Q.Define datatype ?

In the **C** programming language, **data types** are declarations for memory locations or variables that determine the characteristics of the **data** that may be stored and the methods (operations) of processing that are permitted involving them.

Data types simply refers to the type and size of data associated with variables and functions.

The size of integer is basically depends upon the `architecture` of your system. Generally if you have a `16-bit` machine then your `compiler` will must support a int of `size 2 byte.` If your system is of 32 bit,then the compiler must support for 4 byte for integer.

In more details,

- The concept of `data bus` comes into picture yes,16-bit ,32-bit means nothing but the `size of data bus` in your system.
- The data bus size is required for to determine the size of an integer because,The purpose of data bus is to provide data to the processor.The max it can provide to the processor at a single fetch is important and this max size is preferred by the compiler to give a data at time.
- Basing upon this data bus size of your system the compiler is designed to provide max size of the data bus as the size of integer.

Q. define typedef ?

typedef is a keyword used in *C* language to assign alternative names to existing types.

Type Name	**32–bit Size**	**64–bit Size**
char	1 byte	1 byte
short	2 bytes	2 bytes
int	4 bytes	4 bytes

Type Name	32–bit Size	64–bit Size
long	4 bytes	8 bytes
long long	8 bytes	8 bytes

Type Name	32–bit Size	64–bit Size
float	4 bytes	4 bytes
double	8 bytes	8 bytes
long double	16 bytes	16 bytes

13) What is array in C?

Array is a group of similar types of elements. It has contiguous memory location. It makes the code optimized, easy to traverse and easy to sort. More details...

14) What is pointer in C?

A pointer is a variable that refers to the address of a value. It makes the code optimized and makes the performance fast.

17) What is far pointer in C?

A pointer which can access all the 16 segments (whole residence memory) of RAM is known as far pointer.

18) What is dangling pointer in C?

If a pointer is pointing any memory location but meanwhile another pointer deletes the memory occupied by first pointer while first pointer still points to that memory location, first pointer will be known as dangling pointer. This problem is known as dangling pointer problem.

19) What is pointer to pointer in C?

In case of pointer to pointer concept, one pointer refers to the address of another pointer. More details...

20) What is static memory allocation?

In case of static memory allocation, memory is allocated at compile time and memory can't be increased while executing the program. It is used in array. More details...

21) What is dynamic memory allocation?

In case of dynamic memory allocation, memory is allocated at run time and memory can be increased while executing the program. It is used in linked list. More details...

9) What is the use of static variable in C?

A variable which is declared as static is known as static variable. The static variable retains its value between multiple function calls.

```
1. Void function1(){
2. intx=10;//local variable
3. static int y=10;//static variable
4. x=x+1;
5. y=y+1;
6. printf("%d\n",x);//will always print 11
7. printf("%d\n",y);//will always increment value, it will print 11, 12, 13 and so on
8. }
```

Q.What is the use of function in C?

A function in C language provides modularity. It can be called many times. It saves code and we can reuse the same code many times. More details...

Q.What is the difference between call by value and call by reference in C?

We can pass value to function by one of the two ways: call by value or call by reference. In case of call by value, a copy of value is passed to the function, so original value is not modified. But in case of call by reference, an address of value of passed to the function, so original value is modified. More details...

Q. What is array in C?

Array is a group of similar types of elements. It has contiguous memory location. It makes the code optimized, easy to traverse and easy to sort. More details...

Q. What is pointer in C?

A pointer is a variable that refers to the address of a value. It makes the code optimized and makes the performance fast. More details...

Q. What are the usage of pointer in C?

- Accessing array elements
- Dynamic memory allocation
- Call by Reference
- Data Structures like tree, graph, linked list etc.

Q. What is NULL pointer in C?

A pointer that doesn't refer to any address of a value but NULL, is known as NULL pointer. For example:

```
1. int *p=NULL;
```

Q. What is far pointer in C?

A pointer which can access all the 16 segments (whole residence memory) of RAM is known as far pointer.

Q. What is dangling pointer in C?

If a pointer is pointing any memory location but meanwhile another pointer deletes the memory occupied by first pointer while first pointer still points to that memory location, first pointer will be known as dangling pointer. This problem is known as dangling pointer problem.

Q. What is pointer to pointer in C?

In case of pointer to pointer concept, one pointer refers to the address of another pointer. More details...

Q. What is static memory allocation?

In case of static memory allocation, memory is allocated at compile time and memory can't be increased while executing the program. It is used in array. More details...

Q. What is dynamic memory allocation?

In case of dynamic memory allocation, memory is allocated at run time and memory can be increased while executing the program. It is used in linked list. More details...

Q.What functions are used for dynamic memory allocation in C language?

1. malloc()
2. calloc()
3. realloc()
4. free()

More details...

Q. What is the difference between malloc() and calloc()?

malloc(): The malloc() function allocates single block of requested memory. It has garbage value initially.

calloc(): The calloc() function allocates multiple block of requested memory. It initially initializes all bytes to zero.

More details...

Q. What is structure?

Structure is a user-defined data type that allows to store multiple types of data in a single unit. It occupies the sum of memory of all members. More details...

Q. What is union?

Like Structure, union is a user-defined data type that allows to store multiple types of data in a single unit. But it doesn't occupies the sum of memory of all members. It occupies the memory of largest member only. More details...

Q.What is the purpose of sprintf() function?

It is used to print the formatted output into char array.

Q. Can we compile a program without main() function?

Yes, we can compile but it can't be executed.

But, if we use #define, we can compile and run C program without using main() function. For example:

```
1. #include<stdio.h>
2. #define start main
3. void start(){
4. printf("Hello");
5. }
```

Q.What is Linux Kernel?

Linux Kernel is low level system software. It is used to manage the hardware resources for the users. It provides an interface for user-level interaction

Q.Difference b/w OS and kernel.?

The kernel is the part of the operating system. The operating system is the software package that communicates directly to the hardware and our application. The kernel is the lowest level of the operating system. The kernel is the main part of the operating system and is responsible for translating the command into something that can be understood by the computer. The main functions of the kernel are:

1. memory management
2. network management
3. device driver
4. file management
5. process management

Q. What is Swap Space?

Swap space is used to specify a space which is used by Linux to hold some concurrent running program temporarily. It is used when RAM does not have enough space to hold all programs that are executing.

Q.What is the difference between BIOS and Firmware?

BIOS is an acronym for Basic Input/Output System and also known as System BIOS, ROM BIOS, or PC BIOS. It is a type of Firmware used during the booting process (power-on/start up) on IBM PC compatible computers. BIOS Firmware is built into PCs, and it is the first software they run when powered on. The name itself originates from the Basic Input/Output System used in the CP/M operating system in 1975.

Firmware is a combination of persistent memory, program code, and the data stored in it. Typical examples of devices containing Firmware are embedded systems such as traffic lights, consumer appliances, digital watches, computers, computer peripherals, mobile phones, and digital cameras. The Firmware contained in these devices provides the control program for the device

Q. what is IRQ - interrupt request line?

Abbreviation of ***i****nterrupt* ***re****q****u****est line*, and pronounced *I-R-Q*. IRQs are hardware lines over which devices can send interrupt signals to the microprocessor.

Q.what is ARM Architecture ?

An ARM processor is one of a family of CPUs based on the RISC (reduced instruction set computer) architecture developed by Advanced RISC Machines (ARM).

Q.what is difference between Executable fiel and Object file?

Object code is the output of a *compiler* after it processes *source code*.

Source code is the version of a computer program as it is originally *written* (i.e., typed into a computer) by a human in a programming language. A compiler is a specialized program that converts source code into object code.

The object code is usually a *machine code*, also called a *machine language*, which can be understood directly by a specific type of CPU (central processing unit), such as x86 (i.e., Intel-compatible) or PowerPC.

A machine code file can be immediately *executable* (i.e., runnable as a program), or it might require *linking* with other object code files (e.g. *libraries*) to produce a complete executable program.

An object code file can contain not only the object code, but also relocation information that the *linker* uses to assemble multiple object files to form an executable program. It can also contain other information, such as program symbols (names of variables and functions) and *debugging* (i.e., removing errors) information

(1)Executable file is simply understand by machine .

This file is used to perform various operations or functions on a computer. An executable files cannot be read by humans .A file in a format that the computer can directly execute . If you transform a source file into an executable file, you need to

pass it through a compiler or assembler .

(2**) Object file** is the intermediate form between executable and source .

Objective file is the file containing object code, means reload format machine code that is usually not directly executable. This file may also work like a shared library.

Objective files are three types -
(1) Relocatable file (2) Executable file (3) Shared object file

Q.how to Passing Command Line Arguments to a Module ?

module_param takes three parameters: the name of the variable, its type, and a permissions mask to be used for an accompanying sysfs entry. The macro should be placed outside of any function and is typically found near the head of the source file. So hellop would declare its parameters and make them available to insmod as follows:

```
static char *whom = "world";
static int howmany = 1;
module_param(howmany, int, S_IRUGO);
module_param(whom, charp, S_IRUGO);
```

insmod hellop howmany=10 whom="Mom"

To declare an array parameter, use:

module_param_array(name,type,num,perm);

Where name is the name of your array (and of the parameter), type is the type of the array elements, num is an integer variable, and perm is the usual permissions value.

If the array parameter is set at load time, num is set to the number of values supplied.
The module loader refuses to accept more values than will fit in the array.

Q.what is Module?

A module is a part of a program. Programs are composed of one or more independently developed modules that are not combined until the program is linked. A single module can contain one or several routines.

Q. What is driver ?

The software that handles or manages a hardware controller is known as a device **driver.**

Q.what is CP, ELF,SHL , vmlinux , uname -r ,-a ,-m ? - how CP will work ?

uname Command

Purpose

Displays the name of the current operating system.

Syntax

uname [**-a** | **-x** | **-S** *Name*] [**-F**] [**-f**] [**-l**] [**-L**] [**-m**] [**-M**] [**-n**] [**-p**] [**-r**] [**-s**] [**-T** *Name*] [**-u**] [**-v**] [**-W**]

Description

The **uname** command writes to standard output the name of the operating system that you are using.

The machine ID number contains 12 characters in the following digit format: *xxyyyyyymmss*. The *xx* positions indicate the system and is always 00. The *yyyyyy* positions contain the unique ID number for the entire system. The *mm* position represents the model ID. The *ss* position is the submodel number and is always 00. The model ID describes the ID of the CPU Planar, not the model of the System as a whole.

Most machines share a common model ID of 4C.

The machine identifier value returned by the **uname** command may change when new operating system software levels are installed. This change affects applications using this value to access licensed programs. To view this identifier, enter the **uname -m** command.

Contact the appropriate support organization if your application is affected.

Flags

Item	Description
-a	Displays all information specified with the **-m**, **-n**, **-r**, **-s**, and **-v** flags. Cannot be used with the **-x** or **-S***Name* flag. If the **-x** flag is specified with the **-a** flag, the **-x** flag overrides it.
-F	Displays a system identification string comprised of hexadecimal characters. This identification string is the same for all partitions on a particular system.
-f	Similar to the **F** flag, except that the partition number is also used in the calculation of this string. The resulting identification string is

Item	Description
	unique for each partition on a particular system.
-l	Displays the LAN network number.
-L	Displays LPAR number and LPAR name. If LPAR does not exist, -1 is displayed for LPAR number and NULL for LPAR name. If a system is capable of LPAR, but is currently running in Symmetric Multi Processing (SMP) mode, 1 is displayed for LPAR number and NULL for LPAR name.
-m	Displays the machine ID number of the hardware running the system. Note The **-m** flag cannot be used to generate a unique machine identifier for partitions in an LPAR environment.
-M	Displays the system model name. If the model name attribute does not exist, a null string is displayed.
-n	Displays the name of the node. This may be a name the system is known by to a UUCP communications network.
-p	Displays the architecture of the system processor.
-r	Displays the release number of the operating system.
-s	Displays the system name. This flag is on by default.
-S *Name*	Sets the name of the node. This can be the UUCP communications network name for the system.
-T *Name*	Sets the system name. This can be the UUCP communications network name for the system.
-u	Displays the system ID number. If this attribute is not defined, the output is the same as the output displayed by **uname -m**.
-v	Displays the operating system version.
-W	Displays the static workload partition identification number. If the uname command runs in the Global environment, a value of zero is displayed.
-x	Displays the information specified with the **-a** flag as well as the LAN network number, as specified by the **-l** flag.

If you enter a flag that is not valid, the **uname** command exits with an error message, an error return status, and no output.

> **Note:** The uname command does not preserve the new system name and node name values across system reboot.

Exit Status

This command returns the following exit values:

Item	Description
0	The requested information was successfully written.
>0	An error occurred.

Security

Attention RBAC users and Trusted AIX users
This command can perform privileged operations. Only privileged users can run privileged operations. For more information about authorizations and privileges, see Privileged Command Database in *Security*. For a list of privileges and the authorizations associated with this command, see the lssecattr command or the getcmdattr subcommand.

Example

To display the complete system name and version banner, enter:

uname -a

Files

Item	Description
/usr/bin/uname	Contains the **uname** command.

Q. Cross Compilation steps?

Very Simple Guide for Building Cross Compilers Tips

The following is steps to build cross compilers. So far, I tried only for PowerPC and MIPS.

Basically, all you have to do is to follow the following 9 steps.

1. Download sources
2. Set environment variables
3. Build binutils

1. What do you need?

First, you have to obtain the following source codes

- binutils http://www.gnu.org/software/binutils/
- GCC http://www.gnu.org/software/gcc/gcc.html
- newlib http://sources.redhat.com/newlib/
- GDB http://www.gnu.org/software/gdb/gdb.html

2. Set environment variables

First, choose your taget such as powerpc-eabi, powerpc-elf, mips-elf, and so on

For bash simply type

```
% export TARGET=powerpc-eabi
% export PREFIX=/usr/local/$TARGET
% export PATH=$PATH:$PREFIX/bin
```

3. Build binutils

```
% tar xjfv binutils-2.17.tar.bz2
% mkdir build-binutils
% cd build-binutils
% ../binutils-2.17/configure --target=$TARGET --prefix=$PREFIX
% make all
% make install
```

4. Build bootstrap GCC

```
% tar xjfv gcc-4.1.1.tar.bz2
% mkdir build-gcc
% cd build-gcc
% ../gcc-4.1.1/configure --target=$TARGET --prefix=$PREFIX --without-headers
--with-newlib  --with-gnu-as --with-gnu-ld
% make all-gcc
```

% make install-gcc

--with-gnu-as --with-gnu-ld prevents native assembler on certain architectures. (for others, these do not have any effects)

5. Build newlib

Newlib provides standard C library for embedded systems

% tar xzfv newlib-1.14.0.tar.gz
% mkdir build-newlib
% cd build-newlib
% ../newlib-1.14.0/configure --target=$TARGET --prefix=$PREFIX
% make all
% make install

6. Build GCC again with newlib

% cd build-gcc
% ../gcc-4.1.1/configure --target=$TARGET --prefix=$PREFIX --with-newlib --with-gnu-as --with-gnu-ld --disable-shared --disable-libssp
% make all
% make install

7. GDB with PSIM

% tar xjfv gdb-6.3.tar.bz2
% mkdir build-gdb
% cd build-gdb
% ../gdb-6.3/configure --target=$TARGET --prefix=$PREFIX --enable-sim-powerpc --enable-sim-stdio
% make all
% make install

Congratulations! You build your tool chain

8. Compile your code

Now, it's time to compile your code.

% powerpc-eabi-gcc -mcpu=405 hello.c -o hello -msim
% mips-elf-gcc -Tidt.ld -mips4 hello.c -o hello

-T option specifies libraries that include start code.

To Compile with specific Memory map

% powerpc-eabi-gcc -Wl,-Ttext,0x4000,-Tdata,0xf000 hello.c -msim
(-Wl,-Ttext,0x4000,-Tdata,0x10000)

9. Run

% powerpc-eabi-run hello
% mips-elf-run hello

Q. What is **Monolithic kernel and Micro kernel ?**

Monolithic kernel

All the parts of a kernel like the Scheduler, File System, Memory Management, Networking Stacks, Device Drivers, etc., are maintained in one unit within the kernel in **Monolithic Kernel**

Micro kernel

Only the very important parts like IPC(Inter process Communication), basic scheduler, basic memory handling, basic I/O primitives etc., are put into the kernel. Communication happen via message passing. Others are maintained as server processes in User Space.

Q.write a program to reverse the string using recursion ? And find string lenth using recursion ?

Q. write a program to swap the particular nods ?

Q. find the number of bits set in a number?

vmlinuz is the name of the Linux *kernel executable*.

A kernel is a program that constitutes the central core of a computer operating

system. It is the first thing that is loaded into *memory* (which physically consists of RAM chips) when a computer is *booted up* (i.e., started), and it remains in memory for the entire time that the computer is in operation. An executable, also called an *executable file*, is a file that can be run as a program.

vmlinuz is a compressed Linux kernel, and it is *bootable*. Bootable means that it is capable of loading the operating system into memory so that the computer becomes usable and application programs can be run.

vmlinuz should not be confused with *vmlinux*, which is the kernel in a non-compressed and non-bootable form. vmlinux is generally just an intermediate step to producing vmlinuz.

vmlinuz is located in the */boot* directory, which is the directory that contains the files needed to begin booting the system. The file named *vmlinuz* might be the actual kernel executable itself, or it could be a link to the kernel executable, which might bear a name such as */boot/vmlinuz-2.4.18-19.8.0* (i.e., the name of the specific version of the kernel).

ls -l /boot

The Linux kernel is *compiled* by issuing the following command:

```
make bzImage
```

This results in the creation of a file named *bzImage* in a directory such as */usr/src/linux/arch/i386/linux/boot/*.

Compilation is the conversion the kernel's *source code* (i.e., the original form in which the kernel is written by a human) into *object code* (which is understandable directly by a computer's processor). It is performed by a specialized program called a *compiler*, usually one in the *GCC* (GNU Compiler Collection).

bzImage is then copied using the *cp* (i.e., copy) command to the /boot directory and simultaneously renamed *vmlinuz* with a command such as

```
cp /usr/src/linux/arch/i386/linux/boot/bzImage /boot/vmlinuz
```

vmlinuz is not merely a compressed image. It also has *gzip* decompressor code built into it. gzip is one of the most popular compression utilities on Unix-like operating systems.

A compiled kernel named *zImage* file is created on some older systems and is

retained on newer ones for backward compatibility. Both zImage and bzImage are compressed with gzip. The difference is that zImage decompresses into *low memory* (i.e., the first 640kB), and bzImage decompresses into *high memory* (more than 1MB). There is a common misconception that bzImage is compressed with the *bzip2* utility; actually, the *b* just stands for *big*.

The name *vmlinuz* is largely an accident of history. The kernel binary on the original UNIX as developed at Bell Labs was called *unix*. When a new kernel containing support for *virtual memory* was subsequently written at the University of California at Berkeley (UCB), the kernel binary was renamed *vmunix*.

Virtual memory is the use of space on a hard disk drive (HDD) to simulate additional RAM (random access memory) capacity. It was supported by the Linux kernel almost from Linux's inception, in contrast to some other popular operating systems in use at the time, such as MS-DOS.

Thus, it was a natural progression for the Linux kernel to be called *vmlinux*. And because the Linux kernel executable was made into a compressed file and compressed files typically have a *z* or *gz* extension on Unix-like systems, the name of the compressed kernel executable became *vmlinuz*.

Q. Exporting symbols from module ?

Linux kernel allows modules stacking, which basically means one module can use the symbols defined in other modules. But this is possible only when the symbols are exported.

- *let us see how can a module export a symbol and how another module can make use of it.*

A symbol can be exported by a module using the macro EXPORT_SYMBOL().

Let us make use of the very basic hello world module. In this module we have added function called "hello_export" and used the EXPORT_SYMBOL macro to export this function.

hello_export.c

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");
static int hello_init(void)
```

```
{
 printk(KERN_INFO "Hello,world");
 return 0;
}
static hello_export(void) {
printk(KERN_INFO "Hello from another module");
return 0;
}
static void hello_exit(void)
{
 printk(KERN_INFO "Goodbye cruel world");
}
EXPORT_SYMBOL(hello_export);
module_init(hello_init);
module_exit(hello_exit);
```

Makefile required to compile this module is

Makefile

```
ifneq ($(KERNELRELEASE),)
   obj-m := hello_export.o
else
KERNELDIR ?= /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
default:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
clean:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) clean
endif
```

.....

Compile it using the "make" command and then insert it into the kernel using insmod

$make

$insmod hello_export.ko

All the symbols that the kernel is aware of is listed in /proc/kallsyms. Let us search for our symbol in this file.

$ cat /proc/kallsyms | grep hello_export d09c4000 T hello_export[hello_export]

From the output we can see that the symbol we exported is listed in the symbols recognized by the kernel. The first column gives the address of the column and the last column the module to which the symbol belongs to.

If you look into the Module.symvers file in directory where the module was compiled you will see a line similar to the following .

0x06577ca7 hello_export /home/user/hello EXPORT_SYMBOL

Now let us write another module which will make use of the exported symbol. To make use of the symbol we just have to inform the module that the symbols being used is defined externally by using the "extern" keyword. Thus to use the hello_export function we will have to use

extern hello_export();

Once this is done we can use the function any where in the module, as shown in the module below.

hello_use_export.c

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");
extern hello_export();
static int hello_init(void)
```

```
{
        printk(KERN_INFO "Hello,world");
        hello_export();
        return 0;
}
static void hello_exit(void)
{
        printk(KERN_INFO "Goodbye cruel world");
}
module_init(hello_init);
module_exit(hello_exit);
```

Use the same Makefile as shown above only change the module name to hello_use_export.o

Compile and load as done before.

```
$ make
$ insmod hello_use_export.ko
```

If the function has got called successfully we should see the message "Hello from another module" in the kernel logs.

```
$dmesg | tail -5 [22706.056990] Hello,world [23662.239681] Hello,world
[23662.239689] Hello from another module
```

Thus we could call the function defined in another module successfully once it was exported by the module.

The dependecy of hello_use_export on hello_export can be found using the lsmod command

```
$ lsmod |grep hello hello_use_export 590 0 hello_export 683 1 hello_use_export
```

We know the modprobe command can load the module that is passed to it as well as the dependent modules, hence we can also see the working of modprobe using these modules.

Copy the two .ko files, hello_export.ko and hello_use_export.ko to /lib/modules/`uname -r`/

$ cp hello_export.ko /lib/modules/`uname -r`/. $ cp hello_use_export.ko /lib/modules/`uname -r`/.

To use modprobe we have to update the dependencies in the modules.dep file in /lib/modules/`uname -r`/modules.dep . This can be done using the

command depmod.

$ depmod -a $ cat /lib/modules/2.6.32-5-686/modules.dep | grep hello hello_use_export.ko: hello_export.ko hello_export.ko:

The output "hello_use_export.ko: hello_export.ko" signifies that hello_use_export is dependent on hello_export.

Now remove the two modules we inserted before

$ rmmod hello_use_export $ rmmod hello_export

Note: You will not be able to remove the modules in the reverse order cause hello_use_export will be using hello_export. Hence unless hello_use_export is not removed, hello_export can not be removed.

Now use the modprobe command to load hello_use_export, and we will see that hello_export gets loaded automatically.

$ modprobe hello_use_export

$ lsmod |grep hello hello_use_export 590 0 hello_export 683 1 hello_use_export

We can see that even though we did not load the module "hello_export" it got

loaded automatically because "hello_use_export" was dependent on it.

EXPORT_SYMBOL(name);

EXPORT_SYMBOL_GPL(name);

“Either of the above macros makes the given symbol available outside the module.”

“The _GPL version makes the symbol available to GPL-licensed modules only. Symbols must be exported in the global part of the module’s file, outside of any function, because the macros expand to the declaration of a special-purpose variable that is expected to be accessible globally.”

“This variable is stored in a special part of the module executible (an “ELF section”) that is used by the kernel at load time to find the variables exported by the module.”

Probe function

Probe is called *when your device is recognized by the platform*.Generally registering the device requires the d*evice name* and this device name is compared with the driver name(when you are registering the driver you would be specifying *the driver name,which should match with the device name*).

> Once the device gets detected by the kernel it will call the corresponding driver probe for that device.