**LẦN 1**

Ví dụ trong chương trình main.c rất đơn giản

#include <stdio.h>

#include “sum.h”

Int main(void)

{

Int x =0;

X = sum (3,6);

Printf(“x=%d\n”,x);

Return 1;

}

Trong file sum có những gì nữa

Sum.h

#ifndef SUM\_H\_

#define SUM\_H\_

Int sum(int a, int b);

#endif/\*SUM\_H\_\*/

Sum.c

#include”stdio.h”

#include”sum.h”

Int sum(int a, int b)

{

Return a+b;

}

**Lần 2, Thay đổi 1:**

Trong file sum.h

Sum.h

#ifndef SUM\_H\_

#define SUM\_H\_

Int sum\_t = 0;

Int sum(int a, int b);

#endif/\*SUM\_H\_\*/

Trong sum.c

#include”stdio.h”

#include”sum.h”

Int sum(int a, int b)

{

Sum\_t = a+b;

Return sum\_t;

}

Trong hàm main:

#include <stdio.h>

#include “sum.h”

Int sum\_t;

Int main(void)

{

Int x =0;

X = sum (3,6);

Printf(“x =%d\n”,x);

Printf(“sum =%d\n”, sum\_t);

Return 1;

}

* Lỗi sum\_t trong sum\_t = a+b; chưa được khai báo (did you mean sum)

**Lần 3:**

Sửa 2 file sum.c và sum.h như sau:

Sum.h

#ifndef SUM\_H\_

#define SUM\_H\_

Int sum(int a, int b);

#endif/\*SUM\_H\_\*/

Trong sum.c

#include”stdio.h”

#include”sum.h”

Int sum\_t = 0;

Int sum(int a, int b)

{

Sum\_t = a+b;

Return sum\_t;

}

Kết quả x = 9; sum = 9

Vấn đề là trong file main.c printf đúng sum\_t bằng 9 mà mình ko biết vì sao

**Lần 4**

Thử bỏ dòng int sum\_t; trong file main.c là nó báo lỗi ko biết sum\_t là cái gì

Printf(“sum =%d\n”, ***sum\_t***); bởi vì vốn thằng sum\_t trong file main.c chưa được khai báo

Vấn đề là chưa extern mà nó đã nhận thằng sum\_t này nên là ta sẽ thử printf địa chỉ

Trong hàm main:

#include <stdio.h>

#include “sum.h”

Int sum\_t;

Int main(void)

{

Int x =0;

X = sum (3,6);

Printf(“x =%d\n”,x);

Printf(“sum\_t address: %d\n”, &sum\_t);

Printf(“sum =%d\n”, sum\_t);

Return 1;

}

Và trong hàm sum.c cũng in luôn địa chỉ của nó

#include”stdio.h”

#include”sum.h”

Int sum\_t = 0;

Int sum(int a, int b)

{

Sum\_t = a+b;

Printf(“sum\_t address in sum.c: %d\n”, &sum\_t);

Return sum\_t;

}

Kết quả: địa chỉ của nó giống hệt nhau, có nghĩa là chúng ta khai báo địa chỉ giống hệt nhau như vậy 2 biến sum\_t này là 1

Lần 5

Vậy thử từ khoá extern xem như thế nào

Đầu tiên cắt thằng Int sum\_t; này đi (main cũng ko khai báo sum\_t nữa)

#include <stdio.h>

#include “sum.h”

Int main(void)

{

Int x =0;

X = sum (3,6);

Printf(“x =%d\n”,x);

Printf(“sum\_t address: %d\n”, &sum\_t);

Printf(“sum =%d\n”, sum\_t);

Return 1;

}

Và đặt vào sum.h và thêm 1 từ khoá extern

Sum.h

#ifndef SUM\_H\_

#define SUM\_H\_

**Extern** int sum\_t;

Int sum(int a, int b);

#endif/\*SUM\_H\_\*/

Và ko khai báo sum\_t trong sum.c nữa

#include”stdio.h”

#include”sum.h”

Int sum(int a, int b)

{

Sum\_t = a+b;

Printf(“sum\_t address in sum.c: %d\n”, &sum\_t);

Return sum\_t;

}

Main.c báo lỗi undefined reference to sum\_t

Lần 6

Khai báo int sum\_t =0; ở trong sum.c

Được luôn, tức là đầu tiên ta khai báo 1 lần, sau đó extern trong file header thì trên file main.c ko cần khai báo lại nữa.

Kết quả ra là:

Sum\_t address sum.c: -87527014

X = 9

Sum\_t address: -87527014

Sum = 9

Tức là ở đây

Khi khai báo int sum\_t ở file sum.c và qua file sum.h extern cái int sum\_t đó ra thì nó sẽ được sử dụng trong file main.c dù ko cần khai báo lại, tầm vực của biến sum\_t được nhận biết trong cả file main.c

Extern function thì tự tìm hiểu thêm

Từ khoá register, nói về resigter thì phải nói 1 chút về core.

CPU có 3 thành phần cơ bản: CU (control unit: đơn vị xử lý trung tâm), ALU (đơn vị tính toán số học và logic), các thanh ghi.

ALU tính toán và trả về kết quả trực tiếp trên các thanh ghi, ví dụ có thể xử lý: lấy giá trị thanh ghi 1 cộng với giá trị thanh ghi 2 rồi trả về giá trị lưu trữ trên kết quả thanh ghi 3 chẳng hạn.

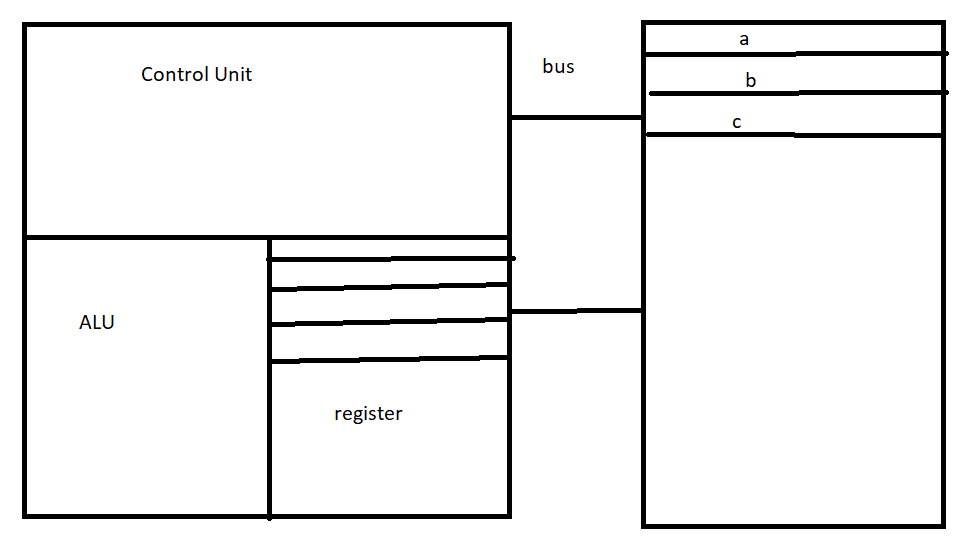
Tức là ví dụ

Biến a cộng biến b

Ko phải là lấy trực tiếp a + b mà thằng ALU tính toán trên các thanh ghi chứ ko thực hiện cộng các giá trị trên memory này được

Mà nó sẽ thực hiện lệnh load xong sau đó store lại, đó gọi là cơ chế load and store của CPU thế nên việc truy xuất, di chuyển dữ liệu mất thời gian hơn.

Để trả về kết quả thật nhanh thì ta khai báo 1 biến được allocate trên thanh ghi để thằng ALU và CU đọc trực tiếp từ các thanh ghi này ra luôn



RAM

Memmory

Thay vì việc được cấp phát trên RAM, trên bộ nhớ

Từ khoá register ít khi được sử dụng

Từ khoá volatile

Nếu chương trình chạy đều đều lần lượt từ trên xuống dưới thì ko nói làm gì cả, có nghĩa là ví dụ ta có 1 biến và việc thay đổi biến đó, ví dụ:

A =5;

/\*Ở dưới này có rất nhiều lệnh nhưng ko tác động đến biến A này\*/

A = 10;

Printf(“A”); /\*Theo trình dịch optimization thì A =5 ko được biên dịch ra nữa và in ra ngay A = 10

Giá trị mà nó truy xuất trước A =5 hay A = 10 nó ko quan tâm \*/

Giờ thử thay

If (A ==5);

{

/\*Thực hiện đoạn code trong này\*/

}

Thường thường thì thằng optimization ở đây khi xuống if A ==5 thì mặc định so sánh A ==5 thì câu lệnh này đúng NHƯNG có 1 vấn đề xảy ra.

Do optimization ko cần lấy lại, có nghĩa là đoạn code trong if ko có câu lệnh nào load lại giá trị A nữa, tức là

1. A =5;





8. If (A==5)
9. {
11. }

Thì optimization chỉ quan tâm dòng số 1 A đã bằng 5 thì đến dòng số 8 thì nó mặc định lấy giá trị 5 so sánh với 5, nó ko cần update lại giá trị của A.

Nhưng giả sử do A ở đây là biến toàn cục thì chương trình của mình viết có thể nhiều luồng chẳng hạn (3 luồng – nhiều chương trình con chạy song song với nhau) thì trên 1 chương trình con A, B nào đấy thì tại dòng số 5 thay đổi A này đi hoặc trên 1 luồng khác, 1 function khác nó biến thành A =6 nhưng mà như vừa nói

Đến dòng số 8 nó ko load lại giá trị của A mà nó so sánh luôn nên thành ra đoạn if đó xảy ra sai sót, tức là A lúc này giá trị = 6 rồi nhưng lúc nhảy vào hàm if này A vẫn bằng 5.

Nếu A = 5 mất rồi thì nó sai và ko xử lý được

Cho nên để tránh trường hợp này xảy ra, ta check lại địa chỉ của A xem giá trị thực tế bằng bao nhiêu.

Thì để check ta khai báo từ khoá volatile

Volatile int A =5;

Tức là khi sử dụng biến A thì nó sẽ check lại giá trị A 1 lần.

Từ khoá volatile sẽ được áp dụng cho các biến được chia sẻ đa luồng hoặc là những biến toàn cục được thay đổi trong ngắt thì chúng ta sẽ sử dụng thêm từ khoá volatile này

Việc lập trình đa luồng tác động đến biến toàn cục rất nhiều nên sẽ rất cần các từ khoá này.

Các từ khoá này có thể thay đổi tính chất của 1 biến, thay đổi truy xuất và optional, optimize complier cho từng biến 1.

Khi cảm thấy rằng có 1 từ khoá có thể sử dụng nhiều hoặc phức tạp mà ta cần bảo vệ biến đó hoặc ta cần extern hay nhiều hàm sử dụng hơn thì chúng ta sẽ sử dụng các từ khoá này.

Ví dụ từ khoá static, biến cục bộ trong 1 hàm nào đó mà nhận biết được biến đó nhưng mà lại muốn các giá trị của nó được gim lại sau các lần gọi nó thì nghĩ ngay đến từ khoá static.

Cái thứ 2 là chúng ta muốn sử dụng chung 1 biến global nhưng qua các file code khác nhau thì chúng ta sử dụng từ khoá extern

Muốn truy xuất nhanh hơn thì sử dụng từ khoá register

Muốn bảo vệ 1 biến trước việc mỗi lần sử dụng biến đó thì luôn luôn check lại giá trị của nó thì sử dụng từ khoá volatile

Structure data type (kiểu dữ liệu có cấu trúc), có 3 cấu trúc:

Structure variable (biến cấu trúc)

Thứ 2 là union thứ 3 là enum

1 kiểu dữ liệu có cấu trúc là gì, bao gồm các thành phần dữ liệu và những thành phần dữ liệu này có thể cùng hoặc ko cùng kiểu dữ liệu với nhau và được group lại thành 1 nhóm

Có thể truy xuất đến từng thành phần của cấu trúc, dùng tên biến của cấu trúc cộng thêm dấu chấm, sau đó đến các thành phần của cấu trúc

Ví dụ con trỏ trỏ đến thành phần của cấu trúc thì dùng cái gì, dấu mũi tên ->.

Câu hỏi:

Kích cỡ của 1 cấu trúc khi ta khai báo thì sẽ được cấp phát sẽ như thế nào và nó sẽ có bao nhiêu byte.

Chú ý: khi khai báo 1 cấu trúc thì nó sẽ ko nằm ở trong bất cứ hàm nào.

Kiểu dữ liệu này:

Struct cat

{

Char book\_name [30];

Char author [30];

Int edn;

Float price;

Char red;

}

Được cấp phát bao nhiêu byte

Struct cat book1;

Printf(“size of cat: %d”, sizeof(struct cat));

Được cấp phát 72 byte, vì sao lại là 72

1\*30+1\*30+4\*1+4\*1+1\*1

Trên các hệ thống, bình thường khi chúng ta load vào RAM luôn nếu giả sử chúng ta load từng byte từng byte một thì nó quá chậm cho nên người ta thay thành đọc 1 lần nhiều hơn, thường thì hệ thống đọc 1 lần là 8 byte, gọi là 1 word.

Hệ thống có thể support đọc hay 2 byte hay 4 byte (nửa word, half word). Nếu nó đọc 2 byte 1 thì nó ko đọc địa chỉ lẻ mà nó chỉ đọc địa chỉ chẵn mà thôi. Như vậy với kiểu char này thì giả sử thằng book name này mất 30 byte (vì mảng kiểu char, mà kiểu char 1 byte cho nên mảng đó được cấp phát 1\*30 byte); ví dụ như hệ thống đọc 1 word là 4 byte một thì nó phải đọc 7 lần mới được 28 byte và nó phải đọc thêm 1 lần nữa là 8 lần thì phải là 32 byte. Tuy nhiên do bookname kiểu char nên có khi ko cần tới 32 byte (mà đọc 1 word 2 byte chẳng hạn)

Đến đoạn char author [30] cũng vậy

Thằng int edn cũng vậy, sẽ được assign vào 1 byte mà có thể chia hết cho 4, float cũng vậy

Nhưng vấn đề là theo lý thuyết thằng char red sẽ chiếm tất cả là 4 byte nên suy ra chương trình ghi ra là 72 byte

Struct cat

{

Char book\_name [30];

Char author [30];

Int edn;

Char red;

Float price;

}

Vẫn là 72 byte, char để đâu cũng là 4 byte, cái này phụ thuộc alligment, khả năng support của hệ thống

Kiểu dữ liệu có cấu trúc này được ứng dụng rất là nhiều thứ, việc chúng ta thấy tất cả các bài toán thực tế ko phải kiểu dữ liệu thuần tuý, ví dụ như chương trình quản lý hardware, thanh ghi

Sử dụng từ khoá typedef để định nghĩa 1 kiểu dữ liệu mới, thay vì kiểu dữ liệu có cấu trúc

Tức là

Typdef struct cat {….};

Thì sau này chỉ cần dùng cat như các kiểu dữ liệu thuần tuý, ko cần từ khoá struct nữa.

Biến union là gì

Định nghĩa kiểu union

Khai báo

Size of

User case

Là kiểu dữ liệu người dùng tự định nghĩa giống như struct nhưng nó ko chiếm bộ nhớ như các thành phần cộng lại của struct mà nó chỉ chiếm bộ nhớ bằng thành phần lớn nhất.

Và nó có thể chứa như biến struct, truy xuất cũng giống luôn.

Nó ko được cấp phát như các thành phần của struct mà nó chỉ cấp phát 1 vùng nhớ bằng vùng nhớ của thành phần lớn nhất trong cái union đấy thôi.

Giá trị mà nó lưu trữ tại một thời điểm là giá trị được lưu trữ của thành viên cuối cùng trong nó thôi.

Union cat

{

Char book\_name [30];

Char author [30];

Int edn;

Char red;

Float price;

}

Union cat book1;

Printf(“size of cat: %d”, sizeof(book1));

Kết quả là size of cat là 32 (do char book\_name [30] được cấp phát thành 32)

Thêm

Gets(book1.book\_name);

//book1.book\_name = “Hello World”;

Printf(“book name: %d”, book1.book\_name);

Printf(“book author: %d”, book1.author);

Printf(“edn: %d”, book1.edn);

Printf(“red: %d”, book1.red);

Printf(“price: %d”, book1.price);

Khi nhập book\_name vào và truy xuất ra từng thành phần này, theo lý thuyết các thành phần này, giả sử như là struct thì nó là từng thành phần riêng lẻ nhau ở các vùng nhớ tách nhau ra, nó rối tiếp nhưng nó liền nhau và tách nhau ra chứ nó ko xếp chồng lên thành 1 đống.

Khi in book\_name này ra và get book\_name này là hello thì in ra book\_name cũng là hello

Kết quả

Size of cat: 32

Hello

Book name: Hello

Book author: hello

Book edn: 1819043144

Book red: H

Book price: 1143139122437582505939828736.0000000

Nhìn vào thằng book author, do các thành phần này cùng truy xuất tới 1 vùng nhớ cho nên nó sẽ in ra chữ hello, book edn là chuyển từ ký tự hello sang 1 mã nào đó (ascii chẳng hạn)

Book red thì nó chỉ lấy 1 giá trị kiểu char, là chữ H

Tóm gọn lại là khi chúng ta truy xuất thì chỉ truy xuất trên 1 vùng nhớ, vậy giá trị thực tế mà nó lưu đầu tiên là bookname.

Union hay ở chỗ và được sử dụng ntn

Được sử như 1 struct biến tạm để gọi là, ví dụ:

Viết 1 chương trình đổi 2 giá trị của cái book1 và book2 cho nhau

Nếu chúng ta sử dụng struct thì phải khai báo struct tạm là struct nó gán với struct kia hoặc là copy vùng nhớ thì phải malloc vùng nhớ thì chúng ta ko phải liên quan tới struct chúng ta đang dùng.

Nếu sử dụng union thì chúng ta có thể di chuyển từng thành phần này mà sử dụng chung 1 cấu trúc thôi

Thay đổi

Trong hàm main

Void main()

{

Union cat book1;

Printf(“size of cat: %d\n”, sizeof(book1));

Gets(book1.book\_name);

//book1.book\_name = “Hello World”;

Printf(“book name: %d”, book1.book\_name);

Printf(“book author: %d”, book1.author);

Printf(“edn: %d”, book1.edn);

Printf(“red: %d”, book1.red);

Printf(“price: %d”, book1.price);

Book1.edn = 0xffff0000;

//book1.book\_name = “Hello World”;

Printf(“book name: %d”, book1.book\_name);

Printf(“book author: %d”, book1.author);

Printf(“edn: %d”, book1.edn);

Printf(“red: %d”, book1.red);

Printf(“price: %d”, book1.price);

}

Và make lại

Nhập hello thì

Size of cat: 32

Hello

Book name: Hello

Book author: hello

Book edn: 1819043144

Book red: H

Book price: 1143139122437582505939828736.0000000

Book name:

Book author:

Book edn: -65536

Book red:

Book price: -nan

Đến đoạn thứ 2 thì book name ko in ra gì cả, author ko in ra gì cả, book edn thì in ra giá trị này.

Book price thì đang bị lỗi

Thì lúc này giá trị của nó được update bởi chính trường khác, trường edn nên là giá trị của vùng nhớ sẽ khác

Biến enum là gì

Được sử dụng để assign cho 1 cái name cho 1 giá trị constant nào đó, thay vì việc ta sử dụng 1 constant nào đấy, 1 số hay là nó có dạng mang tính chất là hard code. Giả sử số tháng trong năm chẳng hạn thì ở ngoài dùng luôn là số 12, thế nhưng nếu chúng ta có thể biến đổi cái hard code này, số 12 này thành number of month thì chúng ta đọc code sẽ dễ hơn và dễ hình dung hơn.