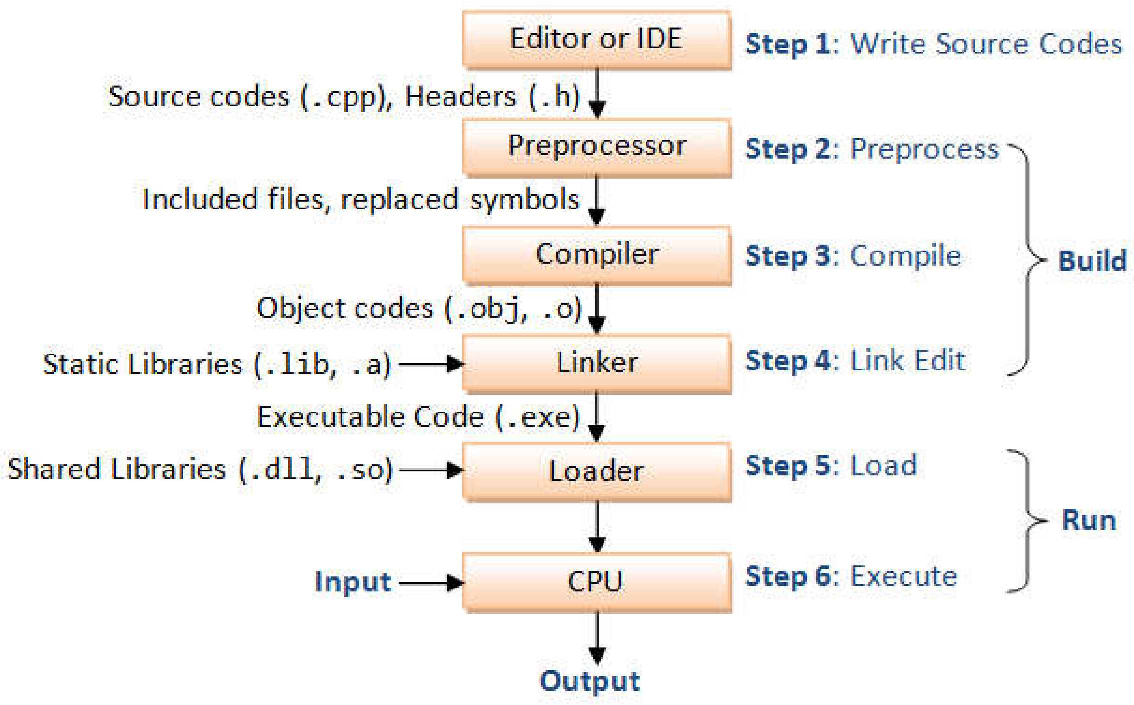
1. Phân biệt Syntax-error, Runtime-error and Logical error?

Syntax-error: Lỗi cú pháp được phát hiện ngay khi biên dịch chương trình. Trình biên dịch sẽ thông báo lỗi tại cửa sổ Output.

Runtime-error: Chương trình đã được biên dịch thành công, gặp lỗi khi chạy chương trình do đầu vào hoặc đầu ra có giá trị không mong muốn.

Logical-error: Chương trình đã được biên dịch và chạy không gặp lỗi Runtime-error. Nhưng kết quả đầu ra không đúng theo yêu cầu do logic xử lý bài toán bị sai.

2. Complier and linker là gì?



3. Vòng lặp vô hạn là gì?

Vòng lặp vô hạn là chuỗi các câu lệnh được thực hiện lặp đi lặp lại không có điểm dừng. Nguyên nhân có thể là:

- Vòng lặp chưa có điều kết thúc vòng lặp.
- Vòng lặp có điều kiện để kết thúc vòng lặp nhưng điều kiện này không bao giờ được đáp ứng.
- Hoặc có thể một logic nào đó dẫn đến lặp vô hạn (ví dụ: lệnh goto có thể tạo ra vòng lặp).
- Chương trình bị dính vòng lặp vô hạn, chương trình có thể bị gây treo PC hoặc bị crash (nếu trong vòng lặp khai bao biến hoặc cấp phát vùng nhớ mà không giải phóng vùng nhớ).

- Do vậy, cần tránh để xảy ra vòng lặp vô hạn trong chương trình.
- Trong một số trường hợp (requirement), lập trình viên không xác định chính xác được số lần lặp để xử lí 1 logic nào đó. Lúc này, lập trình viên phải sử dụng khai báo vòng lặp vô hạn **for()**, **while(), do while()**.

4. Biến static, extern là gì?

Biến local

– Biến local là xuất hiện trong phạm vi cụ thể.
– Biến local chỉ tồn tại trong hàm mà biến được khai báo
– Đôi khi, biến local được gọi là biến tự động (auto) bởi vì các biến được tự động sinh ra khi hàm được thực hiện và sẽ tự động biến mất khi kết thúc hàm.
– Từ khóa **auto** được sử dụng để ám chỉ biến cục bộ.

Biến global và extern

– Biến global là biến được khai báo bên ngoài tất cả các hàm và có giá trị với tất cả các hàm trong chương trình. Tức là các hàm trong chương trình có thể sử dụng biến global để tính toán.
– Biến global tồn tại đến khi nào chương trình kết thúc.
– Có thể định nghĩa 1 biến global trong 1 file (.c/.cpp/.h) và truy cập biến này từ 1 file (.c/.cpp/.h) khác. Để làm điều này, biến phải được khai báo ở cả 2 file và từ khóa **extern** được thêm trong lần khai báo thứ 2.

Biến static

– Biến **static** có thể là global hoặc local. Cả hai đều được khai báo với từ khóa **static** đi kèm.
– Biến **local static** là biến có thể duy trì giá trị từ lần gọi hàm thứ nhất đến các lần gọi hàm tiếp theo. Biến local static tồn tại đến khi chương trình kết thúc.
– Khi tạo 1 biến local static trong hàm, chúng ta nên khởi tạo giá trị cho chúng. Nếu không giá trị biến được gán mặc định bằng 0.
– Biến **global static** là biến global mà chỉ có thể truy cập từ file (.c/.cpp) mà biến đó được định nghĩa.

Biến const

– Trong ngôn ngữ C, tiền xử lý #define được sử dụng để tạo biến với giá trị là hằng số.
– Trong ngôn ngữ C++, xuất hiện một số vấn đề: khi sử dụng #define, tiền xử lí sẽ nhảy thẳng vào source code và thay thế biến bằng giá trị đã định nghĩa. Vì biến #define chỉ tồn tại bên trong file mà nó được định nghĩa, có thể xảy ra trường hợp định nghĩa tên biến giống nhưng khác về giá trị.
– Định nghĩa biến hằng trong C++, chúng ta nên sử dụng từ khóa **const** đi kèm.
– Khi sử dụng từ khóa **const**, phải khởi tạo giá trị ban đầu cho biến.

5. Hàm sprintf() được sử dụng trong trường hợp nào?

int sprintf(char *buffer, const char *format [,argument] ...);

- Hàm sprintf() được dùng để tạo ra chuỗi từ các kiểu dữ liệu nguyên thủy khác nhau (char*, int, float,…)
- Hàm sprintf() trả về số lượng kí tự được ghi ra chuỗi buff.

6. Sư khác nhau giữa memcpy và strcpy?

strcpy()

- Copy chuỗi kí tự nguồn sang chuỗi đích. Hàm strcpy copy kí tự đến khi gặp kí tự NULL thì dừng lại.

memcpy()

- Copy chính xác số byte dữ liệu từ vùng nhớ nguồn sang vùng nhớ đích.

Chú ý: cả 2 hàm không xử lí được trường hợp overlap buffer.

7. Sự khác nhau giữa memcpy() và memmove()?

Cả 2 hàm **memcpy** và **memmove** được sử dụng để copy N byte dữ liệu từ vùng nhớ này sang vùng nhớ khác. Tuy nhiên, trong trường hợp vùng nhớ nguồn overlap với vùng nhớ đích:

- Hàm memmove đảm bảo việc copy dữ liệu và output là chính xác.
- Hàm memcpy KHÔNG đảm bảo việc copy dữ liệu và output là chính xác.

8. Sự khác nhau giữa truyền giá trị và truyền tham biến (con trỏ và tham chiếu)?

Khi truyền đối số kiểu tham trị, chương trình biên dịch sẽ copy giá trị của đối số để gán cho tham số của hàm (không tác động trực tiếp đến biến số truyền vào).

Phương pháp truyền tham biến là cách truyền địa chỉ của đối số cho các tham số tương ứng của hàm được gọi. Với cách truyền tham biến, giá trị của đối số truyền vào có thể bị thay đổi bởi việc gọi hàm.
Truyền tham biến chia ra thành 2 loại: truyền con trỏ (dùng trong C và C++) , truyền tham chiếu (chỉ dùng trong C++)

9. Memory leak là gì? Cách tránh memory leak?

Leak memory được định nghĩa như việc lập trình viên quên không giải phóng (**free** hoặc delete) vùng nhớ đã cấp phát sau khi sử dụng xong. Một lượng nhỏ memory leak không được phát hiện ngay ban đầu. Lượng leak memory sẽ tăng dần theo thời gian. Điều này làm cho ứng dụng giảm performance, hoặc thậm chí gây chết (crash) chương trình khi sử dụng hết memory. Hoặc tệ hơn nữa, việc leak memory của 1 ứng dụng sử dụng hết vùng nhớ của các ứng dụng khác, làm cho các ứng dụng khác có thể bị crash.

10. Ưu và nhược điểm khi sử dụng mảng trong ngôn ngữ C?

Ưu điểm: Mảng có 1 số ưu điểm như sau.

- Dễ hiểu và dễ sử dụng: chỉ cần khai báo <kiểu dữ liệu> tên mảng[kích thước].
- Truy cập đến các phần tử trong mảng nhanh: chúng ta có thể truy cập tới bất kì phần tử nào trong mảng bằng cách chỉ định chỉ số cho phần tử đó. Ví dụ: mảng A[100] gồm 100 phân tử (từ A[0] đến A[99]), để truy xuất tới phần tử thứ i ta chỉ cần gọi giá trị A[i]. Thời gian truy cập phần tử A[0] và thời gian truy cập phần tử A[1000] là như nhau.

Hạn chế: Bên cạnh những ưu điểm, mảng còn tồn tại một số hạn chế sau.

- Kích thước của mảng phải là cố định: Trong cấp phát mảng tĩnh, mảng cần được khai báo với kích thước xác định trước khi chạy chương trình. (vùng nhớ cho mảng được cấp phát khi biên dịch). Trong cấp phát động, vùng nhớ được cấp phát khi chạy chương trình. Như các bạn đã biết, vùng nhớ giành cho mỗi chương trình thường không dự đoán được trước. Nếu khai báo mảng với kích thước lớn, không sử dụng hết sẽ gây lãng phí bộ nhớ, ngược lại nếu kích thước vùng nhớ không đủ dùng, chúng ta không thể mở rộng vùng nhớ thêm được, dẫn đến buffer overrun (tràn vùng nhớ).
- Các byte vùng nhớ cấp phát mảng được sắp xếp liên tục: trong trường hợp vùng nhớ cho chương trình đang bị phân mảnh, chương trình sẽ báo lỗi khi chúng ta khai báo hoặc cấp phát cho mảng với kích thước lớn vì lý do: không đủ vùng nhớ liên tục cho mảng (mặc dù tổng dung lượng vùng nhớ phân mảnh là đủ).
- Việc chèn và xóa phần tử của mảng mất nhiều thời gian: vì vùng nhớ cấp phát cho mảng được sắp xếp liên tục nên việc chèn một phần tử mới vào hoặc xóa phần tử trong mảng trở lên khó khăn. Ví dụ: cho mảng A[100], chúng ta muốn chèn thêm phần tử mới vào vị trí i, tất cả các phần tử thứ i trở đi phải dịch sang vị trí kế tiếp để chèn giá trị vào vị trí thứ i. Việc xóa phần tử trong mảng cũng tương tự như vậy, dịch tất cả các phần tử từ vị trí thứ i+1 sang vị trí liền trước nó.

11. Sự khác nhau giữa cấp phát tĩnh và cấp phát động?

Cấp phát bộ nhớ tĩnh

Khi khai báo các biến (int, float), mảng, chuỗi kí tự, (int a[20], char s[30]).. Trình biên dịch cấp phát vùng nhớ (static memory) cho các biến này. Vì vùng nhớ dành cho các (biến, mảng, chuỗi) là static memory nên việc gán con trỏ tới vùng nhớ tĩnh được gọi là cấp phát bộ nhớ tĩnh. Vùng nhớ được cấp phát trong lúc biên dịch.

Cấp phát bộ nhớ động

Sử dụng hàm malloc(), calloc() để cấp phát bộ nhớ động. Giá trị trả về của malloc(), calloc() là con trỏ trỏ tới vùng nhớ được cấp phát. Vùng nhớ được cấp phát trong lúc chạy chương trình (runtime)

12. Sự khác nhau giữa const char* s và char* const s?

1. Hằng con trỏ – Constant pointer

Khai báo:

```
<Kiểu dữ liệu> * const <Tên con trỏ> = <Địa chỉ khởi tạo>;
```

Đặc điểm:
– Cần gán ngay giá trị địa chỉ khởi tạo cho hằng con trỏ tại câu lệnh khai báo ban đầu.
– Không thể thay đổi địa chỉ đã được khởi gán cho hằng con trỏ (sẽ gây ra lỗi).
– Có thể thay đổi giá trị tại địa chỉ đã khởi gián ban đầu.

2. Con trỏ hằng – Pointer to constant

Khai báo:

```
const <Kiểu dữ liệu> * <Tên con trỏ>;
```

Đặc điểm:
– Không được phép dùng trực tiếp con trỏ hằng để thay đổi giá trị tại vùng nhớ mà con trỏ hằng đang trỏ đến.
– Con trỏ hằng có thể thể thay đổi địa chỉ trỏ tới (hay nói cách khác: nó có thể trỏ đến các ô nhớ khác nhau).

13. Sự khác nhau giữa mảng và chuỗi?

- Mảng: được khai báo <kiểu dữ liệu> tên mảng[<kích thước>]. Trong đó, kiểu dữ liệu có thể là kiểu dữ liệu nguyên thủy: int, float, double, long, char,… hoặc struct, class,…
- Chuỗi là kiểu dữ liệu đặc biệt của mảng. Chuỗi là mảng kí tự (kiểu char) + kí tự NULL ('\0') ở phần tử cuối của chuỗi. Kí tự NULL sẽ được trình biên dịch tự động thêm vào trong quá trình biên dịch.

14. Con trỏ void là gì?

Đặc điểm của con trỏ void:
– Nó có thể lưu trữ địa chỉ của mọi kiểu biến dữ liệu

Hạn chế:
– Ta không thể sử dụng trực tiếp dữ liệu mà con trỏ void trỏ tới bằng toán tử (*), mà cần biến đổi con trỏ (void*) sang thành kiểu dữ liệu tương ứng.

15. Sự khác nhau giữa malloc() và calloc()?

Cả 2 hàm malloc() và calloc() đều được sử dụng để cấp phát vùng nhớ động cho chương trình. Nếu cấp phát vùng nhớ thành công, hàm trả về con trỏ trỏ tới vùng nhớ được cấp phát. Hàm trả về NULL nếu không đủ vùng nhớ.

Hàm malloc() và calloc() trả về NULL trong các trường hợp sau:

- Kích thước vùng nhớ cần cấp phát vượt quá kích thước vật lý của hệ thống
- Tại thời điểm gọi hàm malloc() và calloc(), tạm thời không đủ vùng nhớ để cấp phát. Nhưng application có thể gọi lại nhiều lần malloc() và calloc() để cấp phát vùng nhớ thành công.

Điểm khác nhau giữa hàm malloc() và calloc()

malloc()	calloc()
malloc viết tắt của memory allocation	calloc viết tắt của contiguous allocation
malloc nhận 1 tham số truyền vào là số byte của vùng nhớ cần cấp phát	calloc nhận 2 tham số truyền vào là số block và kích thước mỗi block (byte)
void *malloc(size_t n); Hàm trả về con trỏ trỏ tới vùng nhớ nếu cấp phát thành công, trả về NULL nếu cấp phát fail	void *calloc(size_t n, size_t size); Hàm trả về con trỏ trỏ tới vùng nhớ được cấp phát và vùng nhớ được khởi tạo bằng giá trị 0. Trả về NULL nếu cấp phát fail
Hàm malloc() nhanh hơn hàm calloc()	Hàm calloc() tốn thêm thời gian khởi tạo vùng nhớ. Tuy nhiên, sự khác biệt này không đáng kể.

16. Sự khác nhau giữa việc #include < xxx.h > và #include “xxx.h”?

Lệnh #include <filename> và #include “filename” là tương đương nhau, để include header file của thư viện ngôn ngữ C hoặc header file do lập trình viên tự định nghĩa.

Sự khác nhau giữa #include <filename> and #include “filename” nằm ở khâu tìm kiếm file header của tiền xử lý trước quá trình biên dịch.

- #include <filename>: tiền xử lý (pre-processor) sẽ chỉ tìm kiếm file header (.h) trong thư mục chứa file header của thư viện ngôn ngữ C (thường là thư mục trong bộ cài IDE).
- #include “filename”: Trước tiên, tiền xử lý (pre-processor) tìm kiếm file header(.h) trong thư mục đặt project C/C++. Nếu không tìm thấy, tiền xử lý tìm kiếm file header (.h) trong thư mục chứa file header của thư viện ngôn ngữ C (thường là thư mục trong bộ cài IDE).

17. Từ khóa auto có ý nghĩa gì?

Biến auto là biến local, thời gian tồn tại trong hàm và giá trị của biến auto không được lưu giữa các lần gọi hàm.

18. Sử dụng macro hay function tốt hơn?

Macro	Hàm
Việc định nghĩa macro khó hơn định nghĩa hàm. Nếu không chú ý, chúng ta dễ bị side effect.	Việc định nghĩa đơn giản hơn
không thể debug tìm lỗi của macro trong thời gian thực thi.	Debug đơn giản, dễ bắt lỗi
Macro không cần quan tâm kiểu dữ liệu của tham số và kiểu trả về. Như ví dụ trên, chúng ta có thể truyền kiểu int, float.	Phải chỉ rõ kiểu dữ liệu của tham số và giá trị trả về
Macro tạo ra các inline code, thời gian xử lí inline code ngắn hơn thời gian gọi hàm	Chương trình mất time dịch từ vùng nhớ hàm được lưu trữ sang vùng nhớ goi hàm.
Giả sử macro được gọi 20 lần trong chương trình, 20 dòng code sẽ được chèn vào chương trình trong quá trình tiền xử lí. Điều này làm cho kích thước của chương trình (.EXE, .DLL, LIB,) phình to ra.	Giả sử 1 hàm được gọi 20 lần, sẽ chỉ có 1 bản copy của hàm trong chương trình. Kích thước chương trình nhỏ hơn sử dụng macro.

Tùy thuộc vào tiêu chí thời gian thực thi hay kích thước chương trình, mà lập trình viên quyết định chọn macro hay hàm trong chương trình của mình. Đối với khối chức năng đơn giản ít dòng code, nên sử dụng macro.

19. Sự khác nhau giữa kích thước của biến con trỏ int* và biến con trỏ char* ?

- Kích thước biến con trỏ không phụ thuộc vào kiểu dữ liệu con trỏ trỏ tới vì con trỏ lưu địa chỉ của biến. Địa chỉ của biến là kiểu int.
- Do vậy, kích thước là 2 bytes cho OS 16 bits, 4 bytes cho OS 32 bits, 8 bytes cho OS 64 bits.
- Toán tử sizeof() để tính kích thước của biến (con trỏ, mảng, struct,..)

20. Sự khác nhau giữa struct và union?

struct	**union**
Size của struct ít nhất bằng tổng size của các thành phần của struct. Mình sử dụng từ “ít nhất” là vì size struct còn phụ thuộc vào alignment struct.	Size của union bằng size của thành phần có size lớn nhất trong union.

sizeof(A) = 4 + 10 + 4 = 18 (trong trường hợp struct alignment member = 1 byte) `struct A { int x; char s[10]; float f; };`	sizeof(A) = 10 (kích thước của thành phần lớn nhất trong union) `union A { int x; char s[10]; float f; };`
Tại cùng 1 thời điểm run-time, có thể truy cập vào tất cả các thành phần của struct	Tại cùng 1 thời điểm run-time, chỉ có thể truy cập 1 thành phần của union

21. Sự khác nhau giữa #define và const?

Sử dụng #define để định nghĩa hằng số như: kích thước mảng, không thể sử dụng const để định nghĩa hằng số như: kích thước mảng.

22. Sự khác nhau giữa bộ nhớ stack và heap?

Bộ nhớ stack và heap đều được lưu trên RAM của PC. Dưới đây là một số điểm khác nhau của stack và heap.

Stack	**Heap**
▪ Vùng nhớ được cấp phát khi chương trình được biên dịch.	▪ Vùng nhớ được cấp phát khi chạy chương trình (run-time).
▪ Vùng nhớ stack được sử dụng cho việc thực thi thread. Khi gọi hàm, các biến cục bộ của hàm được lưu trữ vào block của stack (theo kiểu LIFO). Cho đến khi hàm trả về giá trị, block này sẽ được xóa tự động. Hay nói cách khác, các biến cục bộ được lưu trữ ở vùng nhớ stack và tự động được giải phóng khi kết thúc hàm.	▪ Vùng nhớ heap được dùng cho cấp phát bộ nhớ động (malloc( ), new( )). Vùng nhớ được cấp phát tồn tại đến khi lập trình viên giải phóng vùng nhớ bằng lệnh free( ) hoặc delete.
▪ Kích thước vùng nhớ stack được fix cố định. Chúng ta không thể tăng hoặc giảm kích thước vùng nhớ stack. Nếu không đủ vùng nhớ stack, gây ra stack overflow. Hiện tượng này xảy ra khi nhiều hàm lồng	▪ Khi kích thước vùng nhớ heap không đủ cho yêu cầu malloc( ), new. Hệ điều hành sẽ có cơ chế tăng kích thước vùng nhớ heap.

nhau hoặc đệ quy nhiều lần dẫn đến không đủ vùng nhớ.	

23. Sự khác nhau giữa hàm strdup() và strcpy()?

- Hàm strcpy(char* s1, char* s2) copy nội dung vùng nhớ s2 và vùng nhớ s1. Vùng nhớ s1 phải được cấp phát tĩnh hoặc cấp phát động trước đó (malloc( ), new( )). Kích thước vùng nhớ s1 phải đủ để chứa chuỗi s2. Nếu không đủ vùng nhớ, gây ra buffer overrun.
- Hàm strdup(const char* s) tạo ra vùng nhớ mới (gọi hàm malloc( ) để cấp phát vùng nhớ) để lưu chuỗi s. Sau đó, trả về con trỏ trỏ tới vùng nhớ được cấp phát. Nếu cấp phát fail, hàm trả về NULL. Chú ý: phải free vùng nhớ trả về bởi hàm strdup.

22. Sự khác nhau giữa mảng và link list?

Mảng	**Danh sách liên kết**
Vùng nhớ của các phần tử trong mảng được sắp xếp liên tục nhau	Vùng nhớ của các phần tử trong danh sách liên kết được sắp xếp tùy ý (do hệ điều hành). Các phần tử lưu 1 con trỏ trỏ tới phần tử tiếp theo.
Truy cập tới phần tử trong mảng là truy cập trực tiếp dựa vào chỉ số (ví dụ: a[0], a[1], a[2],…, a[n])	Cần phải duyệt tuần tự khi muốn truy cập tới phần tử trong danh sách liên kết.
▪ Kích thước của mảng là hằng số, không thay đổi khi chạy chương trình ▪ Sử dụng mảng không tối ưu được bộ nhớ. Có thể thừa hoặc thiếu bộ nhớ khi xóa hoặc chèn phần tử vào mảng	▪ Kích thước của danh sách liên kết có thể thay đổi khi chạy chương trình. ▪ Sử dụng danh sách liên kết tối ưu được bộ nhớ. Vùng nhớ được cấp phát thêm khi cần chèn thêm phần tử mới, vùng nhớ được free khi xóa phần tử.

define a linked list node:

```
typedef struct node {

   int val;

   struct node * next;
} node_t;
```

Let's create a local variable which points to the first item of the list

```
node_t * head = NULL;

head = malloc(sizeof(node_t));

if (head == NULL) {

    return 1;

}

head->val = 1;

head->next = NULL;
```

23. Mảng con trỏ là gì? Ứng dụng của mảng con trỏ?

Chúng ta có thể sử dụng mảng con trỏ để trỏ tới các phần tử trong mảng dữ liệu (string, struct,) tương ứng. Việc sắp xếp các phần tử trong mảng dữ liệu (string, struct,) dựa vào việc hoán vị các con trỏ. Mảng con trỏ sẽ lưu kết quả sắp xếp mảng (các phần tử mảng dữ liệu thực tế không được hoán vị).

Việc hoán vị con trỏ tăng performance của chương trình so với cách truyền thống.

24. Con trỏ hàm là gì? Ứng dụng con trỏ hàm?

Con trỏ hàm lưu địa chỉ của hàm.

Ứng dụng con trỏ hàm: thay thế cho lệnh switch case.

Function pointers can be useful when you want to create **callback mechanism**, and need to pass address of an function to another function.
They can also be useful when you want to store an array of functions, to call dynamically for example.

25. Con trỏ gần (near pointer) là gì?

26. Con trỏ xa (far pointer) là gì?

27. Alignment struct là gì?

Data alignment là cách mà trình biên dịch sắp xếp dữ liệu trong memory. Việc sắp xếp dữ liệu liên quan tới 2 khái niệm: data alignment và data padding.

Đối với những PC hiện nay, việc đọc/ghi dữ liệu trong bộ nhớ được với kích thước của WORD (OS 32 bit: kích thước WORD: 4 byte) hoặc lớn hơn. Data alignment là cách sắp xếp dữ liệu sao cho kích thước bằng bội số của kích thước 1 word = 4k (byte). (k = 0, 1, 2,). Để alignment data, đôi khi chúng ta cần phải add thêm các byte giả (dummy) vào vị trí thích hợp để đảm bảo data alignment, được gọi là padding data.

Data Alignment làm tăng performance do việc đọc/ghi thao tác trên block data có kích thước bằng bội số của WORD.

29. Mối liên hệ giữa con trỏ và mảng 1 chiều?

Muốn lấy giá trị của phần tử trong mảng 1 chiều ta có thể dùng: ***(A+i)**, i là chỉ số mảng.

30. Mối liên hệ giữa con trỏ và mảng 2 chiều?

Để lấy giá trị của các phần tử mảng 2 chiều ta dùng biểu thức sau: *(*(A+i)+j) và để lấy địa chỉ của phần tử mảng 2 chiều ta dùng biểu thức sau: *(A+i)+j. (trong đó i,j là chỉ số hàng và cột).

31. Macro là gì? Ưu và nhược điểm khi sử dụng macro?

- Marco là 1 tên bất kì (do lập trình viên đặt tên) trỏ tới 1 khối lệnh thực hiện một chức năng nào đó.
- Trong quá trình tiền xử lí (pre-processor), các macro được sử dụng trong chương trình được thay thế bởi các khối câu lệnh tương ứng.
- Định nghĩa macro bằng lệnh #define

32. Sự khác nhau giữa ++x và x++ trong C/C++?

Toán tử tăng trước ++x: tăng giá trị x trước khi thực hiện các phép toán khác trong cùng 1 câu lệnh.

Toán tử tăng sau x++: tăng giá trị x sau khi thực hiện các phép toán khác trong cùng 1 câu lệnh.

33. Tại sao keyword default và break được sử dụng trong lệnh switch case?

Từ khóa default đóng vai trò xử lý những case ngoại lệ khi biểu thức điều kiện không thỏa mãn bất cứ case nào.

Khi câu lệnh trong mỗi case được thực hiên xong, lệnh break giúp thoát switch case. Điều này giúp các câu lệnh của các case bên dưới không được thực hiện.

34. Nên sử dụng switch case hay if else?

- Nên sử dụng switch case trong bài toán mul-ti choice, biểu thức điều kiện tính toán phức tạp nhưng phải có giá trị nguyên.
- Nên sử dụng if else trong bài toán ít trường hợp, các biểu thức điều kiện đơn giản.

35. Sử dụng hàm nào an toàn hơn trong 2 hàm fgets() và gets()?

36. Những ưu điểm khi sử dụng enum thay vì sử dụng #define?

enums được ưa thích vì chúng an toàn và dễ phát hiện hơn.

37. Từ khóa #prama là gì?

Bộ tiền xử lý trong C ở đây không phải là một phần của bộ biên dịch, nhưng có những bước riêng rẽ trong quá trình biên dịch. Theo cách hiểu cơ bản nhất, bộ tiền xử lý trong ngôn ngữ C là các công cụ thay thế văn bản và hướng dẫn trình biên dịch không yêu cầu tiền xử lý trước khi được biên dịch. Chúng tôi hướng đến bộ tiền xử lý C như CPP.

Tất cả các lệnh tiền xử lý bắt đầu với ký thự #. Nó ít nhất không phải là ký tự trắng, để dễ dàng đọc. Dưới đây là danh sách các thẻ tiền xử lý quan trọng.

Directive	Miêu tả
#define	Thay thể cho bộ tiền xử lý macro
#include	Chèn một header đặc biệt từ file khác
#undef	Không định nghĩa một macro tiền xử lý
#ifdef	Trả về giá trị true nếu macro này được định nghĩa

#ifndef	Trả về giá trị true nếu macro này không được định nghĩa
#if	Kiểm tra nếu điều kiện biên dịch là đúng
#else	Phần thay thế cho #if
#elif	#else một #if trong một lệnh
#endif	Kết thúc điều kiện tiền xử lý
#error	In thông báo lỗi trên stderr
#pragma	Thông báo các lệnh đặc biệt đến bộ biên dịch, sử dụng một phương thức được tiêu chuẩn hóa

38. Viết con trỏ tương ứng với giá trị mảng a[i][j][k][m]?

(*(*(*(a+i)+j)+k)+m)

39. Sự khác nhau giữa const và volatite?

Việc khai báo biến có thuộc tính volatile là rất cần thiết nhằm tránh các lỗi phát sinh ngoài ý muốn khi tính năng optimization của compiler được bật. Volatile để chỉ một biến có thể bị thay đổi giá trị một cách "bất thường". Có nghĩa là giá trị của biến có thể bị thay đổi mà không được báo trước. Trong thực tế, có 3 loại biến mà giá trị có thể bị thay đổi như vậy:

• Memory-mapped peripheral registers (thanh ghi ngoại vi có ánh xạ đến ô nhớ)

• Biến toàn cục được truy xuất từ các tiến trình con xử lý ngắt (interrupt service routine)

• Biến toàn cục được truy xuất từ nhiều tác vụ trong một ứng dụng đa luồng.

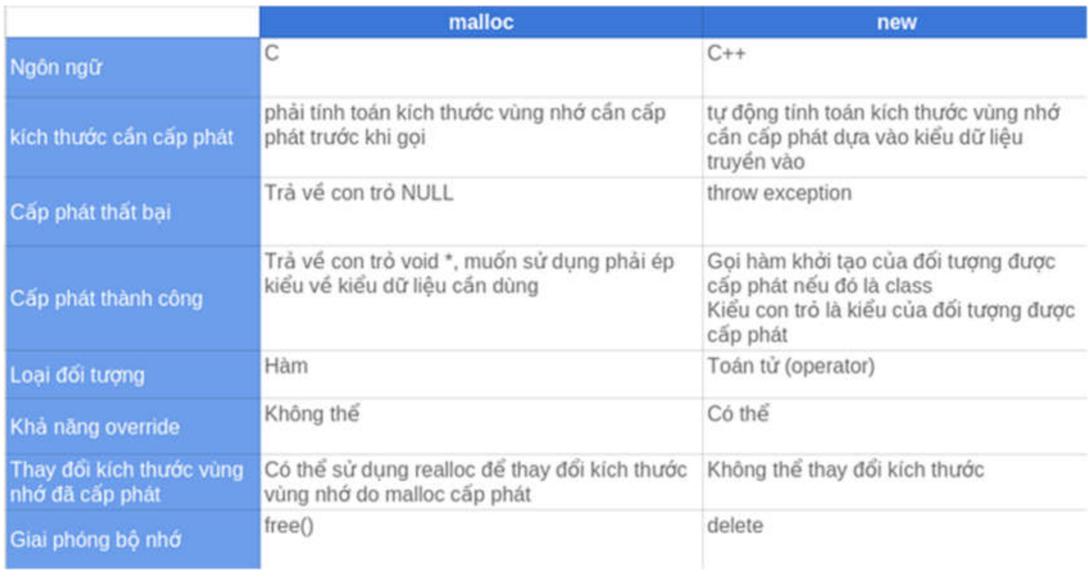
40. Tại sao nên gán NULL cho con trỏ sau khi giải phóng vùng nhớ?

Bất cứ khi nào sử dụng vùng nhớ thông qua con trỏ, chúng ta cũng nên kiểm tra xem con trỏ có khác NULL hay không. Nếu con trỏ khác NULL thì chúng ta hiểu rằng vùng nhớ đó vẫn chưa được giải phóng.

41. Sự khác nhau giữa linker và linkage?

Linker converts an object code into an executable code by linking together the necessary built in functions. The form and place of declaration where the variable is declared in a program determine the linkage of variable.

42. Sự khác nhau giữa malloc() và new()?



	malloc	new
Ngôn ngữ	C	C++
kích thước cần cấp phát	phải tính toán kích thước vùng nhớ cần cấp phát trước khi gọi	tự động tính toán kích thước vùng nhớ cần cấp phát dựa vào kiểu dữ liệu truyền vào
Cấp phát thất bại	Trả về con trỏ NULL	throw exception
Cấp phát thành công	Trả về con trỏ void *, muốn sử dụng phải ép kiểu về kiểu dữ liệu cần dùng	Gọi hàm khởi tạo của đối tượng được cấp phát nếu đó là class Kiểu con trỏ là kiểu của đối tượng được cấp phát
Loại đối tượng	Hàm	Toán tử (operator)
Khả năng override	Không thể	Có thể
Thay đổi kích thước vùng nhớ đã cấp phát	Có thể sử dụng realloc để thay đổi kích thước vùng nhớ do malloc cấp phát	Không thể thay đổi kích thước
Giai phóng bộ nhớ	free()	delete

43. Liệt kê và giải thích các thuộc tính của OOP?

Tính đóng gói (encapsulation)

Gói dữ liệu (data, ~ biến, trạng thái) và mã chương trình (code, ~ phương thức) thành một cục gọi là lớp (class) để dễ quản lí. Trong cục này thường data rất rối rắm, không tiện cho người không có trách nhiệm truy cập trực tiếp, nên thường ta sẽ che dấu data đi, chỉ để lòi phương thức ra ngoài.

Tính kế thừa (inheritance)

Kế thừa là cách tạo lớp mới từ các lớp đã được định nghĩa từ trước

Lớp cha có thể chia sẻ dữ liệu và phương thức cho các lớp con, các lớp con khỏi phải định nghĩa lại những logic chung, giúp chương trình ngắn gọn. Nếu lớp cha là interface, thì lớp con sẽ di truyền những contract trừu tượng từ lớp cha.

Tính đa hình (polymorphism)

Tính đa hình được thể hiện qua việc viết lại các method(hàm) từ class cha thông qua class kế thừa nó hoặc việc triển khai các interface.

Tính trừu tượng

Đây là khả năng của chương trình bỏ qua hay không chú ý đến một số khía cạnh của thông tin mà nó đang trực tiếp làm việc lên, nghĩa là nó có khả năng tập trung vào những cốt lõi cần thiết **(chỉ khai báo)**. Mỗi đối tượng phục vụ như là một "động tử" có thể hoàn tất các công việc một cách nội bộ, báo cáo, thay đổi trạng thái của nó và liên lạc với các đối tượng khác mà không cần cho biết làm cách nào đối tượng tiến hành được các thao tác **(sự override từ lớp con).**

44. Hàm inline() là gì?

Khi được nạp vào ram, mỗi hàm sẽ có địa chỉ nhất định, khi gọi thì cpu sẽ jump tới địa chỉ đó. Viết inline thì compiler sẽ chèn luôn code của hàm đó vào, thay vì chèn địa chỉ, cpu chỉ chạy một mạch mà thôi, vậy nên sẽ nhanh hơn

Chỉ nên dùng cho getter và setter vì nó nhẹ mà hay được gọi.

45. Hàm tạo, hàm hủy là gì?

Hàm tạo – Constructor

Constructor là 1 hàm đặc biệt của class, nó được tự động gọi khi 1 đối tượng của class được khởi tạo. Hàm **Constructor** thường để khởi tạo giá trị ban đầu cho các biến thành viên hoặc setup những vấn đề cần thiết cho class.

Hàm **Constructor** không giống các hàm thành viên khác, phải theo quy tắc.

- Tên của **constructor** phải giống tên của class.
- **Constructor** không có kiểu giả trị trả về (kể cả void)
- Một class có thể có nhiều **constructor**.

```
class Date

{

private:

   int day;

   int month;

   int year;
public:

   //Hàm khởi tạo không có tham số
   Date()

   {

      day = 1;

      month = 1;

      year = 1900;

   }

};
```

Hàm hủy – Destructor

Hàm hủy là hàm đặc biêt của class được tự động gọi khi ta hủy đổi tượng. Nếu như hàm tạo khởi tạo các giá trị cho class thì hàm hủy xóa các dữ liệu của class.

Hàm hủy được xây dựng tuân theo quy tắc:

- Tên hàm trùng với tên class, trước tên hàm có dấu '~'.
- Hàm không có dữ liệu trả về.
- Hàm hủy không có tham số.

```
#include <iostream>

using namespace std;

class Test

{

private:

   int* arr;

   int nId;

public:

   Test(int id)

   {
    //Cap phat bo nho cho mang trong ham tao

    arr = new int[10];

    nId = id;

    cout << "Day la ham tao" << nId << endl;

  }

  ~Test()

  {

    cout << "Day la ham huy" << nId << endl;

    //Giai phong bo nho trong ham huy

    delete[] arr;

  }

};

int main()

{

  //Ham tao duoc goi
  Test tes1(1);

  //Toan tu new goi ham tao cho ptrTest
  Test* ptrTest = new Test(2);

  //Toan tu delete goi ham huy cho ptrTest
  delete ptrTest;

  return 0;

}
```

46. Hàm hủy ảo (virtual destructor) là gì?

47. Hàm tạo copy là gì? Khi nào hàm tạo copy được gọi?

1. **Question 1. What Is Autosar?**
   Answer :
   AUTOSAR (Automotive Open System Architecture) is a standardization initiative of leading automotive manufacturers and suppliers that was founded in autumn of 2003. The goal is the development of a reference architecture for ECU software that can manage the growing complexity of ECUs in modern vehicles.
2. **Question 2. What Is Swc?**
   Answer :
   An SWC file is a package of precompiled Flash symbols and ActionScript code that allows a Flash or Flex developer to distribute classes and assets, or to avoid recompiling symbols and code that will not change. ... They are sometimes referred to as class libraries and cannot be directly executed by the Flash Player.
3. **Question 3. What Is Can And Its Uses?**
   Answer :
   1. CAN is a multi-master broadcast serial bus standard for connecting electronic control unit (ECUs).

   2. Controller–area network (CAN or CAN-bus) is a vehicle bus standard designed to allow microcontrollers a devices to communicate with each other within a vehicle without a host computer.

   3. CAN is a message-based protocol, designed specifically for automotive applications but now also used in other areas such as industrial automation and medical equipment.

   4. The Controller Area Network (CAN) bus is a serial asynchronous bus used in instrumentation applications for industries such as automobiles.

   USES:
   - More reliably, e.g., fewer plug-in connectors that might cause errors.
   - Wiring less complicated, more economic.
   - Easy to implement, changes, too.
   - Additional elements (e.g., control units) are easy to integrate.
   - Installation place exchangeable without electric problems.
   - Wire may be diagnosed.
4. **Question 4. What Are The Can Frame Works?**
   Answer :
   - SOF – 1 Dominant
   - Arbitration Field – 11 bit Identifier, 1 bit RTR (or) 11 bit, 1SRR, 1IDE, 18 bit, 1RTR
   - Control Field – IDE, r0, 4 bits (DLC)
   - Data Field – (0-8) Bytes
   - CRC Field – 15 bits, Delimiter (1 bit recessive)
   - ACK Field – 1 bit, Delimiter (1 bit recessive)
   - EOF – 7 bits recessive
   - IFS – 3 bits recessive
   - Types of frames – Data, remote, Error frame and Overload frame
   - Types of errors – ACK error, Bit error, Stuff error, Form error, CRC error
   - Error frame – 0-12 superposition flags, 8 recessive (Delimiter)

    - Overload frame – 0-12 superposition flags, 8 recessive (Delimiter)

5. **Question 5. Why Can Is Having 120 Ohms At Each End?**
   Answer :
   To minimize the reflection reference, to reduce noise. To ensure that reflection does not cause communication failure, the transmission line must be terminated.
6. **Question 6. Why Can Is Message Oriented Protocol?**
   Answer :
   CAN protocol is a message-based protocol, not an address based protocol. This means that messages are not transmitted from one node to another node based on addresses. Embedded in the CAN message itself is the priority and the contents of the data being transmitted. All nodes in the system receive every message transmitted on the bus (and will acknowledge if the message was properly received). It is up to each node in the system to decide whether the message received should be immediately discarded or kept to be processed.

   A single message can be destined for one particular node to receive, or many nodes based on the way the network and system are designed. For example, an automotive airbag sensor can be connected via CAN to a safety system router node only. This router node takes in other safety system information and routes it to all other nodes on the safety system network. Then all the other nodes on the safety system network can receive the latest airbag sensor information from the router at the same time, acknowledge if the message was received properly, and decide whether to utilize this information or discard it.
7. **Question 7. Can Logic What It Follows?**
   Answer :
   Wired AND logic.
8. **Question 8. What Is Can Arbitration?**
   Answer :
   CAN Arbitration is nothing but the node trying to take control on the CAN bus.
9. **Question 9. How Can Will Follow The Arbitration?**
   Answer :
   CSMA/CD + AMP (Arbitration on Message Priority)

   Two bus nodes have got a transmission request. The bus access method is CSMA/CD+AMP (Carrier Sense Multiple Access with Collision Detection and Arbitration on Message Priority). According to this algorithm both network nodes wait until the bus is free (Carrier Sense). In that case the bus is free both nodes transmit their dominant start bit (Multiple Access). Every bus node reads back bit by bit from the bus during the complete message and compares the transmitted value with the received value.

   As long as the bits are identical from both transmitters nothing happens. The first time there was a difference – in this example the 7th bit of the message – the arbitration process takes place: Node A transmits a dominant level, node B transmits a recessive level. The recessive level will be overwritten by the dominant level.

   This is detected by node B because the transmitted value is not equal to the received value (Collision Detection). At this point of time node B has lost the arbitration, stops the transmission of any further bit immediately and switches to receive mode, because the message that has won the arbitration must possibly be processed by this node (Arbitration on Message Priority)

For example, consider three CAN devices each trying to transmit messages:

- Device 1 – address 433 (decimal or 00110110001 binary)
- Device 2 – address 154 (00010011010)
- Device 3 – address 187 (00010111011)

Assuming all three see the bus is idle and begin transmitting at the same time, this is how the arbitration works out. All three devices will drive the bus to a dominant state for the start-of-frame (SOF) and the two most significant bits of each message identifier.

Each device will monitor the bus and determine success. When they write bit 8 of the message ID, the device writing message ID 433 will notice that the bus is in the dominant state when it was trying to let it be recessive, so it will assume a collision and give up for now. The remaining devices will continue writing bits until bit 5, then the device writing message ID 187 will notice a collision and abort transmission. This leaves the device writing message ID 154 remaining.

It will continue writing bits on the bus until complete or an error is detected. Notice that this method of arbitration will always cause the lowest numerical value message ID to have priority. This same method of bit-wise arbitration and prioritization applies to the 18-bit extension in the extended format as well.

10. **Question 10. What Is The Speed Of Can?**
    Answer :
    40m @1Mbps and if the cable length increases will decrease the speed, due to RLC on the cable.
11. **Question 11. If Master Sends 764 And Slave Sends 744 Which Will Get The Arbitration?**
    Answer :
    Starts from MSB, first nibble is same, Master sends 7, slaves also sends 7 the message with more dominant bits will gain the arbitration, lowest the message identifier higher the priority.
12. **Question 12. Standard Can And Extended Can Difference?**
    Answer :
    Number of identifiers can be accommodated for standard frame are 2power11.

    Number of identifiers more compare to base frame, for extended frame are 2power29.

    - IDE bit – 1 for extended frame.
    - IDE bit – 0 for Standard frame.
13. **Question 13. What Is Bit Stuffing?**
    Answer :
    CAN uses a Non-Return-to-Zero protocol, NRZ-5, with bit stuffing. The idea behind bit stuffing is to provide a guaranteed edge on the signal so the receiver can resynchronize with the transmitter before minor clock discrepancies between the two nodes can cause a problem. With NRZ-5 the transmitter transmits at most five

consecutive bits with the same value. After five bits with the same value (zero or one), the transmitter inserts a stuff bit with the opposite state.

14. **Question 14. What Is The Use Of Bit Stuffing?**
    Answer :
    Long NRZ messages cause problems in receivers:
    • Clock drift means that if there are no edges, receivers lose track of bits.
    • Periodic edges allow receiver to resynchronize to sender clock.
15. **Question 15. What Are The Functions Of Can Transceiver?**
    Answer :
    The transceiver provides differential transmit capability to the bus and differential receive capability to the CAN controller. Transceiver provides an advanced interface between the protocol controller and the physical bus in a Controller Area Network (CAN) node.

    Typically, each node in a CAN system must have a device to convert the digital signals generated by a CAN controller to signals suitable for transmission over the bus cabling (differential output). It also provides a buffer between the CAN controller and the high-voltage spikes that can be generated on the CAN bus by outside sources (EMI, ESD, electrical transients, etc.).

    The can transceiver is a device which detects the signal levels that are used on the CAN bus to the logical signal levels recognized by a microcontroller.
16. **Question 16. Functionality Of Data Link Layer In Can?**
    Answer :
    LLC (Logical Link Control): Overload control, notification, Message filtering and Recovery management functions.
    MAC (Medium Access Control): Encapsulation/ de-capsulation, error detection and control, stuffing and de-stuffing and serialization/de-serialization.
17. **Question 17. What Is Meant By Synchronization?**
    Answer :
    Synchronization is timekeeping which requires the coordination of events to operate a system in unison.
18. **Question 18. What Is Meant By Hard Synchronization And Soft Synchronization?**
    Answer :
    Hard Synchronization to be performed at every edge from recessive-to-dominant edge during Bus Idle. Additionally, Hard Synchronization is required for each received SOF bit. An SOF bit can be received both during Bus Idle, and also during Suspend Transmission and at the end of Interframe Space. Any node disables Hard Synchronization if it samples an edge from recessive to dominant or if it starts to send the dominant SOF bit.

    Two types of synchronization are supported:
    - Hard synchronization is done with a falling edge on the bus while the bus is idle, which is interpreted as a Start of frame (SOF). It restarts the internal Bit Time Logic.
    - Soft synchronization is used to lengthen or shorten a bit time while a CAN frame is received.

19. **Question 19. What Is The Difference Between Function And Physical Addressing?**
    Answer :
    Functional addressing is an addressing scheme that labels messages based upon their operation code or content. Physical addressing is an addressing scheme that labels messages based upon the physical address location of their source and/or destination(s).
20. **Question 20. What Is Kwp2000?**
    Answer :
    KWP 2000(ISO14230) is a Diagnostic communications standard. Specifies possible system configurations using the K & L lines. As 9141-2 but limited to the physical characteristics. Specifies possible system configurations using the K & L lines.

    5 Baud wake up as 9141- 2
    New fast initialisation method
21. **Question 21. What Is Obdii?**
    Answer :
    On-Board Diagnostics in an automotive context is a generic term referring to a vehicle's self-diagnostic and reporting capability.
22. **Question 22. Why Diagnostic Standards?**
    Answer :
    As systems got more complex the link between cause and symptom became less obvious. This meant that electronic systems had to have some level of self diagnosis and to communicate to the outside world. Initially many systems used their own protocols which meant that garages had to have a large number of tools – even to diagnose a single vehicle.
23. **Question 23. What Is Meant By Verification And Validation?**
    Answer :
    Verification and Validation (V&V) is the process of checking that a software system meets specifications and that it fulfills its intended purpose. It is normally part of the software testing process of a project.

    According to the Capability Maturity Model (CMMI-SW v1.1),

    Verification: The process of evaluating software to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.
    Validation: The process of evaluating software during or at the end of the development process to determine whether it satisfies specified requirements.
    Verification shows conformance with specification; validation shows that the program meets the customer's needs.
24. **Question 24. Can You Have Two Transmitters Using The Same Exact Header Field?**
    Answer :
    No that would produce a bus conflict.

    • Unless you have middleware that ensures only one node can transmit at a time.

    For example: use a low priority message as a token to emulate token-passing.

25. **Question 25. Formula For Baudrate Calculation?**
    Answer :
    The baud rate is calculated as:
    baud rate (bits per second) = 18.432 x 10^6 / BRP / (1 + TSEG1 + TSEG2)
26. **Question 26. What Happen When Two Can Nodes Are Sending Same Identifier At A Same Time?**
    Answer :
    Two nodes on the network are not allowed to send messages with the same id. If two nodes try to send a message with the same id at the same time arbitration will not work. Instead, one of the transmitting nodes will detect that his message is distorted outside of the arbitration field.

    The nodes will then use the error handling of CAN, which in this case ultimately will lead to one of the transmitting node being switched off (bus-off mode).

1) What is the use of volatile keyword?

The C's volatile keyword is a qualifier that tells the compiler not to optimize when applied to a variable. By declaring a variable volatile, we can tell the compiler that the value of the variable may change any moment from outside of the scope of the program. A variable should be declared volatile whenever its value could change unexpectedly and beyond the comprehension of the compiler.

In those cases it is required not to optimize the code, doing so may lead to erroneous result and load the variable every time it is used in the program. Volatile keyword is useful for memory-mapped peripheral registers, global variables modified by an interrupt service routine, global variables accessed by multiple tasks within a multi-threaded application.

2) Can a variable be both const and volatile?

The const keyword make sure that the value of the variable declared as const can't be changed. This statement holds true in the scope of the program. The value can still be changed by outside intervention. So, the use of const with volatile keyword makes perfect sense.

3) Can a pointer be volatile?

If we see the declaration volatile int *p, it means that the pointer itself is not volatile and points to an integer that is volatile. This is to inform the compiler that pointer p is pointing to an integer and the value of that integer may change unexpectedly even if there is no code indicating so in the program.

4) What is size of character, integer, integer pointer, character pointer?

- The size of character is 1 byte.
- Size of integer is 4 bytes.
- Size of integer pointer and character is 8 bytes on 64 bits machine and 4 bytes on 32 bits machine.

5) What is NULL pointer and what is its use?

The NULL is a macro defined in C. Null pointer actually means a pointer that does not point to any valid location. We define a pointer to be null when we want to make sure that the pointer does not point to any valid location and not to use that pointer to change anything. If we don't use null pointer, then we can't verify whether this pointer points to any valid location or not.

6) What is void pointer and what is its use?

The void pointer means that it points to a variable that can be of any type. Other pointers points to a specific type of variable while void pointer is a somewhat generic pointer and can be pointed to any data type, be it standard data type(int, char etc) or user define data type (structure, union etc.). We can pass any kind of pointer and reference it as a void pointer. But to dereference it, we have to type the void pointer to correct data type.

7) What is ISR?

An ISR (Interrupt Service Routine) is an interrupt handler, a callback subroutine which is called when an interrupt is encountered.

8) What is return type of ISR?

ISR does not return anything. An ISR returns nothing because there is no caller in the code to read the returned values.

9) What is interrupt latency?

Interrupt latency is the time required for an ISR responds to an interrupt.

10) How to reduce interrupt latency?

Interrupt latency can be minimized by writing short ISR routine and by not delaying interrupts for more time.

11) Can we use any function inside ISR?

We can use function inside ISR as long as that function is not invoked from other portion of the code.

12) Can we use printf inside ISR?

Printf function in ISR is not supported because printf function is not reentrant, thread safe and uses dynamic memory allocation which takes a lot of time and can affect the speed of an ISR up to a great extent.

13) Can we put breakpoint inside ISR?

Putting a break point inside ISR is not a good idea because debugging will take some time and a difference of half or more second will lead to different behavior of hardware. To debug ISR, definitive logs are better.

14) Can static variables be declared in a header file?

A static variable cannot be declared without defining it. A static variable can be defined in the header file. But doing so, the result will be having a private copy of that variable in each source file which includes the header file. So, it will be wise not to declare a static variable in header file, unless you are dealing with a different scenario.

15) Is Count Down_to_Zero Loop better than Count_Up_Loops?

Count down to zero loops are better. Reason behind this is that at loop termination, comparison to zero can be optimized by the compiler. Most processors have instruction for comparing to zero. So, they don't need to load the loop variable and the maximum value, subtract them and then compare to zero. That is why count down to zero loop is better.

16) What are inline functions?

The ARM compilers support inline functions with the keyword __inline. These functions have a small definition and the function body is substituted in each call to the inline function. The argument passing and stack maintenance is skipped and it results in faster code execution, but it increases code size, particularly if the inline function is large or one inline function is used often.

17) Can include files be nested?

Yes. Include files can be nested any number of times. But you have to make sure that you are not including the same file twice. There is no limit to how many header files that can be included. But the number can be compiler dependent, since including multiple header files may cause your computer to run out of stack memory.

18) What are the uses of the keyword static?

Static keyword can be used with variables as well as functions. A variable declared static will be of static storage class and within a function, it maintains its value between calls to that function. A variable declared as static within a file, scope of that variable will be within that file, but it can't be accessed by other files.

Functions declared static within a module can be accessed by other functions within that module. That is, the scope of the function is localized to the module within which it is declared.

19) What are the uses of the keyword volatile?

Volatile keyword is used to prevent compiler to optimize a variable which can change unexpectedly beyond compiler's comprehension. Suppose, we have a variable which may be changed from scope out of the program, say by a signal, we do not want the compiler to optimize it. Rather than optimizing that variable, we want the compiler to load the variable every time it is encountered. If we declare a variable volatile, compiler will not cache it in its register.

20) What is Top half & bottom half of a kernel?

Sometimes to handle an interrupt, a substantial amount of work has to be done. But it conflicts with the speed need for an interrupt handler. To handle this situation, Linux splits the handler into two parts – *Top half and Bottom half*. The top half is the routine that actually responds to the interrupt. The bottom half on the other hand is a routine that is scheduled by the upper half to be executed later at a safer time.

All interrupts are enabled during execution of the bottom half. The top half saves the device data into the specific buffer, schedules bottom half and exits. The bottom half does the rest. This way the top half can service a new interrupt while the bottom half is working on the previous.

21) Difference between RISC and CISC processor.

RISC (Reduced Instruction Set Computer) could carry out a few sets of simple instructions simultaneously. Fewer transistors are used to manufacture RISC, which makes RISC cheaper. RISC has uniform instruction set and those instructions are also fewer in number. Due to the less number of instructions as well as instructions being simple, the RISC computers are faster. RISC emphasise on software rather than hardware. RISC can execute instructions in one machine cycle.

CISC (Complex Instruction Set Computer) is capable of executing multiple operations through a single instruction. CISC have rich and complex instruction set and more number of addressing modes. CISC emphasise on hardware rather that software, making it costlier than RISC. It has a small code size, high cycles per second and it is slower compared to RISC.

22) What is RTOS?

In an operating system, there is a module called the scheduler, which schedules different tasks and determines when a process will execute on the processor. This way, the multi-tasking is achieved. The scheduler in a Real Time Operating System (RTOS) is designed to provide a predictable execution pattern. In an embedded system, a certain event must be entertained in strictly defined time.

To meet real time requirements, the behavior of the scheduler must be predictable. This type of OS which have a scheduler with predictable execution pattern is called Real Time OS(RTOS). The features of an RTOS are

- Context switching latency should be short.
- Interrupt latency should be short.
- Interrupt dispatch latency should be short.
- Reliable and time bound inter process mechanisms.
- Should support kernel preemption.

23) What is the difference between hard real-time and soft real-time OS?

A hard real-time system strictly adheres to the deadline associated with the task. If the system fails to meet the deadline, even once, the system is considered to have failed. In case of a soft real-time system, missing a deadline is acceptable. In this type of system, a critical real-time task gets priority over other tasks and retains that priority until it completes.

24) What type of scheduling is there in RTOS?

RTOS uses pre-emptive scheduling. In pre-emptive scheduling, the higher priority task can interrupt a running process and the interrupted process will be resumed later.

25) What is priority inversion?

If two tasks share a resource, the one with higher priority will run first. However, if the lower-priority task is using the shared resource when the higher-priority task becomes ready, then the higher-priority task must wait for the lower-priority task to finish. In this scenario, even though the task has higher priority it needs to wait for the completion of the lower-priority task with the shared resource. This is called priority inversion.

26) What is priority inheritance?

Priority inheritance is a solution to the priority inversion problem. The process waiting for any resource which has a resource lock will have the maximum priority. This is priority inheritance. When one or more high priority jobs are blocked by a job, the original priority assignment is ignored and execution of critical section will be assigned to the job with the highest priority in this elevated scenario. The job returns to the original priority level soon after executing the critical section.

27) How many types of IPC mechanism you know?

Different types of IPC mechanism are -

- Pipes
- Named pipes or FIFO
- Semaphores
- Shared memory
- Message queue
- Socket

28) What is semaphore?

Semaphore is actually a variable or abstract data type which controls access to a common resource by multiple processes. Semaphores are of two types -

- Binary semaphore – It can have only two values (0 and 1). The semaphore value is set to 1 by the process in charge, when the resource is available.
- Counting semaphore – It can have value greater than one. It is used to control access to a pool of resources.

29) What is spin lock?

If a resource is locked, a thread that wants to access that resource may repetitively check whether the resource is available. During that time, the thread may loop and check the resource without doing any useful work. Suck a lock is termed as spin lock.

30) What is difference between binary semaphore and mutex?

The differences between binary semaphore and mutex are as follows -

- Mutual exclusion and synchronization can be used by binary semaphore while mutex is used only for mutual exclusion.
- A mutex can be released by the same thread which acquired it. Semaphore values can be changed by other thread also.
- From an ISR, a mutex cannot be used.
- The advantage of semaphores is that, they can be used to synchronize two unrelated processes trying to access the same resource.
- Semaphores can act as mutex, but the opposite is not possible.

31) What is virtual memory?

- Virtual memory is a technique that allows processes to allocate memory in case of physical memory shortage using automatic storage allocation upon a request. The advantage of the virtual memory is that the program can have a larger memory than the physical memory. It allows large virtual memory to be provided when only a smaller physical memory is available. Virtual memory can be implemented using paging.

- A paging system is quite similar to a paging system with swapping. When we want to execute a process, we swap it into memory. Here we use a lazy swapper called pager rather than swapping the entire process into memory. When a process is to be swapped in, the pager guesses which pages will be used based on some algorithm, before the process is swapped out again. Instead of swapping whole process, the pager brings only the necessary pages into memory. By that way, it avoids reading in unnecessary memory pages, decreasing the swap time and the amount of physical memory.

32) What is kernel paging?

Paging is a memory management scheme by which computers can store and retrieve data from the secondary memory storage when needed in to primary memory. In this scheme, the operating system retrieves data from secondary storage in same-size blocks called pages. The paging scheme allows the physical address space of a process to be non-continuous. Paging allows OS to use secondary storage for data that does not fit entirely into physical memory.

33) Can structures be passed to the functions by value?

Passing structure by its value to a function is possible, but not a good programming practice. First of all, if we pass the structure by value and the function changes some of those values, then the value change is not reflected in caller function. Also, if the structure is big, then passing the structure by value means copying the whole structure to the function argument stack which can slow the program by a significant amount.

34) Why cannot arrays be passed by values to functions?

In C, the array name itself represents the address of the first element. So, even if we pass the array name as argument, it will be passed as reference and not its address.

35) Advantages and disadvantages of using macro and inline functions?

The advantage of the macro and inline function is that the overhead for argument passing and stuff is reduced as the function are in-lined. The advantage of macro function is that we can write type insensitive functions. It is also the disadvantage of macro function as macro functions can't do validation check. The macro and inline function also increases the size of the executable.

36) What happens when recursive functions are declared inline?

Inlining an recursive function reduces the overhead of saving context on stack. But inline is merely a suggestion to the compiler and it does not guarantee that a function will be inlined. Obviously, the compiler won't be able to inline a recursive function infinitely. It may not inline it at all or it may inline it, just a few levels deep.

37) #define cat(x,y) x##y concatenates x to y. But cat(cat(1,2),3) does not expand but gives preprocessor warning. Why?

The cat(x, y) expands to x##y. It just pastes x and y. But in case of cat(cat(1,2),3), it expands to cat(1,2)##3 instead of 1##2##3. That is why it is giving preprocessor warning.

38) ++*ip increments what?

It increments the value to which ip points to and not the address.

39) Declare a manifest constant that returns the number of seconds in a year using preprocessor? Disregard leap years in your answer.

The correct answer will be -

```
#define SECONDS_IN_YEAR (60UL * 60UL * 24UL * 365UL)
```

Do not forget to use UL, since the output will be very big integer.

40) Using the variable a, write down definitions for the following:

- An integer
- A pointer to an integer
- A pointer to a pointer to an integer
- An array of ten integers
- An array of ten pointers to integers
- A pointer to an array of ten integers
- A pointer to a function that takes an integer as an argument and returns an integer
- Pass an array of ten pointers to a function that take an integer argument and return an integer.

The correct answer is as follows -

- int a;
- int *a;
- int **a;
- int a[10];
- int *a[10];

- int (*a)[10];
- int (*a)(int);
- int (*a[10])(int);

41) Consider the two statements below and point out which one is preferred and why?

-
  ```
  #define B struct A *

  typedef struct A * C;
  ```
- The typedef is preferred. Both statements declare pointer to struct A to something else and in one glance both looks fine. But there is one issue with the define statement. Consider a situation where we want to declare p1 and p2 as pointer to struct A. We can do this by -

-
  ```
  C p1, p2;
  ```
- But doing - B p1, p2, it will be expanded to struct A * p1, p2. It means that p1 is a pointer to struct A but p2 is a variable of struct A and not a pointer.

42) What will be the output of the following code fragment?

-
  ```
  char *ptr;

  if ((ptr = (char *)malloc(0)) == NULL)

  {

     puts("Got a null pointer");

  }

  else

  {

     puts("Got a valid pointer");

  }
  ```
- The output will be “Got a valid pointer”. It is because malloc(0) returns a valid pointer, but it allocates size 0. So this pointer is of no use, but we can use this free pointer and the program will not crash.

43) What is purpose of keyword const?

The const keyword when used in c means that the value of the variable will not be changed. But the value of the variable can be changed using a pointer. The const identifier can be used like this -

```
const int a; or int const a;
```

Both means the same and it indicates that a is an constant integer. But if we declare something like this -

```
const int *p
```

then it does not mean that the pointer is constant but rather it is pointing to an constant integer. The declaration of an const pointer to a non-constant integer will look like this -

```
int * const p;
```

44) What do the following declarations mean?

```
const int a;

int const a;

const int *a;

int * const a;

int const * a const;
```

- The first two means that a is a constant integer.
- The third declaration means that a is a pointer to a constant integer.
- The fourth means that a is a constant pointer to a non-constant integer.
- The fifth means that a is a constant pointer to a constant integer.

  - **45) How to decide whether given processor is using little endian format or big endian format ?**
  - The following program can find out the endianness of the processor.

```
#include<stdio.h>

main ()

{

 union Test

 {

    unsigned int i;

    unsigned char c[2];

 };

 union Test a = {300};

 if((a.c [0] == 1) &&  (a.c [1] == 44))

 {

    printf ("BIG ENDIAN\n");

 }

 else

 {

    printf ("LITTLE ENDIAN\n");

 }

}
```

46) What is the concatenation operator?

The Concatenation operator (##) in macro is used to concatenate two arguments. Literally, we can say that the arguments are concatenated, but actually their value are not concatenated. Think it this way, if we pass A and B to a macro which uses ## to concatenate those two, then the result will be AB. Consider the example to clear the confusion-

```
#define SOME_MACRO(a, b) a##b

main()

{

  int var = 15;

  printf("%d", SOME_MACRO(v, ar));

}
```

Output of the above program will be 15.

47) Infinite loops often arise in embedded systems. How does you code an infinite loop in C?

There are several ways to code an infinite loop -

```
while(1)

{ }

or,

for(;;)

{ }

or,

Loop:

goto Loop
```

But many programmers prefer the first solution as it is easy to read and self-explanatory, unlike the second or the last one.

48) Guess the output:

```
main()

{

 fork();

 fork();

 fork();

 printf("hello world\n");

}
```

It will print “hello world' 8 times. The main() will print one time and creates 3 children, let us say Child_1, Child_2, Child_3. All of them printed once. The Child_3 will not create any child. Child2 will create one child and that child will print once. Child_1 will create two children, say Child_4 and Child_5 and each of them will print once. Child_4 will again create another child and that child will print one time. A total of eight times the printf statement will be executed.

49) What is forward reference w.r.t. pointers in c?

Forward Referencing with respect to pointers is used when a pointer is declared and compiler reserves the memory for the pointer, but the variable or data type is not defined to which the pointer points to. For example

```
struct A *p;

struct A

{

 // members

};
```

50) How is generic list manipulation function written which accepts elements of any kind?

It can be achieved using void pointer. A list may be expressed by a structure as shown below

```
typedef struct

{

 node *next;

 /* data part */

 ......

}node;
```

Assuming that the generic list may be like this

```
typedef struct

{

 node *next;

 void *data;

}node;
```

This way, the generic manipulation function can work on this type of structures.

51) How can you define a structure with bit field members?

Bit field members can be declared as shown below

```
struct A

{

 char c1 : 3;

 char c2 : 4;

 char c3 : 1;

};
```

Here c1, c2 and c3 are members of a structure with width 3, 4, and 1 bit respectively. The ':' indicates that they are bit fields and the following numbers indicates the width in bits.

52) How do you write a function which takes 2 arguments - a byte and a field in the byte and returns the value of the field in that byte?

The function will look like this -

```
int GetFieldValue(int byte, int field )

{

  byte = byte >> field;

  byte = byte & 0x01;

  return byte;

}
```

The byte is right shifted exactly n times where n is same as the field value. That way, our intended value ends up in the 0th bit position. "Bitwise And" with 1 can get the intended value. The function then returns the intended value.

53) Which parameters decide the size of data type for a processor?

Actually, compiler is the one responsible for size of the data type. But it is true as long as OS allows that. If it is not allowable by OS, OS can force the size.

54) What is job of preprocessor, compiler, assembler and linker?

The preprocessor commands are processed and expanded by the preprocessor before actual compilation. After preprocessing, the compiler takes the output of the preprocessor and the source code, and generates assembly code. Once compiler completes its work, the assembler takes the assembly code and produces an assembly listing with offsets and generate object files.

The linker combines object files or libraries and produces a single executable file. It also resolves references to external symbols, assigns final addresses to functions and variables, and revises code and data to reflect new addresses.

55) What is the difference between static linking and dynamic linking ?

In static linking, all the library modules used in the program are placed in the final executable file making it larger in size. This is done by the linker. If the modules used in the program are modified after linking, then re-compilation is needed. The advantage of static linking is that the modules are present in an executable file. We don't want to worry about compatibility issues.

In case of dynamic linking, only the names of the module used are present in the executable file and the actual linking is done at run time when the program and the library modules both are present in the memory. That is why, the executables are smaller in size. Modification of the library modules used does not force re-compilation. But dynamic linking may face compatibility issues with the library modules used.

56) What is the purpose of the preprocessor directive #error?

Preprocessor error is used to throw a error message during compile time. We can check the sanity of the make file and using debug options given below

```
#ifndef DEBUG

#ifndef RELEASE

#error Include DEBUG or RELEASE in the makefile

#endif

#endif
```

57) On a certain project it is required to set an integer variable at the absolute address 0x67a9 to the value 0xaa55. The compiler is a pure ANSI compiler. Write code to accomplish this task.

This can be achieved by the following code fragment:

```
int *ptr;

ptr = (int *)0x67a9;

*ptr = 0xaa55;
```

58) Significance of watchdog timer in Embedded Systems.

The watchdog timer is a timing device with a predefined time interval. During that interval, some event may occur or else the device generates a time out signal. It is used to reset to the original state whenever some inappropriate events take place which can result in system malfunction. It is usually operated by counter devices.

59) Why ++n executes faster than n+1?

The expression ++n requires a single machine instruction such as INR to carry out the increment operation. In case of n+1, apart from INR, other instructions are required to load the value of n. That is why ++n is faster.

60) What is wild pointer?

A pointer that is not initialized to any valid address or NULL is considered as wild pointer. Consider the following code fragment -

```
int *p;

*p = 20;
```

Here p is not initialized to any valid address and still we are trying to access the address. The p will get any garbage location and the next statement will corrupt that memory location.

61) What is dangling pointer?

If a pointer is de-allocated or freed and the pointer is not assigned to NULL, then it may still contain that address and accessing the pointer means that we are trying to access that location and it will give an error. This type of pointer is called dangling pointer.

62) Write down the equivalent pointer expression for referring the same element a[i][j][k][l] ?

We know that a[i] can be written as *(a+i). Same way, the array elements can be written like pointer expression as follows -

```
a[i][j] == *(*(a+i)+j)

a[i][j][k] == *(*(*(a+i)+j)+k)

a[i][j][k][l] == *(*(*(*(a+i)+j)+k)+l)
```

63) Which bit wise operator is suitable for checking whether a particular bit is on or off?

"Bitwise And" (&) is used to check if any particular bit is set or not. To check whether 5'th bit is set we can write like this

```
bit = (byte >> 4) & 0x01;
```

Here, shifting byte by 4 position means taking 5'th bit to first position and "Bitwise And" will get the value in 0 or 1.

64) When should we use register modifier?

The register modifier is used when a variable is expected to be heavily used and keeping it in the CPU’s registers will make the access faster.

65) Why doesn't the following statement work?

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

strcat ( str, '!' ) ;
```

The string function strcat( ) concatenates two strings. But here the second argument is '!', a character and that is the reason why the code doesn't work. To make it work, the code should be changed like this:

```
strcat ( str, "!" ) ;
```

66) Predict the output or error(s) for the following program:

```
void main()

{

   int const * p = 5;

   printf("%d",++(*p));

}
```

The above program will result in compilation error stating “Cannot modify a constant value”. Here p is a pointer to a constant integer. But in the next statement, we are trying to modify the value of that constant integer. It is not permissible in C and that is why it will give a compilation error.

67)Guess the output:

```
#include<stdio.h>
```

```
main()

{

 unsigned int a = 2;

 int b = -10;

 (a + b > 0)?puts("greater than 0"):puts("less than 1");

}
```

Output - greater than 0

If you have guessed the answer wrong, then here is the explanation for you. The a + b is -8, if we do the math. But here the addition is between different integral types - one is unsigned int and another is int. So, all the operands in this addition are promoted to unsigned integer type and b turns to a positive number and eventually a big one. The outcome of the result is obviously greater than 0 and hence, this is the output.

68) Write a code fragment to set and clear only bit 3 of an integer.

```
#define BIT(n) (0x1 << n)

int a;

void SetBit3()

{

   a |= BIT(3);

}



void ClearBit3()

{

   a &= ~BIT(3);

}
```

69) What is wrong with this code?

```
int square(volatile int *p)

{

   return *p * *p;

}
```

The intention of the above code is to return the square of the integer pointed by the pointer p. Since it is volatile, the value of the integer may have changed suddenly and will result in something else which will looks like the result of the multiplication of two different integers. To work as expected, the code needs to be modified like this.

```
int square(volatile int *p)

{

   int a = *p;

   return a*a;

}
```

70) Is the code fragment given below is correct? If so what is the output?

```
int i = 2, j = 3, res;

res = i+++j;
```

The above code is correct, but little bit confusing. It is better not to follow this type of coding style. The compiler will interpret above statement as “res = i++ + j”. So the res will get value 5 and i after this will be 3.

100 embedded c interview questions, your interviewer might ask

BY AMLENDRAON

31

In my previous post, I have created a collection of “c interview questions” that is liked by many people. I have also get the response to create a list of interview questions on “embedded c”. So here I have tried to create some collection of questions that might be asked by your interviewer.

What is the volatile keyword?

The volatile keyword is a type qualifier that prevents the objects from the compiler optimization. According to C standard, an object that has volatile-qualified type may be modified in ways unknown to the implementation or have other unknown side effects. You can also say that the value of the volatile-qualified object can be changed at any time without any action being taken by the code. If an object is qualified by the volatile qualifier, the compiler reloads the value from memory each time it is accessed by the program that means it prevents from to cache a variable into a register. Reading the value from the memory is the only way to check the unpredictable change of the value.

What is the use of volatile keyword?

The volatile keyword is mainly used where we directly deal with GPIO, interrupt or flag Register. It is also used where a global variable or buffer is shared between the threads.

What is the difference between the const and volatile qualifier in C?

The const keyword is compiler-enforced and says that program could not change the value of the object that means it makes the object nonmodifiable type.
e.g,
const int a = 0;
if you will try to modify the value of “a”, you will get the compiler error because “a” is qualified with const keyword that prevents to change the value of the integer variable.

In another side volatile prevent from any compiler optimization and says that the value of the object can be changed by something that is beyond the control of the program and so that compiler will not make any assumption about the object.
e.g,
volatile int a;

When the compiler sees the above declaration then it avoids to make any assumption regarding the “a” and in every iteration read the value from the address which is assigned to the variable.

Can a variable be both constant and volatile in C?

Yes, we can use both constant and volatile together. One of the great use of volatile and const keyword together is at the time of accessing the GPIO registers. In case of GPIO, its value can be changed by the ‘external factors’ (if a switch or any output device is attached with GPIO), if it is configured as an input. In that situation, volatile plays an important role and ensures that the compiler always read the value from the GPIO address and avoid to make any assumption.

After using the volatile keyword, you will get the proper value whenever you are accessing the ports but still here is one more problem because the pointer is not const type so it might be your program change the pointing address of the pointer. So we have to create a constant pointer with volatile keyword.

Syntax of declaration,

int volatile * const PortRegister;

How to read the above declaration,

```
int volatile * const PortRegister;
 |     |     |   |    |
 |     |     |   |    +------> PortRegister is a
 |     |     |   +-----------> constant
 |     |     +---------------> pointer to a
 |     +---------------------> volatile
 +---------------------------> integer
```

Can we have a volatile pointer?

Yes, we can create a volatile pointer in C language.
int * volatile piData; // piData is a volatile pointer to an integer.

The Proper place to use the volatile keyword?

Here I am pointing some important places where we need to use the volatile keyword.

- Accessing the memory-mapped peripherals register or hardware status register.

```
#define COM_STATUS_BIT  0x00000006

uint32_t const volatile * const pStatusReg = (uint32_t*)0x00020000;


unit32_t GetRecvData()
{
 //Code to recv data
   while (((*pStatusReg)  & COM_STATUS_BIT) == 0)
 {
       // Wait until flag does not set
   }

   return RecvData;
}
```

- Sharing the global variables or buffers between the multiple threads.
- Accessing the global variables in an interrupt routine or signal handler.

```
volatile int giFlag = 0;

```

```
ISR(void)
{
 giFlag = 1;
}

int main(void)
{

 while (!giFlag)
 {
    //do some work
 }

 return 0;
}
```

What is ISR?

An ISR refers to the Interrupt Service Routines. These are procedures stored at specific memory addresses which are called when a certain type of interrupt occurs. The Cortex-M processors family has the NVIC that manage the execution of the interrupt.

Can we pass any parameter and return a value from the ISR?

An ISR returns nothing and not allow to pass any parameter. An ISR is called when a hardware or software event occurs, it is not called by the code, so that’s the reason no parameters are passed into an ISR.

In above line, we have already read that the ISR is not called by the code, so there is no calling code to read the returned values of the ISR. It is the reason that an ISR is not returned any value.

What is interrupt latency?

It is an important question that is asked by the interviewer to test the understanding of Interrupt. Basically, interrupt latency is the number of clock cycles that is taken by the processor to respond to an interrupt request. These number of the clock cycle is count between the assertions of the interrupt request and first instruction of the interrupt handler.

Interrupt Latency on the Cortex-M processor family

The Cortex-M processors have the very low interrupt latency. In below table, I have mentioned, Interrupt latency of Cortex-M processors with zero wait state memory systems.

Processors	Cycles with zero wait state memory
Cortex-M0	16
Cortex-M0+	15
Cortex-M3	12
Cortex-M4	12
Cortex-M7	12

How do you measure interrupt latency?

With the help of the oscilloscope, we can measure the interrupt latency. You need to take following steps.

- First takes two GPIOs.
- Configure one GPIO to generate the interrupt and second for the toggling (if you want you can attach an LED).
- Monitor the PIN (using the oscilloscope or analyzer) which you have configured to generate the interrupt.
- Also, monitor (using the oscilloscope or analyzer) the second pin which is toggled at the beginning of the interrupt service routine.
- When you will generate the interrupt then the signal of the both GPIOs will change.

The interval between the two signals (interrupt latency) may be easily read from the instrument.

How to reduce the interrupt latency?

The interrupt latency depends on many factors, some factor I am mentioning in below statements.

- Platform and interrupt controller.
- CPU clock speed.
- Timer frequency
- Cache configuration.
- Application program.

So using the proper selection of platform and processor we can easily reduce the interrupt latency. We can also reduce the interrupt latency by making the ISR shorter and avoid to calling a function within the ISR.

What are the causes of Interrupt Latency?

- The first delay is typically caused by hardware: The interrupt request signal needs to be synchronized to the CPU clock. Depending on the synchronization logic, up to 3 CPU cycles may expire before the interrupt request has reached the CPU core.
- The CPU will typically complete the current instruction, which may take several cycles. On most systems, divide, push-multiple or memory-copy instructions are the most time-consuming instructions to execute. On top of the cycles required by the CPU, additional cycles are often required for memory accesses. In an ARM7 system, the instruction STMDB SP!,{R0-R11, LR} typically is the worst case instruction, storing 13 registers of 32-bits each to the stack, and takes 15 clock cycles to complete.
- The memory system may require additional cycles for wait states.
- After completion of the current instruction, the CPU performs a mode switch or pushes registers on the stack (typically PC and flag registers). Modern CPUs such as ARM generally perform a mode switch, which takes fewer CPU cycles than saving registers.
- Pipeline fill: Most modern CPUs are pipelined. Execution of an instruction happens in various stages of the pipeline. An instruction is executed when it has reached its final stage of the pipeline. Since the mode switch has flushed the pipeline, a few extra cycles are required to refill the pipeline.

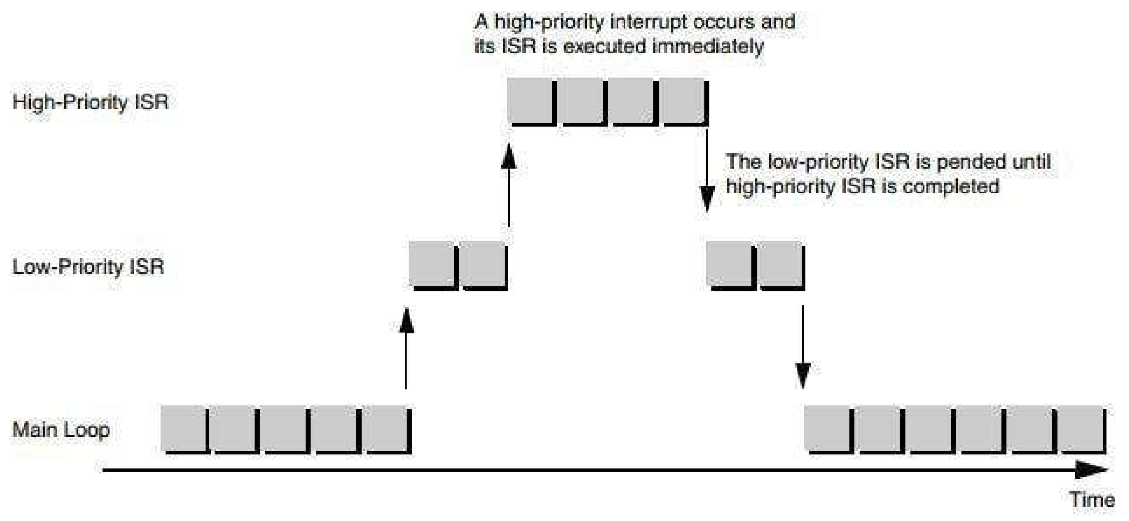
What is nested interrupt?

In a nested interrupt system, an interrupt is allowed to any time and anywhere even an ISR is being executed. But, only the highest priority ISR will be executed immediately. The second highest priority ISR will be executed after the highest one is completed.

The rules of a nested interrupt system are:

- All interrupts must be prioritized.
- After initialization, any interrupts are allowed to occur anytime and anywhere.
- If a low-priority ISR is interrupted by a high-priority interrupt, the high-priority ISR is executed.
- If a high-priority ISR is interrupted by a low-priority interrupt, the high-priority ISR continues executing.

- The same priority ISRs must be executed by time order


A high-priority interrupt occurs and its ISR is executed immediately
High-Priority ISR
The low-priority ISR is pended until high-priority ISR is completed
Low-Priority ISR
Main Loop
Time

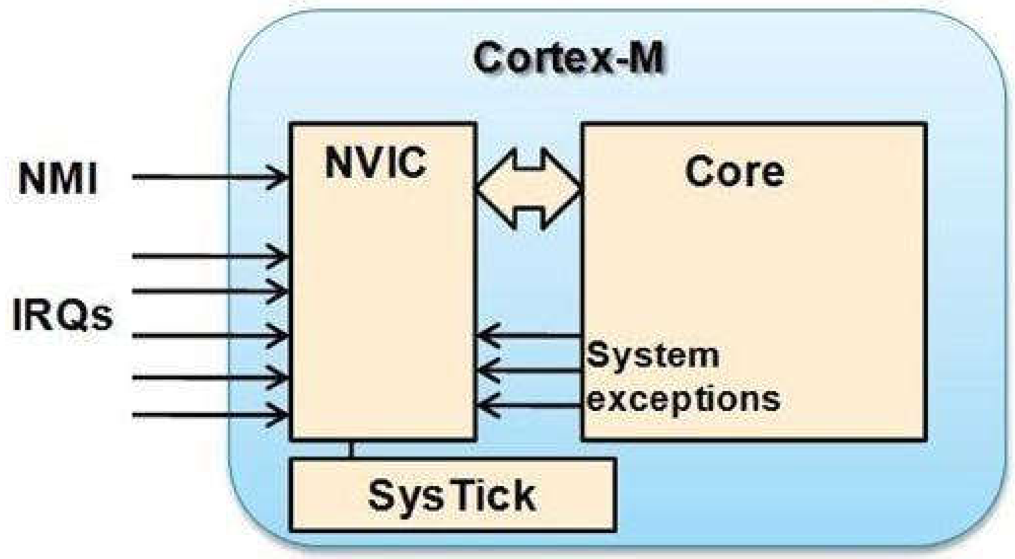
If you want to learn STM32 from scratch, you should follow this course "Mastering Microcontroller with Embedded Driver Development". The course contains video lectures of **18.5-hours** length covering all topics like, Microcontroller & Peripheral Driver Development for STM32 GPIO, I2C, SPI, USART using Embedded C.

Enroll In Course

What is NVIC in ARM Cortex?

The Nested Vector Interrupt Controller (NVIC) in the Cortex-M processor family is an example of an interrupt controller with extremely flexible interrupt priority management. It enables programmable priority levels, automatic nested interrupt support, along with support for multiple interrupt masking, whilst still being very easy to use by the programmer.

The Cortex-M3 and Cortex-M4 processors the NVIC supports up to 240 interrupt inputs, with 8 up to 256 programmable priority levels



Can we use any function inside ISR?

Yes, you can call a function within the ISR but it is not recommended because it can increase the interrupt latency and decrease the performance of the system. If you want to call a nested function within the ISR, you need to read the datasheet of your microcontroller because some vendors have a limit to how many calls can be nested.

One important point also needs to remember that function which is called from the ISR should be re-entrant. If the called function is not re-entrant, it could create the issues.

For example,
If the function is not reentrant and supposes that it is called by another part of the code beside the ISR. So the problem will be invoked when if the ISR calls the same function which is already invoked outside of the ISR?

Can we change the interrupt priority level of Cortex-M processor family?

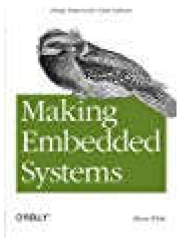
Yes, we can.

What is the start-up code?

A start-up code is called prior to the main function, it creates a basic platform for the application. It is a small block of code that is written in assembly language.

There are following parts of the start-up code.

- Declaration of the Stack area.
- Declaration of the Heap area.
- Vector table.
- Reset handler code.
- Other exception handler code.



Making Embedded Systems: Design Patterns for Grea…

$17.59~~$39.99~~

(52)

Test Driven Development for Embedded C (Pragmatic Pr…

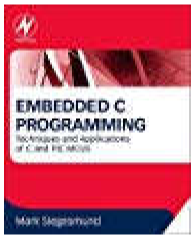
$26.80~~$34.95~~

(32)

C Programming Language, 2nd Edition

$62.85~~$67.00~~

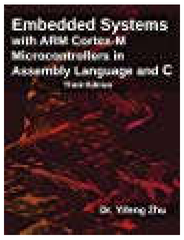
(610)



Embedded C Programming: Techniques and Applicatio…

$50.83~~$59.95~~

(3)


Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C
Third Edition
Dr. Yifeng Zhu

Embedded Systems with ARM Cortex-M Microcon…

What are the start-up code steps?

Start-up code for C programs usually consists of the following actions, performed in the order described:

- Disable all interrupts.

- Copy any initialized data from ROM to RAM.
- Zero the uninitialized data area.
- Allocate space for and initialize the stack.
- Initialize the processor’s stack pointer.
- Create and initialize the heap.
- Enable interrupts.
- Call main.

Infinite loops often arise in embedded systems. How do you code an infinite loop in C?

In embedded systems, infinite loops are generally used. If I talked about a small program to control a led through the switch, in that scenario an infinite loop will be required if we are not going through the interrupt.

There are the different way to create an infinite loop, here I am mentioning some methods.

Method 1:

```
while(1)
{
// task
}
```

Method 2:

```
for(;;)
{
// task
}
```

Method 3:

```
Loop:
goto Loop;
```

How to access the fixed memory location in embedded C?

It is a very basic question that is generally asked by the interviewer.

Let's take an example:
Suppose in an application, you have required accessing a fixed memory address.

```
//Memory address, you want to access
#define RW_FLAG 0x1FFF7800

//Pointer to access the Memory address
volatile uint32_t *flagAddress = NULL;

//variable to stored the read value
uint32_t readData = 0;

//Assign addres to the pointer
flagAddress = (volatile uint32_t *)RW_FLAG;

//Read value from memory
* flagAddress = 12; // Write

//Write value to the memory
readData = * flagAddress;
```

Difference between RISC and CISC processor?

The RISC (reduced instruction set computer) and CISC (Complex instruction set computer) are the processors ISA (instruction set architecture).

There are following difference between both architecture:

	RISC	CISC
Acronym	It stands for ‘Reduced Instruction Set Computer’.	It stands for ‘Complex Instruction Set Computer’.
Definition	The RISC processors have a smaller set of instructions with few addressing nodes.	The CISC processors have a larger set of instructions with many addressing nodes.
Memory unit	It has no memory unit and uses a separate hardware to implement instructions.	It has a memory unit to implement complex instructions.
Program	It has a hard-wired unit of programming.	It has a micro-programming unit.
Design	It is a complex complier design.	It is an easy complier design.
Calculations	The calculations are faster and precise.	The calculations are slow and precise.
Decoding	Decoding of instructions is simple.	Decoding of instructions is complex.
Time	Execution time is very less.	Execution time is very high.
External memory	It does not require external memory for calculations.	It requires external memory for calculations.
Pipelining	Pipelining does function correctly.	Pipelining does not function correctly.

Stalling	Stalling is mostly reduced in processors.	The processors often stall.
Code expansion	Code expansion can be a problem.	Code expansion is not a problem.
Disc space	The space is saved.	The space is wasted.
Applications	Used in high-end applications such as video processing, telecommunications and image processing.	Used in low-end applications such as security systems, home automations, etc.

Images Courtesy: ics.uci.edu

What is the stack overflow?

If your program tries to access the beyond the limit of the available stack memory then stack overflow occurs. In another word you can say that a stack overflow occurs if the call stack pointer exceeds the stack boundary.

If stack overflow occurs, the program can crash or you can say that segmentation fault that is the result of the stack overflow.

What is the cause of the stack overflow?

In the embedded application we have a little amount of stack memory as compare to the desktop application. So we have to work on embedded application very carefully either we can face the stack overflow issues that can be a cause of the application crash.

Here, I have mentioned some causes of unwanted use of the stack.

- Improper use of the recursive function.
- Passing to much arguments in the function.
- Passing a structure directly into a function.
- Nested function calls.
- Creating a huge size local array.

What is the difference between I2c and SPI communication Protocol?

In the embedded system, I2C and SPI both play an important role. Both communication protocols are the example of the synchronous communication but still, both have some important difference.

The important difference between the I2C and SPI communication protocol.

- I2C support half duplex while SPI is the full duplex communication.
- I2C requires only two wire for communication while SPI requires three or four wire for communication (depends on requirement).
- I2C is slower as compared to the SPI communication.
- I2C draws more power than SPI.
- I2C is less susceptible to noise than SPI.
- I2C is cheaper to implement than the SPI communication protocol.
- I2C work on wire and logic and it has a pull-up resistor while there is no requirement of pull-up resistor in case of the SPI.
- In I2C communication we get the acknowledgment bit after each byte, it is not supported by the SPI communication protocol.
- I2C ensures that data sent is received by the slave device while SPI does not verify that data is received correctly.
- I2C support the multi-master communication while multi-master communication is not supported by the SPI.
- One great difference between I2C and SPI is that I2C supports multiple devices on the same bus without any additional select lines (work on the basis of device address) while SPI requires additional signal (slave select lines) lines to manage multiple devices on the same bus.
- I2C supports arbitration while SPI does not support the arbitration.
- I2C support the clock stretching while SPI does not support the clock stretching.
- I2C can be locked up by one device that fails to release the communication bus.

- I2C has some extra overhead due to start and stop bits.
- I2C is better for long distance while SPI is better for the short distance.
- In the last I2C developed by NXP while SPI by Motorola.

What is the difference between Asynchronous and Synchronous Communication?

There are following difference between the asynchronous and synchronous communication.

Asynchronous Communication	**Synchronous Communication**
There is no common clock signal between the sender and receivers.	Communication is done by a shared clock.
Sends 1 byte or character at a time.	Sends data in the form of blocks or frames.
Slow as compare to synchronous communication.	Fast as compare to asynchronous communication.
Overhead due to start and stop bit.	Less overhead.
Ability to communicate long distance.	Less as compared to asynchronous communication.
A start and stop bit used for the data synchronization.	A shared clock is used for the data synchronization.
Economical	Costly

RS232, RS485	I2C, SPI.

What is the difference between RS232 and RS485?

The RS232 and RS485 is an old serial interface. Both serial interfaces are the standard for the data communication. This question is also very important and generally ask by an interviewer.

Some important difference between the RS232 and RS485

Parameter	**RS232**	**RS485**
Line configuration	Single –ended	differential
Numbers of devices	1 transmitter 1 receiver	32 transmitters 32 receivers
Mode of operation	Simplex or full duplex	Simplex or half duplex
Maximum cable length	50 feet	4000 feet
Maximum data rate	20 Kbits/s	10 Mbits/s
signaling	unbalanced	balanced
Typical logic levels	+-5 ~ +-15V	+-1.5 ~ +-6V

Minimum receiver input impedance	3 ~ 7 K-ohm	12 K-ohm
Receiver sensitivity	+-3V	+-200mV

What is the difference between Bit Rate and Baud Rate?

Bit Rate	**Baud Rate**
Bit rate is the number of bits per second.	Baud rate is the number of signal units per second.
It determines the number of bits traveled per second.	It determines how many times the state of a signal is changing.
Cannot determine the bandwidth.	It can determine how much bandwidth is required to send the signal.
This term generally used to describe the processor efficiency.	This term generally used to describe the data transmission over the channel.
Bit rate = baud rate x the number of bits per signal unit	Baud rate = bit rate / the number of bits per signal unit

What is segmentation fault in C?

A segmentation fault is a common problem that causes programs to crash. A core file (core dumped file) also associated with segmentation fault that is used by the developer to finding the root cause of the crashing (segmentation fault).

Generally, the segmentation fault occurs when a program tried to access a memory location that it is not allowed to access or tried to access a memory location in a way that is not allowed (tried to access read-only memory).

What are the common causes of segmentation fault in C?

There are many reasons for the segmentation fault, here I am listing some common causes of the segmentation fault.

- Dereferencing NULL pointers.
- Tried to write read-only memory (such as code segment).
- Trying to access a nonexistent memory address (outside process's address space).
- Trying to access memory the program does not have rights to (such as kernel structures in process context).
- Sometimes dereferencing or assigning to an uninitialized pointer (because might point an invalid memory) can be the cause of the segmentation fault.
- Dereferencing the freed memory (after calling the free function) can also be caused by the segmentation fault.
- A stack overflow is also caused by the segmentation fault.
- A buffer overflow (try to access the array beyond the boundary) is also cause of the segmentation fault.

What is the difference between Segmentation fault and Bus error?

In case of segmentation fault, SIGSEGV (11) signal is generated. Generally, a segmentation fault occurs when the program tries to access the memory to which it doesn't have access to.

In below I have mentioned some scenarios where SIGSEGV signal is generated.

- When trying to de-referencing a NULL pointer.
- Trying to access memory which is already de-allocated (trying to use dangling pointers).
- Using uninitialized pointer(wild pointer).
- Trying to access memory that the program doesn't own (eg. trying to access an array element out of array bounds).

In case of a BUS error, SIGBUS (10) signal is generated. The Bus error issue occurs when a program tries to access an invalid memory or unaligned memory. The bus error comes rarely as compared to the segmentation fault.

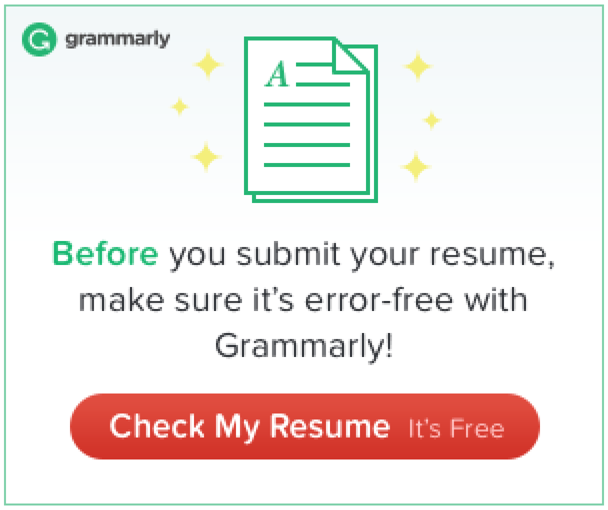
In below I have mentioned some scenarios where SIGBUS signal is generated.

- Non-existent address.
- Unaligned access.
- Paging errors

Size of the integer depends on what?

The C standard is explained that the minimum size of the integer should be 16 bits. Some programing language is explained that the size of the integer is implementation dependent but portable programs shouldn't depend on it.

Primarily size of integer depends on the type of the compiler which has written by compiler writer for the underlying processor. You can see compilers merrily changing the size of integer according to convenience and underlying architectures. So it is my recommendation use the C99 integer data types (uin8_t, uin16_t, uin32_t ..) in place of standard int.


grammarly
Before you submit your resume, make sure it's error-free with Grammarly!
Check My Resume It's Free

Are integers signed or unsigned?

In standard C language, integer data type is by default signed. So if you create an integer variable, it can store both positive and negative value.

For more details on signed and unsigned integer, check out:
A closer look at signed and unsigned integers in C

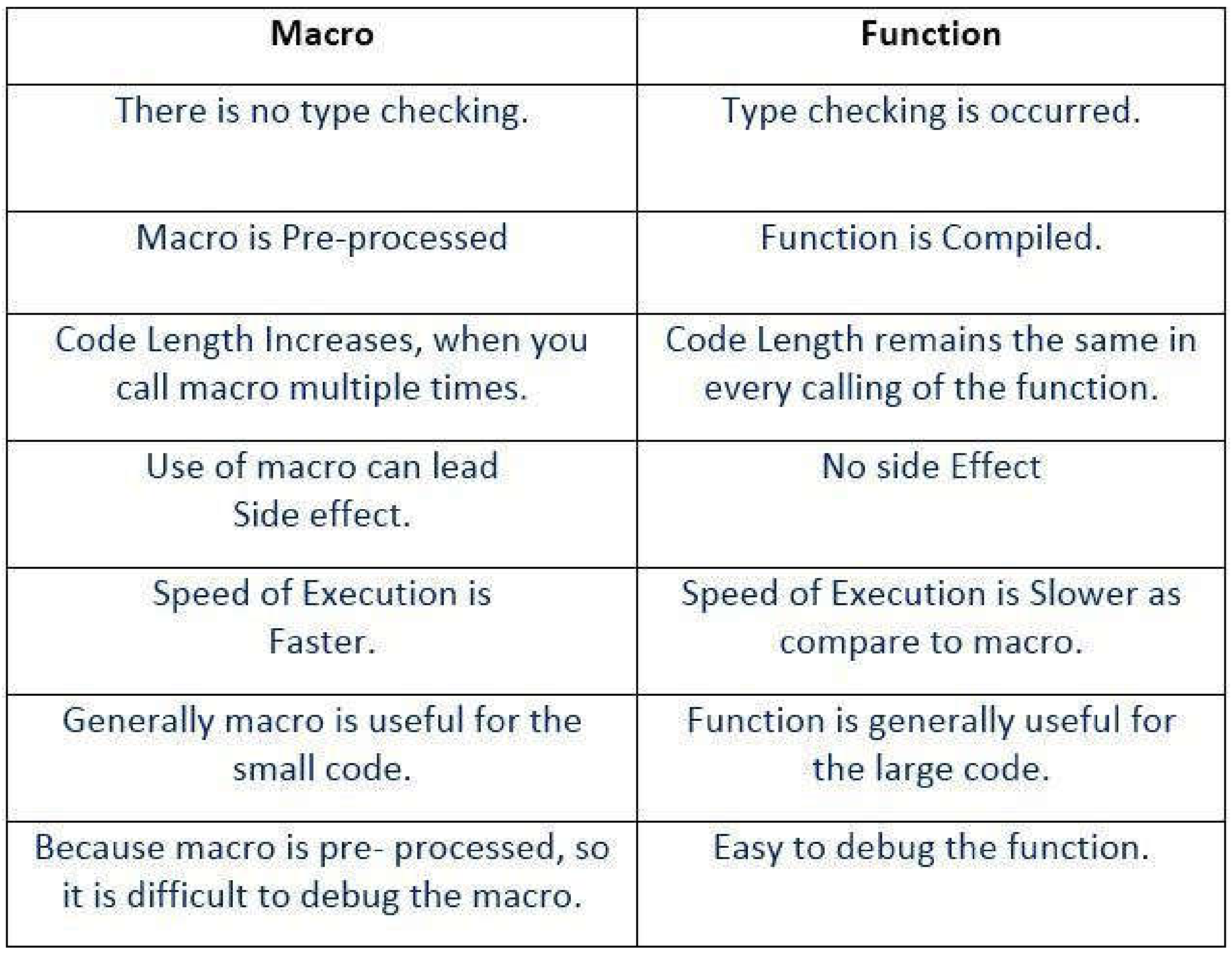
What is a difference between unsigned int and signed int in C?

The signed and unsigned integer type has the same storage (according to the standard at least 16 bits) and alignment but still, there is a lot of difference them, in bellows lines, I am describing some difference between the signed and unsigned integer.

- A signed integer can store the positive and negative value both but beside it unsigned integer can only store the positive value.
- The range of nonnegative values of a signed integer type is a sub-range of the corresponding unsigned integer type.
  For example,
  Assuming size of the integer is 2 bytes.
  signed int -32768 to +32767
  unsigned int 0 to 65535

- When computing the unsigned integer, it never gets overflow because if the computation result is greater than the largest value of the unsigned integer type, it is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type.
  For example,
  Computational Result % (Largest value of the unsigned integer+1)
- The overflow of signed integer type is undefined.
- If Data is signed type negative value, the right shifting operation of Data is implementation dependent but for the unsigned type, it would be Data/ 2pos.
- If Data is signed type negative value, the left shifting operation of Data show the undefined behavior but for the unsigned type, it would be Data x 2pos.

What is the difference between a macro and a function?



Macro	Function
There is no type checking.	Type checking is occurred.
Macro is Pre-processed	Function is Compiled.
Code Length Increases, when you call macro multiple times.	Code Length remains the same in every calling of the function.
Use of macro can lead Side effect.	No side Effect
Speed of Execution is Faster.	Speed of Execution is Slower as compare to macro.
Generally macro is useful for the small code.	Function is generally useful for the large code.
Because macro is pre- processed, so it is difficult to debug the macro.	Easy to debug the function.

What is the difference between typedef & Macros?

typedef:

The C language provides a very important keyword typedef for defining a new name for existing types. The typedef is the compiler directive mainly use with user-defined data types (structure, union or enum) to reduce their complexity and increase the code readability and portability.

***Syntax*,**
typedef type NewTypeName;

Let's take an example,
typedef unsigned int UnsignedInt;

Now UnsignedInt is a new type and using it, we can create a variable of unsigned int.

UnsignedInt Mydata;
In above example, Mydata is variable of unsigned int.

Note: *A typedef creates synonyms or a new name for existing types it does not create new types.*

Macro:

A macro is a pre-processor directive and it replaces the value before compiling the code.One of the major problem with the macro that there is no type checking. Generally, the macro is used to create the alias, in C language macro is also used as a file guard.

***Syntax*,**

#define Value 10

Now Value becomes 10, in your program, you can use the Value in place of the 10.

What do you mean by enumeration in C?

In C language enum is user-defined data type and it consists a set of named constant integer. Using the enum keyword, we can declare an enumeration type by using the enumeration tag (optional) and a list of named integer.

An enumeration increases the readability of the code and easy to debug in comparison of symbolic constant (macro).

The most important thing about the enum is that it follows the scope rule and compiler automatic assign the value to its member constant.

Note: *A variable of enumeration type stores one of the values of the enumeration list defined by that type.*

Syntax of enum,
enum Enumeration_Tag { Enumeration_List };

The Enumeration_Tag specifies the enumeration type name.

The Enumeration_List is a comma-separated list of named constant.

Example,
enum FLASH_ERROR { DEFRAGMENT_ERROR, BUS_ERROR};.

What is the difference between const and macro?

- The const keyword is handled by the compiler, in another hand, a macro is handled by the preprocessor directive.
- const is a qualifier that is modified the behavior of the identifier but macro is preprocessor directive.
- There is type checking is occurred with const keyword but does not occur with #define.
- const is scoped by C block, #define applies to a file.
- const can be passed as a parameter (as a pointer) to the function.In case of call by reference, it prevents to modify the passed object value.

How to set, clear, toggle and checking a single bit in C?

Setting a Bits
Bitwise OR operator (|) use to set a bit of integral data type. “OR” of two bits is always one if any one of them is one.

Number | = (1<< nth Position)

Clearing a Bits
Bitwise AND operator (&) use to clear a bit of integral data type. “AND” of two bits is always zero if any one of them is zero.To clear the nth bit, first, you need to invert the string of bits then AND it with the number.

Number &= ~ (1<< nth Position)

Checking a Bits
To check the nth bit, shift the ‘1’ nth position toward the left and then “AND” it with the number.

Bit = Number & (1 << nth)

Toggling a Bits
Bitwise XOR (^) operator use to toggle the bit of an integral data type. To toggle the nth bit shift the ‘1’ nth position toward the left and “XOR” it.

Number ^= (1<< nth Position)

Write a program swap two numbers without using the third variable?

Let’s assume a, b two numbers, there are a lot of methods two swap two number without using the third variable.

Method 1((Using Arithmetic Operators):

```
#include <stdio.h>

```

```
int main()
{
 int a = 10, b = 5;

 // algo to swap 'a' and 'b'
 a = a + b;  // a becomes 15
 b = a - b;  // b becomes 10
 a = a - b;  // fonally a becomes 5

 printf("After Swapping the value of: a = %d, b = %d\n\n", a, b);

 return 0;
}
```

Method 2 (Using Bitwise XOR Operator):

```
#include <stdio.h>

int main()
{
 int a = 10, b = 5;

 // algo to swap 'a' and 'b'
 a = a ^ b;  // a becomes (a ^ b)
 b = a ^ b;  // b = (a ^ b ^ b), b becomes a
 a = a ^ b;  // a = (a ^ b ^ a), a becomes b

 printf("After Swapping the value of: a = %d, b = %d\n\n", a, b);

```

```
  return 0;
}
```

What is meant by structure padding?

In the case of structure or union, the compiler inserts some extra bytes between the members of structure or union for the alignment, these extra unused bytes are called padding bytes and this technique is called padding.

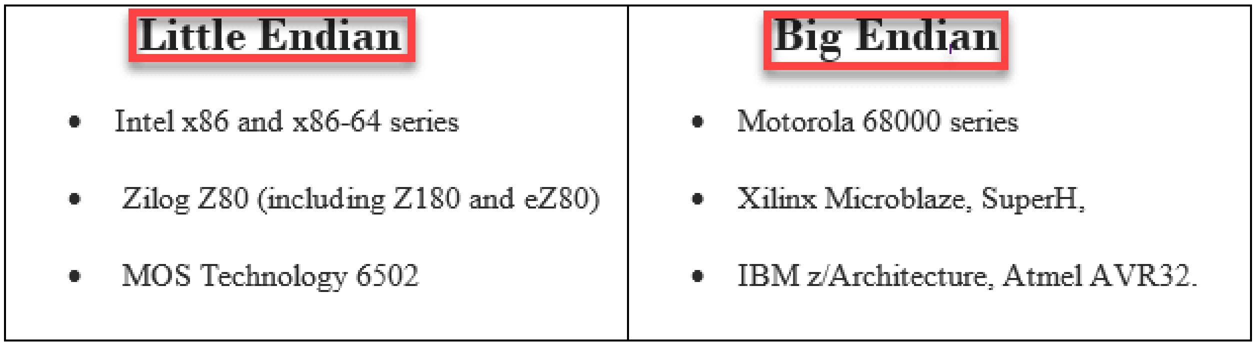
Padding has increased the performance of the processor at the penalty of memory.In structure or union data members aligned as per the size of the highest bytes member to prevent from the penalty of performance.

Note: *Alignment of data types mandated by the processor architecture, not by language.*

What is the endianness?

The endianness is the order of bytes to store data in memory and it also describes the order of byte transmission over a digital link. In memory data store in which order it depends on the endianness of the system, if the system is big-endian then the MSB byte store first (means at lower address) and if the system is little-endian then LSB byte store first (means at lower address).

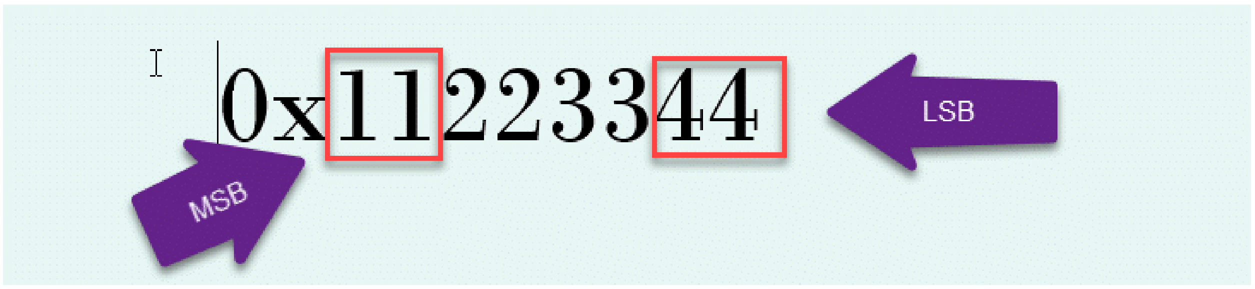
Some examples of the little-endian and big-endian system.



Little Endian	Big Endian
• Intel x86 and x86-64 series	• Motorola 68000 series
• Zilog Z80 (including Z180 and eZ80)	• Xilinx Microblaze, SuperH,
• MOS Technology 6502	• IBM z/Architecture, Atmel AVR32.

What is big-endian and little-endian?

Suppose, 32 bits Data is 0x11223344.



Big-endian

The most significant byte of data stored at the lowest memory address.

Address	Value
00	0x11
01	0x22
02	0x33
03	0x44

Little-endian

The least significant byte of data stored at the lowest memory address.

Address	Value
00	0x44
01	0x33
02	0x22
03	0x11

Note: *Some processor has the ability to switch one endianness to other endianness using the software means it can perform like both big endian or little endian at a time. This processor is known as the Bi-endian, here are some architecture (ARM version 3 and above, Alpha, SPARC) who provide the switchable endianness feature.*

Write a c program to check the endianness of the system.

Method 1:

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <inttypes.h>

int main(void)
{
uint32_t u32RawData;
uint8_t *pu8CheckData;
u32RawData = 0x11223344; //Assign data

pu8CheckData = (uint8_t *)&u32RawData; //Type cast

if (*pu8CheckData == 0x44) //check the value of lower address
{
printf("little-endian");
}
else if (*pu8CheckData == 0x11) //check the value of lower address
{
printf("big-endian");
}


return 0;
}
```

Method 2:

```
#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>

```

```
typedef union
{

uint32_t u32RawData;  // integer variable
uint8_t  au8DataBuff[4]; //array of character

}RawData;


int main(void)
{
RawData uCheckEndianess;
uCheckEndianess.u32RawData = 0x11223344; //assign the value

if (uCheckEndianess.au8DataBuff[0] == 0x44) //check the array first index value
{
printf("little-endian");
}
else if (uCheckEndianess.au8DataBuff[0] == 0x11) //check the array first index value
{
printf("big-endian");
}

return 0;
}
```

How to convert little endian to big endian vice versa in C?

Method 1:

```c
#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>

//Function to change the endianess
uint32_t ChangeEndianness(uint32_t u32Value)
{
uint32_t u32Result = 0;
u32Result |= (u32Value & 0x000000FF) << 24;
u32Result |= (u32Value & 0x0000FF00) << 8;
u32Result |= (u32Value & 0x00FF0000) >> 8;
u32Result |= (u32Value & 0xFF000000) >> 24;
return u32Result;
}


int main()
{
uint32_t u32CheckData  = 0x11223344;
uint32_t u32ResultData =0;
u32ResultData = ChangeEndianness(u32CheckData);  //swap the data
printf("0x%x\n",u32ResultData);
u32CheckData = u32ResultData;
u32ResultData = ChangeEndianness(u32CheckData);//again swap the data
printf("0x%x\n",u32ResultData);
return 0;
}
```

What is static memory allocation and dynamic memory allocation?

According to C standard, there are four storage duration, static, thread (C11), automatic, and allocated. The storage duration determines the lifetime of the object.

The static memory allocation:

Static Allocation means, an object has external or internal linkage or declared with static storage-class. It's initialized only once, prior to program startup and its lifetime is throughout the execution of the program. A global and static variable is an example of static memory allocation.

The dynamic memory allocation:

In C language, there are a lot of library functions (malloc, calloc, or realloc,..) which are used to allocate memory dynamically. One of the problems with dynamically allocated memory is that it is not destroyed by the compiler itself that means it is the responsibility of the user to deallocate the allocated memory.

When we allocate the memory using the memory management function, they return a pointer to the allocated memory block and the returned pointer is pointing to the beginning address of the memory block. If there is no space available, these functions return a null pointer.

What is the memory leak in C?

A memory leak is a common and dangerous problem. It is a type of resource leak. In C language, a memory leak occurs when you allocate a block of memory using the memory management function and forget to release it.

```
int main ()
{

  char * pBuffer = malloc(sizeof(char) * 20);

  /* Do some work */

  return 0; /*Not freeing the allocated memory*/
}
```

Note: *once you allocate a memory than allocated memory does not allocate to another program or process until it gets free.*

What is the difference between malloc and calloc?

The malloc and calloc are memory management functions. They are used to allocate memory dynamically. Basically, there is no actual difference between calloc and malloc except that the memory that is allocated by calloc is initialized with 0.

In C language,calloc function initialize the all allocated space bits with zero but malloc does not initialize the allocated memory. These both function also has a difference regarding their number of arguments, malloc take one argument but calloc takes two.

What is the purpose of realloc( )?

The realloc function is used to resize the allocated block of memory. It takes two arguments first one is a pointer to previously allocated memory and the second one is the newly requested size.

The calloc function first deallocates the old object and allocates again with newly specified size. If the new size is lesser to the old size, the contents of the newly allocated memory will be same as prior but if any bytes in the newly created object goes beyond the old size, the values of the exceeded size will be indeterminate.

Syntax:
void *realloc(void *ptr, size_t size);

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main ()
{
char *pcBuffer = NULL;
/* Initial memory allocation */
pcBuffer = malloc(8);

strcpy(pcBuffer, "aticle");
printf("pcBuffer = %s\n", pcBuffer);

/* Reallocating memory */
pcBuffer = realloc(pcBuffer, 15);

strcat(pcBuffer, "world");
printf("pcBuffer = %s\n", pcBuffer);

//free the allocated memory
free(pcBuffer);

return 0;
}
```

Output:

```
pcBuffer = aticle
pcBuffer = aticleworld
```

***Note:** It should be used for dynamically allocated memory but if a pointer is a null pointer, realloc behaves like the malloc function.*

What is the return value of malloc (0)?

If the size of the requested space is zero, the behavior will be implementation-defined. The return value of the malloc could be a null pointer or it shows the behavior of that size is some nonzero value.It is suggested by the standard to not use the pointer to access an object that is returned by the malloc while size is zero.

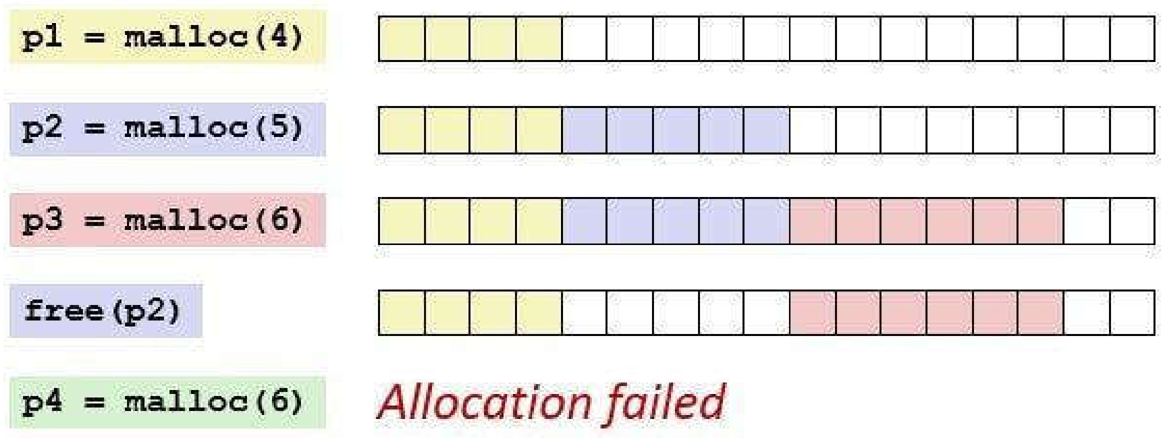
What is dynamic memory fragmentation?

The memory management function is guaranteed that if memory is allocated, then it would be suitably aligned to any object which has the fundamental alignment. The fundamental alignment is less than or equal to the largest alignment that's supported by the implementation without an alignment specification.

One of the major **problems with dynamic memory allocation** is fragmentation, basically, fragmentation occurred when the user does not use the memory efficiently. There are two types of fragmentation, external fragmentation, and internal fragmentation.

The external fragmentation is due to the small free blocks of memory (small memory hole) that is available on the free list but program not able to use it. There are different types of free list allocation algorithms that used the free memory block efficiently.

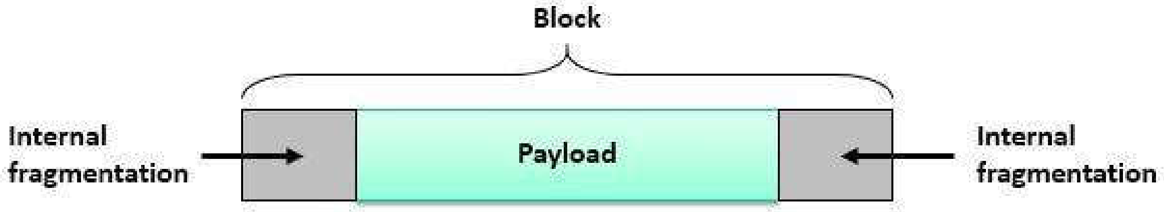
To understand the external fragmentation, consider a scenario where a program has 3 contiguous blocks of memory and the user frees the middle block of memory. In that scenario, you will not get a memory, if the required block of memory is larger than a single block of memory (but smaller or equal to the aggregate of the block of memory).



The internal fragmentation is wasted of memory that is allocated for rounding up the allocated memory and in bookkeeping (infrastructure), the bookkeeping is used to keep the information of the allocated memory.

Whenever we called the malloc function then it reserves some extra bytes (depend on implementation and system) for bookkeeping. This extra byte is reserved for each call of malloc and become a cause of the internal fragmentation.

If size of the payload is smaller than the block size, then internal fragmentation occur.



For example,
See the below code, the programmer may think that system will be allocated 8 *100 (800) bytes of memory but due to bookkeeping (if 8 bytes) system will be allocated 8*100 extra bytes. This is an internal fragmentation, where 50% of the heap waste.

```
char *acBuffer[100];

int main()
{
 int iLoop = 0;
 while(iLoop < 100)
 {
 acBuffer[iLoop ] =  malloc(8);
 ++iLoop;

 }

}
```

How is the free work in C?

When we call the memory management functions (malloc, calloc or realloc) then these functions keep extra bytes for bookkeeping.

Whenever we call the free function and pass the pointer that is pointing to allocated memory, the free function gets the bookkeeping information and release the allocated memory. Anyhow if you or your program change the value of the pointer that is pointing to the allocated address, the calling of free function give the undefined result.

```
____ The allocated block ____
/                          \
+--------+--------------------+
| Header | Your data area ... |
+--------+--------------------+
         ^
         |
+-- Returned Address
```

For example,

```
#include <stdio.h>
#include <stdlib.h>


int main()
{
char *pcBuffer = NULL;
pcBuffer  =  malloc(sizeof(char) *  16); //Allocate the memory

pcBuffer++; //Increment the pointer

free(pcBuffer); //Call free function to release the allocated memory

return 0;
}
```

What is a Function Pointer?

A **function pointer** is similar to the other pointers but the only difference is that it points to a function instead of the variable.

In the other word, we can say, a function pointer is a type of pointer that store the address of a function and these pointed function can be invoked by function pointer in a program whenever required.

How to declare a pointer to a function in c?

The syntax for declaring function pointer is very straightforward. It seems like difficult in beginning but once you are familiar with function pointer then it becomes easy.

The declaration of a pointer to a function is similar to the declaration of a function. That means function pointer also requires a return type, declaration name, and argument list. One thing that you need to remember here is, whenever you declare the function pointer in the program then declaration name is preceded by the * (Asterisk) symbol and enclosed in parenthesis.

***For example*,**

```
void ( *fpData )( int );
```

For the better understanding, let's take an example to describe the declaration of a function pointer in c.
e.g,

```
void ( *pfDisplayMessage) (const char *);
```

In above expression, pfDisplayMessage is a pointer to a function taking one argument, const char *, and returns void.

When we declare a pointer to function in c then there is a lot of importance of the bracket. If in the above example, I remove the bracket, then the meaning of the above expression will be change and it becomes void *pfDisplayMessage (const char *). It is a declaration of a function which takes the const character pointer as arguments and returns void pointer.

Where can the function pointers be used?

There are a lot of places, where the function pointers can be used. Generally, function pointers are used in the implementation of the callback function, finite state machine and to provide the feature of polymorphism in C language …etc.

Write a program to check an integer is a power of 2?

Here, I am writing a small algorithm to check the power of 2. If a number is a power of 2, function return 1.

```
int CheckPowerOftwo (unsigned int x)
{
  return ((x != 0) && !(x & (x - 1)));
}
```

What is the output of the below code?

```
#include <stdio.h>

int main()
{
   int x = -15;

   x = x << 1;

   printf("%d\n", x);
}
```

Output:

undefined behavior.

What is the output of the below code?

```
#include <stdio.h>

int main()
{
   int x = -30;

   x = x >> 1;

```

```
    printf("%d\n", x);
}
```

Output:

implementation-defined.

Write a program to count set bits in an integer?

```
unsigned int NumberSetBits(unsigned int n)
{
  unsigned int CountSetBits= 0;
  while (n)
  {
    CountSetBits += n & 1;
    n >>= 1;
  }
  return CountSetBits;
}
```

What is void or generic pointers in C?

A void pointer is a generic pointer. It has no associated data type that's why it can store the address of any type of object and type-casted to any types.

According to C standard, the pointer to void shall have the same representation and alignment requirements as a pointer to a character type.A void pointer declaration is similar to the normal pointer, but the difference is that instead of data types we use the void keyword.

Syntax:
void * Pointer_Name;

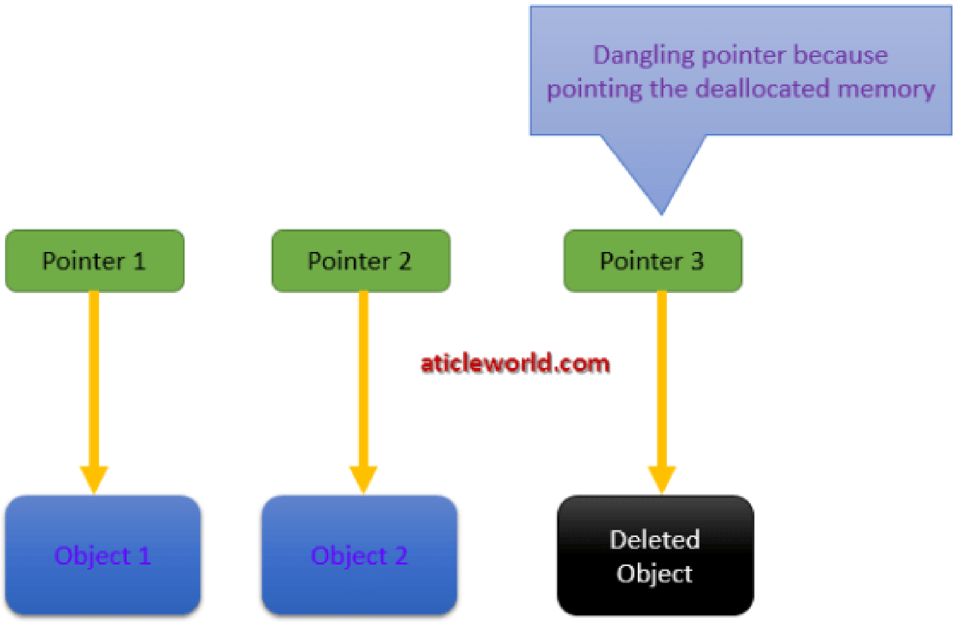
What is the advantage of a void pointer in C?

There are following advantages of a void pointer in c.

- Using the void pointer we can create a generic function that can take arguments of any data type. The memcpy and memmove library function are the best examples of the generic function, using these function we can copy the data from the source to destination.
  e.g.
  void * memcpy (void * dst, const void * src, size_t num);
- We have already know that void pointer can be converted to another data type that is the reason malloc, calloc or realloc library function return void *. Due to the void * these functions are used to allocate memory to any data type.
- Using the void * we can create a generic linked list.For more information see this link: How to create generic Link List.

What are dangling pointers?

Generally, **daggling pointers** arise when the referencing object is deleted or deallocated, without changing the value of the pointers. It creates the problem because the pointer is still pointing the memory that is not available. When the user tries to dereference the daggling pointers than it shows the undefined behavior and can be the cause of the segmentation fault.



For example,

```
#include<stdio.h>
#include<stdlib.h>

int main()
{
int *piData = NULL;

piData = malloc(sizeof(int)* 10); //creating integer of size 10.

free(piData); //free the allocated memory

*piData = 10; //piData is dangling pointer

return 0;

}
```

In simple word, we can say that dangling pointer is a pointer that not pointing a valid object of the appropriate type and it can be the cause of the undefined behavior.

$45.86~~$58.99~~

(330)

Elegoo EL-KIT-008 Mega 2560 Project The Most C…

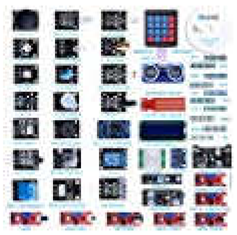
$59.99

(512)

LAFVIN UNO Project Super Starter Kit for Arduino UN…

$27.99~~$36.88~~

(106)



ELEGOO Upgraded 37 in 1 Sensor Modules Kit with Tu…

$27.99~~$42.99~~

(114)

Elegoo EL-CB-001 UNO R3 Board ATmega328P AT…

$11.86~~$15.99~~

What is the wild pointer?

A pointer that is not initialized properly prior to its first use is known as the wild pointer. Uninitialized pointers behavior is totally undefined because it may point some arbitrary location that can be the cause of the program crash, that's is the reason it is called a wild pointer.

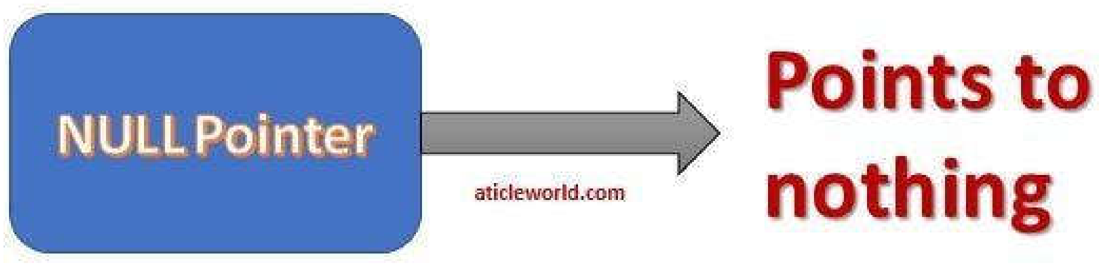
In the other word, we can say every pointer in programming languages that are not initialized either by the compiler or programmer begins as a wild pointer.

Note: *Generally, compilers warn about the wild pointer.*

Syntax,
int *piData; //piData is wild pointer.

What is a NULL pointer?

According to C standard, an integer constant expression with the value 0, or such an expression cast to type void *, is called a null pointer constant. If a null pointer constant is converted to a pointer type, the resulting pointer, called a null pointer.



***Syntax*,**
int *piData = NULL; // piData is a null pointer

What are the post increment and decrement operators?

When we use post-increment (++) operator on an operand then the result is the value of the operand and after getting the result, the value of the operand is incremented by 1. The working of the post-decrement (-) operator is similar to the post-increment operator but the difference is that the value of the operand is decremented by 1.

Note*: incrementation and decrementation by 1 are the types specified.*

Which one is better: Pre-increment or Post increment?

Nowadays compiler is enough smart, they optimize the code as per the requirements. The post and pre increment both have own importance we need to use them as per the requirements.

If you are reading a flash memory byte by bytes through the character pointer then here you have to use the post-increment, either you will skip the first byte of the data. Because we already know that in case of pre-increment pointing address will be increment first and after that, you will read the value.

Let’s take an example of the better understanding,
In below example code, I am creating a character array and using the character pointer I want to read the value of the array. But what will happen if I used pre-increment operator? The answer to this question is that ‘A’ will be skipped and B will be printed.

```
#include <stdio.h>

int main(void)
{

 char acData[5] ={'A','B','C','D','E'};
 char *pcData = NULL;

 pcData = acData;

 printf("%c ",*++pcData);

 return 0;
}
```

But in place of pre-increment if we use post-increment then the problem is getting solved and you will get A as the output.

```
#include <stdio.h>

int main(void)
{

 char acData[5] ={'A','B','C','D','E'};
 char *pcData = NULL;

 pcData = acData;

 printf("%c ",*pcData++);

 return 0;
}
```

Besides that, when we need a loop or just only need to increment the operand then pre-increment is far better than post-increment because in case of post increment compiler may have created a copy of old data which takes extra time. This is not 100% true because nowadays compiler is so smart and they are optimizing the code in a way that makes no difference between pre and post-increment. So it is my advice, if post-increment is not necessary then you have to use the pre-increment.

Note: Generally post-increment is used with array subscript and pointers to read the data, otherwise if not necessary then use pre in place of post-increment.Some compiler also mentioned that to avoid to use post-increment in looping condition.
iLoop = 0.

```
while (a[iLoop ++] != 0)
{
// Body statements
}
```

Are the expressions *ptr ++ and ++*ptr same ?

Both expressions are different. Let’s see a sample code to understand the difference between both expressions.

```
#include <stdio.h>

int main(void)
{
   int aiData[5] = {100,200,300,400,500};

 int *piData = aiData;

   ++*piData;

 printf("aiData[0] = %d, aiData[1] = %d, *piData = %d", aiData[0], aiData[1], *piData);

   return 0;
}
```

Output: 101 , 200 , 101

Explanation:
In the above example, two operators are involved and both have the same precedence with a right to left associativity. So the above expression ++*p is

equivalent to ++ (*p). In another word, we can say it is pre-increment of value and output is 101, 200, 101.

```
#include <stdio.h>

int main(void)
{
   int aiData[5] = {100,200,30,40,50};

 int *piData = aiData;

   *++piData;

 printf("aiData[0] = %d, aiData[1] = %d, *piData = %d", aiData[0], aiData[1], *piData);

   return 0;
}
```

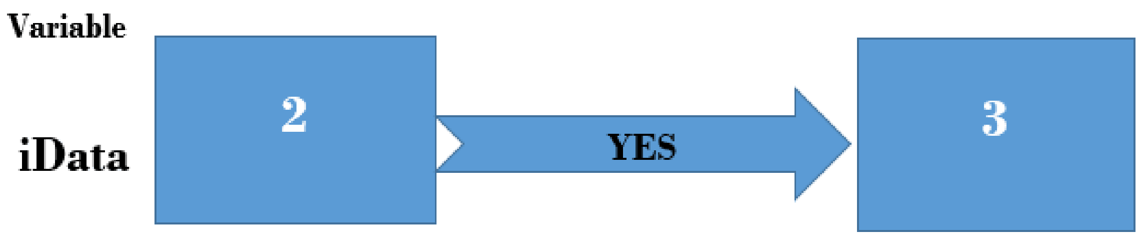
Output: 100, 200, 200

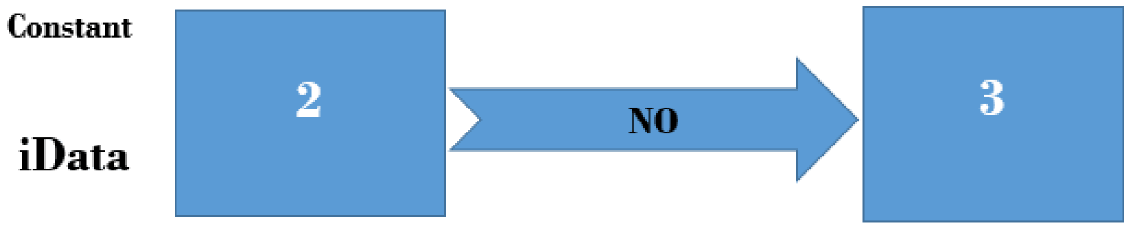
Explanation:
In the above example, two operators are involved and both have the same precedence with the right to left associativity. So the above expression *++p is equivalent to *(++p). In another word you can say it is pre-increment of address and output is 100, 200,200.

What does the keyword const mean?

A const is only a qualifier, it changes the behavior of a variable and makes it read-only type. When we want to make an object read-only type, then we have to declare it as const.

Syntax
const DataType Identifier = Value;
e.g.
const int iData = 0





At the time of declaration, const qualifier only gives the direction to the compiler that the value of declaring object could not be changed. In simple word, const means not modifiable (cannot assign any value to the object at the runtime).

When should we use const in a C program?

There are following places where we need to use the const keyword in the programs.

- In call by reference function argument, if you don't want to change the actual value which has passed in function.

Eg.
int PrintData (const char *pcMessage);

- In some places, const is better then macro because const handle by the compiler and have a type checking.
  Eg.
  const int ciData = 100;
- In the case of I/O and memory mapped register const is used with the volatile qualifier for efficient access.
  Eg.
  const volatile uint32_t *DEVICE_STATUS = (uint32_t *) 0x80102040;
- When you don't want to change the value of an initialized variable.

What is the meaning of below declarations?

1. const int a;
2. int const a;
3. const int *a;
4. int * const a;
5. int const * a const;

1. The "a" is a constant integer.
2. Similar to first, "a" is a constant integer.
3. Here "a" is a pointer to a const integer, the value of the integer is not modifiable, but the pointer is not modifiable.
4. Here "a" is a const pointer to an integer, the value of the pointed integer is modifiable, but the pointer is not modifiable.
5. Here "a" is a const pointer to a const integer that means the value of pointed integer and pointer both are not modifiable.

Differentiate between a constant pointer and pointer to a constant?

Constant pointer:

A constant pointer is a pointer whose value (pointed address) is not modifiable. If you will try to modify the pointer value, you will get the compiler error.

A constant pointer is declared as follows :
Data_Type * const Pointer_Name;

Let’s see the below example code when you will compile the below code get the compiler error.

```
#include<stdio.h>

int main(void)
{
   int var1 = 10, var2 = 20;

   //Initialize the pointer
   int *const ptr = &var1;

   //Try to modify the pointer value
   ptr = &var2;

   printf("%d\n", *ptr);

   return 0;
}
```

Pointer to a constant:

In this scenario the value of pointed address is constant that means we can not change the value of the address that is pointed by the pointer.

A constant pointer is declared as follows :
Data_Type const* Pointer_Name;

Let’s take a small code to illustrate a pointer to a constant:

```
#include<stdio.h>

int main(void)
{
   int var1 = 100;
   // pointer to constant integer
   const int* ptr = &var1;

   //try to modify the value of pointed address
   *ptr = 10;

   printf("%d\n", *ptr);

   return 0;
}
```

What are the uses of the keyword static?

In C language, the static keyword has a lot of importance. If we have used the static keyword with a variable or function, then only internal or none linkage is worked. I have described some simple use of a static keyword.

- A static variable only initializes once, so a variable declared static within the body of a function maintains its prior value between function invocations.
- A global variable with static keyword has an internal linkage, so it only accesses within the translation unit (.c). It is not accessible by another translation unit. The static keyword protects your variable to access from another translation unit.
- By default in C language, linkage of the function is external that it means it is accessible by the same or another translation unit. With the help of the static keyword, we can make the scope of the function local, it only accesses by the translation unit within it is declared.

What is the difference between global and static global variables?

In simple word, they have different linkage.

A static global variable ===>>> **internal linkage.**
A non-static global variable ===>>> **external linkage.**

So global variable can be accessed outside of the file but the static global variable only accesses within the file in which it is declared.

Differentiate between an internal static and external static variable?

In C language, the external static variable has the internal linkage and internal static variable has no linkage. So the life of both variable throughout the program but scope will be different.

A external static variable ===>>> **internal linkage.**
A internal static variable ===>>> **none .**

Can static variables be declared in a header file?

Yes, we can declare the static variables in a header file.

What is the difference between declaration and definition of a variable?

Declaration of variable in c

A variable declaration only provides sureness to the compiler at the compile time that variable exists with the given type and name, so that compiler proceeds for further compilation without needing all detail of this variable. In C language, when we declare a variable, then we only give the information to the compiler, but there is no memory reserve for it. It is only a reference, through which we only assure to the compiler that this variable may be defined within the function or outside of the function.

Note: *We can declare a variable multiple time but defined only once.*
eg,
extern int data;
extern int foo(int, int);
int fun(int, char); // extern can be omitted for function declarations

Definition of variable in c

The definition is action to allocate storage to the variable. In another word, we can say that variable definition is the way to say the compiler where and how much to create the storage for the variable generally definition and declaration occur at the same time but not almost.

eg,
int data;
int foo(int, int) { }

Note: *When you define a variable then there is no need to declare it but vice versa is not applicable.*

What is the difference between pass by value by reference in c and pass by reference in c?

Pass By Value:

- In this method value of the variable is passed. Changes made to formal will not affect the actual parameters.
- Different memory locations will be created for both variables.
- Here there will be temporary variable created in the function stack which does not affect the original variable.

Pass By Reference :

- In Pass by reference, an address of the variable is passed to a function.
- Whatever changes made to the formal parameter will affect the value of actual parameters(a variable whose address is passed).
- Both formal and actual parameter shared the same memory location.
- it is useful when you required to returns more than 1 values.

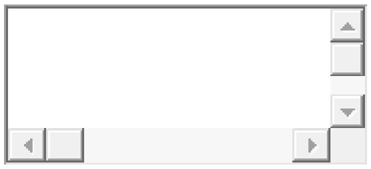
What is a reentrant function?

In computing, a computer program or subroutine is called reentrant if it can be interrupted in the middle of its execution and then safely be called again ("re-entered") before its previous invocations complete execution. The interruption could be caused by an internal action such as a jump or call, or by an external action such as an interrupt or signal. Once the reentered invocation completes, the previous invocations will resume correct execution.

What is the inline function?

An inline keyword is a compiler directive that only suggests the compiler to substitute the body of the function at the calling the place. It is an optimization technique used by the compilers to reduce the overhead of function calls.

***for example*,**



```
static inline void Swap(int *a, int *b)
{
  int tmp= *a;
  *a= *b;
  *b = tmp;
}
```

What is the advantage and disadvantage of the inline function?

There are few important advantage and disadvantage of the inline function.

Advantages:-
1) It saves the function calling overhead.
2) It also saves the overhead of variables push/pop on the stack, while function calling.
3) It also saves the overhead of return call from a function.
4) It increases locality of reference by utilizing instruction cache.
5) After inlining compiler can also apply intraprocedural optimization if specified. This is the most important one, in this way compiler can now focus on dead code elimination, can give more stress on branch prediction, induction variable elimination etc..

Disadvantages:-
1) May increase function size so that it may not fit in the cache, causing lots of cache miss.
2) After inlining function, if variables number which are going to use register increases than they may create overhead on register variable resource utilization.
3) It may cause compilation overhead as if somebody changes code inside an inline function then all calling location will also be compiled.
4) If used in the header file, it will make your header file size large and may also make it unreadable.
5) If somebody used too many inline functions resultant in a larger code size than it may cause thrashing in memory. More and number of page fault bringing down your program performance.
6) It’s not useful for an embedded system where large binary size is not preferred at all due to memory size constraints.

What is the virtual memory?

The virtual memory is the part of memory management techniques and it creates an illusion that the system has a sufficient amount memory.In another word you can say that virtual memory is a layer of indirection.

Consider the two statements and find the difference between them?

```
struct sStudentInfo {

char Name[12];
int Age;
float Weight;
int RollNumber;

};


#define STUDENT_INFO struct sStudentInfo*

typedef struct sStudentInfo* studentInfo;

statement 1
STUDENT_INFO p1, p2;

statement 2
studentInfo q1, q2;
```

Both statements looking same but actually, both are different to each other.

Statement 1 will be expanded to struct sStudentInfo * p1, p2. It means that p1 is a pointer to struct sStudentInfo but p2 is a variable of struct sStudentInfo.

In statement 2, both q1 and q2 will be a pointer to struct sStudentInfo.

What are the limitations of I2C interface?

- Half duplex communication, so data is transmitted only in one direction (because of the single data bus) at a time.
- Since the bus is shared by many devices, debugging an I2C bus (detecting which device is misbehaving) for issues is pretty difficult.
- The I2C bus is shared by multiple slave devices if anyone of these slaves misbehaves (pull either SCL or SDA low for an indefinite time) the bus will be stalled. No further communication will take place.
- I2C uses resistive pull-up for its bus. Limiting the bus speed.
- Bus speed is directly dependent on the bus capacitance, meaning longer I2C bus traces will limit the bus speed.

Here, I have mentioned some questions for you. If you know, please write in comment box. Might be your comment helpful for other.

- What is the difference between flash memory, EPROM, and EEPROM?
- What is the difference between Volatile & Non Volatile Memory?
- What are the differences between a union and a structure in C?
- What is the difference between RS232 and UART?
- Is it possible to declare struct and union one inside other? Explain with example.
- How to find the bug in code using the debugger if the pointer is pointing to an illegal value.
- What is watchdog timer?
- What is the DMA?
- What is RTOS?
- What are CAN and its uses?
- Why is CAN having 120 ohms at each end?
- Why is CAN message-oriented protocol?
- What is the Arbitration in the CAN?
- Standard CAN and Extended CAN difference?
- What is the use of bit stuffing?
- How many types of IPC mechanism do you know?
- What is semaphore?
- What is the spinlock?
- Convert a given decimal number to hex.

- What is the difference between heap and stack memory?
- What is socket programming?
- How can a double pointer be useful?
- What is the difference between binary semaphore and mutex?

100 embedded c interview questions, your interviewer might ask

BY **AMLENDRA**ON

31

In my previous post, I have created a collection of “c interview questions” that is liked by many people. I have also get the response to create a list of interview questions on “embedded c”. So here I have tried to create some collection of questions that might be asked by your interviewer.

What is the volatile keyword?

The volatile keyword is a type qualifier that prevents the objects from the compiler optimization. According to C standard, an object that has volatile-qualified type may be modified in ways unknown to the implementation or have other unknown side effects. You can also say that the value of the volatile-qualified object can be changed at any time without any action being taken by the code. If an object is qualified by the volatile qualifier, the compiler reloads the value from memory each time it is accessed by the program that means it prevents from to cache a variable into a register. Reading the value from the memory is the only way to check the unpredictable change of the value.

What is the use of volatile keyword?

The volatile keyword is mainly used where we directly deal with GPIO, interrupt or flag Register. It is also used where a global variable or buffer is shared between the threads.

What is the difference between the const and volatile qualifier in C?

The const keyword is compiler-enforced and says that program could not change the value of the object that means it makes the object nonmodifiable type.
e.g,
const int a = 0;
if you will try to modify the value of “a”, you will get the compiler error because “a” is qualified with const keyword that prevents to change the value of the integer variable.

In another side volatile prevent from any compiler optimization and says that the value of the object can be changed by something that is beyond the control of the program and so that compiler will not make any assumption about the object.
e.g,
volatile int a;

When the compiler sees the above declaration then it avoids to make any assumption regarding the “a” and in every iteration read the value from the address which is assigned to the variable.

Can a variable be both constant and volatile in C?

Yes, we can use both constant and volatile together. One of the great use of volatile and const keyword together is at the time of accessing the GPIO registers. In case of GPIO, its value can be changed by the ‘external factors’ (if a switch or any output device is attached with GPIO), if it is configured as an input. In that situation, volatile plays an important role and ensures that the compiler always read the value from the GPIO address and avoid to make any assumption.

After using the volatile keyword, you will get the proper value whenever you are accessing the ports but still here is one more problem because the pointer is not const type so it might be your program change the pointing address of the pointer. So we have to create a constant pointer with volatile keyword.

Syntax of declaration,

int volatile * const PortRegister;

How to read the above declaration,

```
int volatile * const PortRegister;
|   |   | |  |
|   |   | |  +------> PortRegister is a
|   |   | +-----------> constant
|   |   +---------------> pointer to a
|   +---------------------> volatile
+---------------------------> integer
```

Can we have a volatile pointer?

Yes, we can create a volatile pointer in C language.
int * volatile piData; // piData is a volatile pointer to an integer.

The Proper place to use the volatile keyword?

Here I am pointing some important places where we need to use the volatile keyword.

- Accessing the memory-mapped peripherals register or hardware status register.

```
#define COM_STATUS_BIT  0x00000006

uint32_t const volatile * const pStatusReg = (uint32_t*)0x00020000;


unit32_t GetRecvData()
{
 //Code to recv data
   while (((*pStatusReg)  & COM_STATUS_BIT) == 0)
 {
       // Wait until flag does not set
   }

   return RecvData;
}
```

- Sharing the global variables or buffers between the multiple threads.
- Accessing the global variables in an interrupt routine or signal handler.

```
volatile int giFlag = 0;

```

```
ISR(void)
{
  giFlag = 1;
}

int main(void)
{

  while (!giFlag)
  {
     //do some work
  }

  return 0;
}
```

What is ISR?

An ISR refers to the Interrupt Service Routines. These are procedures stored at specific memory addresses which are called when a certain type of interrupt occurs. The Cortex-M processors family has the NVIC that manage the execution of the interrupt.

Can we pass any parameter and return a value from the ISR?

An ISR returns nothing and not allow to pass any parameter. An ISR is called when a hardware or software event occurs, it is not called by the code, so that’s the reason no parameters are passed into an ISR.

In above line, we have already read that the ISR is not called by the code, so there is no calling code to read the returned values of the ISR. It is the reason that an ISR is not returned any value.

What is interrupt latency?

It is an important question that is asked by the interviewer to test the understanding of Interrupt. Basically, interrupt latency is the number of clock cycles that is taken by the processor to respond to an interrupt request. These number of the clock cycle is count between the assertions of the interrupt request and first instruction of the interrupt handler.

Interrupt Latency on the Cortex-M processor family

The Cortex-M processors have the very low interrupt latency. In below table, I have mentioned, Interrupt latency of Cortex-M processors with zero wait state memory systems.

Processors	**Cycles with zero wait state memory**
Cortex-M0	16
Cortex-M0+	15
Cortex-M3	12
Cortex-M4	12
Cortex-M7	12

How do you measure interrupt latency?

With the help of the oscilloscope, we can measure the interrupt latency. You need to take following steps.

- First takes two GPIOs.
- Configure one GPIO to generate the interrupt and second for the toggling (if you want you can attach an LED).
- Monitor the PIN (using the oscilloscope or analyzer) which you have configured to generate the interrupt.
- Also, monitor (using the oscilloscope or analyzer) the second pin which is toggled at the beginning of the interrupt service routine.
- When you will generate the interrupt then the signal of the both GPIOs will change.

The interval between the two signals (interrupt latency) may be easily read from the instrument.

How to reduce the interrupt latency?

The interrupt latency depends on many factors, some factor I am mentioning in below statements.

- Platform and interrupt controller.
- CPU clock speed.
- Timer frequency
- Cache configuration.
- Application program.

So using the proper selection of platform and processor we can easily reduce the interrupt latency. We can also reduce the interrupt latency by making the ISR shorter and avoid to calling a function within the ISR.

What are the causes of Interrupt Latency?

- The first delay is typically caused by hardware: The interrupt request signal needs to be synchronized to the CPU clock. Depending on the synchronization logic, up to 3 CPU cycles may expire before the interrupt request has reached the CPU core.
- The CPU will typically complete the current instruction, which may take several cycles. On most systems, divide, push-multiple or memory-copy instructions are the most time-consuming instructions to execute. On top of the cycles required by the CPU, additional cycles are often required for memory accesses. In an ARM7 system, the instruction STMDB SP!,{R0-R11, LR} typically is the worst case instruction, storing 13 registers of 32-bits each to the stack, and takes 15 clock cycles to complete.
- The memory system may require additional cycles for wait states.
- After completion of the current instruction, the CPU performs a mode switch or pushes registers on the stack (typically PC and flag registers). Modern CPUs such as ARM generally perform a mode switch, which takes fewer CPU cycles than saving registers.
- Pipeline fill: Most modern CPUs are pipelined. Execution of an instruction happens in various stages of the pipeline. An instruction is executed when it has reached its final stage of the pipeline. Since the mode switch has flushed the pipeline, a few extra cycles are required to refill the pipeline.

What is nested interrupt?

In a nested interrupt system, an interrupt is allowed to any time and anywhere even an ISR is being executed. But, only the highest priority ISR will be executed immediately. The second highest priority ISR will be executed after the highest one is completed.

The rules of a nested interrupt system are:

- All interrupts must be prioritized.
- After initialization, any interrupts are allowed to occur anytime and anywhere.
- If a low-priority ISR is interrupted by a high-priority interrupt, the high-priority ISR is executed.
- If a high-priority ISR is interrupted by a low-priority interrupt, the high-priority ISR continues executing.

- The same priority ISRs must be executed by time order

What is NVIC in ARM Cortex?

The Nested Vector Interrupt Controller (NVIC) in the Cortex-M processor family is an example of an interrupt controller with extremely flexible interrupt priority management. It enables programmable priority levels, automatic nested interrupt support, along with support for multiple interrupt masking, whilst still being very easy to use by the programmer.

The Cortex-M3 and Cortex-M4 processors the NVIC supports up to 240 interrupt inputs, with 8 up to 256 programmable priority levels

---

Can we use any function inside ISR?

Yes, you can call a function within the ISR but it is not recommended because it can increase the interrupt latency and decrease the performance of the system. If you want to call a nested function within the ISR, you need to read the datasheet of your microcontroller because some vendors have a limit to how many calls can be nested.

One important point also needs to remember that function which is called from the ISR should be re-entrant. If the called function is not re-entrant, it could create the issues.

For example,
If the function is not reentrant and supposes that it is called by another part of the code beside the ISR. So the problem will be invoked when if the ISR calls the same function which is already invoked outside of the ISR?

Can we change the interrupt priority level of Cortex-M processor family?

Yes, we can.

What is the start-up code?

A start-up code is called prior to the main function, it creates a basic platform for the application. It is a small block of code that is written in assembly language.

There are following parts of the start-up code.

- Declaration of the Stack area.
- Declaration of the Heap area.
- Vector table.
- Reset handler code.
- Other exception handler code.

Making Embedded Systems: Design Patterns for Grea…

$17.59~~$39.99~~

(52)

Test Driven Development for Embedded C (Pragmatic Pr…

$26.80~~$34.95~~

(32)

C Programming Language, 2nd Edition

$62.85~~$67.00~~

(610)

Embedded C Programming: Techniques and Applicatio…

$50.83~~$59.95~~

(3)

Embedded Systems with ARM Cortex-M Microcon…

$69.50

(46)

The AVR Microcontroller and Embedded Systems Using …

$25.00

(25)

Embedded C

$44.14~~$64.99~~

(12)

Programming Embedded Systems in C and C++

$42.36

(29)

What are the start-up code steps?

Start-up code for C programs usually consists of the following actions, performed in the order described:

- Disable all interrupts.
- Copy any initialized data from ROM to RAM.
- Zero the uninitialized data area.
- Allocate space for and initialize the stack.
- Initialize the processor's stack pointer.
- Create and initialize the heap.
- Enable interrupts.
- Call main.

Infinite loops often arise in embedded systems. How do you code an infinite loop in C?

In embedded systems, infinite loops are generally used. If I talked about a small program to control a led through the switch, in that scenario an infinite loop will be required if we are not going through the interrupt.

There are the different way to create an infinite loop, here I am mentioning some methods.

Method 1:

```
while(1)
{
// task
}
```

Method 2:

```
for(;;)
{
```

```
// task
}
```

Method 3:

```
Loop:
goto Loop;
```

How to access the fixed memory location in embedded C?

It is a very basic question that is generally asked by the interviewer.

Let’s take an example:
Suppose in an application, you have required accessing a fixed memory address.

```
//Memory address, you want to access
#define RW_FLAG 0x1FFF7800

//Pointer to access the Memory address
volatile uint32_t *flagAddress = NULL;

//variable to stored the read value
uint32_t readData = 0;

//Assign addres to the pointer
flagAddress = (volatile uint32_t *)RW_FLAG;

//Read value from memory
* flagAddress = 12; // Write

//Write value to the memory
readData = * flagAddress;
```

Difference between RISC and CISC processor?

The RISC (reduced instruction set computer) and CISC (Complex instruction set computer) are the processors ISA (instruction set architecture).

There are following difference between both architecture:

	RISC	CISC
Acronym	It stands for ‘Reduced Instruction Set Computer’.	It stands for ‘Complex Instruction Set Computer’.
Definition	The RISC processors have a smaller set of instructions with few addressing nodes.	The CISC processors have a larger set of instructions with many addressing nodes.
Memory unit	It has no memory unit and uses a separate hardware to implement instructions.	It has a memory unit to implement complex instructions.
Program	It has a hard-wired unit of programming.	It has a micro-programming unit.
Design	It is a complex complier design.	It is an easy complier design.
Calculations	The calculations are faster and precise.	The calculations are slow and precise.
Decoding	Decoding of instructions is simple.	Decoding of instructions is complex.

Time	Execution time is very less.	Execution time is very high.
External memory	It does not require external memory for calculations.	It requires external memory for calculations.
Pipelining	Pipelining does function correctly.	Pipelining does not function correctly.
Stalling	Stalling is mostly reduced in processors.	The processors often stall.
Code expansion	Code expansion can be a problem.	Code expansion is not a problem.
Disc space	The space is saved.	The space is wasted.
Applications	Used in high-end applications such as video processing, telecommunications and image processing.	Used in low-end applications such as security systems, home automations, etc.

Images Courtesy: ics.uci.edu

What is the stack overflow?

If your program tries to access the beyond the limit of the available stack memory then stack overflow occurs. In another word you can say that a stack overflow occurs if the call stack pointer exceeds the stack boundary.

If stack overflow occurs, the program can crash or you can say that segmentation fault that is the result of the stack overflow.

What is the cause of the stack overflow?

In the embedded application we have a little amount of stack memory as compare to the desktop application. So we have to work on embedded application very carefully either we can face the stack overflow issues that can be a cause of the application crash.

Here, I have mentioned some causes of unwanted use of the stack.

- Improper use of the recursive function.
- Passing to much arguments in the function.
- Passing a structure directly into a function.
- Nested function calls.
- Creating a huge size local array.

What is the difference between I2c and SPI communication Protocol?

In the embedded system, I2C and SPI both play an important role. Both communication protocols are the example of the synchronous communication but still, both have some important difference.

The important difference between the I2C and SPI communication protocol.

- I2C support half duplex while SPI is the full duplex communication.
- I2C requires only two wire for communication while SPI requires three or four wire for communication (depends on requirement).
- I2C is slower as compared to the SPI communication.
- I2C draws more power than SPI.
- I2C is less susceptible to noise than SPI.
- I2C is cheaper to implement than the SPI communication protocol.
- I2C work on wire and logic and it has a pull-up resistor while there is no requirement of pull-up resistor in case of the SPI.
- In I2C communication we get the acknowledgment bit after each byte, it is not supported by the SPI communication protocol.
- I2C ensures that data sent is received by the slave device while SPI does not verify that data is received correctly.

- I2C support the multi-master communication while multi-master communication is not supported by the SPI.
- One great difference between I2C and SPI is that I2C supports multiple devices on the same bus without any additional select lines (work on the basis of device address) while SPI requires additional signal (slave select lines) lines to manage multiple devices on the same bus.
- I2C supports arbitration while SPI does not support the arbitration.
- I2C support the clock stretching while SPI does not support the clock stretching.
- I2C can be locked up by one device that fails to release the communication bus.
- I2C has some extra overhead due to start and stop bits.
- I2C is better for long distance while SPI is better for the short distance.
- In the last I2C developed by NXP while SPI by Motorola.

What is the difference between Asynchronous and Synchronous Communication?

There are following difference between the asynchronous and synchronous communication.

Asynchronous Communication	**Synchronous Communication**
There is no common clock signal between the sender and receivers.	Communication is done by a shared clock.
Sends 1 byte or character at a time.	Sends data in the form of blocks or frames.
Slow as compare to synchronous communication.	Fast as compare to asynchronous communication.
Overhead due to start and stop bit.	Less overhead.

Ability to communicate long distance.	Less as compared to asynchronous communication.
A start and stop bit used for the data synchronization.	A shared clock is used for the data synchronization.
Economical	Costly
RS232, RS485	I2C, SPI.

What is the difference between RS232 and RS485?

The RS232 and RS485 is an old serial interface. Both serial interfaces are the standard for the data communication. This question is also very important and generally ask by an interviewer.

Some important difference between the RS232 and RS485

Parameter	**RS232**	**RS485**
Line configuration	Single –ended	differential
Numbers of devices	1 transmitter 1 receiver	32 transmitters 32 receivers
Mode of operation	Simplex or full duplex	Simplex or half duplex
Maximum cable length	50 feet	4000 feet

Maximum data rate	20 Kbits/s	10 Mbits/s
signaling	unbalanced	balanced
Typical logic levels	+-5 ~ +-15V	+-1.5 ~ +-6V
Minimum receiver input impedance	3 ~ 7 K-ohm	12 K-ohm
Receiver sensitivity	+-3V	+-200mV

What is the difference between Bit Rate and Baud Rate?

Bit Rate	**Baud Rate**
Bit rate is the number of bits per second.	Baud rate is the number of signal units per second.
It determines the number of bits traveled per second.	It determines how many times the state of a signal is changing.
Cannot determine the bandwidth.	It can determine how much bandwidth is required to send the signal.
This term generally used to describe the processor efficiency.	This term generally used to describe the data transmission over the channel.

Bit rate = baud rate x the number of bits per signal unit Baud rate = bit rate / the number of bits per signal unit

What is segmentation fault in C?

A segmentation fault is a common problem that causes programs to crash. A core file (core dumped file) also associated with segmentation fault that is used by the developer to finding the root cause of the crashing (segmentation fault).

Generally, the segmentation fault occurs when a program tried to access a memory location that it is not allowed to access or tried to access a memory location in a way that is not allowed (tried to access read-only memory).

What are the common causes of segmentation fault in C?

There are many reasons for the segmentation fault, here I am listing some common causes of the segmentation fault.

- Dereferencing NULL pointers.
- Tried to write read-only memory (such as code segment).
- Trying to access a nonexistent memory address (outside process's address space).
- Trying to access memory the program does not have rights to (such as kernel structures in process context).
- Sometimes dereferencing or assigning to an uninitialized pointer (because might point an invalid memory) can be the cause of the segmentation fault.
- Dereferencing the freed memory (after calling the free function) can also be caused by the segmentation fault.
- A stack overflow is also caused by the segmentation fault.
- A buffer overflow (try to access the array beyond the boundary) is also cause of the segmentation fault.

What is the difference between Segmentation fault and Bus error?

In case of segmentation fault, SIGSEGV (11) signal is generated. Generally, a segmentation fault occurs when the program tries to access the memory to which it doesn't have access to.

In below I have mentioned some scenarios where SIGSEGV signal is generated.

- When trying to de-referencing a NULL pointer.
- Trying to access memory which is already de-allocated (trying to use dangling pointers).
- Using uninitialized pointer(wild pointer).
- Trying to access memory that the program doesn't own (eg. trying to access an array element out of array bounds).

In case of a BUS error, SIGBUS (10) signal is generated. The Bus error issue occurs when a program tries to access an invalid memory or unaligned memory. The bus error comes rarely as compared to the segmentation fault.

In below I have mentioned some scenarios where SIGBUS signal is generated.

- Non-existent address.
- Unaligned access.
- Paging errors

Size of the integer depends on what?

The C standard is explained that the minimum size of the integer should be 16 bits. Some programing language is explained that the size of the integer is implementation dependent but portable programs shouldn't depend on it.

Primarily size of integer depends on the type of the compiler which has written by compiler writer for the underlying processor. You can see compilers merrily changing the size of integer according to convenience and underlying architectures. So it is my recommendation use the C99 integer data types (uin8_t, uin16_t, uin32_t ..) in place of standard int.

Are integers signed or unsigned?

In standard C language, integer data type is by default signed. So if you create an integer variable, it can store both positive and negative value.

For more details on signed and unsigned integer, check out:
A closer look at signed and unsigned integers in C

What is a difference between unsigned int and signed int in C?

The signed and unsigned integer type has the same storage (according to the standard at least 16 bits) and alignment but still, there is a lot of difference them, in bellows lines, I am describing some difference between the signed and unsigned integer.

- A signed integer can store the positive and negative value both but beside it unsigned integer can only store the positive value.
- The range of nonnegative values of a signed integer type is a sub-range of the corresponding unsigned integer type.
  For example,
  Assuming size of the integer is 2 bytes.
  signed int -32768 to +32767
  unsigned int 0 to 65535
- When computing the unsigned integer, it never gets overflow because if the computation result is greater than the largest value of the unsigned integer type, it is reduced modulo the number that is one greater than the

largest value that can be represented by the resulting type.
For example,
Computational Result % (Largest value of the unsigned integer+1)

- The overflow of signed integer type is undefined.
- If Data is signed type negative value, the right shifting operation of Data is implementation dependent but for the unsigned type, it would be Data/ 2pos.
- If Data is signed type negative value, the left shifting operation of Data show the undefined behavior but for the unsigned type, it would be Data x 2pos.

What is the difference between a macro and a function?

What is the difference between typedef & Macros?

typedef:

The C language provides a very important keyword typedef for defining a new name for existing types. The typedef is the compiler directive mainly use with user-defined data types (structure, union or enum) to reduce their complexity and increase the code readability and portability.

***Syntax*,**
typedef type NewTypeName;

Let’s take an example,
typedef unsigned int UnsignedInt;

Now UnsignedInt is a new type and using it, we can create a variable of unsigned int.

UnsignedInt Mydata;
In above example, Mydata is variable of unsigned int.

Note: *A typedef creates synonyms or a new name for existing types it does not create new types.*

Macro:

A macro is a pre-processor directive and it replaces the value before compiling the code.One of the major problem with the macro that there is no type checking. Generally, the macro is used to create the alias, in C language macro is also used as a file guard.

Syntax,

#define Value 10

Now Value becomes 10, in your program, you can use the Value in place of the 10.

What do you mean by enumeration in C?

In C language enum is user-defined data type and it consists a set of named constant integer. Using the enum keyword, we can declare an enumeration type by using the enumeration tag (optional) and a list of named integer.

An enumeration increases the readability of the code and easy to debug in comparison of symbolic constant (macro).

The most important thing about the enum is that it follows the scope rule and compiler automatic assign the value to its member constant.

Note: *A variable of enumeration type stores one of the values of the enumeration list defined by that type.*

Syntax of enum,
enum Enumeration_Tag { Enumeration_List };

The Enumeration_Tag specifies the enumeration type name.

The Enumeration_List is a comma-separated list of named constant.

Example,
enum FLASH_ERROR { DEFRAGMENT_ERROR, BUS_ERROR};.

What is the difference between const and macro?

- The const keyword is handled by the compiler, in another hand, a macro is handled by the preprocessor directive.
- const is a qualifier that is modified the behavior of the identifier but macro is preprocessor directive.
- There is type checking is occurred with const keyword but does not occur with #define.
- const is scoped by C block, #define applies to a file.
- const can be passed as a parameter (as a pointer) to the function.In case of call by reference, it prevents to modify the passed object value.

How to set, clear, toggle and checking a single bit in C?

Setting a Bits
Bitwise OR operator (|) use to set a bit of integral data type. “OR” of two bits is always one if any one of them is one.

Number | = (1<< nth Position)

Clearing a Bits
Bitwise AND operator (&) use to clear a bit of integral data type. “AND” of two

bits is always zero if any one of them is zero.To clear the nth bit, first, you need to invert the string of bits then AND it with the number.

Number &= ~ (1<< nth Position)

Checking a Bits
To check the nth bit, shift the '1' nth position toward the left and then "AND" it with the number.

Bit = Number & (1 << nth)

Toggling a Bits
Bitwise XOR (^) operator use to toggle the bit of an integral data type. To toggle the nth bit shift the '1' nth position toward the left and "XOR" it.

Number ^= (1<< nth Position)

Write a program swap two numbers without using the third variable?

Let's assume a, b two numbers, there are a lot of methods two swap two number without using the third variable.

Method 1((Using Arithmetic Operators):

```
#include <stdio.h>

int main()
{
  int a = 10, b = 5;

  // algo to swap 'a' and 'b'
  a = a + b;  // a becomes 15
  b = a - b;  // b becomes 10
  a = a - b;  // fonally a becomes 5

  printf("After Swapping the value of: a = %d, b = %d\n\n", a, b);

  return 0;
}
```

Method 2 (Using Bitwise XOR Operator):

```
#include <stdio.h>

int main()
{
  int a = 10, b = 5;

  // algo to swap 'a' and 'b'
  a = a ^ b;  // a becomes (a ^ b)
  b = a ^ b;  // b = (a ^ b ^ b), b becomes a
  a = a ^ b;  // a = (a ^ b ^ a), a becomes b

  printf("After Swapping the value of: a = %d, b = %d\n\n", a, b);

  return 0;
}
```

What is meant by structure padding?

In the case of structure or union, the compiler inserts some extra bytes between the members of structure or union for the alignment, these extra unused bytes are called padding bytes and this technique is called padding.

Padding has increased the performance of the processor at the penalty of memory.In structure or union data members aligned as per the size of the highest bytes member to prevent from the penalty of performance.

Note: *Alignment of data types mandated by the processor architecture, not by language.*

What is the endianness?

The endianness is the order of bytes to store data in memory and it also describes the order of byte transmission over a digital link. In memory data store in which order it depends on the endianness of the system, if the system is big-endian then the MSB byte store first (means at lower address) and if the system is little-endian then LSB byte store first (means at lower address).

Some examples of the little-endian and big-endian system.

What is big-endian and little-endian?

Suppose, 32 bits Data is 0x11223344.

Big-endian

The most significant byte of data stored at the lowest memory address.

Little-endian

The least significant byte of data stored at the lowest memory address.

***Note**: Some processor has the ability to switch one endianness to other endianness using the software means it can perform like both big endian or little endian at a time. This processor is known as the Bi-endian, here are some architecture (ARM version 3 and above, Alpha, SPARC) who provide the switchable endianness feature.*

Write a c program to check the endianness of the system.

Method 1:

```
#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>

int main(void)
{
uint32_t u32RawData;
uint8_t *pu8CheckData;
u32RawData = 0x11223344; //Assign data

pu8CheckData = (uint8_t *)&u32RawData; //Type cast

if (*pu8CheckData == 0x44) //check the value of lower address
{
printf("little-endian");
}
else if (*pu8CheckData == 0x11) //check the value of lower address
{
printf("big-endian");
}


return 0;
}
```

Method 2:

```
#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>

typedef union
{

uint32_t u32RawData;  // integer variable
uint8_t  au8DataBuff[4]; //array of character

}RawData;


int main(void)
{
RawData uCheckEndianess;
uCheckEndianess.u32RawData = 0x11223344; //assign the value

if (uCheckEndianess.au8DataBuff[0] == 0x44) //check the array first index value
{
printf("little-endian");
}
else if (uCheckEndianess.au8DataBuff[0] == 0x11) //check the array first index value
{
printf("big-endian");
}

return 0;
}
```

How to convert little endian to big endian vice versa in C?

Method 1:

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <inttypes.h>

//Function to change the endianess
uint32_t ChangeEndianness(uint32_t u32Value)
{
uint32_t u32Result = 0;
u32Result |= (u32Value & 0x000000FF) << 24;
u32Result |= (u32Value & 0x0000FF00) << 8;
u32Result |= (u32Value & 0x00FF0000) >> 8;
u32Result |= (u32Value & 0xFF000000) >> 24;
return u32Result;
}


int main()
{
uint32_t u32CheckData  = 0x11223344;
uint32_t u32ResultData =0;
u32ResultData = ChangeEndianness(u32CheckData);  //swap the data
printf("0x%x\n",u32ResultData);
u32CheckData = u32ResultData;
u32ResultData = ChangeEndianness(u32CheckData);//again swap the data
printf("0x%x\n",u32ResultData);
return 0;
}
```

What is static memory allocation and dynamic memory allocation?

According to C standard, there are four storage duration, static, thread (C11), automatic, and allocated. The storage duration determines the lifetime of the object.

The static memory allocation:

Static Allocation means, an object has external or internal linkage or declared with static storage-class. It's initialized only once, prior to program startup and its lifetime is throughout the execution of the program. A global and static variable is an example of static memory allocation.

The dynamic memory allocation:

In C language, there are a lot of library functions (malloc, calloc, or realloc,..) which are used to allocate memory dynamically. One of the problems with dynamically allocated memory is that it is not destroyed by the compiler itself that means it is the responsibility of the user to deallocate the allocated memory.

When we allocate the memory using the memory management function, they return a pointer to the allocated memory block and the returned pointer is pointing to the beginning address of the memory block. If there is no space available, these functions return a null pointer.

What is the memory leak in C?

A memory leak is a common and dangerous problem. It is a type of resource leak. In C language, a memory leak occurs when you allocate a block of memory using the memory management function and forget to release it.

```
int main ()
{

 char * pBuffer = malloc(sizeof(char) * 20);

 /* Do some work */

 return 0; /*Not freeing the allocated memory*/
}
```

Note: *once you allocate a memory than allocated memory does not allocate to another program or process until it gets free.*

What is the difference between malloc and calloc?

The malloc and calloc are memory management functions. They are used to allocate memory dynamically. Basically, there is no actual difference between calloc and malloc except that the memory that is allocated by calloc is initialized with 0.

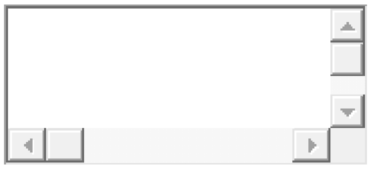
In C language,calloc function initialize the all allocated space bits with zero but malloc does not initialize the allocated memory. These both function also has a difference regarding their number of arguments, malloc take one argument but calloc takes two.

What is the purpose of realloc( )?

The realloc function is used to resize the allocated block of memory. It takes two arguments first one is a pointer to previously allocated memory and the second one is the newly requested size.

The calloc function first deallocates the old object and allocates again with newly specified size. If the new size is lesser to the old size, the contents of the newly allocated memory will be same as prior but if any bytes in the newly created object goes beyond the old size, the values of the exceeded size will be indeterminate.

Syntax:
void *realloc(void *ptr, size_t size);



```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main ()
{
char *pcBuffer = NULL;
/* Initial memory allocation */
pcBuffer = malloc(8);

strcpy(pcBuffer, "aticle");
printf("pcBuffer = %s\n", pcBuffer);

/* Reallocating memory */
pcBuffer = realloc(pcBuffer, 15);

strcat(pcBuffer, "world");
printf("pcBuffer = %s\n", pcBuffer);

//free the allocated memory
free(pcBuffer);

return 0;
}
```

Output:

```
pcBuffer = aticle
pcBuffer = aticleworld
```

Note: *It should be used for dynamically allocated memory but if a pointer is a null pointer, realloc behaves like the malloc function.*

What is the return value of malloc (0)?

If the size of the requested space is zero, the behavior will be implementation-defined. The return value of the malloc could be a null pointer or it shows the behavior of that size is some nonzero value.It is suggested by the standard to not use the pointer to access an object that is returned by the malloc while size is zero.

What is dynamic memory fragmentation?

The memory management function is guaranteed that if memory is allocated, then it would be suitably aligned to any object which has the fundamental alignment. The fundamental alignment is less than or equal to the largest alignment that's supported by the implementation without an alignment specification.

One of the major problems with dynamic memory allocation is fragmentation, basically, fragmentation occurred when the user does not use the memory efficiently. There are two types of fragmentation, external fragmentation, and internal fragmentation.

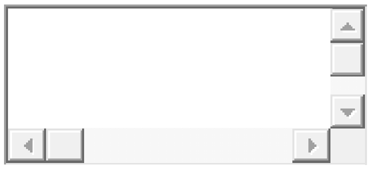
The external fragmentation is due to the small free blocks of memory (small memory hole) that is available on the free list but program not able to use it. There are different types of free list allocation algorithms that used the free memory block efficiently.

To understand the external fragmentation, consider a scenario where a program has 3 contiguous blocks of memory and the user frees the middle block of memory. In that scenario, you will not get a memory, if the required block of memory is larger than a single block of memory (but smaller or equal to the aggregate of the block of memory).

The internal fragmentation is wasted of memory that is allocated for rounding up the allocated memory and in bookkeeping (infrastructure), the bookkeeping is used to keep the information of the allocated memory.

Whenever we called the malloc function then it reserves some extra bytes (depend on implementation and system) for bookkeeping. This extra byte is reserved for each call of malloc and become a cause of the internal fragmentation.

***For example*,**
See the below code, the programmer may think that system will be allocated 8 *100 (800) bytes of memory but due to bookkeeping (if 8 bytes) system will be allocated 8*100 extra bytes. This is an internal fragmentation, where 50% of the heap waste.



```
char *acBuffer[100];

int main()
{
 int iLoop = 0;
 while(iLoop < 100)
 {
 acBuffer[iLoop ] =  malloc(8);
 ++iLoop;

 }

}
```

How is the free work in C?

When we call the memory management functions (malloc, calloc or realloc) then these functions keep extra bytes for bookkeeping.

Whenever we call the free function and pass the pointer that is pointing to allocated memory, the free function gets the bookkeeping information and release the allocated memory. Anyhow if you or your program change the value of the pointer that is pointing to the allocated address, the calling of free function give the undefined result.

```
____ The allocated block ____
/                            \
+--------+--------------------+
| Header | Your data area ... |
+--------+--------------------+
         ^
         |
         +-- Returned Address
```

For example,

```
#include <stdio.h>
#include <stdlib.h>


int main()
{
char *pcBuffer = NULL;
pcBuffer  =  malloc(sizeof(char) *  16); //Allocate the memory

pcBuffer++; //Increment the pointer

free(pcBuffer); //Call free function to release the allocated memory

return 0;
}
```

What is a Function Pointer?

A function pointer is similar to the other pointers but the only difference is that it points to a function instead of the variable.

In the other word, we can say, a function pointer is a type of pointer that store the address of a function and these pointed function can be invoked by function pointer in a program whenever required.

How to declare a pointer to a function in c?

The syntax for declaring function pointer is very straightforward. It seems like difficult in beginning but once you are familiar with function pointer then it becomes easy.

The declaration of a pointer to a function is similar to the declaration of a function. That means function pointer also requires a return type, declaration name, and argument list. One thing that you need to remember here is, whenever you declare the function pointer in the program then declaration name is preceded by the * (Asterisk) symbol and enclosed in parenthesis.

For example,

```
void ( *fpData )( int );
```

For the better understanding, let’s take an example to describe the declaration of a function pointer in c.
e.g,
void (*pfDisplayMessage) (const char *);

In above expression, pfDisplayMessage is a pointer to a function taking one argument, const char *, and returns void.

When we declare a pointer to function in c then there is a lot of importance of the bracket. If in the above example, I remove the bracket, then the meaning of the above expression will be change and it becomes void *pfDisplayMessage (const char *). It is a declaration of a function which takes the const character pointer as arguments and returns void pointer.

Where can the function pointers be used?

There are a lot of places, where the function pointers can be used. Generally, function pointers are used in the implementation of the callback function, finite state machine and to provide the feature of polymorphism in C language …etc.

Write a program to check an integer is a power of 2?

Here, I am writing a small algorithm to check the power of 2. If a number is a power of 2, function return 1.

```
int CheckPowerOftwo (unsigned int x)
{
  return ((x != 0) && !(x & (x - 1)));
}
```

What is the output of the below code?

```
#include <stdio.h>

int main()
{
   int x = -15;

   x = x << 1;

   printf("%d\n", x);
}
```

Output:

undefined behavior.

What is the output of the below code?

```
#include <stdio.h>

 int main()
 {
    int x = -30;

    x = x >> 1;

    printf("%d\n", x);
 }
```

Output:

implementation-defined.

Write a program to count set bits in an integer?

```
unsigned int NumberSetBits(unsigned int n)
{
  unsigned int CountSetBits= 0;
  while (n)
  {
    CountSetBits += n & 1;
    n >>= 1;
  }
  return CountSetBits;
}
```

What is void or generic pointers in C?

A void pointer is a generic pointer. It has no associated data type that's why it can store the address of any type of object and type-casted to any types.

According to C standard, the pointer to void shall have the same representation and alignment requirements as a pointer to a character type.A void pointer declaration is similar to the normal pointer, but the difference is that instead of data types we use the void keyword.

Syntax:
void * Pointer_Name;

What is the advantage of a void pointer in C?

There are following advantages of a void pointer in c.

- Using the void pointer we can create a generic function that can take arguments of any data type. The memcpy and memmove library function are the best examples of the generic function, using these function we can copy the data from the source to destination.
  e.g.
  void * memcpy (void * dst, const void * src, size_t num);
- We have already know that void pointer can be converted to another data type that is the reason malloc, calloc or realloc library function return void *. Due to the void * these functions are used to allocate memory to any data type.
- Using the void * we can create a generic linked list.For more information see this link: How to create generic Link List.

What are dangling pointers?

Generally, daggling pointers arise when the referencing object is deleted or deallocated, without changing the value of the pointers. It creates the problem because the pointer is still pointing the memory that is not available. When the user tries to dereference the daggling pointers than it shows the undefined behavior and can be the cause of the segmentation fault.

For example,

```
#include<stdio.h>
#include<stdlib.h>

int main()
{
int *piData = NULL;

piData = malloc(sizeof(int)* 10); //creating integer of size 10.

free(piData); //free the allocated memory

*piData = 10; //piData is dangling pointer

return 0;

}
```

In simple word, we can say that dangling pointer is a pointer that not pointing a valid object of the appropriate type and it can be the cause of the undefined behavior.

What is the wild pointer?

A pointer that is not initialized properly prior to its first use is known as the wild pointer. Uninitialized pointers behavior is totally undefined because it may point some arbitrary location that can be the cause of the program crash, that's is the reason it is called a wild pointer.

In the other word, we can say every pointer in programming languages that are not initialized either by the compiler or programmer begins as a wild pointer.

Note: *Generally, compilers warn about the wild pointer.*

***Syntax*,**
int *piData; //piData is wild pointer.

What is a NULL pointer?

According to C standard, an integer constant expression with the value 0, or such an expression cast to type void *, is called a null pointer constant. If a null pointer constant is converted to a pointer type, the resulting pointer, called a null pointer.

***Syntax*,**
int *piData = NULL; // piData is a null pointer

What are the post increment and decrement operators?

When we use post-increment (++) operator on an operand then the result is the value of the operand and after getting the result, the value of the operand is incremented by 1. The working of the post-decrement (-) operator is similar to the post-increment operator but the difference is that the value of the operand is decremented by 1.

Note: *incrementation and decrementation by 1 are the types specified.*

Which one is better: Pre-increment or Post increment?

Nowadays compiler is enough smart, they optimize the code as per the requirements. The post and pre increment both have own importance we need to use them as per the requirements.

If you are reading a flash memory byte by bytes through the character pointer then here you have to use the post-increment, either you will skip the first byte of the data. Because we already know that in case of pre-increment pointing address will be increment first and after that, you will read the value.

Let's take an example of the better understanding,
In below example code, I am creating a character array and using the character pointer I want to read the value of the array. But what will happen if I used pre-increment operator? The answer to this question is that 'A' will be skipped and B will be printed.

```
#include <stdio.h>

int main(void)
{

 char acData[5] ={'A','B','C','D','E'};
 char *pcData = NULL;

 pcData = acData;

 printf("%c ",*++pcData);

 return 0;
}
```

But in place of pre-increment if we use post-increment then the problem is getting solved and you will get A as the output.

```
#include <stdio.h>

int main(void)
{

 char acData[5] ={'A','B','C','D','E'};
 char *pcData = NULL;

 pcData = acData;

 printf("%c ",*pcData++);

return 0;
}
```

Besides that, when we need a loop or just only need to increment the operand then pre-increment is far better than post-increment because in case of post increment compiler may have created a copy of old data which takes extra time. This is not 100% true because nowadays compiler is so smart and they are optimizing the code in a way that makes no difference between pre and post-increment. So it is my advice, if post-increment is not necessary then you have to use the pre-increment.

Note: Generally post-increment is used with array subscript and pointers to read the data, otherwise if not necessary then use pre in place of post-increment.Some compiler also mentioned that to avoid to use post-increment in looping condition.
iLoop = 0.

```
while (a[iLoop ++] != 0)
{
// Body statements
}
```

Are the expressions *ptr ++ and ++*ptr same ?

Both expressions are different. Let’s see a sample code to understand the difference between both expressions.

```
#include <stdio.h>

int main(void)
{
   int aiData[5] = {100,200,300,400,500};

 int *piData = aiData;

   ++*piData;

 printf("aiData[0] = %d, aiData[1] = %d, *piData = %d", aiData[0], aiData[1], *piData);

   return 0;
}
```

Output: 101 , 200 , 101

Explanation:
In the above example, two operators are involved and both have the same precedence with a right to left associativity. So the above expression ++*p is equivalent to ++ (*p). In another word, we can say it is pre-increment of value and output is 101, 200, 101.

```
#include <stdio.h>

int main(void)
{
   int aiData[5] = {100,200,30,40,50};

 int *piData = aiData;

 *++piData;

printf("aiData[0] = %d, aiData[1] = %d, *piData = %d", aiData[0], aiData[1], *piData);

 return 0;
}
```

Output: 100, 200, 200

Explanation:
In the above example, two operators are involved and both have the same precedence with the right to left associativity. So the above expression *++p is equivalent to *(++p). In another word you can say it is pre-increment of address and output is 100, 200,200.

What does the keyword const mean?

A const is only a qualifier, it changes the behavior of a variable and makes it read-only type. When we want to make an object read-only type, then we have to declare it as const.

Syntax
const DataType Identifier = Value;
e.g.
const int iData = 0

At the time of declaration, const qualifier only gives the direction to the compiler that the value of declaring object could not be changed. In simple word, const means not modifiable (cannot assign any value to the object at the runtime).

When should we use const in a C program?

There are following places where we need to use the const keyword in the programs.

- In call by reference function argument, if you don't want to change the actual value which has passed in function.
  Eg.
  int PrintData (const char *pcMessage);
- In some places, const is better then macro because const handle by the compiler and have a type checking.
  Eg.
  const int ciData = 100;
- In the case of I/O and memory mapped register const is used with the volatile qualifier for efficient access.
  Eg.
  const volatile uint32_t *DEVICE_STATUS = (uint32_t *) 0x80102040;
- When you don't want to change the value of an initialized variable.

What is the meaning of below declarations?

1. const int a;
2. int const a;
3. const int *a;
4. int * const a;
5. int const * a const;

1. The "a" is a constant integer.
2. Similar to first, "a" is a constant integer.
3. Here "a" is a pointer to a const integer, the value of the integer is not modifiable, but the pointer is not modifiable.
4. Here "a" is a const pointer to an integer, the value of the pointed integer is modifiable, but the pointer is not modifiable.
5. Here "a" is a const pointer to a const integer that means the value of pointed integer and pointer both are not modifiable.

Differentiate between a constant pointer and pointer to a constant?

Constant pointer:

A constant pointer is a pointer whose value (pointed address) is not modifiable. If you will try to modify the pointer value, you will get the compiler error.

A constant pointer is declared as follows :
Data_Type * const Pointer_Name;

Let's see the below example code when you will compile the below code get the compiler error.

```
#include<stdio.h>

int main(void)
{
   int var1 = 10, var2 = 20;

   //Initialize the pointer
   int *const ptr = &var1;

   //Try to modify the pointer value
   ptr = &var2;

   printf("%d\n", *ptr);

   return 0;
}
```

Pointer to a constant:

In this scenario the value of pointed address is constant that means we can not change the value of the address that is pointed by the pointer.

A constant pointer is declared as follows :
Data_Type const* Pointer_Name;

Let's take a small code to illustrate a pointer to a constant:

```
#include<stdio.h>

int main(void)
{
   int var1 = 100;
   // pointer to constant integer
   const int* ptr = &var1;

   //try to modify the value of pointed address
   *ptr = 10;

   printf("%d\n", *ptr);

   return 0;
}
```

What are the uses of the keyword static?

In C language, the static keyword has a lot of importance. If we have used the static keyword with a variable or function, then only internal or none linkage is worked. I have described some simple use of a static keyword.

- A static variable only initializes once, so a variable declared static within the body of a function maintains its prior value between function invocations.
- A global variable with static keyword has an internal linkage, so it only accesses within the translation unit (.c). It is not accessible by another translation unit. The static keyword protects your variable to access from another translation unit.
- By default in C language, linkage of the function is external that it means it is accessible by the same or another translation unit. With the help of the static keyword, we can make the scope of the function local, it only accesses by the translation unit within it is declared.

What is the difference between global and static global variables?

In simple word, they have different linkage.

A static global variable ===>>> **internal linkage.**
A non-static global variable ===>>> **external linkage.**

So global variable can be accessed outside of the file but the static global variable only accesses within the file in which it is declared.

Differentiate between an internal static and external static variable?

In C language, the external static variable has the internal linkage and internal static variable has no linkage. So the life of both variable throughout the program but scope will be different.

A external static variable ===>>> **internal linkage.**
A internal static variable ===>>> **none .**

Can static variables be declared in a header file?

Yes, we can declare the static variables in a header file.

What is the difference between declaration and definition of a variable?

Declaration of variable in c

A variable declaration only provides sureness to the compiler at the compile time that variable exists with the given type and name, so that compiler proceeds for further compilation without needing all detail of this variable. In C language, when we declare a variable, then we only give the information to the compiler, but there is no memory reserve for it. It is only a reference, through which we only assure to the compiler that this variable may be defined within the function or outside of the function.

***Note:** We can declare a variable multiple time but defined only once.*
eg,
extern int data;
extern int foo(int, int);
int fun(int, char); // extern can be omitted for function declarations

Definition of variable in c

The definition is action to allocate storage to the variable. In another word, we can say that variable definition is the way to say the compiler where and how much to create the storage for the variable generally definition and declaration occur at the same time but not almost.

eg,
int data;
int foo(int, int) { }

Note: *When you define a variable then there is no need to declare it but vice versa is not applicable.*

What is the difference between pass by value by reference in c and pass by reference in c?

Pass By Value:

- In this method value of the variable is passed. Changes made to formal will not affect the actual parameters.
- Different memory locations will be created for both variables.
- Here there will be temporary variable created in the function stack which does not affect the original variable.

Pass By Reference :

- In Pass by reference, an address of the variable is passed to a function.
- Whatever changes made to the formal parameter will affect the value of actual parameters(a variable whose address is passed).
- Both formal and actual parameter shared the same memory location.
- it is useful when you required to returns more than 1 values.

What is a reentrant function?

In computing, a computer program or subroutine is called reentrant if it can be interrupted in the middle of its execution and then safely be called again ("re-entered") before its previous invocations complete execution. The interruption could be caused by an internal action such as a jump or call, or by an external action such as an interrupt or signal. Once the reentered invocation completes, the previous invocations will resume correct execution.

What is the inline function?

An inline keyword is a compiler directive that only suggests the compiler to substitute the body of the function at the calling the place. It is an optimization technique used by the compilers to reduce the overhead of function calls.

for example,

```
static inline void Swap(int *a, int *b)
{
  int tmp= *a;
  *a= *b;
  *b = tmp;
}
```

What is the advantage and disadvantage of the inline function?

There are few important advantage and disadvantage of the inline function.

Advantages:-
1) It saves the function calling overhead.
2) It also saves the overhead of variables push/pop on the stack, while function calling.
3) It also saves the overhead of return call from a function.
4) It increases locality of reference by utilizing instruction cache.
5) After inlining compiler can also apply intraprocedural optimization if specified. This is the most important one, in this way compiler can now focus on dead code elimination, can give more stress on branch prediction, induction variable elimination etc..

Disadvantages:-
1) May increase function size so that it may not fit in the cache, causing lots of cache miss.
2) After inlining function, if variables number which are going to use register increases than they may create overhead on register variable resource utilization.
3) It may cause compilation overhead as if somebody changes code inside an inline function then all calling location will also be compiled.
4) If used in the header file, it will make your header file size large and may also make it unreadable.
5) If somebody used too many inline functions resultant in a larger code size than it may cause thrashing in memory. More and number of page fault bringing down your program performance.
6) It’s not useful for an embedded system where large binary size is not preferred at all due to memory size constraints.

What is the virtual memory?

The virtual memory is the part of memory management techniques and it creates an illusion that the system has a sufficient amount memory.In another word you can say that virtual memory is a layer of indirection.

Consider the two statements and find the difference between them?

```
struct sStudentInfo {

char Name[12];
int Age;
float Weight;
int RollNumber;

};


#define STUDENT_INFO struct sStudentInfo*

typedef struct sStudentInfo* studentInfo;

statement 1
STUDENT_INFO p1, p2;

statement 2
studentInfo q1, q2;
```

Both statements looking same but actually, both are different to each other.

Statement 1 will be expanded to struct sStudentInfo * p1, p2. It means that p1 is a pointer to struct sStudentInfo but p2 is a variable of struct sStudentInfo.

In statement 2, both q1 and q2 will be a pointer to struct sStudentInfo.

What are the limitations of I2C interface?

- Half duplex communication, so data is transmitted only in one direction (because of the single data bus) at a time.
- Since the bus is shared by many devices, debugging an I2C bus (detecting which device is misbehaving) for issues is pretty difficult.
- The I2C bus is shared by multiple slave devices if anyone of these slaves misbehaves (pull either SCL or SDA low for an indefinite time) the bus will be stalled. No further communication will take place.
- I2C uses resistive pull-up for its bus. Limiting the bus speed.
- Bus speed is directly dependent on the bus capacitance, meaning longer I2C bus traces will limit the bus speed.

Here, I have mentioned some questions for you. If you know, please write in comment box. Might be your comment helpful for other.

- What is the difference between flash memory, EPROM, and EEPROM?
- What is the difference between Volatile & Non Volatile Memory?
- What are the differences between a union and a structure in C?
- What is the difference between RS232 and UART?
- Is it possible to declare struct and union one inside other? Explain with example.
- How to find the bug in code using the debugger if the pointer is pointing to an illegal value.
- What is watchdog timer?
- What is the DMA?
- What is RTOS?
- What are CAN and its uses?
- Why is CAN having 120 ohms at each end?
- Why is CAN message-oriented protocol?
- What is the Arbitration in the CAN?
- Standard CAN and Extended CAN difference?
- What is the use of bit stuffing?
- How many types of IPC mechanism do you know?
- What is semaphore?
- What is the spinlock?
- Convert a given decimal number to hex.

- What is the difference between heap and stack memory?
- What is socket programming?
- How can a double pointer be useful?
- What is the difference between binary semaphore and mutex?