**Programming Questions and Answers**

**C Language:**

**1. Memory Layout of a C program?**

The register variables are stored in CPU registers and the rest of variables are stored in memory.

Now let’s see how memory organized when a c program is run.

Memory Layout consists of

1) Code Segment or Text Segment ( Code of the Program )

2) Initialized Data Segment Initialized Static Variables

Initialized Global Variables

3) Uninitialized Data Segment Uninitialized Static Variables

Uninitialized Global Variables

4) Heap Segment Dynamically Allocated Memory

5) Stack Segment Initialized and uninitialized automatic variables

**1. Code Segment**

This Segment is used to store executable code of the program. The size of this segment does not change during run time.

**2, 3 . Data Segment**

This segment stores static and global variables. It is further subdivided Uninitialized and initialized data segments. The initialized data segment store all the initialized static and global variables. The uninitialized data segment stores all the uninitialized data segment.

The reason for division of this segment is that all uninitialized variables can be collectively assigned to zero.

The size of this segment is also fixed and does not change during run time.

**4. Heap Segment**

This segment is used for dynamically allocated memory. It is responsibility of the programmer to allocate memory from the heap.

The size of this segment is dynamic i.e, it may change during run time.

**5. Stack Segment**

Automatic variables are stored in this segment. The size of this segment keeps on changing during run time.

**2. What is storage class specifier?**

Storage class specifiers in C language tells the compiler where to store a variable,

how to store the variable, what is the initial value of the variable and life time of the

variable.

Syntax: storage\_specifier data\_type variable \_name

Different Types of storage class specifier in c :

There are four storage class specifier in C Language. They are,

1. Automatic

2. Register

3. Static

4. Extern

When a storage class is not specified in the declaration, compiler assumes default storage class based on place declaration.

A storage class decides about these four aspects of the variable

1) Life time - Time between creation and destruction of the variable

2) Scope - Locations where the variable is available to use.

3) Initial Value - Default value taken by uninitialized variable.

4) Place of storage - Place of memory where the storage is allocated for the variable.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Storage Class** | **Place Of declaration** | **Life time** | **Scope** | **Initial Value** | **Place of Storage** | **Linkage** |
| auto | Inside the  function | With in the function only | Local | Garbage | Memory  ( Stack ) | None |
| Static ( Local ) | Inside the function | **through out the program.**  Retain the value of the variable between function calls | Local | zero | Memory  ( Data ) | None |
| Static  ( Global ) | Outside the function | **through out the program.**  variable is limited to with in the file | With in the file | zero | Memory  ( Data ) | Internal |
| register | Inside the function | With in the function only | Local | Garbage | CPU registers | None |
| Extern | Outside or inside function. | Till the end of the main program | Global | Zero | Memory  ( Data ) | External |
| None | Outside function | Till the end of the main program | Global | Zero | Memory  ( Data ) | External |

**4. What is difference between static local and static global variable?**

Ans:

**Static global :**

**Scope : With in the File, Life time: through out the program.**

Static variable has scope only in the file in which it is declared. it can't be accessed in any other file

but its value remains intact if code is running in some other file means lifetime is in complete

program .

**Static local:**

**Scope : With in the function , Life time: t****hrough out the program.**

static local variable has scope in that function in which it is declared. means it can't be used in other

functions in the same file also, means scope is limited to the function in which it is declared while

its life time is also through out the program.

**5. Difference between static variable and global variable**

Global variables are variables which are defined in the outside the function. They have external linkage, which means that in other source files, the same name refers to the same location in memory

Static global variables are private to the source file where they are defined and do not conflict with other variables in other source files which would have the same name.

If you declare a variable with in function is called as a static local, it retains the value of variable between function calls, but scope is local only. But life of the variable is through out the program.

**6. const char \*ptr vs char \*const ptr**

Const can be applied to the declaration of any variable to specify that its value will not be changed

const int a; //constant integer val

int const a; //constant value to integer same as 1 st one

const int \*a; //pointer to constant integer we cant change value but we can assign another address to

pointer a like \*a= &b ,but cont change if \*a=10 cannot change to another val 19

int \* const a; //constant pointer to integer we cnt change the pointer but we can change the val …

int const \* a const; //we cant change pointer and value also

int \*a ( Pointer to Variable ) , const int \*ptr; ( Pointer to constant ), Constant pointer to variable. int \*const ptr; constant pointer to constant const int \*const ptr;

**7. What is volatile keyword?**

Ans: volatile just tells the compiler or force the compiler to "not to do the optimization" for that

variable.

so compiler would not optimize the code for that variable and reading the value from

the specified location

**8. Difference between Register keyword and Volatile keyword.**

Ans: Use of register keyword should increase the performance of the system where as the use of volatile will decrease performance due to multiple accesses to memory.

The registers do not have the address so the operator & cannot be used on them.

**9. What is dangling pointer, NULL pointer and void pointer?**

Void pointer or generic pointer is a special type of pointer that can be pointed at objects of any data type.

Pointer assigned value NULL is called as NULL pointer

When a pointer is pointing to non-existing memory location is called dangling pointer.

Null pointer is a pointer which is pointing to nothing. Null pointer points to empty location in

memory. Value of null pointer is 0. We can make a pointer to point to null as below.

The following are the differences between strcpy() and memcpy():  
  
- memcpy() copies specific number of bytes from source to destination in RAM, where as strcpy() copies a constant / string into another string.  
  
- memcpy() works on fixed length of arbitrary data, where as strcpy() works on null-terminated strings and it has no length limitations.  
  
- memcpy() is used to copy the exact amount of data, whereas strcpy() is used of copy variable-length null terminated strings.

Array is collection of similar data items.

**Pointer Uses**

Some of the Uses of pointers are

I) Accessing array elements,

II) Returning more than one value from a function

III) Accessing Dynamically allocated memory

IV) Implementing data structures like linked list , trees.

A pointer is variable that stores memory address.

Call by value, only the value of arguments are sent to function while in call by reference addresses of arguments are sent to function. In call by value method any changes made to the formal arguments will not effect to actual argument.

**Question: How to deallocate dynamically allocate memory without using “free()” function.**

Solution: Standard library function [realloc()](http://www.cplusplus.com/reference/clibrary/cstdlib/realloc/) can be used to deallocate previously allocated memory. Below is function declaration of “realloc()” from “stdlib.h”

*filter\_none*

*edit*

*play\_arrow*

*brightness\_4*

|  |
| --- |
| void \*realloc(void \*ptr, size\_t size); |

If “size” is zero, then call to realloc is equivalent to “free(ptr)”. And if “ptr” is NULL and size is non-zero then call to realloc is equivalent to “malloc(size)”.

**Question: Zombie process and Orphan process.**

Orphan - Parent exit , Init process becomes the parent of child process. So child is terminated and process table gets deleted by os.

Zombie - When the child terminates it gives exit status to parent. Meanwhile time suppose your parent is in sleep state and unable to receive any status from child. Though the child exit but the process occupies space in process table

Zombie Process:

A process which has finished the execution but still has entry in the process table to report to its parent process is known as a zombie process. A child process always first becomes a zombie before being removed from the process table. The parent process reads the exit status of the child process which reaps off the child process entry from the process table.

In the following code, the child finishes its execution using exit() system call while the parent sleeps for 50 seconds, hence doesn’t call [wait()](https://en.wikipedia.org/wiki/Wait_(system_call)) and the child process’s entry still exists in the process table.

|  |
| --- |
| // A C program to demonstrate Zombie Process.  // Child becomes Zombie as parent is sleeping  // when child process exits.  #include <stdlib.h>  #include <sys/types.h>  #include <unistd.h>  int main()  {      // Fork returns process id      // in parent process      pid\_t child\_pid = fork();        // Parent process      if (child\_pid > 0)          sleep(50);        // Child process      else          exit(0);        return 0;  } |

Note that the above code may not work with online compiler as fork() is disabled.

Orphan Process:

A process whose parent process no more exists i.e. either finished or terminated without waiting for its child process to terminate is called an orphan process.

In the following code, parent finishes execution and exits while the child process is still executing and is called an orphan process now.

However, the orphan process is soon adopted by init process, once its parent process dies.

*brightness\_4*

|  |
| --- |
| // A C program to demonstrate Orphan Process.  // Parent process finishes execution while the  // child process is running. The child process  // becomes orphan.  #include<stdio.h>  #include <sys/types.h>  #include <unistd.h>    int main()  {      // Create a child process      int pid = fork();        if (pid > 0)          printf("in parent process");        // Note that pid is 0 in child process      // and negative if fork() fails      else if (pid == 0)      {          sleep(30);          printf("in child process");      }        return 0;  } |

Q: Difference between inline and #define macro

The major difference between inline functions and macros is the way they are handled. Inline functions are parsed by the compiler, whereas macros are expanded by the C++ pre processor.

When the pre processor comes across any occurrences of SUM(first, last) in the code, then that text will be replaced by (first + last). When would one want to use a macro? Usually when what you’re substituting for is very simple, and does not justify the overhead of a function call. Remember that function calls do incur overhead.

Inline functions, as mentioned earlier, are parsed by the compiler directly instead of the preprocessor. Inline functions look very similar to regular functions.

The difference between an inline function and a regular function is that wherever the compiler finds a call to an inline function, it writes a copy of the compiled function definition. However, with a regular function, a normal function call is generated.

The reason C++ has inline functions and macros is to eliminate the overhead incurred by function calls. However, the tradeoff of this is the fact that the program size increases with both macros and inline functions. Remember that inline functions look like regular functions, but macros are implemented with text replacement.

Debugging macros is also difficult. This is because the preprocessor does the textual replacement for macros, but that textual replacement is not visible in the source code itself. Because of all this, it’s generally considered a good idea to use inline functions over macros.

Source : <https://www.programmerinterview.com/c-cplusplus/inline-vs-macro/>

Inline functions provides following advantages over macros.

* Since they are functions so type of arguments is checked by the compiler whether they are correct or not.
* There is no risk if called multiple times. But there is risk in macros which can be dangerous when the argument is an expression.
* They can include multiple lines of code without trailing backlashes.
* Inline functions have their own scope for variables and they can return a value.
* Debugging code is easy in case of Inline functions as compared to macros.

Inline functions can provide scope for variables (particularly static ones), preprocessor macros can only do this in code blocks {...}, and static variables will not behave exactly the same way.

**Q: struct hi { char jj; static int i; };**

Ans: Error  **expected specifier-qualifier-list before ‘static’**

Coming to struct - A struct is a derived type. You define a variable of a particular declared struct type. This variable, say X, is referred to as one whole data item. The structure in turn can be local to a function (in which case it will be given memory on stack), it can be a global (in which case it will be in the BSS or data segment) or it can be allocated dynamically from heap. Whatever the case, ***all its members are continuous in memory.***  
So, it is possible to only make the whole structure with all its elements of static type, *because it is not possible to separate one member out to the data segment by making it static.*

If it were possible to make one member as static, *then that would defeat the purpose of a struct variable,* with the static member always in data segment and the rest of the members elsewhere in memory with different life spans.

Codes:-

int ToggleBit(int num,int pos)

{

return num ^ ( 1 << pos ) ;

}

int ResetBit(int num,int pos)

{

return num & ( ~ ( 1 << pos ) );

}

int SetBit(int num,int pos)

{

return num | ( 1 << pos );

}

int PrintAllBits(int num)

{

int i=0;

for(i= ( sizeof(int)\*8)-1; i>=0 ; i-- )

printf("%d ", (( num>>i ) & 1 ) );

printf("\n");

return 0;

}

endian

{

unsigned int i = 1;

char \*c = (char\*)&i;

if (\*c)

printf("Little endian"); // intel family

else

printf("Big endian"); // Motorola family

getchar();

return 0;

}

unsigned int reverseBits(unsigned int num)

{

unsigned int rev =0 ,pos = (sizeof(int) \* 8 ) -1 ;

while(num)

{

if (num & 1 )

rev = rev | 1<<pos;

num = num >> 1;

pos--;

}

return rev;

}

int Check\_Pwr\_of\_4(int num)

{

if ( !( num & ( num -1) ) && (num & 55555555) )

printf("Power of 4 \n");

else

printf("Not A power of 4\n");

return 0;

}

int Check\_PositiveorNegative(int num)

{

if ( num \* -1 > num )

printf("- %d \n",num);

else

printf("+ %d \n",num);

return 0;

}

int PrintAllBits(int num)

{

int i=0;

for(i= ( sizeof(int)\*8) -1; i>=0 ; i-- )

printf("%d ", (( num>>i ) & 1 ) );

printf("\n");

return 0;

}

int PowerOf2\_Logic2(int num)

{

if ( num & ( num -1 ) )

printf(" Not A power of 2 \n");

else

printf(" Power of 2 \n");

return 0;

}

int CountNumberOfSetBits(int num)

{

int i=0,count=0;

for(i= ( sizeof(int)\*8) -1; i>=0 ; i-- )

if ( (( num>>i ) & 1 ) == 1 )

count++;

printf("Total Set Bit Count = %d\n",count);

return count;

}

int PowerOf2(int num)

{

// Logic is if Power 2, only one bit is 1 all are zero

if ( CountNumberOfSetBits(num) == 1 )

printf(" Power Of 2\n");

else

printf("Not A Power of 2\n");

return 0; }

int odd\_or\_even(int num)

{

// Logic is, if bit is odd, it should have 0th bit is 1

if ( num & 1 )

printf(" Odd \n");

else

printf(" Even Number \n");

return 0;

}

void palidrome(int num)

{

int tmp = num;

int val=0;

while(tmp )

{

val = val \* 10 + ( tmp%10);

tmp = tmp/10;

}

if ( val == num )

fprintf(stdout," Paligrame\n");

else

fprintf(stdout," Paligrame kadhu\n");

return;

}

int myatoi(char \*str)

{

int i,tmp=0;

for(i=0;str[i];i++)

{

if ( str[i] >= '0' && str[i] <= '9' )

tmp=tmp\*10+ str[i]-'0';

else break;

}

return tmp;

}

**Virtual Address :-**

Virtual addresses as the name says are virtual, i.e. they do not point to

any address in RAM directly but need to be converted into physical

address by MMU at run time.

**Drivers :-**

Ques :- What is character Driver.

Ans:-

Character device drivers normally perform I/O in a byte stream.

Generally provide interfaces to the application layer.

Char driver infrastructure is composed of bunch of logical file system though which a kernel service can interact with applications.

Vfs and sysfs most popular choice among logical file system

Vfs is file system abstraction layer which provide common file api to user mode applications

Vfs api’s Perform operations on file descriptor

Ques:- How can you find memory leak happens in kernel

Ques: Difference Between Semaphore and Mutex and binary semaphore

Ques: check linked list is palidrome or not

Ques: Find detection of loop in linked list

Ques: Find architecture size logic without using sizeof

Ques: Find nth node from end of the list in linked list