C- Interview Question:-

==========

1> To get mirror image of a binary tree.?

To get the mirror image of the binary tree we have to just traverse the tree in post oreder and in a bottom up fashion we have to swap the right son to left

code :

```
void MirrorImage(struct node *R){
      struct node *temp=NULL;
      if(R==NULL||(R->left==NULL &&R->right==NULL))
            return;
      MirrorImage(R->left);
      MirrorImage(R->right);
      temp=R->left;
      R->left=R->right;
      R->right=temp;
}
```

==========

```
#include<stdio.h>
void f1(int a[]){
int *p=0;
int i=0;
while(i++ > 3)
p=&a[i]; //error
*p=0;
}
void main() {
int a[10];
f1(a);
}
```

ANSWER: Segmentation fault.

P is not assigned with any address as while loop does not execute (since i++ is

not greater than 3). thereby p is NULL , *p will result in error.

The same program executes fine if while executes (change condition in loop as -

while(i++ < 3))

==========

Count the number of set bits in a number. Now optimize for speed. Now optimize for size

```
int BitCount (unsigned int u){
```

```
    unsigned int uCount=0 ;
    for(; u; u&=(u-1))
      uCount++;
    return uCount ;
}
```

==========

Given a singly linked list, print out its contents in reverse order.

Can you do it without using any extra space

This can be done via recursion.

```
function printLinkedLits( Node head){
    while( head->next! null)
    printLinkedList( head->next );
    printf(head);
}
```

Recursion will use the internal stack space .. you can do following::

First traverse the link list and reverse it then traverse it and print

it If there is any constraint that list should be as it is then reverse it again

==========

Count the number of set bits in a number without using a loop.

```
int countBits( int n ){
    if( n == 0 ) return 0;
    if( n&1 ) return 1 + countBits( n>>1 );
    else return countBits( n >> 1 );
}
```

==========

What is the difference between functors, call back functions and function pointers?

1. functors are class objects which overload operator(),

2. function pointers are pointer to function, eg: vtbl is a array of function pointers...

3. callback functions are function pointers passed as parameters of function, which can be called back when an event occurs...eg: when a thread is created,we send a callback function to be called when thread starts...

Ans. Functors are functions with a state. In C++ you can realize them as a class with one or more private members to store the state and with an overloaded operator () to execute the function. Functors can encapsulate C and C++ function pointers employing the concepts templates and polymorphism. You can

build up a list of pointers to member functions of arbitrary classes and call them all through the same interface without bothering about their class or the need of a pointer to an instance. All the functions just have got to have the same return-type and calling parameters. Sometimes functors are also known as closures. You can also use functors to implement callbacks.

Function Pointers are pointers, i.e. variables, which point to the address of a function. You must keep in mind, that a running program gets a certain space in the main-memory. Both, the executable compiled program code and the used variables, are put inside this memory. Thus a function in the program code is, like e.g. a character field, nothing else than an address. It is only important how you, or better your compiler/processor, interpret the memory a pointer points to.

Function Pointers provide the concept of callback functions. A callback is done just like a normal function call would be done: You just use the name of the function pointer instead of a function name.

==========

What are call back functions?

A callback function is a function that is called through a function pointer

A call back function is the one in which a pointer is passed as an argument to the current function.

Whenever the function pointer is accessed in the current function the other function is called .

==========

Compare and contrast dynamic array & linked list

Dynamic array is good for direct,random access but the array has to be resized every time a new element is added.Faster than linked list as the allocation of memory dynamically for every node causes a lot of overhead especially if the list is small.

Linked list is good for sequential access and need not resize every time a new element is added due to dynamic allocation of memory.

==========

Why is malloc preferred over calloc?

There are two differences. First, is in the number of arguments. Malloc() takes a single argument (memory required in bytes), while calloc() needs two arguments (number of variables to allocate memory, size in bytes of a single variable). Secondly, malloc() does not initialize the memory allocated, while calloc() initializes the allocated memory to ZERO.

Here are more opinions and answers from FAQ Farmers:

The difference between malloc and calloc are: 1. malloc() allocates byte of memory, whereas calloc()allocates block of memory.

Calloc(m, n) is essentially equivalent to p = m*malloc(n); memset(p, 0, m * n); The zero fill is all-bits-zero, and does not therefore guarantee useful null pointer values (see section 5 of this list) or floating-point zero values. Free is properly used to free the memory allocated by calloc.

Malloc(s); returns a pointer for enough storage for an object of s bytes. Calloc(n,s); returns a pointer for enough contiguous storage for n objects, each of s bytes. The storage is all initialized to zeros.

Simply, malloc takes a single argument and allocates bytes of memory as per the argument taken during its invocation. Where as calloc takes two aguments, they are the number of variables to be created and the capacity of each vaiable (i.e. the bytes per variable).

This one is false:

I think calloc can allocate and initialize memory, if the asked memory is available contiguously where as malloc can allocate even if the memory is not available contiguously but available at different locations.

A less known difference is that in operating systems with optimistic memory allocation, like Linux, the pointer returned by malloc isn't backed by real memory until the program actually touches it.

calloc does indeed touch the memory (it writes zeroes on it) and thus you'll be sure the OS is backing the allocation with actual RAM (or swap). This is also why it is slower than malloc (not only does it have to zero it, the OS must also find a suitable memory area by possibly swapping out other processes)

==========

How is the memory allocation done for a union?

A union will allocate only the space required by one element(the element that occupies the maximum size).Suppose you have a union consisting of 4 variables(of different data types),than the union will allocate space only to the variable that requires the maximum memory space. For example:

```
union shirt{
char size;
int chest;
}mine;
```

The union variable 'mine' will be allocate only 4 bytes(the compiler knows,that tha character will require only one byte, while an integer needs 4 bytes .It allots 'mine' a total of only 4 bytes

==========

How will you determine the page size of a *nix machine using C code ? Hint: Use malloc()

start a fresh process.

char *ptr = (char*) malloc(1);

allocate one byte but the OS gets a full page and attaches it to the process address space.

start dereferencing ptr and incrementing it in an infinite loop, until SEGV gets generated.

while(1){

*ptr++ = NULL;

pagesize++;

}

write a signal handler to catch SIGSEGV, there print out the value of 'pagesize'

---

most memory corruption detectors work on the principle that when memory beyond the page allocated is accessed a SEGV is generated, so it replaces calls to malloc with its own malloc call where it places data at the end of the page instead of the beginning, so when a pointer overwrites a buffer, it also crosses the pagelimit and sigsegv is raised, so you get the exact spot of access violation/memory corruption.

==========

How can u find the size of structure.?

struct *ptr ;

int size = (char *)(ptr+1) - (char *)ptr ;

This has to do with padding. Structures are usually padded with extra bytes for proper memory alignment. Thus size(struct) does not return the exact size of struct. we need to declare a struct as __(Packed)__ and then do a size of operation. There might be a better way.

==========

If a number is power of 2

for unsigned integer numbers:

bool isPow2(int n){

return !(n & (n-1)) && n;

}

==========

for unsigned integer numbers:

bool isPow2(int n){

return !(n & (n-1)) && n;

}

==========

return num&(num-1)==0

==========

Global and local variables - where are they defined.

When should we use a local variable, a global variable.

how do we optimize using these tactics.

Global and static variables are stored in bss section of the data segment. Local variables are stored on the stack and are valid until the program exits from the function in which they are defined.

==========

What should not be done in an ISR (Interrupt Service Routine)?

ISR's should not invoke functions which may cause ``blocking'' of the caller. An ISR must not perform I/O, A call to a device driver may block the system if the caller needs to wait for the device.

One should never sleep in an Interrupt handler. Things that might cause blocking/sleeping are:

1. Using semaphore to lock a shared data structure

2. Calling functions that might sleep

3. Allocating memory

Also one should not perform time consuming tasks in an ISR.

You can allocate memory in ISR at least in Linux. Use function get_free_pages(). If the memory is not available, it will return immediately and thus does not block

.==========

Write code for memcpy function

Basic first version - copies one byte at a time.

```
void memcpy(void* src, void* dst, size_t len) {
char* p = (char*)src;
char* q = (char*)dst;
while(len--)
*p++ = *q++;
}
int memcpy(void* dest,const void* src,int num){
while(num--)
*(char*)dest=*(char*)src;
dest=(char*)dest+1;
src=(char*)src+1;
}
```

==========

You can put print messages in the interrupt handler to print the struct reg arg and int # for debug purposes.

Usually the kernel doesn't do floating point computation and hence it doesn't save/restore the context of floating point regs (this is about the linux kernel on x86). There can be kernels which might do FP computations and they might need to save/restore FP regs and computations.

Interrupt operations only allow non blocking operations;

print statement is a nonblocking operation and floating point computation may not be.

==========

What are upper half and bottom half in device drivers.Why are they used?

upper half and lower half are the terms related to interrupt handler and hardware devices only via drivers raise interrupts in this context , they asked u in form of device driver.

Each device has to register its interrupt handler (which is the part of device driver. Now for devices which raise interrupt frequently or in case when same interrupt handler is used via devices so to increase the service performance driver designers design the handlers in such a way that whenever an interrupt occurs the OS does the most important part of handler "upper half" to respond to interrupt,create a data structure containing device specific data called "lower half" for later processing when CPU becomes available. This way interrupt handlers can be used in case when the interrupt raise frequency is high.

These are called top halves and bottom halves. When a device raises an interrupt the isr should be executed fastly as the other processes are stopped while the isr is being executed, that means interrupt latency should be very less. When an interrupt is raised you just check the Interrupt status register to find what the interrupt is for(read/write/error) and differ the work of reading the data, taking actions on based on interrupts to be done later. In Linux this is done a technique called bottom halves, like softirqs, tasklets.

==========

System call? What happens in low level.

System calls are for kernel service. Using 0x80 INT, it jumps to kernel context.

For this u shd have asmlinkage pre-knowledge is required.

While implementation, u need to increment macro count from particular header file then define ur call and u need to put in corresponds...location

==========

```
struct {
char c;
int a;
char d;
}
```

What is the size here? What is the padding? Can we reorder these to reduce the padding? Why we need padding?

byte padding will be done by compiler to reduce number of machine cycles to read. if the integers are stored in the address which is divisible by its size (say 4) then only cpu can read whole 4 bytes in a single cycle other wise cpu machine cycles will be more hence byte padding will be done.

==========

How will you remove white space from a string?

```
char s[]="a b cde f";
int i,j;
for(i=j=0;s[j];j++) {
if(s[j] != ' ') {
s[i]=s[j]; i++;
}
}
```

}
s[i]='\0';

definition :: Defines the memory area (allocates the memory) for the variable and the definition occures once through the program(memory is allocated once)

Declaration :: Tells about the signature of the variable (type and size to be considered). but the declaration can occur many times.

OR For a variable, the definition is the statement that actually allocates memory. For example, the statement:

long int var;

is a definition. On the other hand, an extern reference to the same variable:

extern long int var;

is a declaration, since this statement doesn?t cause any memory to be allocated. Here?s another example of a declaration:

typedef MyType short;

Declaration of a variable is stating that it exists.

Definition of a variable says it exists and it is here, thus it implies that it is declared.

That doesn't only relate to variables but functions, structs/classes. enums... did I forget anything?

The following are declared but not defined:

extern int var;

void function();

struct myStruct;

class myClass;

enum myEnum;

The following are defined:

```
# int var;
# void function() {
# }
#
# struct myStruct {
# // stuff in here
# };
#
# class myClass {
# // stuff in here
# };
#
# enum myEnum {
# // stuff in here
# };
```

Why extern int var; is declared and int var; is defined?

extern is called a storage class specifier. It is used to tell the compiler that the variable or function has external linkage. That is, that it can be accessed from another file. There are various storage class specifers:

```
1. extern int var;   //declaration. var is an int in another file with external linkage
2. extern int var = 20;  //definition. var is created here with a value of 20
3.                       //and is to accessible from other files.
4. int var = 20;     //definition. Same as extern int var = 20;
5. static int var = 20;  //definition. var is an int with a value of 20 that has internal linkage
6.                       //var CANNOT by accessed from another file.
```

==========

What is main difference between declaring variable and volatile variable?

Declaring a variable "volatile" tells the compiler that it can't assume that the variable is only changed by the program code. If the variable is not declared volatile, sometimes the compiler can make certain optimizations since it can assume that it knows everything about how the variable will be used. When it is declared volatile, the variable may be changed by an external process (for example, it could be connected to an I/O device), so it must always be fetched from memory.

==========

What is difference between new and malloc?

both malloc and new functions are used for dynamic memory allocations and the basic difference is: malloc requires a special "typecasting" when it allocates memory for eg. if the pointer used is the char pointer then after the processor allocates memory then this allocated memory needs to be typecasted to char pointer i.e (char*).but new does not requires any typecasting. Also, free is the keyword used to free the memory while using malloc and delete the keyword to free memory while using new, otherwise this will lead the memory leak.

Besides the basic syntactical difference: malloc is a C function which will allocate the amount of memory you ask and thats it. new is a C++ operator which will allocate memory AND call the constructor of the class for which's object memory is being allocated.

Similarily, free is a C function which will free up the memory allocated. but delete is a C++ operator which will free up the allocated memory AND call the destructor of the object.

1.malloc requires the type casting during declaration , where as new doesn't needed the type casting during decleration

2. when ever we use new for allocating memory along with this it calls the constructor of the class for which object memory is allocated

3. in case of malloc free is the word used to clear the memory, where as delete is the format used in case of new to free the memory after usage

4. malloc is function, where as new is operator..so the time required for execution is less in case of new (it being a operator) as compared to malloc(it being a function)

1. malloc is a function call, while new is an operator. This difference is syntactic; behind the scenes, they both perform pretty much the same work to allocate the memory, and operator new also invokes any required constructors. There is a commonplace urban myth that operators are somehow faster in your code than functions; this is not correct, as any operator (except for mathematical operations that correspond directly to a single machine-code instruction) invocation amounts to a function call in any case. 2. malloc can fail, and returns a NULL pointer if memory is exhausted. Operator new never returns a NULL pointer, but indicates failure by throwing an exception instead. There is also a nothrow() version of operator new, which does return NULL on failure.

==========

Difference Between Malloc and Calloc

There are two differences. First, is in the number of arguments. Malloc() takes a single argument (memory required in bytes), while calloc() needs two arguments (number of variables to allocate memory, size in bytes of a single variable). Secondly, malloc() does not initialize the memory allocated, while calloc() initializes the allocated memory to ZERO.

Here are more opinions and answers from FAQ Farmers:

* The difference between malloc and calloc are: 1. malloc() allocates byte of memory, whereas calloc()allocates block of memory.

* Calloc(m, n) is essentially equivalent to p = m*malloc(n); memset(p, 0, m * n); The zero fill is all-bits-zero, and does not therefore guarantee useful null pointer values (see section 5 of this list) or floating-point zero values. Free is properly used to free the memory allocated by calloc.

* Malloc(s); returns a pointer for enough storage for an object of s bytes. Calloc(n,s); returns a pointer for enough contiguous storage for n objects, each of s bytes. The storage is all initialized to zeros.

* Simply, malloc takes a single argument and allocates bytes of memory as per the argument taken during its invocation. Where as calloc takes two aguments, they are the number of variables to be created and the capacity of each vaiable (i.e. the bytes per variable).

This one is false:

* I think calloc can allocate and initialize memory, if the asked memory is available contiguously where as malloc can allocate even if the memory is not available contiguously but available at different locations.

1. malloc takes only the size of the memory block to be allocated as input parameter.

2. malloc allocates memory as a single contiguous block.

3. if a single contiguous block cannot be allocated then malloc would fail.

1. calloc takes two parameters: the number of memory blocks and the size of each block of memory

2. calloc allocates memory which may/may not be contiguous.

3. all the memory blocks are initialized to 0.

4. it follows from point 2 that calloc will not fail if memory can beallocated in non-contiguous blocks when a single contiguous blockcannot be allocated.

calloc, malloc, free, realloc - Allocate and free dynamic memory

Synopsis

```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
void *malloc(size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
```

Description

calloc() allocates memory for an array of nmemb elements of size bytes each and returns a pointer to the allocated memory. The memory is set to zero.

malloc() allocates size bytes and returns a pointer to the allocated memory. The memory is not cleared.

free() frees the memory space pointed to by ptr, which must have been returned by a previous call to malloc(), calloc() or realloc(). Otherwise, or if free(ptr) has already been called before, undefined behaviour occurs. If ptr is NULL, no operation is performed.

realloc() changes the size of the memory block pointed to by ptr to size bytes. The contents will be unchanged to the minimum of the old and new sizes; newly allocated memory will be uninitialized. If ptr is NULL, the call is equivalent to malloc(size); if size is equal to zero, the call is equivalent to free(ptr). Unless ptr is NULL, it must have been returned by an earlier call to malloc(), calloc() or realloc(). If the area pointed to was moved, a free(ptr) is done.

```
volatile unsigned long *t0, *t2;
      t0 = (volatile unsigned long *)v_ram;
      t2 = (volatile unsigned long *)(v_ram + 0x00200000);
      *t0 = 0x87654321;
      *t2 = 0x12345678;
```

```
*((volatile unsigned short *) 0xfffece08) = 0x03FF;
  volatile unsigned int *MuxConfReg;
  MuxConfReg =(volatile unsigned int *) ((unsigned int) FUNC_MUX_CTRL_0); //Configuration
Registers 0xfffe1000

  *MuxConfReg &= ~(0x02000000);   /* bit 25 */
```

```
volatile u32 *macen = (volatile u32*)MAC0_ENABLE;   /* 0xB0520000 */
```

3.5 What is the output of the following ?

```
    #include <stdio.h>

      int      main ( void ){
        char a[] = "abc";
        char b[] = "xyz";

        *a = *b++;
        puts ( a );
        return 0;
      }
```

Ans The only operation that can be performed directly on an array value are the application of the sizeof and address (&) operators.

For sizeof, the array must be bounded and the result is the number of

storage units occupied by the array. The result of & is a pointer

to (the first element of) the array. And, in the above program we

are trying to increment array type, b, which is not allowed. It is an error.

```
#include <stdio.h>

int        main ( void ){
  union {
     int     _i;
     float   _f;
  }_u = { 10 };

  printf ( "%f\n", u.f );
  return 0;
}
```

Ans Before I give an answer to this question, let us discuss

how a union object is represented in the memory:

* The size of an union object is equal to size of it's largest member.

* Address of every member of an union object is same.

* A union can not have an instance of itself, but a pointer to instance of itself.

Let me introduce two terms: Object Representation and

Object Interpretation. Simply speaking, object representation

is the pattern of bits of an object in the storage unit. And,

object interpretation is the meaning of the representation.

In the above example, union members _i and _f reside in the same

memory location, say M. The union object _u has been initialized

to 10 which by default is assigned to the first member of the

union. Since _i and _f has the same location, their object

representation is also the same, but their interpretation would be

different. Let us assume that a C implementation (i.e., compiler) uses 2's complement for integers and IEEE 754 floating point format for float. Though, the bit pattern for _i and _f appear identical in the storage unit, their interpretation is different. Hence, if you thought that the above printf would print 10.000000, then you are wrong. The output is unspecified.

BIT-FIELDS

---

* A structure member may be declared to consist of a specified number of bits (including a sign bit, if any). Such a member is called a bit-field; its width is preceded by a colon.

* A bit-field will have a type that is signed int, or unsigned int. A bit-field is interpreted as a signed or unsigned integer type consisting of the specified number of bits.

* An implementation (i.e., a compiler) may allocate storage large enough to store the bit-field.

* If one bit-field follows another bit-field, then allocation for these fields would be given in the same storage unit, if sufficient storage space is available.

Ex: int a: 2;
int b: 2; will be stored in the same int.

* Similarly, if two bit-fields are next to each other and if they can not be stored in the same storage unit, then whether the two bit-fields overlap each other or are allocated two storage units are implementation -defined.

Ex: int a: 28;

int b: 10;

Here, whether A and B are stored separately in two ints or four bits

of B are stored in first int and the rest in second int, is implementation

-defined. Assume size of int to be 4 bytes.

* Another point is that the order of allocation of bit-fields (left-to-right

or right-to-left) in the storage unit is implementation-defined.

* A bit-field of non-zero width without the variable (identifier) name is considered as padding. A special case, a bit-field member with a width of 0, is considered as no other bit-fields are to be packed in

the same storage unit in which the previous bit-field member, if any, was stored.

* The address operator (&) can not be applied to bit-field members; hence, there are no pointers to or arrays of bit-field members.

==========

What is difference between define and typedef in c?

Answer

defines are handled by a preprocessor (a program run before the actual c compiler) which works like replace all in you editor.

Typedef is handled by the c compiler itself, and is an actual definition of a new type.

Answer

The distinction given between #define and typedef has one significant error: typedef does not in fact create a new type. According to Kernighan & Richie, the authors of the authoritative and universally acclaimed book, "The C Programming Language":

It must be emphasized that a typedef declaration does not create a new type in any sense; it merely adds a new name for some existing type. Nor are there any new semantics: variables declared this way have exactly the same properties as variables whose declarations are spelled out explicitly. In effect, typedef is like #define, except that since it is interpreted by the compiler, it can cope with textual substitutions that are beyond the capabilities of the preprocessor.

Answer :: There are some more subtleties though. The type defined with a typedef is exactly like its counterpart as far as its type declaring power is concerned BUT it cannot be modified like its counterpart. For example, let's say you define a synonim for the int type with:

typedef int MYINT

Now you can declare an int variable either with

int a;

or

MYINT a;

But you cannot declare an unsigned int (using the unsigned modifier) with

unsigned MYINT a;

although

unsigned int a;

would be perfectly acceptable.

Answer

typedefs can correctly encode pointer types.where as #DEFINES are just replacements done by the preprocessor.

For example,

1. typedef char *String_t;
2. #define String_d char *

3. String_t s1, s2; String_d s3, s4;

s1, s2, and s3 are all declared as char *, but s4 is declared as a char, which is probably not the intention.

Answer

typedef also allows to delcare arrays,

1. typedef char char_arr[];
2. char_arr my_arr = "Hello World!\n";

This is equal to

1. char my_arr[] = "Hello World!\n";

This may lead to obfuscated code when used too much, but when used correctly it is extremely useful to make code more compat and easier to read.

A #define is just a macro, i.e. it will be processed/expanded by the preprocessor.
When the actual C compiler starts doing its work there will be no sign of any
INT32 anymore. The typedef language facility is handled by the C compiler
itself: it introduces another name for a type. It's an awful hack actually: when
the C compiler parses that name in the right context the name is supposed
the alias for the type, otherwise it's supposed to be an identifier name:

Correct about the #define part. A typedef is just a new name for an already
existing type. defines are handled by the preprocessor while typedefs are
handled by the C compiler itself.

==========

Is this a correct method of allocating a 3-dimensional array?

```
/* assume all malloc's succeed */
#define     MAT 2
#define     ROW 2
#define     COL 2
   char ***ptr = malloc ( MAT * sizeof *ptr );

   for ( i = 0; i < MAT; i++ )
      ptr[i] = malloc ( ROW * sizeof **ptr );

   for ( i = 0; i < MAT; i++ )
      for ( j = 0; j < ROW; j++ )
         ptr[i][j] = malloc ( COL * sizeof ***ptr );
```

typedef Vs #define

There are two differences between define and typedef.

Firstly, typedef obeys scoping rules just like variables, whereas define

stays valid until the end of the file (or until a matching undef).

Secondly, some things can be done with typedef that cannot be done with define.

Examples:

Code:

```
typedef int* int_p1;
int_p1 a, b, c;  // a, b, and c are all int pointers.

#define int_p2 int*
int_p2 a, b, c;  // only the first is a pointer!
```

Code:

typedef int a10[10];

a10 a, b, c; // create three 10-int arrays

Code:

typedef int (*func_p) (int);

func_p fp // func_p is a pointer to a function that

    // takes an int and returns an int

What is the difference between typedef & Macros?

Typedef is used to create a new name to an already existing data type. Redefine the name creates conflict with the previous declaration.

eg:

typedef unsigned int UINT32

Macros [#define] is a direct substitution of the text before compling the whole code. In the given example its just a textual substitution. where there is a posibility of redifining the macro

eg:

#define chPointer char *

#undef chPointer

#define chPointer int *

==========

What's the difference between using a typedef or a define for a user-defined type?

In general, typedefs are preferred, in part because they can correctly encode pointer types. For example, consider these declarations:

```
typedef char *String_t;
#define String_d char *
String_t s1, s2;
String_d s3, s4;
```

s1, s2, and s3 are all declared as char *, but s4 is declared as a char, which is probably not the intention.

#defines do have the advantage that #ifdef works on them On the other hand, typedefs have the advantage that they obey scope rules (that is, they can be declared local to a function or block).

==========

What's the difference between using a typedef or a define for a user-defined type?

In general, typedefs are preferred, in part because they can correctly encode pointer types. For example, consider these declarations:

```
typedef char *String_t;
#define String_d char *
String_t s1, s2;
String_d s3, s4;
```

s1, s2, and s3 are all declared as char *, but s4 is declared as a char, which is probably not the intention.

#defines do have the advantage that #ifdef works on them On the other hand, typedefs have the advantage that they obey scope rules (that is, they can be declared local to a function or block).

==========

With respect to function parameter passing, what is the difference between call-by-value and call-by-reference? Which method does C use?

In the case of call-by-reference, a pointer reference to a variable is passed into a function instead of the actual value. The function's operations will effect the variable in a global as well as local sense. Call-by-value (C's method of parameter passing), by contrast, passes a copy of the variable's value into the function. Any changes to the variable made by function have only a local effect and do not alter the state of the variable passed into the function.

==========

How to declare a pointer to a function?

Use something like this

```
int myfunc(); // The function.
int (*fp)();  // Pointer to that function.
fp = myfunc;  // Assigning the address of the function to the pointer.
(*fp)();      // Calling the function.
fp();         // Another way to call the function.
```

==========

Is *(*(p+i)+j) is equivalent to p[i][j]? Is num[i] == i[num] == *(num + i) == *(i + num)? :: Yes

((p+i)+j) == p[i][j].

So is

num[i] == i[num] == *(num + i) == *(i + num)

==========

What is a memory leak?

Its an scenario where the program has lost a reference to an area in the memory.

Its a programming term describing the loss of memory. This happens when the program allocates some memory but fails to return it to the system

What is the difference between an array of pointers and a pointer to an array?

This is an array of pointers

int *p[10];

This is a pointer to a 10 element array

int (*p)[10];

==========

What is a void pointer? Why can't we perform arithmetic on a void * pointer?

The void data type is used when no other data type is appropriate. A void pointer is a pointer that may point to any kind of object at all. It is used when a pointer must be specified but its type is unknown.

The compiler doesn't know the size of the pointed-to objects incase of a void * pointer. Before performing arithmetic, convert the pointer either to char * or to the pointer type you're trying to manipulate

==========

One dimensional array

```
int *myarray = malloc(no_of_elements * sizeof(int));
//Access elements as myarray[i]
```

Two dimensional array

Method1

```
int **myarray = (int **)malloc(no_of_rows * sizeof(int *));
for(i = 0; i < no_of_rows; i++)
{
 myarray[i] = malloc(no_of_columns * sizeof(int));
}
// Access elements as myarray[i][j]
```

==========