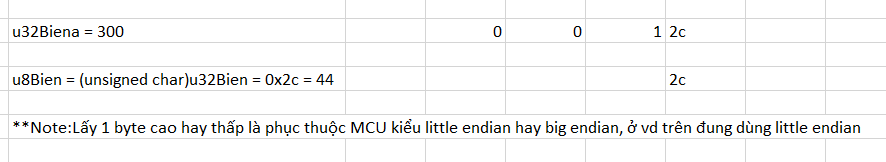
Và các biến còn lại sẽ share chung byte với biến có size lớn nhất đó



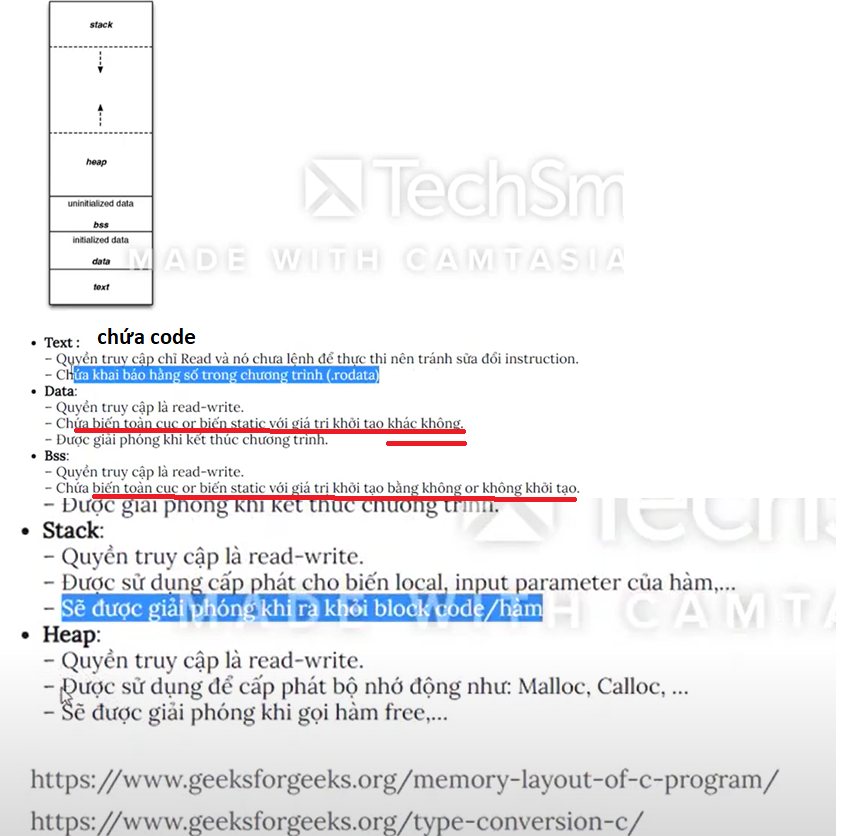
**Bài 4: Quản lý vùng nhớ**

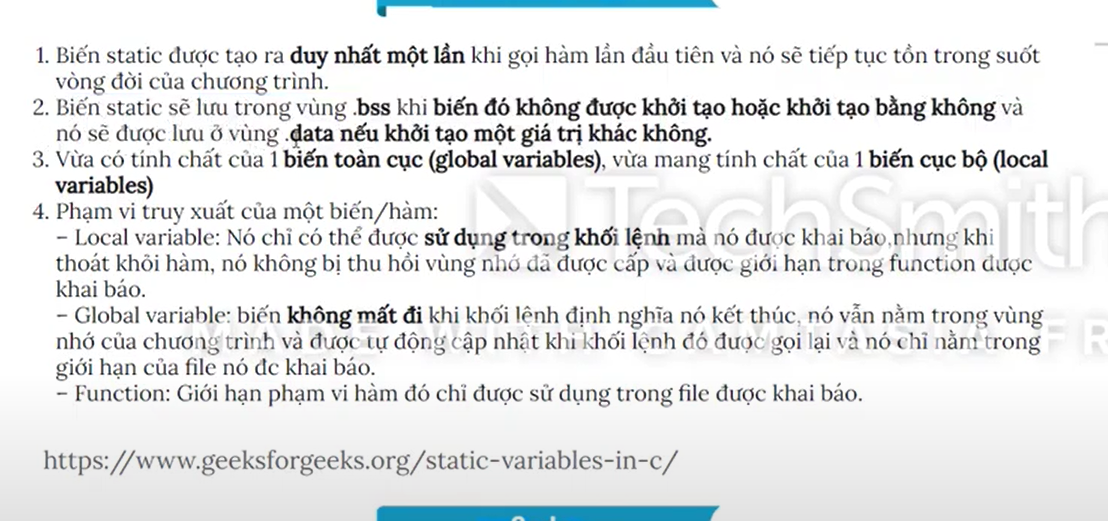
**4.1.Ép kiểu**

Từ kiểu dữ liệu cao về kiểu dữ liệu thấp thì mất dữ liệu

****

**4.2.Phân vùng**

****

**Bài 5: Biến static **

Xem code để hiểu rõ hơn

    /\*pham vi biến này chỉ dược sử dụng trong function này\*/

    /\*ra khỏi function sẽ không sử dụng được biến này, nhưng cũng ko bị mất đi \*/

    /\*giá trị KHỞI TẠO cho biến static phải là hằng số\*/

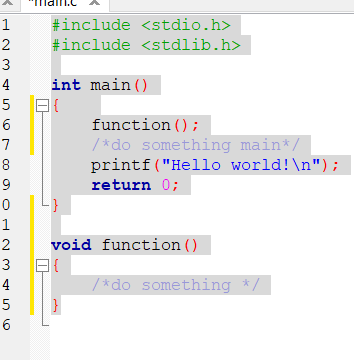
    /\*có thể gọi cả function để thay đổi giá trị\*/

    static unsigned int u32Static = 1;

    //32Static = 1; không đúng

**Bài 6: Từ khóa inline**

**1.khi không có inline**

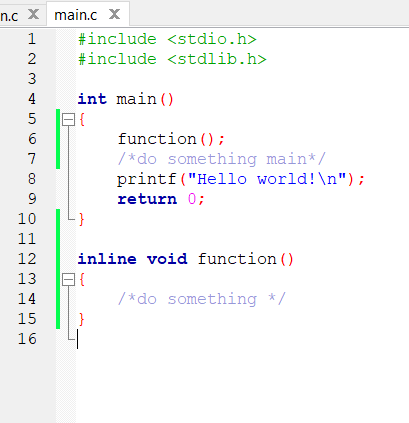
**** - khi tới câu lệnh gọi hàm function trong hàm main:

+ địa chỉ câu lệnh tiếp theo sẽ được lưu lại trong stack hoặc thanh ghi nào đó

+thanh ghi R14(Link register) lưu thông tin trả về các lời gọi hàm và ngoại lệ

+rồi sẽ nhảy vào thực hiện hàm function

2.khi có inline

-Khi tới câu lệnh gọi hàm function trong hàm main:

+copy các câu lệnh của hàm function vào hàm main để thực hiện luôn

+không cần lưu địa chỉ câu lệnh tiếp theo

Ưu điểm: nhanh hơn so với không dùng

* Nhưng nếu gọi nhiều lần hàm function đó thì code size sẽ tăng lên

**Bài 8.1** **Bài tập dịch bit**

Dịch trái: a = b<<n = b\*(2^n)

Dịch phải:a= b>>n = b/(2^n)

Vd1. Thanh ghi a 10 bit, thanh ghi b chứa 8 bit cao của biến a:

b = a>>2;

hoăc b =a/4;

\*\* lấy n bit thấp của thanh ghi a cho vào thanh ghi x

x = a % (2^n)

vd2: chuyển 2 bit thấp của a vào 2 bit 4,5 của thanh ghi c

/\*lấy 2 biến thấp của thanh ghi a gán vào x \*/

x = a & 0x03

hoặc x = a%4

/\*chuyển x vào 2 bit [5:4] của c \*/

c = x<<4 = (a%4) << 4;

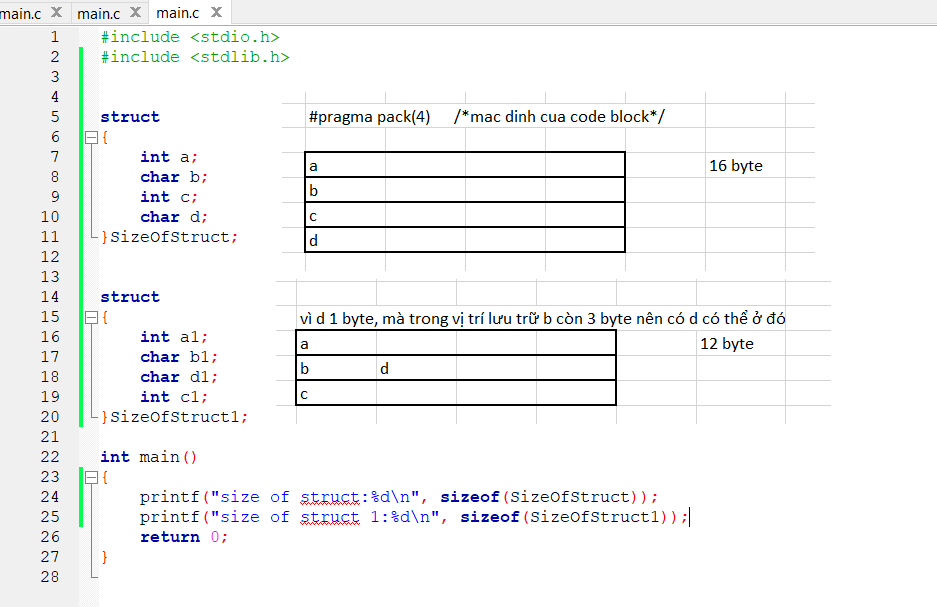
**Bài 9: Struct**

**1.**

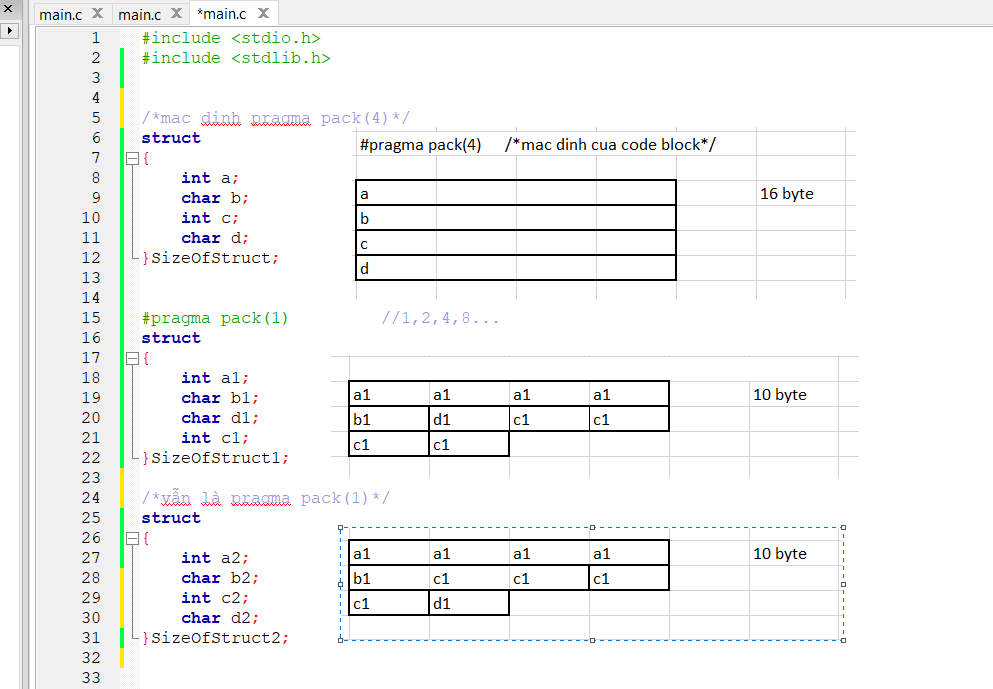
**2. size của struct**

Tính size theo thứ tự khia báo trong struct

Với mặc định của keilC, size của struct sẽ tính theo size của biến lớn nhất trong đó (tạm thấy sizeofstruct chia hết cho size của biến lớn nhất trong struct đó)

****

Thay đổi với #pragma pack() // 1,2,4,8…



**Bài 10: Marcro**

**1.**Function marcro vs function bình thường

Marcro (#define)

* được thay trực tiếp vào hàm main trong quá trình processing
* Code nhanh hơn, chương trình chạy nhanh hơn nhưng Size Code lớn

Function bình thường:

* khi được gọi, chương trình sẽ boost vào stack(lưu thông tin để biết khi trở lại, tương tự như interrupt) sau đó nhảy vào function đó, rồi nhảy ra
* Chương trình chạy chậm hơn nhưng size code nhỏ hơn

**2.if vs #if**

#if dùng để khi trên 1 project nhưng muốn chạy được trên nhiều con chip (xem video)

Vd:

/\*define f103 thì đoạn code cho f103 sẽ được thực chạy, đoạn còn lại sẽ bỏ qua\*/

#define STM32F103

#if STM32

{

Code cho STM32f103

} #else

{

Code cho Stm32f407

} #endif;

**Bài 11: Từ khóa ENUM**

Trong nhúng, có thể dùng để liệt kê các lỗi, khi chương trình bị lỗi thì sẽ return về các lỗi này (bằng chữ)

**Enumerate** : Liệt kê

Kiểu liệt kê (enum) là một trong số các kiểu dữ liệu do người lập trình tự định nghĩa

 Liệt kê những danh từ để thay thế cho những con số, giúp người đọc chương trình dễ hiểu hơn (chứ không giúp chương trình chạy nhanh hơn).

Cách khai báo: giống struct

**enum** **DaysOfWeek**

{

SUNDAY,

MONDAY,

TUESDAY,

WEDNESDAY,

THURSDAY,

FRIDAY,

SATURDAY

};

không cần chúng ta trực tiếp gán giá trị cho các tên hằng số, **compiler** đã tự động khởi tạo giá trị cho chúng, bắt đầu với giá trị 0 và tăng dần

Bên cạnh việc tự động gán giá trị cho từng phần tử được liệt kê, chúng ta cũng có thể chủ động thay đổi giá trị cho chúng (**nhưng chỉ có thể thay đổi giá trị trong phần khai báo**), một enum sau khi đã định nghĩa xong thì không thể thay đổi những giá trị của danh sách các phần tử nữa

**Bài 14: Từ khóa EXTERN**

Từ khóa EXTERN được dùng khi bạn có một biến toàn cục được sử dụng chung ở các file khác nhau.

Giả dụ project của bạn có 2 file main.c và dem.c, bạn muốn 1 biến u32checking được sử dụng ở cả 2 file. Nếu khai báo int u32checking ở cả 2 file thì sẽ bị lỗi hoặc dẫn đến kết quả không chính xác.

Giải pháp ở đây là bạn khai báo biến toàn cục int u32checking ở file main.c, còn ở các file khác nếu có sử dụng thì bạn khai báo extern int u32checking, khi đó chương trình sẽ hiểu và tìm biến toàn cục u32checking đã được khai báo ở file main.c để sử dụng

**Bài 15: Từ khóa \_\_weak**

Cung cấp 1 function trống để người dùng dev

Người dùng có thể gọi function đó ra để tự dev theo ý họ hoặc phần cứng

**Bài 16: Từ khóa Constant**

1.Khai báo biến là constant thì không thể thay đổi giá trị được

Vd: const int u32constant = 10;

Nhưng có thể thay đổi được giá trị đó nếu dùng pointer

Tạo 1 pointer trỏ tới địa chỉ biến u32constant, thay đổi giá trị \*pointer thì cũng thay đổi giá trị của u32constant.

2.Pointer to Constant

không thể thay doi giá trị được lưu ở địa chỉ(giá trị của biến) mà pointer trỏ tới

có thể thay đổi địa địa chỉ mà pointer trỏ tới

3.Constant\_Pointer

có thể thay đổi giá trị ở địa chỉ mà pointer trỏ tới

nhưng không thể thay đổi địa chỉ mà ponter trỏ tới

**Bài 17: Cấp phát bộ nhớ**

sau khi cấp phát bộ nhớ thì con trỏ tựa như 1 mảng, ưu điểm hơn mảng khi không bị thừa hoặc thiếu bộ nhớ  
vd. Khi dùng mảng a[100] sẽ dễ gây ra bị thừa

int\* ptr;

    /\*cấp phát 5 ô nhớ cho con trỏ ptr\*/

    ptr = (int\*)malloc(5 \* sizeof(int));

2 cách: malloc vs calloc

/\*

    so sánh                 calloc                  malloc

    giá trị khởi tạo        Có, all = 0             không, giá trị cũ, ngẫu nhiên

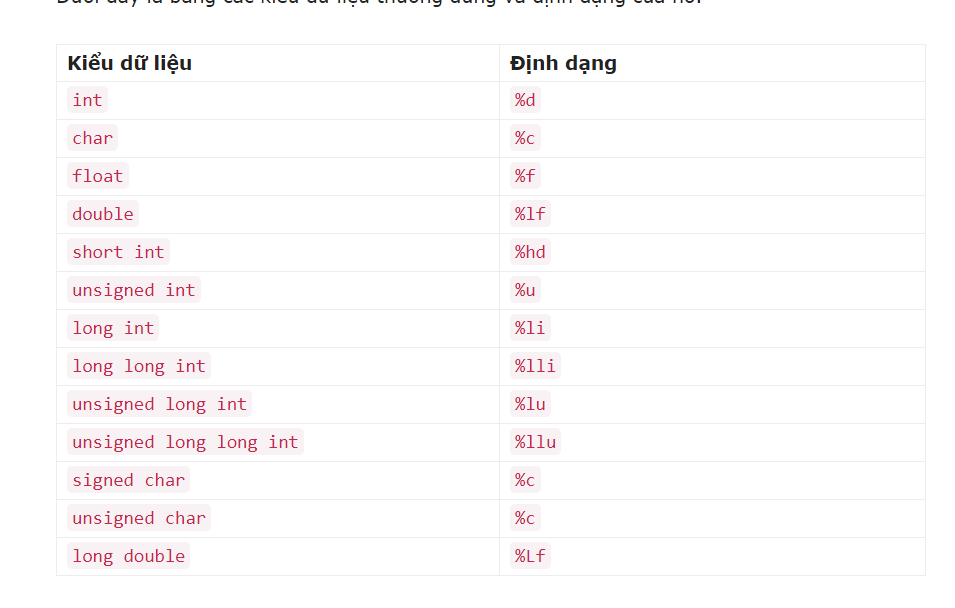
    số lượng tham số        2 (số lượng và size)    1 (số lượng NHÂN size)          1

\*/

Giải phóng bộ nhớ: free(ptr);

**Bài 22**

Không printf %d cho biến uint32\_t được



**Bài 24: Typedef**

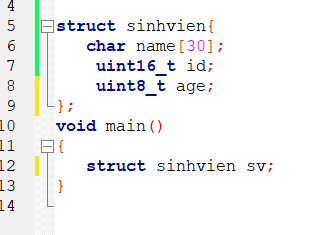
Công dụng tương tự như define

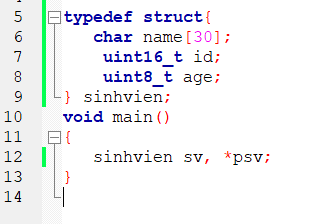
Gán **1 kiểu dữ liệu** thành 1 cách gọi dễ nhớ hơn

Lợi ích tương tự như việc sử dụng define

**Typedef với struct:**

Khi khai báo struct mới, thay vì dùng



Thì có thể dùng

**Bài 25: Khác biệt include**

#include < filename > VS #include “ filename ”

VD: #include<stdio.h> vs #include “stm32f1xx.h”

* 2 lệnh trên là tương đương nhau, và dùng để thêm các file header trong lập trình C
* Sự khác biệt giữa 2 cú pháp trên nằm ở quá trình tìm kiếm file header của tiền xử lý trước khi biên dịch

#include <stdio.h> : tiền xử lý sẽ chỉ tiềm kiếm file header (.h) trong thư mục chứa file header của ngôn ngữ C( thường là bộ cài IDE)

#inlude “stm32f1xx”: tiền xử lý tìm kiếm file header (.h) trong thư mục cài đặt project C/C++ trước.

Nếu không thấy, tiền xử lý sẽ tìm kiếm file header (.h) trong thư mục chứa file header của ngôn ngữ C( thường là bộ cài IDE)