Bài 15 Đoạn mã sau có vấn đề gì trong đoạn mã sau không?

void myfunction(char \*q) {  
memcpy(q, "hello", 5);   
}  
  
int main(void) {  
char \*p;  
myfunction(p);  
return 0;  
}

**p trong main là một con trỏ chưa được khởi tạo (garbage). Truyền p vào myfunction và dùng memcpy(q, "hello", 5) ghi vào vùng nhớ mà q trỏ tới là undefined behavior — chương trình có thể sập (segfault), ghi đè bộ nhớ khác, hoặc hành vi bất định khác.**

**Ngoài ra, memcpy sao chép 5 byte nhưng không đặt ký tự null '\0'; nếu mục đích là sao chép chuỗi C, cần sao chép 6 byte (bao gồm '\0').**

**Bài 16 Hàm hoạt động ra sao? Các biến được cấp phát ở đâu?**

**int a;  
char \*b;  
const char c[20] = "I am a string";  
  
void func(char d, int c\_param, char \*f) {  
int g = 0;  
static int h = 2;  
char \*i = NULL;  
i = (char \*)malloc(20);  
if (i != NULL) {  
free(i);  
}  
}**

**1. Hoạt động của hàm:  
Khi hàm func() được gọi, các tham số d, c\_param và f được truyền vào thông qua stack. Bên trong hàm, các biến cục bộ g và i được khởi tạo trên stack. Biến h là biến tĩnh cục bộ nên không được tạo lại mỗi lần gọi, mà được cấp phát cố định trong vùng nhớ tĩnh (data segment).**

**Tiếp theo, lệnh malloc(20) sẽ cấp phát 20 byte trên heap và gán địa chỉ của vùng nhớ đó cho con trỏ i. Nếu việc cấp phát thành công (i ≠ NULL), hàm sẽ thực hiện free(i) để giải phóng vùng nhớ vừa được cấp phát, tránh rò rỉ bộ nhớ. Sau khi hàm kết thúc, các biến cục bộ và tham số bị hủy cùng với stack frame của hàm, trong khi các biến tĩnh và biến toàn cục vẫn tồn tại.**

**2. Vị trí cấp phát của các biến:**

| **Tên biến** | **Phạm vi** | **Kiểu** | **Vị trí cấp phát** | **Ghi chú** |
| --- | --- | --- | --- | --- |
| **a** | **Toàn cục** | **Thường** | **BSS segment** | **Chưa khởi tạo, giá trị mặc định = 0** |
| **b** | **Toàn cục** | **Con trỏ** | **BSS segment** | **Ban đầu NULL** |
| **c** | **Toàn cục** | **Mảng hằng** | **Read-only data** | **Lưu chuỗi hằng** |
| **d, c\_param, f** | **Tham số hàm** | **–** | **Stack** | **Tạo khi gọi hàm** |
| **g** | **Cục bộ** | **Thường** | **Stack** | **Bị hủy khi hàm kết thúc** |
| **h** | **Cục bộ** | **Static** | **Data segment** | **Tồn tại suốt chương trình** |
| **i** | **Cục bộ** | **Con trỏ** | **Stack (chính con trỏ)** | **Trỏ đến vùng nhớ heap** |
| **malloc(20)** | **–** | **Heap** | **Heap** | **Được giải phóng bằng free(i)** |

**Bài 17 a và b được cấp phát và lưu trữ ở đâu?**

**void main(void) {  
char a[4] = "foo";  
char \*b = "bar";  
}**

| **Biến** | **Kiểu** | **Vị trí cấp phát** | **Ghi chú** |
| --- | --- | --- | --- |
| **a** | **Mảng cục bộ** | **Stack** | **Lưu nội dung "foo", bị hủy khi hàm kết thúc** |
| **b** | **Con trỏ cục bộ** | **Stack (chính con trỏ)** | **Trỏ tới chuỗi hằng "bar" nằm trong read-only data segment** |

**Bài 18 Đoạn mã sau có vấn đề gì không? Với RAM = 2000 byte?**

**void myfunction() {  
char \*q;  
for (int i = 0; i < 1000; i++) {  
q = (char \*)malloc(8);  
if (q != NULL) {  
memcpy(q, "hello", 6);  
}  
}  
}**

**Đoạn mã trên có vấn đề nghiêm trọng về rò rỉ bộ nhớ (memory leak).**

**Trong vòng lặp for, chương trình thực hiện malloc(8) 1000 lần, nhưng không có bất kỳ lệnh free() nào để giải phóng vùng nhớ đã được cấp phát, gây rò rỉ bộ nhớ.**

**Bài 19 Đoạn mã sau có vấn đề gì không? Làm sao để in ra "hello"?**

**void myfunction(char \*q) {  
q = (char \*)malloc(8);  
if (q) {  
memcpy(q, "hello", 6);  
}  
}  
  
void main(void) {  
char \*p = NULL;  
myfunction(p);  
}**

**Đoạn mã có vấn đề vì con trỏ p được truyền theo cơ chế truyền giá trị, nên khi q được cấp phát trong hàm, địa chỉ này không được gán trở lại cho p. Sau khi gọi hàm, p vẫn là NULL, nên không thể in ra "hello".**

**Để khắc phục, có thể:**

* **Truyền địa chỉ của con trỏ (char \*\*) vào hàm, hoặc**
* **Trả về con trỏ từ hàm rồi gán cho p.**

**Ví dụ dùng con trỏ cấp 2:**

**void myfunction(char \*\*q) {**

**\*q = (char \*)malloc(8);**

**if (\*q) memcpy(\*q, "hello", 6);**

**}**

**void main(void) {**

**char \*p = NULL;**

**myfunction(&p);**

**printf("%s", p); // In ra hello**

**free(p);**

**}**

**Bài 20 Các biến sau được lưu ở đâu?**

**int a = 5;  
int d;  
char b[6] = "hello";  
const int c = 4;  
  
void myfunction(void)  
{  
static int g = 5;  
const int h = 6;  
char i;  
}**

| **Biến** | **Phạm vi** | **Kiểu** | **Vị trí lưu trữ** | **Ghi chú** |
| --- | --- | --- | --- | --- |
| **a** | **Toàn cục** | **Khởi tạo** | **Data segment** | **Có giá trị khởi tạo 5** |
| **d** | **Toàn cục** | **Chưa khởi tạo** | **BSS segment** | **Giá trị mặc định 0** |
| **b** | **Toàn cục** | **Mảng ký tự, khởi tạo** | **Data segment** | **Lưu chuỗi "hello"** |
| **c** | **Toàn cục** | **Hằng số** | **Read-only data segment** | **Không thể thay đổi lúc chạy** |
| **g** | **Cục bộ static** | **Khởi tạo** | **Data segment** | **Tồn tại suốt chương trình** |
| **h** | **Cục bộ const** |  | **Stack** | **Giá trị 6, hủy khi hàm kết thúc** |
| **i** | **Cục bộ thường** |  | **Stack** | **Tạo khi vào hàm, hủy khi thoát hàm** |

**Bài 21 Đoạn mã sau có vấn đề gì không?**

**void main(void) {  
char array[20];  
for (int i = 0; i < 20; i++) {  
array[i] = i;  
}  
  
int \*p = (int \*)array;  
p++;  
p++;  
printf("Value at p: 0x%x\n", \*p);  
}**

**Đoạn mã trên có vấn đề về truy cập bộ nhớ và ép kiểu con trỏ:**

1. **Ép kiểu từ char\* sang int\* có thể gây lỗi căn chỉnh (alignment) trên một số kiến trúc vi xử lý.**
   * **Biến array được khai báo kiểu char[20], địa chỉ có thể không được căn chỉnh theo 4 byte như int\* yêu cầu.**
   * **Khi ép kiểu int \*p = (int \*)array;, nếu địa chỉ array không chia hết cho 4, việc truy cập \*p có thể gây lỗi “bus error” hoặc hành vi không xác định.**
2. **Tăng con trỏ p hai lần tương đương với nhảy 8 byte (vì mỗi bước tăng là 4 byte trên hệ thống 32-bit). Khi đó p trỏ vào array[8], nhưng vì kiểu dữ liệu ban đầu là char, việc đọc 4 byte liên tiếp tại vị trí này để in ra có thể không phản ánh giá trị mong muốn.**

**Bài 22 Có vấn đề gì với hàm copy sau?**

**int copy(char \*scr, char \*dst, unsigned int size) {  
char \*s1;  
while (size--) {  
\*s1++ = \*scr++;  
\*dst++ = \*s1;  
}  
return 0;  
}**

**Hàm trên có lỗi vì con trỏ s1 không được khởi tạo, dẫn đến truy cập vùng nhớ không hợp lệ khi gán \*s1++ = \*scr++. Ngoài ra, logic sao chép sai, không thực hiện đúng việc copy từ scr sang dst.**

**Bài 23 Đoạn mã sau có vấn đề gì không?**

**void func(void) {   
char \*p = malloc(10);   
strcpy(p, "Embedded");   
}**

**Đoạn mã trên có vấn đề rò rỉ bộ nhớ (memory leak) vì con trỏ p được cấp phát bằng malloc nhưng không được free trước khi hàm kết thúc.  
Ngoài ra, không kiểm tra kết quả malloc, có thể dẫn đến lỗi nếu cấp phát thất bại.**

**Bài 24 Đoạn mã sau có hợp lệ không?**

**void test(void) {   
char \*str = "Hello";   
str[0] = 'h';   
}**

**Đoạn mã trên không hợp lệ.**

**Lý do:**

* **Biến str trỏ tới chuỗi hằng trong vùng nhớ chỉ đọc.**
* **Câu lệnh str[0] = 'h'; cố gắng ghi vào vùng nhớ này → gây lỗi runtime (undefined behavior).**

**Câu lệnh đúng:**

**char str[] = "Hello";**

**str[0] = 'h';**

**Bài 25 Các biến a, b, c, d được lưu ở đâu trong bộ nhớ?**

**int a = 10;   
const int b = 5;   
static int c = 3;   
int func(void) {   
int d = 2;   
return a + b + c + d;   
}**

**- a: biến toàn cục, được lưu ở vùng Data (RAM) vì có khởi tạo.**

**- b: biến hằng toàn cục, được lưu ở vùng Read-Only Data (ROM/Flash).**

**- c: biến tĩnh toàn cục, được lưu ở vùng Data (RAM).**

**- d: biến cục bộ, được lưu ở Stack khi hàm func được gọi.**

**Bài 26 Đoạn mã sau có vấn đề gì?**

**void recursion(int x) {   
int a[100];   
recursion(x + 1);   
}**

**Đoạn mã trên có vấn đề gọi đệ quy không có điều kiện dừng, dẫn đến:**

* **Hàm recursion sẽ gọi chính nó vô hạn.**
* **Mỗi lần gọi sẽ tạo mảng a[100] trên stack, làm stack ngày càng đầy.**
* **Kết quả: tràn stack (stack overflow), chương trình bị lỗi khi chạy.**

**Bài 27 Đoạn mã sau có in ra đúng giá trị ban đầu không? Tại sao?**

**int main(void) {   
int a = 0x12345678;   
char \*p = (char \*)&a;   
p[0] = 0xAA;   
printf("a = 0x%x\n", a);   
}**

**Đoạn mã trên không in ra đúng giá trị ban đầu của a.**

**Lý do:**

* **Con trỏ p trỏ đến địa chỉ của a và sửa byte thấp nhất p[0] = 0xAA.**
* **Trên hệ thống little-endian, byte thấp lưu trước, nên giá trị a sau khi gán sẽ bị thay đổi thành 0x123456AA.**
* **Vì vậy, giá trị in ra không còn là 0x12345678 ban đầu.**

**Bài 28 Đoạn mã sau có vấn đề gì?**

**char \*getString(void) {   
char str[] = "hello";   
return str;   
}   
int main(void) {   
char \*s = getString();   
printf("%s\n", s);   
}**

**Đoạn mã trên có vấn đề trả về con trỏ trỏ tới biến cục bộ.**

**Cụ thể:**

* **str là mảng cục bộ, được cấp phát trên stack.**
* **Khi getString kết thúc, vùng nhớ của str bị giải phóng.**
* **Con trỏ s trong main trỏ tới vùng nhớ không còn hợp lệ ⇒ hành vi không xác định (undefined behavior) khi dùng printf.**

**Bài 29**

**1. Kết quả in ra là gì?  
2. Nếu bỏ từ khóa static, kết quả có khác không?  
3. Biến count được lưu ở vùng nhớ nào?**

**void counter(void) {   
static int count = 0;   
count++;   
printf("%d ", count);   
}   
  
int main(void) {   
for (int i = 0; i < 3; i++)   
counter();   
return 0;   
}**

1. **Kết quả in ra:**

**1 2 3**

**→ Vì biến count là static, giữ giá trị giữa các lần gọi hàm.**

1. **Nếu bỏ từ khóa static:  
   Kết quả sẽ là**

**1 1 1**

**→ Vì mỗi lần gọi hàm, biến count được khởi tạo lại về 0.**

1. **Vị trí lưu biến count:  
   Biến count (static) được lưu ở vùng Data (RAM), không nằm trên stack.**

**Bài 30**

**1. Code này có biên dịch được không?  
2. Nếu không, compiler báo lỗi gì? Tại sao?**

**int main(void) {  
register int x = 5;  
printf("%d\n", x);  
printf("%p\n", &x);  
return 0;  
}**

1. **Code này không biên dịch được.**
2. **Lỗi compiler:  
   Thường báo như sau (tuỳ trình biên dịch):**

**error: address of register variable ‘x’ requested**

**Nguyên nhân:**

* **Biến x được khai báo với từ khóa register, compiler có thể lưu x trong thanh ghi CPU, không có địa chỉ bộ nhớ cụ thể.**
* **Lệnh &x yêu cầu lấy địa chỉ của x, điều này không hợp lệ với biến register, dẫn đến lỗi biên dịch.**

**sizeof(struct ab) = 8 byte**

* **int a chiếm 4 byte.**
* **char b chiếm 1 byte.**
* **Compiler sẽ thêm 3 byte đệm (padding) để căn chỉnh cấu trúc theo biên 4 byte → tổng = 4 + 1 + 3 = 8 byte.**

**sizeof(struct ba) = 5 byte**

* **Nhờ #pragma pack(push, 1) → bỏ căn chỉnh, cấu trúc được đóng gói 1 byte.**
* **char a = 1 byte, int b = 4 byte → tổng = 1 + 4 = 5 byte, không có padding.**

**Bài 32 sizeof(mystruct) sẽ in ra gì?**

**typedef struct {  
char x;  
int y;  
} mystruct;**

**sizeof(mystruct) sẽ in ra 8 byte (trên hệ thống 32-bit hoặc 64-bit thông thường).**

* **char x chiếm 1 byte.**
* **Tiếp theo là int y (4 byte) → để căn chỉnh theo biên 4 byte, compiler sẽ thêm 3 byte padding sau x.**
* **Tổng: 1 (x) + 3 (padding) + 4 (y) = 8 byte.**

**Bài 33 sizeof(struct A) là bao nhiêu trên hệ thống 32-bit?**

**struct A {   
char a;   
int b;   
short c;   
};**

**Trên hệ thống 32-bit, sizeof(struct A) = 12 byte.**

* **char a → 1 byte**
* **Padding 3 byte để căn int b theo biên 4 byte**
* **int b → 4 byte**
* **short c → 2 byte**
* **Padding 2 byte ở cuối để tổng kích thước là bội số của 4 (biên lớn nhất trong struct)**

**Tổng:  
1 (a) + 3 (padding) + 4 (b) + 2 (c) + 2 (padding) = 12 byte**

**Bài 34 Ảnh hưởng của #pragma pack(1) là gì? Kết quả sizeof(struct B) là bao nhiêu trên hệ thống 32-bit?**

**#pragma pack(push, 1)   
struct B {   
char a;   
int b;   
short c;   
};   
#pragma pack(pop)**

**Ảnh hưởng của #pragma pack(1):**

* **Chỉ thị này yêu cầu compiler đóng gói cấu trúc với căn chỉnh 1 byte,  
  → nghĩa là không chèn byte đệm (padding) giữa các thành phần trong struct.**

**Tính sizeof(struct B) trên hệ thống 32-bit:**

* **char a → 1 byte**
* **int b → 4 byte (ngay sau a, không padding)**
* **short c → 2 byte (ngay sau b, không padding)**

**Tổng: 1 + 4 + 2 = ✅ 7 byte.**

**Bài 35. Tính sizeof(struct Outer) trên hệ thống 32-bit.**

**struct Inner {   
char c;   
int d;   
};   
  
struct Outer {   
int a;   
struct Inner in;   
char e[3];   
};**

* **sizeof(struct Inner) = 8 byte.**
* **sizeof(struct Outer) = 16 byte:**
* **a: 4 byte**
* **in: 8 byte**
* **e[3]: 3 byte + 1 byte padding cuối để căn 4 byte.**

**Bài 36 Trong đoạn mã sau, ab.b có giá trị là bao nhiêu?**

**union data {  
int a;  
char b;  
} ab;  
  
ab.a = 64;  
ab.a = 2500; // 0x09C4**

**Giá trị của ab.b là 0xC4 (theo hệ 16) hay 196 (theo hệ 10).**

* **ab.a = 2500 → 2500₁₀ = 0x09C4.**
* **Trong union, các thành viên dùng chung vùng nhớ, nên ab.b trỏ tới byte thấp nhất của a (trên hệ thống little-endian).**
* **Byte thấp của 0x09C4 là 0xC4.**

**Bài 37 sizeof(struct Test) là bao nhiêu trên hệ thống 32-bit?**

union Data {   
int i;   
char c[4];   
};   
  
struct Test {   
char flag;   
union Data data;   
};

* sizeof(union Data) = 4 byte (do thành phần lớn nhất là int).
* sizeof(struct Test) = 8 byte:
  + flag: 1 byte + 3 byte padding để căn 4 byte.
  + data: 4 byte.

Bài giải

* **Little-endian:**  
  Dữ liệu được lưu **byte thấp trước** → kết quả in ra là:
* 78 56 34 12
* **Big-endian:**  
  Dữ liệu được lưu **byte cao trước** → kết quả in ra là:
* 12 34 56 78

Bài 39 sizeof(struct Flags) là bao nhiêu?

struct Flags {   
unsigned int a : 1;   
unsigned int b : 3;   
unsigned int c : 4;   
};

Bài giải

Tổng số bit:

* a: 1 bit
* b: 3 bit
* c: 4 bit  
  → Tổng = 8 bit = 1 byte.

Tuy nhiên, kiểu chứa là unsigned int (4 byte trên hệ 32-bit), và các bit field cùng kiểu sẽ được **đóng gói trong cùng một unsigned int** nếu còn chỗ.  
 Cả 3 trường nằm trong **cùng một unsigned int**, nên kích thước của struct Flags = kích thước của một unsigned int = **4 byte**.

Bài 40

1. Giá trị flag, mode, value sẽ in ra bao nhiêu (giả sử hệ thống little-endian)?  
   2. Tại sao giá trị thay đổi khi chạy trên big-endian?  
   3. Giải thích cách compiler ánh xạ bit field vào raw.

union Data {   
struct {   
unsigned int flag : 3;   
unsigned int mode : 5;   
unsigned int value : 8;   
} info;   
unsigned short raw;   
};   
  
int main(void) {   
union Data d;   
d.raw = 0xABCD;   
printf("flag=%u, mode=%u, value=%u\n", d.info.flag, d.info.mode, d.info.value);   
}

1. Trên hệ thống little-endian, raw = 0xABCD → sắp xếp byte CD AB.
   * flag = 5
   * mode = 19
   * value = 171
2. Trên big-endian, thứ tự byte đổi thành AB CD nên các bit được ánh xạ khác → giá trị các trường thay đổi.
3. Compiler ánh xạ bit field từ bit thấp lên trong kiểu cơ sở.
   * flag: bit 0–2
   * mode: bit 3–7
   * value: bit 8–15  
     Sự khác biệt endianness làm thay đổi cách đọc các bit này trong bộ nhớ.

Bài 41

1. sizeof(struct Packet) là bao nhiêu trên 32-bit?  
2. Giải thích tại sao sizeof không phải luôn bằng 2?  
3. Nếu ta gán data.raw = 0xAB, giá trị của id, type, err lần lượt là gì?

struct Packet {   
unsigned char header;   
union {   
struct {   
unsigned int id : 4;   
unsigned int type : 3;   
unsigned int err : 1;   
} bits;   
unsigned char raw;   
} data;   
}

1. ✅ sizeof(struct Packet) = **2 byte**

* header: 1 byte
* data: union có raw (1 byte) và bit field (1 byte) → kích thước union là 1 byte.  
  → Tổng = 1 + 1 = **2 byte**.

1. sizeof không phải luôn bằng 2 vì compiler có thể **thêm padding** để căn hàng theo biên của kiểu lớn nhất trong struct. Trong ví dụ này các thành phần đều là unsigned char, nên không có padding thêm. Nếu có kiểu lớn hơn (ví dụ int), kích thước có thể tăng do căn chỉnh bộ nhớ.
2. Khi data.raw = 0xAB → 0xAB = 1010 1011₂ (little-endian, LSB ở bit 0):

* id (4 bit thấp): 1011₂ = **11**
* type (3 bit kế): 010₂ = **2**
* err (1 bit cao): 1 = **1**

Kết quả:

id = 11, type = 2, err = 1

Bài 42

1. sizeof(union Config) là bao nhiêu?  
2. Tại sao cả hai struct đều có cùng vùng nhớ?  
3. Nếu gán security.key = 0x7F, giá trị normal.mode có thay đổi không?

union Config {  
struct {  
unsigned int enable : 1;  
unsigned int speed : 2;  
unsigned int mode : 3;  
} normal;  
  
struct {  
unsigned int secure : 1;  
unsigned int key : 7;  
} security;  
};

**1. sizeof(union Config)**

* normal: gồm các bit field 1 + 2 + 3 = 6 bit → compiler sẽ gom trong 1 biến unsigned int (thường là 4 byte trên hệ thống 32-bit).
* security: gồm 1 + 7 = 8 bit, cũng nằm trong 1 unsigned int → kích thước 4 byte.
* Trong union, kích thước bằng kích thước **thành phần lớn nhất** →  
   sizeof(union Config) = 4 byte.

**2. Tại sao hai struct dùng chung vùng nhớ**

Trong union, **tất cả các thành viên bắt đầu từ cùng một địa chỉ**.  
Do đó, normal và security **chia sẻ cùng một vùng nhớ** — ghi vào một thành viên sẽ ảnh hưởng trực tiếp đến thành viên còn lại.

**3. Ảnh hưởng khi gán security.key = 0x7F**

* security.key chiếm 7 bit thấp (bit 1 → bit 7),
* normal cũng dùng chung vùng nhớ → khi ta thay đổi security.key, các bit tương ứng trong normal cũng thay đổi.  
  → Cụ thể, các bit thuộc normal.enable, normal.speed, normal.mode sẽ bị ghi đè theo giá trị mới.

**Kết luận:**  
 normal.mode **có thể thay đổi** khi gán security.key = 0x7F, vì cả hai trường cùng nằm trong chung một từ 32-bit của union.

**Bài 43**

1. In ra kết quả gì (với little-endian)?  
2. Giải thích tại sao việc truy cập ctrl.bits và ctrl.reg lại ảnh hưởng trực tiếp nhau?  
3. Nếu muốn MODE chiếm bit 4–5 thay vì 1–2, sửa khai báo thế nào?

typedef union {   
struct {   
unsigned int EN : 1;   
unsigned int MODE : 2;   
unsigned int RSV : 1;   
unsigned int CLK : 4;   
} bits;   
unsigned char reg;   
} CTRL\_REG;   
  
int main(void) {   
CTRL\_REG ctrl = { .reg = 0xB2 };   
printf("EN=%u, MODE=%u, CLK=%u\n", ctrl.bits.EN, ctrl.bits.MODE, ctrl.bits.CLK);   
}

1. **Kết quả in ra (little-endian)**  
   0xB2 = 1011 0010₂

* EN (bit 0) = 0
* MODE (bit 1–2) = 01₂ = 1
* CLK (bit 4–7) = 1011₂ = 11

**Kết quả:**

EN=0, MODE=1, CLK=11

1. **Giải thích**  
   bits và reg nằm trong cùng một union → chúng dùng **chung vùng nhớ**.  
   Gán vào reg sẽ thay đổi từng bit trong bits, và ngược lại.
2. **Sửa để MODE chiếm bit 4–5**  
   Đổi thứ tự và số bit trong struct:

struct {

unsigned int EN : 1;

unsigned int RSV : 1;

unsigned int CLK : 2;

unsigned int MODE : 2; // chiếm bit 4–5

unsigned int PAD : 2;

} bits;

MODE lúc này bắt đầu sau 4 bit đầu tiên → nằm ở bit 4–5 như yêu cầu.

Bài 44

1. sizeof(union Frame) là bao nhiêu?  
2. Nếu bytes.low = 0xAA và bytes.high = 0xBB, giá trị của info.len, info.type, info.crc là gì?  
3. Giải thích tại sao giá trị có thể khác nhau giữa compiler (GCC vs ARMCC)?

union Frame {   
struct {   
unsigned short len : 10;   
unsigned short type : 3;   
unsigned short crc : 3;   
} info;   
  
struct {   
unsigned char low;   
unsigned char high;   
} bytes;   
};

1. **Kích thước sizeof(union Frame)**

* info: tổng 10 + 3 + 3 = 16 bit → compiler sẽ lưu trong unsigned short (2 byte).
* bytes: gồm low (1 byte) + high (1 byte) = 2 byte.  
  → Kích thước của union bằng kích thước thành phần lớn nhất = **2 byte**.

sizeof(union Frame) = 2.

1. **Giá trị các trường khi bytes.low = 0xAA, bytes.high = 0xBB**  
   Giá trị tổng trong union:

high = 0xBB

low = 0xAA

=> raw = 0xBBAA = 1011 1011 1010 1010₂

Trên hệ thống **little-endian**, bit 0 bắt đầu từ low.

* len (10 bit thấp): 1110101010₂ = 0x3AA = 938
* type (3 bit kế): 011₂ = 3
* crc (3 bit cao): 101₂ = 5

Kết quả:

len = 938, type = 3, crc = 5

1. **Sự khác nhau giữa compiler (GCC vs ARMCC)**  
   Cách **ánh xạ bit field** (bit ordering trong struct) không được chuẩn hóa hoàn toàn.

* **GCC** thường gán bit 0 cho bit thấp nhất trong byte thấp (little-endian logic).
* **ARMCC** có thể ánh xạ bit field từ MSB → LSB trong cùng một từ, dẫn đến len, type, crc được tách theo thứ tự ngược lại.

👉 Vì vậy, khi biên dịch cùng code, **cách tách bit sẽ khác nhau → giá trị trường khác nhau** tùy compiler.