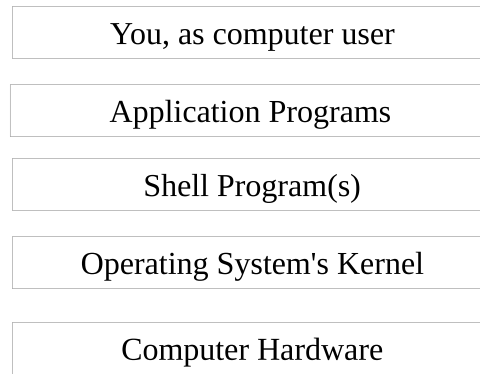
C Programming in Linux Environment

Srinivasulu Karasala

1. Computer and You

When you sit in front of a computer and working on it, following sub systems are involved:



At the top, you are present as a computer user. You may be running (also called executing) an application program. Following are some sample application programs you could be running on the computer:

- Running a Video game application
- Running a media player application to play audio or video files
- Running a browser application to browse the Internet
- Running a Paintbrush program to draw a nice picture
- Running a Notepad or WordPad or MS-Word to write or read some document
- Running Turbo-C compiler to write and build a program

When a computer is powered-on, CPU present in the hardware executes the BIOS program present in the ROM or Flash memory. Note that CPU can execute a program only if that program is present in the ROM(Flash) or RAM (DRAM/SDRAM). CPU can't execute a program that is present in the hard-disk or CD-ROM. On power-on no valid content is present in the RAM. So CPU executes the BIOS program present in the ROM. This program loads the Operating System's kernel from the hard-disk into RAM and executes the kernel. This kernel after initializing all the hardware, it loads the shell program present in the hard disk into RAM and executes. Note that OS kernel automatically runs the shell program.

Now the shell program allows the user to run other application programs. When user runs an application, it will also loaded into RAM (from hard disk) and get executed. When the application completes or user closes it, it will be removed from the RAM. Computer booting sequence is shown in the following figure.

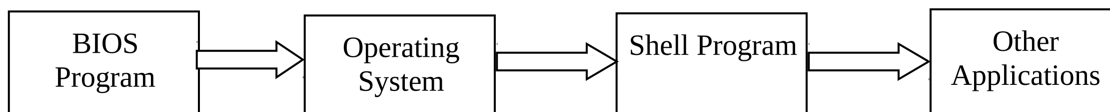


Figure 1. Computer booting sequence

2. More about Shell Programs

The program which allows us to run other programs is called a shell program. There are two types of shell programs, one is command shell, and other is graphical shell.

Graphical shell program allows us to run other programs by clicking program icons or names, with a mouse. Or by touching icon of program with a finger on tablet PC or smart phones.

The command shell allows us to run other programs by typing the program names. So to use command shell, one should know the names of the programs. We call these names (names of programs) as commands. So command shell allows us to run commands by typing the command names. But understand that we are actually typing a program name to run that program. When we type a command, we may also type arguments to the command. These arguments acts as 'options' or 'inputs' to the command. Look at the following four examples:

```
$ ls
$ ls -l
$ ls /bin
$ ls -l /bin
```

In the first case we are typing only command name '`ls`', to list the names of the files in the current directory. Note '`ls`' is short form for 'List'. In the second case we are typing option '**`-l`**' to the command. This '`-l`' option is to list the files in the long format. That is with more details about each file.

In the third case we are giving the name of the directory as input argument. So the command, lists the files present in that directory. In the last case, both option argument and input arguments are given. Most of the times options will have '-' as prefix. So it is also possible to interchange position of input and option arguments.

Shell program's purpose is to allow the user to run other application programs. But shell program itself is like any other application program. Only difference is that, shell is automatically started by the operating system. Other application programs are started by the user, through the shell program.

Once upon a time, operating systems used to have only command shells. Examples are old Unix operating system and DOS operating system. But now a days all most all operating systems have got graphical shell. Graphical shells are very convenient to use. So they are very popular. But command shells are much more powerful, as it offers lot more commands (programs) than with graphical shells. Each of these command also takes many options as arguments and makes each command much more flexible and powerful. It is also possible to combine two or more commands to perform a complex task. One can also type all the required commands in a text file and can use this text file as a new command. Such text files containing commands are called shell scripts or simply scripts. Writing such shell scripts is called shell programming.

High end Linux servers and embedded Linux devices may not be running the graphical shell. One need to interact with such systems using command shell only. Command shell is for power users like system administrators, system programmers and embedded programmers. Where as graphical shell is for normal users and kids.

Now all most all operating systems that are used in desktop computers will have graphical shells. When you power-on these desktop computers, you will see the graphical shell running. Note that graphical shell is the one, which displays Desktop and task-bar to you. However if command shell is also required, one can start the command shell program, from the graphical shell.

Now command shell runs and displays a command window. User can run large number of programs by typing their names (i.e. commands) in this command shell window. So graphical shell and command shell can be present at the same time. We normally call this command shell window, as 'command terminal' or 'terminal window' or simply 'terminal'.

Also note that for Linux computers, we can disable the graphical shell. So when power-on such Linux computers, you will get command line shell directly. As mentioned earlier, Linux servers and some embedded Linux devices may have only command shells.

Review Questions

1. What is the purpose of shell programs? explain in your own words
2. What are two types of shell programs?
3. Do all the Linux computers, will have graphical shell, explain.
4. What are the advantages and disadvantages of command shell and graphical shell
5. Is it possible to have graphical shell and command shell at the same time
6. What are command arguments?
7. Command arguments can be option or input, explain the difference
8. What is the difference between shell programs and other application programs?
9. What are shell scripts?
10. Does the Windows OS got the command shell?

GUI and CLI (console) Programs

Every program we run, interacts with us, by giving output, or by taking input, or by doing the both. We can classify these programs into two types, based on how these programs are interacting with the users. Programs that interact with the users through graphical objects, mouse and keyboard are called GUI programs.

Second type of programs give only text as output and takes text as input through the keyboard. These second type of programs are called CLI (command line interface) programs. CLI programs are also some times referred as *console programs.*

Activity:
In a command terminal type the command 'date' to know the today's date and time. Try the 'cal' (short name for calender) command with various arguments as given below:

```
$ cal
$ cal 2013
$ cal -3
$ cal -m 6
$ cal 8 2013
```

First command displays the calender of current month. Second one calender of year 2013. Third one shows calender of previous, current and next months. Fourth command shows calender of 6th month of current year. The last command displays the calender of 8th (August) month of year 2013. So this is the flexibility possible with the CLI programs. But you have to remember

all these possible options. That is the downside of CLI programs. One can do all these things with GUI calender program.

Another advantage of CLI programs is that, they are much easier to develop than their GUI counterparts. Because of this reason, there exist hundreds of CLI programs (commands) already. In fact most of the programs we are developing as part learning C, are also commands only. Main difference between commands and our executable programs is that, command executable files are present in the special directories like `/bin, /usr/bin, /sbin, and /usr/sbin;` where as our executable files are present in our directories. If our executable files are copied to the above standard directories, then our programs also can be used as commands. To know the commands available on a Linux, one can list the files present in the above directories. Every file in these directories, is a command only.

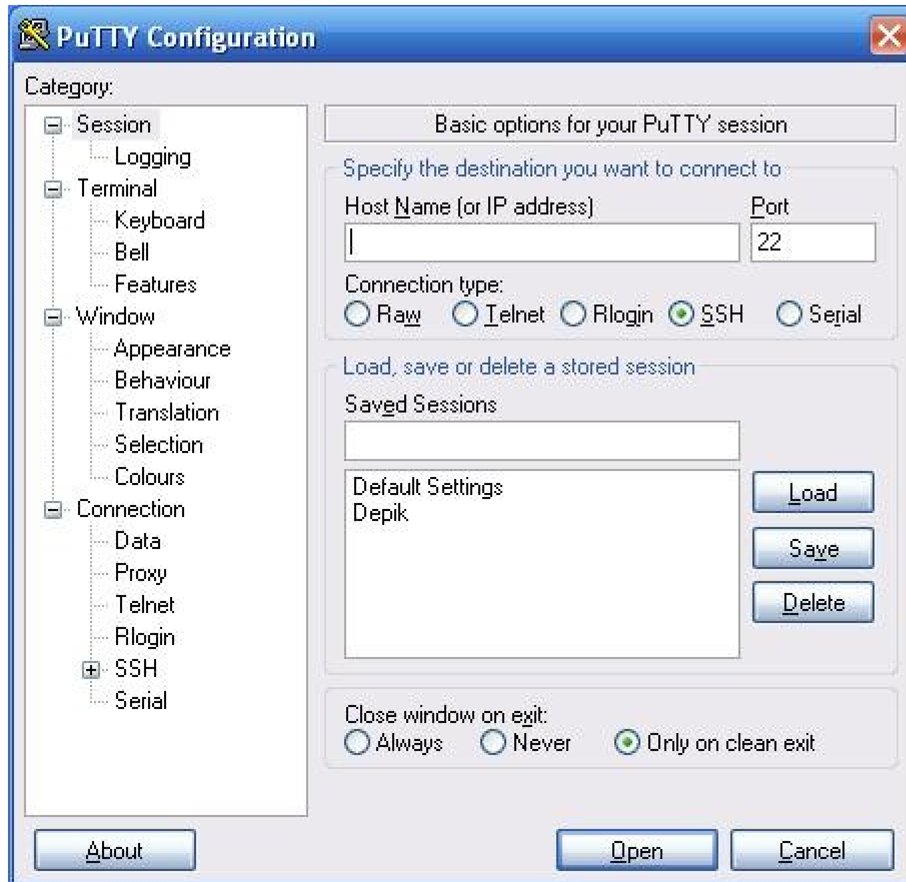
Review Questions

1. Explain the difference between CLI programs and GUI programs in your own words
2. What are the advantages of CLI applications over GUI applications.
3. What are the disadvantages in using CLI applications, write your own observations and feelings in using CLI programs.
4. What is the difference between standard commands and your executable files? How to make your executable file as a command?

2. Practicing C Programs on a Linux

One can access DEPIK's Linux server and can practice C programming on this Linux system. To access DEPIK's Linux system, one should have a Windows PC with Internet connection.

First download the putty.exe application from www.putty.org website. Run the putty program from Windows OS. In the Host Name(or IP address) field, enter the IP address of DEPIK's Linux server. Click the open button present at the bottom of the windows as shown in the following picture.



A command shell terminal will appear on your Windows PC. On the terminal you will see ***login as:*** prompt. Enter your user name and press enter. Next it prompts for the password. Type in the password and press enter (password supplied by DEPIK). Now you will see the shell command prompt. Now one can run various commands, by typing the commands names at this command prompt. By default the putty window's font is very small and background color is black. But you can change the font size and colors through the 'Putty Configuration' window shown above.

```
login as: santhosh
santhosh@69.55.55.8's password:
Welcome to Ubuntu 12.04 LTS (GNU/Linux 3.2.0-24-virtual i686)
santhosh@depiks:~$
```

Linux OS organizes the files on the hard disk, in a tree (inverted tree is more correct) like directory structure. The top most directory is called the root directory. The name of the root

directory is represented with a single letter `/` (forward slash character). When a command terminal is started, it will associate with one directory. This directory is called 'present working directory' or simply 'current directory'. One can change to any other directory by giving change directory (cd) command.

Most of the Linux commands contain two or three letters only. So easy to type, but a bit difficult to remember. It is important to note that, a command is a name of an application (executable) program. When the command is entered, that application will be loaded into memory and gets executed.

When a command shell started it displays the following prompt:

```
karasala@depiks:~$
```

In the prompt, *'karasala'* is the user name, '*depiks*' is the name of the computer. Between : and $ characters is the name of the present working directory. The letter ~ indicates present working directory is user's home directory. But now onwards we will show only the $ as prompt.

The 'pwd' (Present Working Directory) command displays the name of current working directory. So by using this command one can know the name of the current directory. In fact as mentioned earlier, the name of current directory is always displayed within the shell prompt itself.

```
$ pwd
```

The list command **'ls'** and change directory command '**cd'** are the most commonly used commands. Using '**cd**' command one can move to different directories and using 'ls' (list) command one can list (display) the files and directories present in the current directory. The '**ls -l**' command lists the files and directories in the long form. The '**ls /bin**' command lists the files present in the /bin directory. Most of the commands take options and arguments. For '**ls'** command '**-l'** is the option. And '**/bin**' is the argument giving the name of the directory to list the files.

```
$ ls
$ ls -l
$ ls /bin
$ ls -l /bin
```

The '**cd**' command stands for "Change Directory". Using this command we can change to a new directory. For example if you want to change to "/bin" directory you can use the following command.

```
$ cd  /bin
```

Now you can give 'pwd' command to verify that you are in "/bin" directory. You can also give 'ls' command to display the files and directories present in the "/bin" directory. Just giving 'cd' command without specifying any directory name, will change to your home directory.

Change to your home directory. Now you can give the following command to create a new directory with name 'cprogs'.

```
$ mkdir cprogs
```

Now give 'ls' command and verify that new directory is present in your home directory. Now you can change to the newly created directory by giving the following command.

```
$ cd cprogs
```

Give 'pwd' command and verify that you really moved to new directory.

The most important command is 'vi' command, which is used to create a C program files. Using 'vi' is covered in the next section. Assume that you have created "hello.c" file using the 'vi' command. If you want to copy the "hello.c" file to another file with name "hello2.c", then you can use 'cp' (copy) command as shown below:

```
$ cp hello.c  hello2.c
```

Verify with 'ls' command that "hello2.c" is present. Now you can change the name of the "hello.c" file to "x.c" by using 'mv' (move) command.

```
$ mv hello.c  x.c
```

Verify with 'ls' command that, instead of "hello.c" file, "x.c" file is present. You can remove a file by using 'rm' (remove) command as shown below:

```
$ rm x.c
```

The ‘cat’ (concatenate) command is useful to display the contents of a file. For example you want to print the contents of hello.c file you can use the ‘cat’ command as shown below:

```
$ cat  hello.c
```

The ‘cat’ command is also useful to create small files. Enter the following command to create abc.txt file.

```
$ cat > abc.txt
```

Once the above command is given, the cat command is waiting for you to type few lines of text. So type some lines and finally enter Control-d character (Press control key and D simultaneously). Now cat command will create the abc.txt file and saves all the lines you typed into that file. Verify this by giving ‘ls’ command. You may display this new file by using same cat command as shown below:

```
$ cat  abc.txt
```

So far in this section, you have learned the following commands:

```
ls, cd, mkdir,  pwd,  cp, mv, rm, cat
```

But there exist hundreds of commands in the Linux. One can find these commands in the **/bin** directory and **/usr/bin** directory. So list the files present in those directories. Every file in these directories, is an executable file, and hence it is a command. To know the information about these commands, one can use 'man' (manual) command. The 'man' is short form for manual page. The 'man' command displays the manual page for the given command. The 'man' command can also display the manual pages for the library functions. Following are the examples:

```
$ man  ls
$ man  strcpy
```

The first command displays the manual page for list command . Second command displays the manual page for string copy function. The 'man' command is like editor displaying the content of a file. One can use arrow keys to move up and down. To exit from this display of manual page, enter letter 'q'.

4. Using vi editor to create C program file

Enter the following command to create a 'hello.c' file.

```
$ vi  hello.c
```

When you enter the above command, if 'hello.c' file is not already existing you will see empty window space. If file is already present, you will see its contents.

The vi editor works in two modes. One is ‘command mode’ and another is ‘insert mode’. In command mode whatever we type, the vi editor interprets them as commands. So we can enter commands to delete set of lines, copy set of line, paste the lines copies or deleted etc.. In the insert mode, whatever we type it will get inserted into the file.

When we start vi editor it will be in the insert mode. Press letter 'i' to go to INSERT mode. You will see "-- INSERT --" word in the last line. So you are in INSERT mode of vi editor

While in INSERT mode, type a C program, by using all alpha numeric characters and punctuation characters. You may also use all ARROW keys, 'Backspace' key and 'Delete' keys.

Once you finish typing the C program, press 'Esc' key to come to the COMMAND mode from INSERT mode. In the COMMAND mode you will not see "-- INSERT --" string at the bottom line.

While in command mode, we can enter two types of commands. These long commands and short commands. Long commands start with a Colon key and end with Enter key. As we enter this long command it will get displayed in the last line. Where as short commands will have one or more letters. Following are some short commands you can use:

dd	Deletes the current line on which cursor is present
2dd	Deletes 2 lines
5dd	Deletes 5 lines
100dd	Deletes 100 lines from the current line onwards
yy	Copies the current line into internal buffer
8yy	Copies 8 lines from the current lines
p	pastes the last deleted or copied lines after the current line
P	pastes the last deleted or copies lines before the current line

These are only a few short commands there are many more such commands, which you can type while you are in the command mode of vi editor.

Following are some long commands:

`:x`	Save and exit from the editor
`:q!`	Quit without saving the changes
`:w`	Write to the file without quitting from the vi editor
`:w new.c`	Write the content of the current file to a new file 'new.c'
`:r abc.c`	Read another file 'abc.c' and copy that into the current file

All the above long commands should end with ‘Enter’ key. These are only few of the many long commands available.

So after entering the complete program in insert mode, switch to command mode by pressing ‘Esc’ key and type the following two characters, followed by 'Enter' key save and exit from the editor.

```
:x
```

Then the file 'hello.c' will be saved and you will come out of the vi editor. Now you will see your favorite shell prompt.

Give 'ls' command at command prompt, and verify that 'hello.c' file is present in your directory.

5. Using nano editor to create C program file

We strongly recommend students to use 'vi' as editor. However one can also use 'nano' editor to create and edit the text files or C program files.

```
$ nano hello.c
```

This nano editor is similar to Turbo-C editor. One can directly type the program. Finally to save the program, press Control-X and select 'Yes' to save the file. But as a Linux programmer, one should learn vi editor. So we recommend 'vi' editor only.

6. Compile and run the above written 'hello.c' program

Enter the following command to compile and build the executable file '*a.out*' from the '*hello.c*' program

```
$ gcc hello.c
```

If you do not see any errors, then compilation should be successful and executable file will be created with name '***a.out***'. Give 'ls' command at command prompt, and verify that 'a.out' file is present in your directory. You can run this executable file by entering its name at command prompt as shown below:

```
$ ./a.out
```

The compiler by default produces the executable file with name ***a.out***. However one can change the name of output file with **-o** option. The ***-o*** option allows us to specify the name of the output file. The following command produces the executable file with name ***hello***.

```
$ gcc hello.c -o hello
```

One can run the 'hello' program, just like 'a.out' program as given below. The ./ is required before the 'hello' to tell the shell program, to look for hello program in the current directory. If ./ is not given, shell looks for the 'hello' program only in certain directories like /bin, /usr/bin/, /usr/local/bin etc.. As our 'hello' program is not in those directories, shell will display command not found error. When ./ is given, shell looks for 'hello' in the current directory and executes.

```
$ ./hello
```

7. More about files

File is a fundamental object in an operating system. Each file is a set of bytes, stored on a storage device. Typical storage devices are hard disks, CD-ROMs, USB pen drivers, Memory cards etc.. Every file will have a name and size. Size tells the number of bytes present in the file. Operating system allow users to organize files into various directories.

Linux operating system also maintains additional information about each file. This additional information includes, date and time of file creation, owner's ID, group ID, and permissions. Broadly files can be classified into two groups, text files and non-text files. Typically non-text files are called binary files. But note that every file contains a set of bytes. And each byte has 8 bits.

Writing a C program involves creating a text file by using some text editors like 'vi' or' nano'. We generate an executable file from this text by using 'gcc' (C compiler) command. We run these executable files.

2. Developing simple C programs

2.1 Introduction to programming

Programming is all about writing a set of instructions for a computer to execute. These set of instructions are called a program. Next we can ask the computer to execute this program (i.e. instructions) so that computer will perform some useful work. For example, we can write a simple program, which will take marks of all the students of a class as input, and output the following useful information:

- Highest Mark
- Average Mark
- Number of students failed
- Number of students passed with second class
- ..
- ..
- Number of students who got less than 10 marks

Computer can execute only simple small instructions called machine instructions. So writing a program (instructions) with such small instructions is tedious and time consuming.

So we specify by using high-level instructions. Let us call these high level instructions as statements. Next we use a command (program) called compiler, to convert these high level statements in to low level machine instructions. Compiler reads the statements from the C program file, converts them to machine instructions and stores in an executable file. When we run this executable file, these low-level machine instructions along with the data are loaded into the computer's memory (RAM) and CPU will faithfully execute these machine instructions.

The rules and regulations we use to write these high level statements are called a computer language. In other words computer language specifies the rules and regulations to be followed to write the statements. These rules and regulations are also called syntax or grammar of the computer language. There exist so many computer languages, each having its own grammar.

2.2 C Programs with different complexities

A set of statements written to do some useful job is called a program. In C language, a program is organized in a single C file or multiple C files. Each C file contains a set of functions and global variable definitions. Each function in turn consists of a set of statements.

A minimum C program just contains a single function called ***main***. The following sections will show you different programs with different complexities. Just enter these programs using *vi* editor, compile and run them.

2.2.1 Simple Program with *main()* function

The following is the simple program containing a single file with single C function called *main()* function. This is the minimal program possible. A program at the minimum contains only a *main()* function. In the figure below, C file is represented using double line rectangle, main function is represented using a single line rectangle. Other functions are represented using dashed line rectangles. This program does not contain any other functions. A sample program of this type is given after the figure.

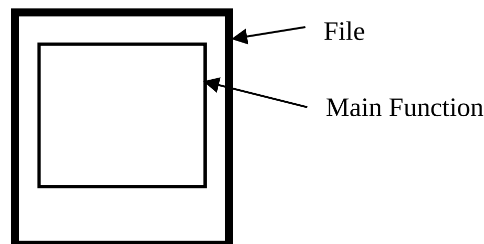


Figure 2.1 C program file with single main function

prg2_1.c

```
#include <stdio.h>

int main()
{
  printf("Hello world\n");
}
```

2.2.2 Simple Program with main() function and global data

The following program also contains only a *main()* function. But it also contains some global data variables. In the figure data variables are represented with black strips.

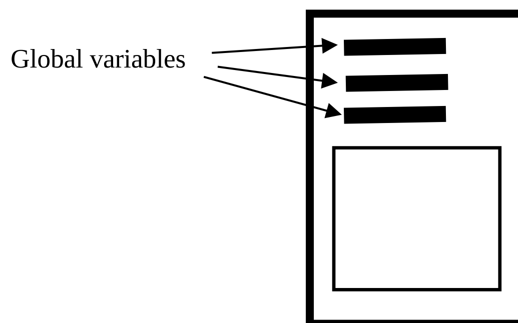


Figure 2.2 C program file with single main function and global data

prg2_2.c

```
#include <stdio.h>

int a = 10;
int b = 20;
int c;

int main()
{
  printf("Value of ‘a’ is %d\n", a);
  printf("Value of ‘b’ is %d\n", b);
  printf("Value of ‘c’ is %d\n", c);
}
```

2.2.3 Program with three functions and no global data

The following program contains two other functions besides a *main()* function. Note that these functions are represented using normal line rectangles.

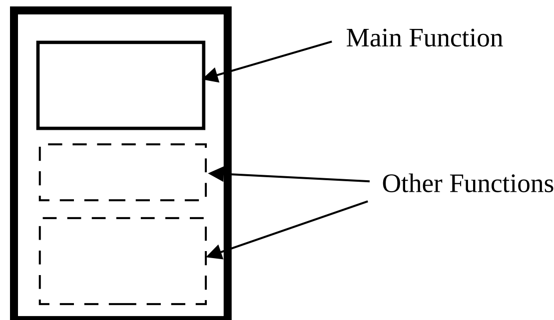


Figure 2.3 C program file with single main function and two additional functions

prg2_3.c

```
#include <stdio.h>

int main()
{
  abc();
  xyz();
}

xyz()
{
  printf(“I am xyz() function\n”);
}
```

```
int abc()
{
  int a;

  printf("Enter some number to store in variable a : ");
  scanf("%d", &a);
  printf("Value of ‘a’ is %d\n", a);
}
```

2.2.4 Program with global data and two functions

The following program contains two functions (one main and one other) and two global variables.

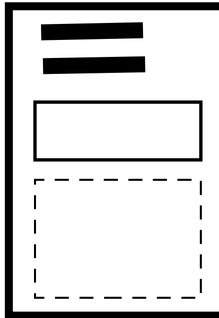


Figure 2.4 C program file with two functions and global data

prg2_4.c

```
#include <stdio.h>

int a;
int b;
int c;

int main()
{
  a = 12;
  b = 34;
  c = 56;

  abc();
}

int abc()
{
  printf("Value of ‘a’ is %d\n", a);
  printf("Value of ‘b’ is %d\n", b);
  printf("Value of ‘c’ is %d\n", c);
}
```

2.2.5 Program with global data and many functions

The following figure represents a program in single file with multiple other functions and a *main()* function.

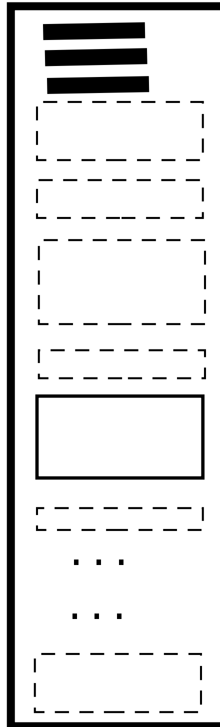


Figure 2.5 C Program file with multiple functions and global data

2.2.6 Large Program with multiple C files

The following figure represents a program with multiple C files. Some C files contain only functions, some files contain only global data, and some with both functions and global data. But note that, there will be only one *main()* function. A program will never have more than one *main()* function.

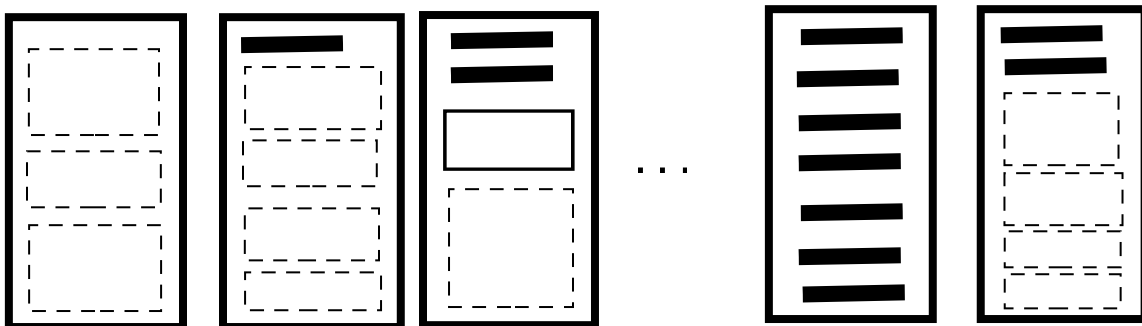


Figure 2.6 C Program with multiple files

2.3 The *main()* Function

As shown in the above figures, all programs contain a single *main()* function. Besides the *main()* function a program may contain additional functions. The importance of *main()* function is that, when a program is executed the execution starts with the *main()* function. So the statements present in the *main()* function are executed first.

2.4 Library Functions

The statements present in *main()* function are calling other functions such as *printf()*, *scanf()* and *abc()*. The *printf()* and *scanf()* functions are called library functions. These are already written and available for us to use them. Where as functions like *abc()*, are our own functions and we have written them. There exist a lot of library functions like *printf()* and *scanf()* which we can call from our programs to do something. For example we use *printf()* function to write something on the display monitor and *scanf()* function to read something from the keyboard.

2.5 Organization of a Function

A function (either a *main()* function or any other function) contains a set of statements. These statements can be classified into two broad types. These are:

- Local data definition statements
- Executable statements

Data definition statements

Data definition statements in a function define local variables. Each variable represents a memory location and can hold some data. Only the executable statements present within this function can access these local variables. Executable statements present in other functions cannot access these local variables.

However as we saw in the organization of C program, a C file can have global data definition statements. These global data definitions statements are present out side the functions (shown in triple lines in figures 1.x above. These global data variables can be accessed or used by executable statements present in any function.

Executable Statements

Executable statements can be present only inside the functions. Executable statements cannot be placed outside a function. Executable statements present in the functions will get executed when that function is called. Following are the different types of executable statements.

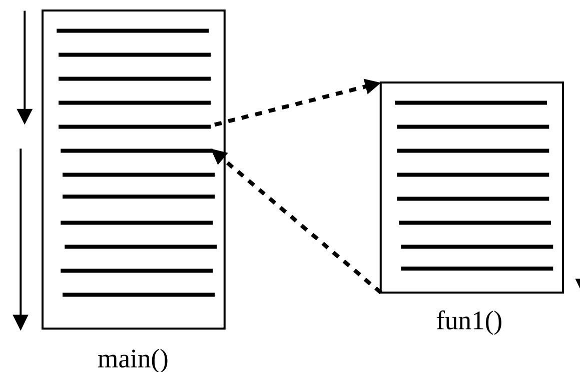
- Function invocation (or Function Calling) statement
- Assignment statement
- if statement
- if-else statement

- if-else if-else statement
- switch case statement
- for statement
- while statement
- do-while statement
- break statement
- continue statement
- return statement
- block statement

We will learn all these types of executable statements and data definition statements as we proceed further.

2.6 Program Execution Sequence

When a program is executed, the *main()* function is called first. The statements present in the main function are executed one by one. When all the statements present in the main function are executed then the main function finishes and program execution also completes. If any statement in the main function is a *function invocation statement*, then execution goes to the called function. After executing all the statements in the called function, execution returns to the next statement after the function invocation statement of main function. This is illustrated in the following figure:



2.7 Things to remember

- A smallest C program contains a single function with name *'main'*.
- When we run a program, the *main()* function will get executed.
- Some executable statements in *main()* function can call other functions. These functions could be either our own functions (written by us) or library functions present in the libraries.
- When all the statements in *main()* function are executed the program execution will get complete.
- Library functions are useful functions written, compiled and stored in library files.
- Even though a function is present in a C file, it will get executed only when it is invoked/called from some other function. Only exception to this rule is *main()* function, which will get called automatically when the program is executed.

```
Activity
--------
01. Read the material given in this chapter thoroughly. Answer the following
    review questions to check your understanding.
02. Go through the sample programs given in this chapter. Try to understand
    each and every line of program. Finally enter this program in your
    computer and run.
04. If you have any doubt send us a mail to  depik.help@gmail.com

Review Questions
----------------
01. Explain the organization of simple to complex C programs.
02. What are library functions and give examples of library functions.
03. What is the difference between main() function and other functions?
04. What the two broad types of statements that could be present inside
    a function?
05. If a C file containing a main() function and 4 other functions is
    compiled and executed. In the main() function, there are no
    function invocation statements to call the other 4 functions.
    If so, will these four functions will get executed or not?

Assignments
-----------
01. Enter all the programs listed in this chapter (prg2_1.c to prg2_4.c)
    using vi editor, compile and run.
```

3. Data definition statements & variables

We write programs to process the data. Data is stored in variables. Using data definition statements we define the variables. Using executable statements, we specify the operations to be performed on the data.

3.1 Variable Types

Data definition statements are used to define the variables. Variables are memory locations for storing the data items. Following are some data definition statements.

```
int        a;
short      b;
char       c;
float      f;
```

In the data definition statement we are specifying the name and type of variables. Type specifies the size of the variable and how to interpret the content of variable. Size specifies how many bits or bytes are used for this variable. If a variable is occupying 32 bits (4 bytes), then these 32 bits can be interpreted as a signed integer or unsigned integer or a real number (floating point number).

Variable types can be classified into two broad groups. These are integer types and floating types. Integers are used to store whole numbers without any fractions. Where as floating type is used to store real numbers, i.e. numbers with decimal point. Again integer types can be classified into signed integers and unsigned integers. Following are the types in each these groups:

Signed Integer types

- `char`
- `short`
- `int`
- `long`

Unsigned integer types

- `unsigned char`
- `unsigned short`
- `unsigned int`
- `unsigned long`

The 'char' type variable occupies 8 bits or 1 byte. The 'short' type variable occupies 16 bits or 2 bytes. Both 'int' and 'long' type variables occupies 32 bits or 4 bytes. Signed integers are represented in two's compliment form.

Real or Floating types

- `float`

- `double`

The 'float' type variable occupies 32 bits or 4 bytes. The 'double' type occupies 64 bits or 8 bytes. Floating variables are represented according to IEEE standard. In this standard, for 32 bit float variables, 23 bits are used to store the fraction, 8 bits to store the exponent and 1 bit to store the sign.

Pointer Types

We can also define pointer type variables. These variables are used to store the addresses. We will learn about these types of variables in later chapters.

3.2 Local variables and Global variables

If variables are defined in side a function, these are called local variables of that function. These variables can be used only in the executable statements of that function. Global variables are defined by keeping the data definition statements out side the function. Typically global data definition statements are kept at the beginning of the file. Global variables can be used by all the functions.

Program 3.1

In this program we will learn the following:

- Defining local variables by using data definition statements
- Assigning values to variables by using assignment statements
- Printing the content of variables by calling *printf()* library function.

prg3_1.c

```
#include <stdio.h>

int main()
{
  int     a;
  int     b;
  float   c;

  a = 20;
  b = 30;
  c = 10.825;

  printf("a = %d\n", a);
  printf("b = %d\n", b);
  printf("c = %f\n", c);
}
```

This program contains only a *main()* function. In this function first 3 statements are called data definition statements. Using these statements we are defining three variables. When we define a variable, we specify its type and name. Depending on the type of the variable, required number of bytes of memory is allocated to the variable. So each variable represents a location in memory of

certain size. The *int* type variables will occupy 4 bytes (32 bits), *short* type variables will occupy 2 bytes (16 bits) and *char* type variables will occupy 1 byte (8bits).

The remaining six statements of main function are executable statements. The first three executable statements are *assignment statements*. In the first assignment statement, we are assigning a value of 20 to the variable *a*. That is, we are storing a value of 20 in the memory location associated with name *a*.

The last three executable statements are *function invocation* statements. Function invocation statements are used to call another function from the current function. From this main function we are calling *printf()* function to display the contents of variables.

3.3 printf() Function

The *printf()* function is the most widely used library function. We use this function to write some text on to the display device. When we invoke a function, we pass arguments to the function. These arguments are like input to the function. Most of the functions take fixed number of arguments. Where as special functions like *printf()* and *scanf()* takes variable number of arguments. We specify these arguments inside the parenthesis after the function name. When multiple arguments are passed these are separated by the commas.

In the above program there are three *printf()* statements. We are passing two arguments to these functions. The first argument should be always a string. This string may or may not contain a format specification that starts with character %. For each format specification character present in the string there should be one additional argument. If there are no format specification characters in the first string argument, then the *printf()* does not take any more arguments.

The *printf()* function writes the character by character present in the string to the output device. However when it gets format specification that starts with % character, this function takes the next argument and converts that argument into the format specified by the format specification and writes it on to the output device. The characters after the format specification are displayed as it is. If one more format specification comes, the next argument is written in that format. This process continues until complete string is written.

Program 3.2

In this program we will learn the following:

- Defining local variables by using data definition statements
- Reading the numbers into these variables from the user, by calling *scanf()* function
- Printing the content of variables by using *printf()* library function.

prg3_2.c

```
#include <stdio.h>

int main()
{
  int    a;
  int    b;
  float  c;
```

```
  printf("Enter two integers and one float value\n");
  scanf("%d", &a);
  scanf("%d", &b);
  scanf("%f", &c);

  printf("Following are the values you entered\n");
  printf("a = %d\n",a);
  printf("b = %d\n",b);
  printf("c = %f\n",c);
}
```

In this program we are using *scanf()* library function to read from the keyboard input and write into the variables. The *scanf()* function we are passing two parameters. First parameter is the format string specifying the input format and second parameter is the address of the variable into which this value is loaded (or assigned).

We used three *scanf()* function invocation statements to read the three variables. Next we are displaying the content of these variables by using *printf()* library function.

Note the use of &a &b and &c in *scanf()* function. In *printf()* we are using only a b and c.

3.4 scanf() Function

The *scanf()* function is also a special function like *printf()* function that takes variable number of arguments. The number of format specifications present in the first argument string decides the number of additional arguments. The *scanf()* function reads character by character from the keyboard and converts them according to format specification, and places it in the location specified through additional argument. The additional arguments should pass the address of the variable where the converted value should be placed. The address operator *&* is used to pass the address of the variable. So when we pass *&a* to the *scanf()* function, we are passing the memory address of variable a. The *scanf()* function keeps the value read from keyboard at that address.

Program 3.3

In this program we will learn the following:

- Defining global variables by using data definition statements
- We are using these global variables in two functions. Function *fun1()* is initializing these variables by reading data from the user. Function *fun2()* is printing the content of these variables. Note that only global variable can be accessed from multiple functions, where as local variables can be used only in the function in which they are defined.
- The *main()* function is calling both these functions.

prg3_3.c

```
#include <stdio.h>

int             a;
unsigned int    b;
float           c;
```

```
void fun1()
{
  printf(“Enter two integers and one float value\n”);
  scanf("%d", &a);
  scanf("%u", &b);
  scanf("%f", &c);
}

int fun2()
{
  printf(“Following are the values you entered\n”);
  printf("a = %d\n",a);
  printf("b = %u\n",b);
  printf("c = %f\n",c);
  printf(“c = %.2f\n”,c);
}

int main()
{
  fun1();
  fun2();
}
```

3.5 Reading small size integer variables

We use `“%d”` as format specification in *scanf()* function to read integer variables. But when we want to read small integer variables like `char`, and `short` variables we must use different format specification. This is illustrated in the following program. Note that `char` variable is capable of storing from –128 to 127. Where as `unsigned char` variable can store from 0 to 255. The `char` variable occupies 8 bits or 1 byte of space. The unsigned short variables take values from 0 to 65535 where as short (signed) variables can hold from –32768 to 32767.

prg3_4.c

```
#include <stdio.h>

int main()
{
  char           c;
  short          s;
  unsigned char  uc;
  unsigned short us;

  printf("Enter small integer value from -128 to 127 : ");
  scanf("%hhd",&c);
  printf("You entered %d\n",c);

  printf("Enter small unsigned integer value from 0 to 255 : ");
  scanf("%hhu",&uc);
  printf("You entered %u\n",uc);

  printf("Enter short integer value from -32768 to 32767 : ");
```

```
  scanf("%hd",&s);
  printf("You entered %d\n",s);

  printf("Enter short unsigned integer value from 0 to 65535 : ");
  scanf("%hu",&us);
  printf("You entered %u\n",us);
}
```

3.6 Reading and Printing text characters

So far we studied about storing integer and float (numbers with decimal point) numbers in variables. But another use of variables is for storing the text characters. Text characters consists of small letters, capital letters, numerical digits, punctuation marks and special characters like # $ etc.. Each of these text characters is associated with a standard code number called ASCII code. The standard ASCII codes range from 0 to 127. For example the ASCII codes of small letters 'a' to 'z' will range from 97 to 122, and ASCII codes of capital letters 'A' to 'Z' will range from 65 to 90. Similarly the ASCII codes of digits '0' to '9' will range from 48 to 57.

The ASCII codes 0 to 31 represent a control characters. These control characters represent non-printable characters. The ASCII code 0 represents a special character called NULL character. This NULL has got a special significance, which we discuss later.

We normally use variables of type 'char' to store the text characters. This is because, the ASCII codes range from 0 to 127, so these codes can be easily fit in char variables.

prg3_5.c

```
#include <stdio.h>

int main()
{
  char   ch1, ch2, ch3;

  ch1 = 65;
  ch2 = ‘b’;
  ch3 = ‘3’;

  printf(“ ch1 = %c\n”, ch1);
  printf(“ ch2 = %c\n”, ch2);
  printf(“ ch3 = %c\n”, ch3);

  printf(“ ch1 = %d\n”, ch1);
  printf(“ ch2 = %d\n”, ch2);
  printf(“ ch3 = %d\n”, ch3);
}
```

The above program illustrates various things. When we want to keep some text character in a variable, we can assign the ASCII code to that variable. In the above program we are assigning ASCII code 65 (code of letter 'A') directly. But easy way is to assign the character with single quotes. A character between the single quotes will represent its ASCII code.

When we want print the content of char variables as text character, then we must use %c as format. If we want to print the content of char variable as a number, we can use usual %d format. This is illustrated in the above program by printing all the variables both with %c first and with %d later.

The following program shows how to read a ASCII code of a character by using a *scanf()* function. In the previous section we studied how to read a small integer into a character variable by using *scanf()* function. The following program is printing the character it has read, in both character form and integer form. This program will be useful to find the ASCII code of any character.

prg3_6.c

```
#include <stdio.h>

int main()
{
  char ch;

  printf("Enter some character: ");
  scanf("%c",&ch);

  printf("Entered character : %c \n", ch);
  printf("Its ASCII code is : %d \n", ch);
}
```

3.7 Printing and reading integers in other notations

We normally define integer variables by using ‘int’ type. However when we define variables by using ‘char’ or ‘short’ type, they are also integer variables. Only difference is that these are small integers. The ‘char’ type defines integer of 8 bits. The ‘short’ type defines 16 bit integer.

Normally we can print all sizes of integer variables in decimal form by using %d or %u as format. The %u prints integer as unsigned value. However we can print integers in Octal or Hex format by using %o or %x as format. This is illustrated in the following program. It is assumed that the student will know the differences between octal, decimal and hex (hexadecimal) notations.

prg3_7.c

```
#include <stdio.h>

int main()
{
  char   c;
  short  s;
  int    i;

  c = 20;
  s = 40;
  i = 120;

  printf(“ c = %d, s = %d, i = %d\n”, c, s, i);
  printf(“ c = %o, s = %o, i = %o\n”, c, s, i);
```

```
  printf(" c = %x, s = %x, i = %x\n", c, s, i);
}
```

In the above program we are printing three integer variables of different sizes in decimal, octal and hex formats.

The following program is reading one integer in octal notation and second in hex notation. Finally it is printing both the integers in decimal as well as in octal or hex.

prg3_8.c

```
#include <stdio.h>

int main()
{
  unsigned int a, b;

  printf("Enter some octal number\n");
  scanf("%o", &a);
  printf("Enter some hexadecimal number\n");
  scanf("%x", &b);
  printf("Octal number is %o, its equivalent decimal is %d\n",a,a);
  printf("Hex number is %x, its equivalent decimal is %d\n",b,b);
}
```

3.8 Printing and reading float variables

Similar to integer variables, which are available in three sizes (8 bit, 16 bit and 32 bit) floating point variables are also available in two sizes. These are 32 bit and 64 bit. Remember that, all sizes of integer variables can be printed by using any of %d, %u, %x, and %o formats. But while reading integer variables of 16 bit and 8 bits, you have to prefix 'h' and 'hh' respectively, before d, u, x, or o.

Similarly, both the sizes of floating point variables can be printed by using %f, %e or %g formats. However while reading double type floating variables, we need to prefix 'l' before f, e or g.

Another important aspect of printing floating number is controlling the number of digits after the decimal point. If we give `"%.3f"` as format, it prints only 3 digits after decimal point. In this way we can control the precision of printing floating point numbers.

prg3_9.c

```
#include <stdio.h>

int main()
{
  float f1, f2;
  double d1, d2;
  printf("Enter two float numbers\n");
  scanf("%f%f", &f1, &f2);
  printf("Sum of %.3f and %.4f is %f\n", f1, f2, f1+f2);
```

```
  printf("Difference of %f and %f is %.3f\n", f1, f2, f1-f2);
  printf("Enter two double float numbers\n");
  scanf("%lf%lf",&d1,&d2);
  printf("You entered %.8f and %.8f\n");
}
```

3.9 sizeof Operator

The *sizeof* operator is useful to find the memory size of any variable. The size is always given in number of bytes. The *sizeof* operator is also used to get the size of any *type* by specifying *type* name.

prg3_10.c

```
#include <stdio.h>

int main()
{
  char c; short s; int i; float f;
  int sz;

  sz = sizeof c;
  printf("size of c is %d\n", sz);
  printf("size of s is %d\n", sizeof s);
  printf("size of int type is %d\n", sizeof(int));
  printf("size of f %d size of double %d\n", sizeof f,sizeof (double));
}
```

```
Review Questions
----------------
01. Distinguish following three groups of variables
       - Integers
       - Floating point numbers
       - Pointers
    Again distinguish between signed and unsigned integers. Also integers
    of various sizes and floats of various sizes.
02. What is the maximum and minimum numbers that can be stored in 'char'
    variables and 'unsigned char' variable.
03. What is the maximum and minimum numbers that can be stored in 'short'
    variables and 'unsigned short' variable.
04. Do you know how -128 and 127 are arrived as minimum and maximum values
    that can be stored in a 'char' type variable.
05. What is the format specification to read integer into a char variable
    using scanf() statement?
06. What is the format specification to read character (ASCII code) into
    a char variable using scanf() statement?
07. In the above questions we are reading into a char variable, then what
    is the difference in these two formats?
08. What is the format specification to read short integer using scanf()
    statement?
09. you have a char variable. You can print it both by using %c and %d.
    What is the difference between them.
10. What are the ASCII codes of 'A' and 'Z' ?
11. What are the ASCII codes of 'a' and 'z' ?
12. What are the ASCII codes of '0' and '9' ?
13. What is the ASCII code of NULL character?
14. What is the difference between global variables and local variables?
15. How to print floating numbers with various precisions?
```

16. What is 'sizeof' operator?

Assignments

```
-----------
i)   Every program should have only one function. That is main().
ii)  Whenever asked to read some number, it is assumed that you have
     to define a variable and use scanf() function to read the number.
```

1. Write a program which defines a single integer variable and reads a value into this variable from the user by using scanf() function. Next it prints the same variable three times using %o %d and %x formats. When you run this program and enter a value as 100 then what are the three values printed by this program? First find the answer and verify the answer by running the program.

2. Write a program which reads a float variable from the user and prints it four times. First time with one digit after decimal point, next with two digits and so on.

3. Write a program which reads a single text character from the user and prints next three subsequent characters. For example if you enter 'S' it prints 'T' 'U' and 'V'. Verify what do get if you enter 'Z'. Hint: When you read a character you will get its ASCII code into the variable. Next increment and print it as character. Repeat this two more times to print two more characters after this.

4. This program is also similar to the previous program, but prints three characters before the given character.

5. Write program which reads an integer from the user and prints it by using following printf() function statements. And observe the differences in display formats.

```
printf(" :%d:   \n", val);
printf(" :%10d: \n", val);
printf(" :%5d:  \n", val);
printf(" :%08d: \n", val);
```

6. Write program which reads an float value from the user and prints it by using following printf() function statements. And observe the differences in display formats.

```
printf(" :%f:     \n", val);
printf(" :%10.4f: \n", val);
printf(" :%.0f:   \n", val);
printf(" :%08.2f: \n", val);
```

7. Write a program which defines 10 integer variables and reads 10 numbers from the user into them. You should use 10 separate scanf() function statements. You are not supposed to use any loop statements, as we have not yet learned them. Next print these 10 numbers in two columns such that each number occupies exactly 10 digits (with leading zeros) and there will be 10 space characters between two integers. So total there will be five rows.

8. Read 5 floating numbers from the user and print them such that one number will get printed on each line. But ensure that all decimal points will get aligned on the same column.

4. Executable Statements and Expressions

Executable statements can be present only inside the functions. Executable statements cannot be placed outside the function. Executable statements present in the functions will get executed when that function is called. Following are the different types of executable statements.

- Assignment statement
- if statement
- if-else statement
- if-else if-else statement
- switch case statement
- for statement
- while statement
- do-while statement
- break statement
- continue statement
- Function invocation (Calling) statement
- Return statement
- Block statement

4.1 Understanding Expressions and Operators

Executable statements are built by using the *expressions*. Before writing executable statements one should have good understanding of expressions. Expressions are built by using variables (also called operands), constants and operators. Following are some sample expressions:

```
5
a
b + 3
a < 5
a & b
(2 * a) + (4 * b) + 7
```

Each line above, is an example of expression. The first expression contains only a constant value 5. So when this expression evaluated, the result is going to be same always and it is 5. The second expression contains single variable ‘a’. The value of this expression depends only on the value of the variable ‘a’. The third line contains an expression with one variable and one constant. Both are connected using arithmetic operator +.

Expressions can be built by using different types of operators. Even though we use mostly arithmetic operators, but the following types of operators are also used in the expressions. Many times we may use expressions with multiple types of operators.

- Arithmetic operators (+ - * / % ++ --)
- Conditional operators (== < > >= <= !=)
- Logical operators (&& || !)
- Bit operators (& | ^ ~ << >>)

Following are some examples of expressions with above operators.

```
a < 5
a == b
(a < 5) && (a > 1)
a != b
a | b
!a
```

4.2 Arithmetic expressions

We can print the value of expression by using *printf()* function as shown in the following program.

prg4_1.c

```
#include <stdio.h>

int main()
{
  int   a;
  int   b;

  printf("Enter the value of a and b\n");
  scanf("%d%d",&a,&b);

  printf("a+b is %d\n", a+b);
  printf("a-b is %d\n", a-b);
  printf("a*b is %d\n", a*b);
  printf("a/b is %d\n", a/b);
  printf("a%b is %d\n", a%b);
}
```

4.3 Conditional expressions

When conditional expression is evaluated it results in Boolean value 1 indicating TRUE or 0 indicating FALSE. We can verify this with the following program. This program prints the result of various conditional expressions.

prg4_2.c

```
#include <stdio.h>

int main()
{
  int   a;
  int   b;

  printf("Enter the value of a and b\n");
```

```
  scanf("%d%d",&a,&b);


  printf("Result of a < b is %d\n", a < b);
  printf("Result of a > b is %d\n", a > b);
  printf("Result of a <= b is %d\n", a <= b);
  printf("Result of a >= b is %d\n", a >= b);
  printf("Result of a == b is %d\n",a == b);
  printf("Result of a != b is %d\n",a != b);
}
```

4.4 Logical expressions

When we evaluate logical expression, it also results in boolian values TRUE (1) or FALSE(0) similar to conditional expressions. We build logical expressions by using either boolian variables or expressions that results boolean values.

In C language, there is no variable of type boolian. Any non float (or double) variable like chat, short, int, or any pointer variable can be used as a boolian variable. A boolian variable having value 0 (zero) is considered as FALSE, any value other than 0 is considered TRUE.

prg4_3.c

```
#include <stdio.h>

int main()
{
  int   a, b, c;

  printf("Enter the value of a, b and c\n");
  scanf("%d%d%d",&a,&b,&c);

  printf("Result of (a < b) && (a < c) is %d\n", (a < b) && (a < c));
  printf("Result of (a > b) && (a > c) is %d\n", (a > b) && (a > c));
  printf("Result of (a == b)&&(a == c) is %d\n", (a == b) && (a == c));
  printf("Result of a && b is %d\n", a && b);
  printf("Result of a || b is %d\n", a || b);
  printf("Result of !a is %d\n", !a);
  printf("Result of !b is %d\n", !b);
}
```

4.5 Expressions with Bit operators

The bit operators will perform logical operations on the individual bits of variables. All the resulted bits will be the result of the expression. For example, when we perform bitwise AND operation on variable ‘a’ and ‘b’. All the 32 bits of variable ‘a ‘ are ANDed with the corresponding 32 bits of variable ‘b’. This results in 32 bits. This resulted 32 bits are result of this expression. Following is the sample program illustrating the use of bit operators.

prg4_4.c

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

```
int main()
{
  int   a;
  int   b;

  printf("Enter the value of a and b\n");
  scanf("%d%d",&a,&b);


  printf("Result of a & b is %d\n", a & b);
  printf("Result of a | b is %d\n", a | b);
  printf("Result of a ^ b is %d\n", a ^ b);
  printf("Result of ~a is %d\n",     ~a);
  printf("Result of ~b is %d\n",     ~b);
  printf("Result of a << 2 is %d\n", a << 2);
  printf("Result of b >> 2 is %d\n", b >> 2);
}
```

4.6 Things to Remember

- Expressions are made with variables (also called operands), constants and operators.
- There are four types of operators, resulting in four types of expressions. These are Arithmetic, Logical, Conditional and Bit wise.

```
Review Questions
----------------
01. Distinguish between executable statements and data definition statements.
02. Is it possible for executable statements to be present out side of
    functions?
03. Is it possible for data definition statements to be present out side of
    functions?
04. What are the various types of executable statements used in C programming?
05. The Return statement must be present in every function. Do you agree
    with this statement?
06. What do you understood by block statement?
07. What is an expression?
08. What are the constants, operators and operands in an expression?
09. What are the various groups of operators?
10. List operators in each of the above groups.
11. What is the difference between binary operator and unary operator?

Assignments
-----------
i)   Every program should have only one function. That is main().
ii)  Whenever asked to read some number, it is assumed that you have
     to define a variable and use scanf() function to read the number.
iii) Printing should be done by using printf() function and by using
     expression inside the printf() as shown below:
         printf("%d \n", number << lsbits);

1.  Ask user to enter an integer number and number of bits to shift left.
    Print the number after shifting it left by given number of bits.
    Note that you should have an idea about the value that will get printed.

2.  Ask user to enter an integer number and number of bits to shift right.
```

Print the number after shifting it right by given number of bits.

3. **Ask user to enter two integers and print the logical AND, and bitwise AND of these two integers.**

4. **Ask user to enter two integers and print the logical OR, and bitwise OR of these two integers.**

5. **Ask user to enter two integers and print the exclusive OR of these two integers.**

6. **Ask user to enter a single integer and prints its bitwise complement and logical complement values.**

7. **Ask user to enter two characters and read them into two char variables. Print the result of expression char1 < char2. Where char1 and char2 are the char variables into which user entered characters are read.**

8. **Ask user to enter two characters and read them into two char variables. Print the result of expression char1 > char2. Where char1 and char2 are the char variables into which user entered characters are read.**

9. **Ask user to enter two characters and read them into two char variables. Print the result of expression char1 == char2. Where char1 and char2 are the char variables into which user entered characters are read.**

5. Assignment statements

A C program consists of functions. Each function consists of statements. We can classify these statements into various types. One most widely used type of statement is assignment statement. Assignment statement is used to assign the result of an expression into some variable. Following is the syntax of Assignment statement.

```
variable =  <expression>;
```

The operator `'='` is called assignment operator. This is different from `'=='` (equality condition) operator. The left hand side (LHS) of assignment operator should be a placeholder to keep the result of right hand side (RHS) expression. Common placeholders are variables.

5.1. Assignment statements using Arithmetic expressions

Following are the examples of assignment statements using arithmetic expressions.

```
i =  (p * t * r) / 100;

peri = 2 * ( breadth + length);
```

Following is simple sample program showing the usage of arithmetic expressions and assignment.

prg5_1.c

```
#include <stdio.h>

int main()
{
  int num1, num2;
  int result;

  printf("Enter number 1 :");
  scanf("%d",&num1);

  printf("Enter number 2 :");
  scanf("%d",&num2);

  result = num1 + num2;
  printf("%d + %d = %d\n",num1,num2,result);

  result = num1 - num2;
  printf("%d - %d = %d\n",num1,num2,result);

  result = num1 * num2;
  printf("%d * %d = %d\n",num1,num2,result);

  result = num1 / num2;
  printf("%d / %d = %d\n",num1,num2,result);

  result = num1 % num2;
  printf("%d / %d = %d\n",num1,num2,result);
```

```
}
```

5.2 Assignment statements using conditional expressions

Following are the examples of assignment statements using conditional expressions.

```
a =  b < c;

eq =  x != y;
```

Note that, it is possible to store (or assign) the result of conditional expression into a variable. The result will be one (1) if condition is True, zero (0) if condition is False.

Following is simple sample program showing the usage of logical expressions and assignment.

prg5_2.c

```
#include <stdio.h>

int main()
{
  int num1, num2;
  int result;

  printf("Enter number 1 :");
  scanf("%d",&num1);

  printf("Enter number 2 :");
  scanf("%d",&num2);

  result = num1 > num2;
  printf("%d > %d = %d\n",num1,num2,result);

  result = num1 >= num2;
  printf("%d >= %d = %d\n",num1,num2,result);

  result = num1 < num2;
  printf("%d < %d = %d\n",num1,num2,result);

  result = num1 <= num2;
  printf("%d <= %d = %d\n",num1,num2,result);

  result = num1 == num2;
  printf("%d == %d : %d\n",num1,num2,result);

  result = num1 != num2;
  printf("%d != %d = %d\n",num1,num2,result);
}
```

5.3. Assignment statements using logical expressions

Following are the examples of assignment statements using logical expressions:

```
a =  b && c;

z =  x || y;
```

Note that it is possible to store (or assign) the result of logical expression into a variable. The result will be one (1) if result of expression is true, zero (o) if result is false.

Following is simple sample program showing the usage of logical expressions and assignment.

prg5_3.c

```
#include <stdio.h>

int main()
{
  int num1, num2;
  int result;

  printf("Enter number 1 :");
  scanf("%d",&num1);

  printf("Enter number 2 :");
  scanf("%d",&num2);

  result = num1 && num2;
  printf("%d && %d : %d\n",num1,num2,result);

  result = num1 || num2;
  printf("%d || %d : %d\n",num1,num2,result);

  result = !num1;
  printf("!%d : %d\n",num1,result);

}
```

5.4. Assignment statements using bit wise expressions

Following are the examples of assignment statements using logical expressions:

```
a =  b & c;

z =  x | y;

b =  a << 5;

c =  d >> 2;
```

Following is simple sample program showing the usage of bit wise expressions and assignment.

prg5_4.c

```
#include <stdio.h>

int main()
```

```
{
  int num1, num2;
  int result;
  int lshift, rshift;

  printf("Enter number 1 :");
  scanf("%d",&num1);

  printf("Enter number 2 :");
  scanf("%d",&num2);

  printf("Enter left shift value :");
  scanf("%d",&lshift);

  printf("Enter right shift value :");
  scanf("%d",&rshift);

  result = num1 & num2;
  printf("%d & %d : %d\n",num1,num2,result);

  result = num1 | num2;
  printf("%d | %d : %d\n",num1,num2,result);

  result = num1 ^ num2;
  printf("%d ^ %d : %d\n",num1,num2,result);

  result = ~num1;
  printf("~%d : %d\n",num1,result);

  result = num1 << lshift;
  printf("%d << %d : %d\n",num1, lshift, result);

  result = num2 >> rshift;
  printf("%d >> %d : %d\n",num2, rshift, result);

}
```

Review Questions
----
01. What are the assignment statements?
02. What is assignment operator
03. What is equality operator?
04. When you assign the result of a conditional expression to an variable, what could be the possible content of variable?

Assignments
----
i) Write all the following programs by using a single main() function. In every function use the assignment statement to store the result in a variable. Finally print this result variable by using printf() function.

1. Write a main() function which asks user to enter the size of a side of a square in centimeters. This function should read that value and should print the area of square in square centimeters. Use only integers.

2. Rewrite the above program by using a floating variables. But this

program should read the size in centimeters and should print the area in square meters.

3. **Write a program having single main function. This function asks user to enter length and breadth of a rectangle in centimeters. Use integer variable to read these length and breadth. Next the function should print area and perimeter of the rectangle.**

4. **Write a program having single main function. This function asks user to enter length, breadth and height of a block in centimeters. Use integer variable to read these length, breadth and height. Next the function should print the surface area and volume of the block.**

5. **Ask user to enter the radius of a circle in centimeters. Use floating point variable. Now compute the circumference in centimeters and and area in square meters. Also print the area of square whose perimeter is same as circumference of the above circle. This area also should be printed only in square meters. You can include math.h file and can use constant M_PI, which is set more precisely than 3.14.**

6. **Ask user to enter principle, annual rate of interest and time in years. Use all float variables only. Print the interest and also total (i.e. interest + principle).**

7. **Read temperature in Fahrenheit degrees from the user and print it in Celsius degrees. Use floating point variables.**

   Formula: Celsius = (5.0/9.0) * (fahrenheit - 32)

8. **Read temperature in Celsius degrees from the user and print it in Fahrenheit degrees. Use floating point variables.**

9. **This program gives you ideal weight based on your height. Ask user to enter his height in feet and inches. Convert height into centimeters. Subtract 100 from the centimeters. Finally print ideal weight i.e. the remaining centimeters as weight in kilograms. For example if someone's height is 168 centimeters, his ideal weight is 68 Kilograms. Assume one Inch is 2.5 centimeters. One foot is equal to 12 inches.**

.

6. Conditional execution statements

If a function contains only simple assignment statements, then that function will get executed sequentially statement after statement. But most of the time, we want some statements to be executed when some condition is true. We may want some other statements to get execute if condition is false. For this kind of requirements following statements are used:

- If statement
- If – Else Statement
- If – Else if – Else if – Else statement
- Switch statement

6.1 If Statement

Following is the syntax of if statement.

```
if (<expression>)
  <statement>;
```

An *if* statement takes an expression and evaluates the expression to True or False. If expression results in a zero value it is considered as False. Any other value is considered as True. If expression results in True (any non-zero value), the statement after the 'if' condition will get executed. If expression results in zero value, the statement after 'if' will be skipped.

The statement after 'if' could be a single statement or it could be a block statement containing multiple statements. A **block statement** is a set of statements enclosed in the braces { }. All these statement enclosed in the braces acts like single statement for the 'if' condition. So based on the condition all statements (within braces) will get executed or all will get skipped.

All of the following are valid *if* statements.

```
if(a)
  printf("a is non zero value\n");

if(a+b)
  printf("The sum of a and b is non zero\n");

if(--b)
  printf("After decrementing b, it is non zero\n");

if(b++)
  printf("b is non zero, b is incremented\n");

if(a < b)
{
  printf("a is less than b\n");
  printf(“This is second statement in a block statement\n”);
  printf(“if condition is true, all three statements will exec\n”
}
```

```
  if(a > b)
    printf("a is greater than b\n");

  if(a & b)
    printf("a AND(bit wise) b is non zero\n");

  if(a << b)
    printf("a<<b is true\n");


  if(a >> b)
    printf("a>>b is true\n");

  if(a && b)
    printf("a && b is true\n");

  if((a < b) || (a < c))
    printf("(a < b) || (a < c) is true\n");

  if(5)
    printf("5 is true\n");
```

It is very important for the students to understand the type of expressions used in side the parenthesis of *if* statement. Students should note that the expression could be any thing including a variable or constant or any type of expression as shown in the above sample *if* statements.

Observe that in the third *if* statement, we used a block statement. So when condition is True, all the three *printf()* statements present inside the block statement will get execute. If condition is wrong none of the statements will get executed.

Following is the sample program showing the usage of *if* statement.

prg6_1.c

```
#include <stdio.h>

int main()
{
  int bill;
  int discount;

  printf("Please enter your bill amount in rupees:");
  scanf("%d",&bill);

  if(bill > 10000)
    printf("Congrats you are eligible for 20 %% discount\n");

  printf("Thanks for shopping with us, Visit again\n");
}
```

6.2 If – Else Statement

Following is the syntax if statement.

```
if (<expression>)
  <statement_1>;
else
  <statement_2>;
```

When expression results in True, the statement_1 will get executed, if expression is False, the statement_2 will get executed. Note that the statements after ‘if’ and ‘else’ could be a simple statements or block statements containing multiple statements in braces.

prg6_2.c

```
#include <stdio.h>

int main()
{
  int bill;
  int discount;

  printf("Please enter your bill amount in rupees:");
  scanf("%d",&bill);

  if(bill <= 0)
  {
    printf(“Invalid bill amount\n”);
    return 0;
  }
  if(bill < 1000)
    printf(“Sorry you will not get any discount\n”);
  else
  {
    discount = (bill * 10)/100;
    printf(“You got %d discount\n”,discount);
    printf(“You just need to pay %d rupees\n”, bill – discount);
  }
}
```

In the above program the statement after first ‘if’ condition is a block statement. In this block statement there are two statements. First statement is calling *printf()* function. The second statement is a *return statement*. When a return statement is executed the execution flow will return from this function and no further statements in this function will get executed. The return statement is useful for two purposes. One is to return from the middle of a current function to the function that called this function. Second is to return a value to the calling function. Once all the statements in the function are executed, control will automatically return to the calling function. There is no need to keep return statement at the end of every function. However a function returning a value must always use the return statement to return some value to the calling function.

6.3 If – Else If – Else If – Else Statement

Following is the syntax of this statement.

```
if (<expression_1>)
  <statement_1>;

else if(<expression_2>)
  <statement_2>;

else if(<expression_3>)
  <statement_3>;

else
  <statement_4>;
```

This statement works like this. First it evaluates the expression_1. If it is True, it executes the statement_1 and skips evaluation of all other expressions and execution of their corresponding statements. If expression_1 results in False, then it skips execution of statement_1 and evaluates expression_2. If it is True, it executes statement_2 and skips evaluation of all other expressions and execution of corresponding statements. If no expression results in True, then finally the statement after last 'else' will get executed.

In other words, this statement executes only one statement corresponding to the first expression that results in True. If no expression results in True, it executes the statement after the last 'else'.

Some times the last 'else' may not be present. In such cases, only the first statement whose expression results in True is get executed. If no expression results in True, then no statement will get executed. Following is the syntax of such usage.

```
if (<expression_1>)
  <statement_1>;

else if(<expression_2>)
  <statement_2>;

else if(<expression_3>)
  <statement_3>;
```

Note that using multiple if statements as shown below is clearly different from 'if-else if-else' statement. In the following multiple if statements, all the 'if' expressions are always evaluated and the corresponding statement is get executed if expression results in True. So if all the expression are resulted in True then all the statements will get executed. Where as in 'if-else if – else' any one of the statements will get executed.

```
if (<expression_1>)
  <statement_1>;

if(<expression_2>)
  <statement_2>;
```

```
if(<expression_3>)
  <statement_3>;
```

Note that in all the above examples, the <statement_x> could be a single statement or block statement.

prg6_3.c

```
#include <stdio.h>

int main()
{
  int bill;
  int discount;

  printf("Please enter your bill amount in rupees:");
  scanf("%d",&bill);

  if(bill < 1000)
    discount = 0;

  else if((bill >= 1000) && (bill < 2000))
    discount = 5;

  else if((bill >= 1000) && (bill < 2000))
    discount = 10;

  else if((bill >= 2000) && (bill < 3000))
    discount = 15;

  else
    discount = 20;

  bill = bill – ((bill * discount)/100);
  printf(“Your final bill is %d\n”, bill);
}
```

Following is another sample program showing the usage of “if else-if else” construct.

prg6_4.c

```
#include <stdio.h>

int main()
{
  int day;

  printf("Enter day i.e., 1 for Sunday, 2 for Mon,.... 7 for Sat:");
  scanf("%d",&day);

  if(day == 1)
    printf(“Today is holiday, enjoy as you wish\n”);

  else if(day == 2)
    printf(“Study Mathes\n”);
```

```
  else if(day == 3)
    printf("Study English\n");

  else if(day == 4)
    printf("Study Science\n");

  else if(day == 5)
    printf("Study Social\n");

  else if(day == 6)
    printf("Study Telugu\n");

  else if(day == 7)
    printf("Study Hindi\n");
  else
    printf("Invalid day\n");
}
```

6.4 Switch statement

Following is the syntax of switch statement.

```
switch(<expression>)
{
    case <val>:
      <statement>;
      <statement>;
      break;

    case <val>:
      <statement>;
      <statement>;
      break;

    case <val>:
      <statement>;
      <statement>;
      break;

    default:
      <statement>;
      <statement>;
      break;
}
```

The switch statement works as follows. The expression in parenthesis after the ‘switch’ keyword is evaluated. This expression should result an integer. After each ‘case’ keyword, an integer value is present. Based on the integer value of expression the control flow will jump to the case whose integer is matched with this expression result. Next all the statements after this case will get executed. The ‘break’ statement will cause the control flow to jump outside of the total switch

statement. If no ‘break’ statement is present, execution will continue to execute the statements belonging to other cases also.

The ‘default’ keyword represents a special case. When an expression integer does not match with any of the case integers, then control will switch to this default case.

prg6_5.c

```
#include <stdio.h>

int main()
{
  int day;

  printf("Enter day i.e., 1 for Sunday, 2 for Mon,.... 7 for Sat:");
  scanf("%d",&day);

  switch(day)
  {
    case 1:
     printf(“Today is holiday, enjoy as you wish\n”);
     break;

    case 2:
      printf(“Study Mathes\n”);
      break;

    case 3:
      printf(“Study English\n”);
      break;

    case 4:
      printf(“Study Science\n”);
      break;

    case 5:
      printf(“Study Social\n”);
      break;

    case 6:
      printf(“Study Telugu\n”);
      break;

    case 7:
      printf(“Study Hindi\n”);
      break;

    default:
      printf(“Invalid day\n”);
  }
}
```

```
Review Questions
----------------
01. Give list of possible examples of expressions that can be used inside
    the parenthesis of 'if' statement.
02. What is a block statement?
```

```
03. Write syntax for a 'if-else if-else if-else' statement.
04. What is the difference between 'if-else if-else if-else' statement
    and 'if-else if-else if' statement?
05. What is the difference between 'if-else if-else if-else if' statement
    and 'if-if-if-if' statement.
06. Write syntax for 'switch case' statement.
07. What is the use of 'break' statement inside a 'switch case' statement
    and what will happen if 'break' statements are not present between
    various cases?
08. What is the use of 'default' case in side a 'switch case' statement?
    And what will happen if no 'default' case is present?
09. Is it possible to replace every 'if else-if .. else-if else' statement
    with a switch case statement?

Assignments
-----------
1. Write a main() function which reads a character from the
   user and if character is a small letter it prints equivalent
   capital letter. If it is capital letter it prints equivalent
   small letter. If it is neither small nor capital it prints the
   same letter.

2. Write a main() function which reads an integer from the user
   and prints statement "It is Even number" or "It is odd number"
   based on the number is even or odd.

3. Write a main() function which reads two integer numbers, one is big number
   and another is small number. Next it prints whether small number is
   a factor of big number or not. Note a small number is a factor to
   big number if small number divides big number without leaving any
   remainder.

4. Write a main() function, which asks user to enter any small or
   capital letter. If user enters any other character, this function
   should display an error message like "Invalid input" and should
   return from the function. If it is a valid letter, it should print
   the next letter. Following are examples:

     If user enters   b     your program should print c
     If user enters   X     your program should print Y
     If user enters   z     your program should print a
     If user enters   Z     your program should print A
     If not English char    your program should print "Invalid char"

5. Ask user to enter two English letter characters. Read them into two
   char variables. You should print "Identical" if they are identical
   characters. Note that they are identical even if they are of different
   cases. Print "Not Identical" if they are not identical.
   Print "Invalid characters" if any of the characters is not a valid
   english letter character. Following are examples:

       a    b    ---->    Not Identical
       B    b    ---->    Identical
       K    K    ---->    Identical
       a    a    ---->    Identical
       5    s    ---->    Invalid characters
       $    #    ---->    Invalid characters

6. Ask user to enter the year. Print "Leaf Year" or "NOt a Leaf year"
   based on the year is a leaf or not. Leaf year is the one that is divisible
   by 4 without leaving reminder. Use modulo operator % to get the remainder
   of division.
```

7. Array variables

Arrays allow us to define a large number of variables of same type with a single name. To access individual variables, we use index along with the array name. Following are examples of individual variables and array variables.

```
char ch;
char name[40]; // 40 character variables

int maxmark;
int marks[60]; // 60 integers

short maxheight;
short heights[50]; // 50 short variables

float f;
float fvalues[20];  // 20 float variables
```

Every array variable definition contains the following:

- Type of the elements of the array
- Name of the array variable
- Number of elements of array in brackets
- Semicolon

So far we have seen how to declare individual variables. The above definitions are showing both single variable definition and array variable definitions.

ch is a single variable which can hold one byte of data.
name[40] is an array of 40 char variables. So we can store 40 small integers or 40 ASCII codes in that array.

maxmark is a single variable which can hold one integer data.
marks[60] is an array of 60 integer variables. We can store 60 student marks in this marks array.

The variable *f* is a single float variable, whereas *fvalues* is an array of 20 float values. So we can store 20 floating-point numbers in that array.

7.1 Using Array Variables

The following program shows the simple use of array variables. In this program, we are defining an array of four integers, an array of four floats, an array of four shorts and an array of four char integers. We are also defining one integer ‘sum’ and float ‘product’. Basically this program shows that, instead of defining separate variables, we can define arrays. Arrays are very helpful when we want to define hundreds of variables of same type.

prg7_1.c

```
int main()
{
  int   iarr[4], sum;
  float farr[4], product;
  short sarr[4];
  char  carr[4];

  printf("Enter four integer numbers\n");
  scanf("%d%d%d%d", &iarr[0], &iarr[1], &iarr[2], &iarr[3]);
  sum = iarr[0] + iarr[1] + iarr[2] + iarr[3];
  printf("Sum of four integers is %d\n", sum);

  printf("Enter four float numbers\n");
  scanf("%f%f%f%f", &farr[0], &farr[1], &farr[2], &farr[3]);
  product = farr[0] * farr[1] * farr[2] * farr[3];
  printf("product of four float numbers is %.3\n", product);

  printf("Enter four short numbers\n");
  scanf("%hd%hd%hd%hd", &sarr[0], &sarr[1], &sarr[2], &sarr[3]);
  sum = sarr[0] + sarr[1] + sarr[2] + sarr[3];
  printf("Sum of four short integers is %d\n", sum);

  printf("Enter four small (char) numbers\n");
  scanf("%hhd%hhd%hhd%hhd", &carr[0], &carr[1], &carr[2], &carr[3]);
  sum = carr[0] + carr[1] + carr[2] + carr[3];
  printf("Sum of four char integers is %d\n", sum);
}
```

7.2 Array variable Initialization

The following program illustrates how array variables can be initialized.

prg7_2.c

```
int   arr1[5];
int   arr2[4]   = {2, 10, 45, 36};
float farr[]    = {4.5, 25.23, 71.98, 111.23, 123.45, 149.1, 200.7};
char  carr[10]  = {1, 2, 3, 4, 5};

int main()
{
  printf("Value of arr1[0] is %d\n", arr1[0]);
  printf("Value of arr2[2] is %d\n", arr2[2]);
  printf("Value of farr[5] is %f\n", farr[5]);
  printf("Value of carr[0] is %d\n", carr[0]);
}
```

In the above program, we are defined four arrays. First two are integer arrays. Third one is a float array. And final one is char array.

First array is not initialized. All global variables, which are not initialized, will have zero as initial values. Second integer array is initialized. The float array is also initialized, but its size is not

specified. The compiler automatically decides the size based on the number of initialized values The character array is initialized with small integer numbers.

The following program is using an array of 5 integers, instead of using 5 separate variables.

prg7_3.c

```
int main()
{
  int   arr[5];

  printf(“Enter two numbers\n”);
  scanf(“%d%d”, &arr[0], &arr[1]);

  arr[2] = arr[0] + arr[1];
  arr[3] = arr[0] – arr[1];
  arr[4] = arr[0] * arr[1];
  printf(“%d, %d , %d, %d, %d\n”,arr[0],arr[1],arr[2],arr[3],arr[4]);
}
```

7.3 char Arrays and Strings

A char variable can store 8 bit integer. As we saw in the previous chapters, we also use char variable to store text character (i.e. ASCII code of text character). In the same way we can use char arrays to store multiple small integers as well as to store multiple text characters. A char array that is used to store text characters is called a string. Another important point about a string is that, the last character in the string must be always a NULL character.

In the following program we are defining two character arrays and storing a string of text characters in them. Because of this reason we will call them, as text strings are simply strings. There are 18 characters in the first string. The 18th character or the last character is ?. So ASCII code of ‘?’ is stored in the 17th location (Note that first char ‘H’ is stored in 0th location).

But now the important point is that, a NULL character whose ASCII code is 0 (zero), is stored after the last character ‘?’. So in the following text[50] array total 19 locations are occupied, which includes a NULL. For the second char array str[], we are not specifying the size. The size is decided by the number characters present in the initialized string plus 1 to keep NULL char. So the size of this array should be 17. We can use sizeof operator to print the size of this array.

Note the usage of `%s` format specification to display the string. We are using this first time here.

prg7_4.c

```

char  text[50]  = “Hello how are you?”;
char  str[]     = “Thanks I am fine”;

int main()
{
  printf(“String text = %s \n”, text);
  printf(“%d %d %d \n”, text[0], text[17], text[18]);

  printf(“String str  = %s \n”, str);
```

```
  printf(“size of str is %d\n”, sizeof(str));
}
```

The following program illustrates how to read the strings from the user.

prg7_5.c

```
char  text[50];
char  str[80];

int main()
{
  printf(“Enter first string: ”);
  scanf(“%s”, text);

  printf(“Enter second string: ”);
  scanf(“%[^\n]”, str);
  printf(“first string is : %s  and Second is %s \n”, text, str);
}
```

The first string is read by using `%s` as format. This way we can read string containing only one word, that is string without any spaces. Where as the format `%[^\n]` will allow us to read a string till new line character (\n) is entered. So using this format we can enter a string with multiple words.

```
Review Questions
----------------
01. What are array variables?
02. How to access individual element of an array?
03. How to initialize the array variables?
04. Is it possible to define an array without specifying size?
05. What is the difference between character array and string?
06. How to initialize the string?
07. What is the format specification to print a string?
08. What is the format specification to read a string using scanf()?
09. There are two ways of reading a string using scanf() function. What
    is the difference between them?

Assignments
-----------
i)  All programs should contain only a main() function.
ii) As we have not yet covered loop statements, you are not supposed to use
    them while writing the following programs.

1.  Define an array of six char variables. Read 6 subjects marks from the
    user into these 6 variables. Compute and print the total and average marks.
    Note that we are using char variables, as marks are always less than or
    equal to 100. Use correct format specification in scanf() function to
    read char variables.

2.  Define an array of four floats. Read four float number from the user
    into this array. Compare first and second number. If second number is
    bigger, then swap first and second numbers. (Note you may need to use
    some other variable while swapping). Similarly compare first and third,
```

and if third is bigger swap these two numbers. Finally compare first and fourth and swap if fourth is bigger. Print all the four numbers at the end.

Note: swapping means interchanging the contents of both variables. To swap variable 'a' and 'b'. First copy 'a' to temporary variable 't'. Then copy 'b' to 'a'. Finally copy 't' to 'b'

3. **Define an array 6 char variables. Ask user to enter a string of exactly 5 characters. Read the string using scanf() function.**
   Print the first 5 characters individually by using %c format.
   Print all the 6 characters of array as decimal numbers by using %d format.
   What could be the value of 6th character.
   Finally print the string by using %s format.

8. Loop statements

Loop statements are useful to execute a single statement or a set of statements (i.e block statement) multiple times. For example if we want to print hello world 10 times then we can use loop statement. In fact the loop statements are very useful to work with the array variables. For example we defined an array of 100 integers. Now we want to read 100 integers from the user by using scanf() function. If we do not have loop statements then we end up writing 100 scanf() statements. Because of this reason, arrays and loop statements will always go together.

There are three kinds of loop statements in C language. These are:

- **while** loop statement
- **do-while** loop statement
- **for** loop statement

All the above three loop statements will serve the same purpose but with subtle differences.

8.1 while() loop

The *while loop* has got the following syntax:

```
while(<expression>)
  <statement>;
```

This while loop first executes the expression present inside the parenthesis. If this expression result is FALSE, it will skip the execution of the following statement. If result is TRUE, it executes the statement after the parenthesis and it continues above steps till expression inside the parenthesis gives FALSE. Note that the statement after the while loop could be a block statement containing multiple statements inside the braces.

Following is the example.

prg8_1.c

```
int i = 0;
int array[10];

int main()
{
  while(i<10)
  {
    printf("Hello World\n");
    i++;
  }

  i = 0;
  while(i<10)
  {
    printf(“Enter number %d : “, 1+1);
    scanf(“%d”, &array[i]);
    i++;
```

```
  }
}
```

The following program prints the required multiplication table.

prg8_2.c

```
#include <stdio.h>

int main()
{
  int tableNum;
  int i;

  printf("Enter multiplication table number to print:");
  scanf("%d",&tableNum);

  i = 1;
  while(i <= 10)
  {
    printf("%d x %d  = %d \n", tableNum, i, tableNum * i);
    i++;
  }
}
```

8.2 do-while() loop

Following is the syntax for *do-while loop*:

```
do
{

  <statement>;

} while(<expression>);
```

The do-while statement first executes all the statements present after do. Next it executes the expression present in the parenthesis after ‘while’ keyword. If expression’s result is True it continues the do loop execution again. If false it comes out of *do-while loop*.

Following is the example.

prg8_3.c

```
int main()
{
  i = 0;

  do
  {
    printf("Hello world\n");
    i++;
```

```
  } while(i<10);
}
```

The difference between while() and do while() is that, do-while () always executes the "statements in loop" at least once. Where as while() loop may or may not execute the "statements in loop"

8.3 for() Loop

The for() loop statement has got the following syntax:

```
for(<statement1>; <expression2>; <statement3>)
  <statement>;
```

As shown in the above format, *for* statement consists of 'for' keyword followed by three statements separated by semicolons inside the parenthesis. The *for* statement works as follows.

First it executes the first statement in parenthesis. Next it evaluates the second expression and result of this second expression is verified for True or False. If result is True, then for statement executes the statement after the parenthesis. Next it executes the statement 3. Next it continues with evaluating expression till it fails.

We can always convert every *for loop* into equivalent *while loop*. So the above *for loop* is equal to the following *while loop*.

```
<statement1>;
while(<expression2>)
{
  <statement>;
  <statement3>;
}
```

Now look at the following for loop statement.

```
for(i=0; i<10; i++)
  printf("Hello world\n");
```

First it executes the first statement:

```
  i = 0;
```

So i becomes 0.

Next it executes the second statement:

```
  i < 10;
```

As i is zero this statement gives TRUE result. So it executes the statement after parenthesis. That is:

```
  printf("hello world\n");
```

Next it executes the third statement in parenthesis, that is:

```
i++;
```

With this third statement i becomes 1.
Next it jumps back to second statement (`i < 10`) and continues till second statement gives FALSE.

In the above example, statement to execute has only single statement. If in each iteration of the loop, we want execute multiple statements, then we can use block statement that contains multiple statements as shown below:

```
for(i=0; i<10; i++)
{
  printf("Enter value\n");
  scanf("%d", &ival[i]);
  printf("value %d is stored in element %d\n", ival, i);
}
```

The above statement executes the three statements in for loop 10 times.

8.4 break statement in loops

When ‘break’ statement is executed in side any loop, the control will come out of the loop and statement after the loop will get executed.

We also used ‘break’ statement already in ‘switch-case statement. After each ‘case’, we are keeping statements to execute for that case. The last statement under each case is ‘break’ statement. This ‘break’ makes control to exit from the switch statement. If no break statement is present, then statements of next case will get executed.

The following program uses *for loop* to read 10 numbers from the user. However if user enters any number equal to or less than zero, it breaks from the *for loop* immediately. After *for loop* it is displaying all the number in array.

prg8_4.c

```
int arr[10];

int main()
{
  int i,j;

  for(i=0; i<10; i++)
  {
    printf("Enter %dth number\n", i);
    scanf(“%d”, &arr[i]);
    if(arr[i] <= 0)
      break;
  }
  for(j=0; j<i; j++)
```

```
    printf(“%dth number is %d\n”, i, arr[j]);
}
```

8.5 continue statement in loops

When ‘continue’ statement is executed in side any loop, the execution will skip the remaining statements in the loop and proceeds with next iteration of loop.

The following program contains two for loops. In first for loop, we are reading 10 numbers from the user. In the second for loop we are printing the square of each number. But if number happens to be less than or equal to zero, we are skipping the remaining statements in the loop, and continuing with the next iteration.

prg8_5.c

```
int arr[10];

int main()
{
  int i, sqr;

  for(i=0; i<10; i++)
  {
    printf("Enter %dth number\n", i);
    scanf(“%d”, &arr[i]);
  }

  for(i=0; i<10; i++)
  {
    if(arr[i] <= 0)
      continue;

    sqr = arr[i] * arr[i];
    printf(“square of %dth number is %d\n”, i, sqr);
  }
}
```

8.6 Sample Programs using loops and arrays

Sample Program 6

This program just reads 10 integers from the user and prints all the 10 integers later.

prg8_6.c

```
#include <stdio.h>

int array[10];

int main()
{
  int i=0;
```

```
  while(i < 10)
  {
    printf ("Enter number %d : ",i+1);
    scanf("%d", &array[i]);
    i++;
  }

  printf("Following are the numbers entered by you\n");
  for(i=0; i<10; i++)
    printf(" Number %d is %d \n", i+1, array[i]);
}
```

Sample Program 7

This program finds the maximum mark from the marks present in the initialized array.

prg8_7.c

```
#include <stdio.h>

int student_marks[6] = {73, 82, 56, 89, 92, 68};

int main()
{
  int maxmark,i;

  maxmark = student_marks[0];

  for(i=1; i<6; i++)
  {
    if(maxmark < student_marks[i])
      maxmark = student_marks[i];
  }

  printf("Maximum mark of student is %d\n", maxmark);
}
```

Sample Program 8

This program computes and prints the average mark from the marks present in the initialized array.

prg8_8.c

```
#include <stdio.h>

int student_marks[6] = {73, 82, 56, 89, 92, 68};

int main()
{
  int total = 0;
  int avg,i;

  for(i=0; i<6; i++)
```

```
  {
    total  = total + student_marks[i];
  }
  avg = total / 6;
  printf("Avarage mark of student is %d\n", avg);
}
```

Sample Program 9

This program prints whether a student is failed or passed in ordinary, second class or first class.

prg8_9.c

```
#include <stdio.h>

int student_marks[6] = {73, 82, 56, 89, 92, 68};

int main()
{
  int total = 0;
  int i;
  int fail=0;

  for(i=0; i<6; i++)
  {
    if(student_marks[i] < 35)
    {
      fail = 1;
      break;
    }
    total  = total + student_marks[i];
  }

  if(fail)
    printf("Student is failed\n");
  else if (total >= (6*60))
    printf("Student passed in first class\n");
  else if (total >= (6*50))
    printf("Student passed in second class\n");
  else
    printf("Student passed as ordinary\n");
}
```

Sample Program 10

This program takes marks of students of a class. The program continues to accept marks till –1 is entered. In this way the program need not know number of students in the class.

prg8_10.c

```
#include <stdio.h>

int students[100];

int main()
{
```

```
  int count = 0;
  int marks;

  do
  {
    printf("Enter marks of student, if no more students enter -1\n");
    scanf("%d", &marks);
    students[count++] = marks;
  }while(marks >= 0);

  count--;
  printf("Thanks for entering marks of %d students\n", count);
}
```

Sample Program 11

This program reads a string containing multiple words from the user. And displays the string. It uses a While loop to count the number of characters present in the string. It counts characters till NULL character (ASCII code 0) is encountered.

prg8_11.c

```
#include <stdio.h>

char string[100];

int main()
{
  int ii = 0;

  printf("Enter a string with multiple words\n");
  scanf("%[^\n]", string);

  printf("Your string is: %s\n", string);
  while(string[ii] != 0)
    ii++;
  printf("Number of characters in the string are %d\n", ii);
}
```

Sample Program 12

This program defines two string arrays, each of size 100. Next it reads a string from the user in to string1. Next it copies string1 into the second string. Finally it prints the content of second string.

prg8_12.c

```
#include <stdio.h>

char string1[100];
char string2[100];

int main()
{
  int ii = 0;
```

```
  printf("Enter a string with multiple words\n");
  scanf("%[^\n]", string1);

  printf("Your string is: %s\n", string1);

  while(string1[ii] != 0)
  {
    string2[ii] = string1[ii];
    ii++;
  }
  string2[ii] = 0;

  printf("New string is: %s\n", string2);
}
```

Sample Program 13

This program reads a string from the user and counts the number of vowels present in the string and finally prints the vowel count.

prg8_13.c

```
#include <stdio.h>

char string[100];

int main()
{
  int ii = 0;
  int vowels = 0;

  printf("Enter a string with multiple words\n");
  scanf("%[^\n]", string);

  printf("Your string is: %s\n", string);

  while(string[ii] != 0)
  {
    switch(string[ii])
    {
      case 'a': case 'e': case 'i': case 'o': case 'u':
      case 'A': case 'E':
      case 'I':
      case 'O':
      case 'U':
        vowels++;
    }
    ii++;
  }

  printf("Number of vowels in the given string are: %d\n", vowels);
}
```

Sample Program 14

This program reads a string from the user and counts the number of space characters present in the string. Next it prints the number of words in the string by assuming that only space character is lies between two words. But note that there could be multiple space characters in between the two words. Sometimes there could be space characters even before the first word as well as after the last word. But we are not taking care of those situations.

prg8_14.c

```
#include <stdio.h>

char string[100];

int main()
{
  int ii = 0;
  int words = 0;

  printf("Enter a string with multiple words\n");
  scanf("%[^\n]", string);

  printf("Your string is: %s\n", string);

  while(string[ii] != 0)
  {
    if(string[ii] == ' ')
      words++;
    ii++;
  }
  words++;

  printf("Number of words in the given string are: %d\n", words);
}
```

Sample Program 15

The following program reads the two strings and appends the second string at the end of the first string.

prg8_15.c

```
#include <stdio.h>

char string1[100];
char string2[100];

int main()
{
  int ii = 0;
  int jj = 0;

  printf("Enter a string1 with multiple words\n");
  scanf("%[^\n]", string1);

  __fpurge(stdin);
```

```
  printf("Enter a string2 with multiple words\n");
  scanf("%[^\n]", string2);

  printf("Your string1 is: %s\n", string1);
  printf("Your string2 is: %s\n", string2);

  while(string1[ii] != 0)
    ii++;

  while(string2[jj] != 0)
  {
    string1[ii] = string2[jj];
    ii++;
    jj++;
  }
  string1[ii] = 0;

  printf("String1 after appending string2 is: %s\n", string1);
}
```

Sample Program 16

The following program is printing the given number in bit by bit, i.e. in binary format. In a For loop, we are always testing the 31st bit. Next we are shifting the ‘val’ left by one bit, so that 30th bit will come to 31st position. This we are continuing for all the 32 bits.

prg8_16.c

```
#include <stdio.h>


int main()
{
  int i;
  int val;

  printf("Enter an integer\n");
  scanf("%d", &val);

  for(i=0; i<32; i++)
  {
    if(val & 0x80000000)
      printf("1");
    else
      printf("0");
    val = val << 1;
  }

  printf("\n");
}
```

Sample Program 17

The following program reads an integer from the user and displays the integer in binary form. The ‘maskbit’ is initially set to 0x80000000. That is 31st bit set to 1. When we do bit wise &

operation, the 31st bit will get tested. For the next iteration we are shifting the 'maskbit' right by one bit so that 30th bit will be tested. In this way, in the For loop, we are testing and printing all the 32 bits of the given integer. Note that in the above program we are shifting 'val' and keeping the mask bit constant.

Next this program is trying to change the given bit to the given value that is 0 or 1. This program is reading the bit number to change and then bit value of this bit number. To set the given bit to 1, we are moving 1 to that bit position and performing bit wise OR operation. To set the given bit to 0, we are moving 1 to that bit position and then complementing. With this complement operation all other bits will become 1 and this bit at given bit poison becomes zero. Finally we are doing bit wise AND operation to set the given bit.

Finally we are displaying all the bits once again to see the effect of our change.

prg8_17.c

```
#include <stdio.h>

int main()
{
  int i;
  int val,bitno,bitval;
  unsigned int maskbit;

  printf("Enter an integer\n");
  scanf("%d", &val);

  printf("Following is the 32 bit integer in binary format\n");
  maskbit = 0x80000000;
  for(i=0; i<32; i++)
  {
    if(val & maskbit)
      printf("1");
    else
      printf("0");
    maskbit = maskbit >> 1;
  }
  printf("\n");

  printf("Enter the bit number (0 to 31) you would like to modify: ");
  scanf("%d", &bitno);
  if((bitno > 31) || (bitno < 0))
  {
    printf("Sorry invalid bit number, exiting.. ");
    return 0;
  }
  printf("Enter the bit value (0 or 1) you would like to set there: ");
  scanf("%d", &bitval);
  if((bitval > 1) || (bitval < 0))
  {
    printf("Sorry invalid bit value, exiting.. ");
    return 0;
  }
  if(bitval == 1)
    val = val | (1 << bitno);
```

```
  else
    val = val & ~(1 << bitno);

  printf("32 bit integer in binary format after change\n");
  maskbit = 0x80000000;
  for(i=0; i<32; i++)
  {
    if(val & maskbit)
      printf("1");
    else
      printf("0");
    maskbit = maskbit >> 1;
  }
  printf("\n");
}
```

Review Questions
----
01. What is infinite loop and how to write it?
02. What is 'break' statement, and where is it useful?
03. What is 'continue' statement, and where do you use it?
04. Which one do you like 'for' or 'while'?

Assignments
----
i) All programs should contain only a main() function. No more additional functions.

01. Define an array of 6 shorts. Ask user to enter the marks of 6 subjects. Compute and display whether student is failed, or passed in ordinary, second class or first class. Assume pass mark as 35.

02. Define an array of 6 chars. Ask user to enter the marks of 6 subjects. Just print whether student is passed or failed. If failed allow 5 grace marks. These grace marks can be added to subjects in which student has failed. Compute whether with grace marks, student is passed or failed. Note that even though student has got 34 marks in 5 subjects still she/he is passed because of 5 grace marks. I hope you got the point.

03. Define an array of 100 integers. First ask user how many students have taken the exam. Next ask user to enter marks of all these students. Compute and display the highest mark, lowest mark and average mark. Use only a single loop. Within a single loop check for lowest, highest and total.

Strings

04. Define a string of 100 characters and read a string from the user. Count the number of capital letters (>= 'A' && <= 'Z'), number of small letters and number of other (not capital and not small) characters. Finally display all these three numbers. Note: Use if-else if-else if-else statement inside the loop.

05. Define and read a string from the user. Next convert each capital letter to small letter (just add 32 to capital letter) and each small letter to capital (just subract 32 from small letter). Finally print the modified string.

06. Read a string and count the number of vowels by using If, Else-If,

Else-If statements. Finally print the vowels count.

07. Define two strings. Read one string from the user. Copy the first string into the second string from the reverse direction. For example if first string is "Hello world", the second string should be "dlrow olleH". Print the second string.

08. The above program is reversing a string by copying it into second string. It is some what easy. Now try to reverse a string without using second string. First find the length of the string. Set ii=0 and jj to string length minus 1. So jj is pointing to last character. Next swap these two characters. Increment ii and decrement jj. Continue this till ii is less than jj.

09. Read a string and count the number of words present in it. Assume that there could be any number of space characters between two words. Also there could be any number of spaces before the first word and after the last word.

Bits

Read all the numbers only in hex by using %x as format. Define all numbers as 'unsigned int'.

10. Write a program to read hex number and print it in binary format. Next print the total number of 1 bits. You should count the 1 bits while printing them, so that you have only a single loop.

11. This is same as above program, you should print the number of zero bits.

12. Write a program to read hex number and print it in binary, next rotate it left by one bit and print it again in binary. To rotate left, first check the 31st bit. If it is zero just shift left. If it is one, shift left and OR with 1.

13. Write a program to read hex number and print it in binary, next rotate it right by four bits and print it again in binary.

14. Read hex number, print it in binary, complement it and print again in binary. Both binary numbers should be one below the other.

15. Read two hex numbers, print both of them in binary one below the other and finally print the bit wise AND of these two number below these two numbers.

16. Read two hex numbers, print both of them in binary one below the other and finally print the bit wise OR of these two number below these two numbers.

17. Read two hex numbers, print both of them in binary one below the other and finally print the bit wise XOR of these two number below these two numbers.

9. Structure variables

Structure variables are like compound variables that hold many variables (may be of different type) in it. So to define a structure variable first we should declare the structure template. In this structure template we specify the field variables (their name and types) that are going to present in side that structure. Next by using this structure template name, we can define the structure variables of that type. Following are the various structure template examples.

```
// structure template declarations

struct point
{
  int x;
  int y;
};


struct line
{
  struct point  p1;
  struct point  p2;
  int           line_width;
};

struct book
{
  char title[40];
  char author[40];
  char publisher[40];
  int  price;
  int  edition;
};

struct student
{
  char name[40];
  int  phone;
  char address[50];
};

// Structure variable definitions

struct point  pt;
struct point  points[10];

struct line   ln;
struct line   lines[5];

struct book   mybook;
struct book   mybooks[20];

struct student st;
struct student sts[40];
```

In the above sample code, we declared various structure templates first. Next by using these template names, we defined the structure variables. It is very important to distinguish structure templates and structure variables.

prg9_1.c

```
#include <stdio.h>

struct point
{
  int x;
  int y;
};


struct line
{
  struct point  p1;
  struct point  p2;
  int           line_width;
};

int main()
{
  struct point p;
  struct point pts[5];
  struct line  ln;
  int          i;

  //Following statements will fill point variable p
  printf("Enter x and y co-ordinates of p\n”);
  scanf("%d%d",&p.x, &p.y);

  //Following statements will fill array of 10 point structures
  for(i=0; i<5; i++)
  {
    printf("Enter x and y co-ordinates of %d th point\n”, i);
    scanf("%d%d",&pts[i].x, &pts[i].y);
  }

  //Following statements will fill the line structure
  printf(“Enter x and y of point p1\n”);
  scanf(“%d%d”, &ln.p1.x, &ln.p1.y);

  printf(“Enter x and y of point p2\n”);
  scanf(“%d%d”, &ln.p2.x, &ln.p2.y);

  printf(“Enter line width of line\n”);
  scanf(“%d”, &ln.line_width);

  //Now You add statements to display contents of p, pts[5] and ln
}
```

The following program shows how to fill and print the struct student variable.

prg9_2.c

```
#include <stdio.h>

struct student
{
  char name[40];
  int  phone;
  char addrs[50];
};

int main()
{
  struct student s;

  // Following statements will fill the structure variable ‘s’
  printf("Enter student’s name:\n”);
  scanf(“%[^\n]”, s.name);

  printf(“Enter student’s phone:\n”);
  scanf(“%d”, &s.phone);

  __fpurge(stdin);
  printf(“Enter student’s address:\n”);
  scanf(“%[^\n]”, s.addrs);

  // Following statement will print the content of structure variable s
  printf(“Name: %s, Phone: %d, Address: %s\n”,s.name, s.phone,s.addrs);
}
```

In the above program to fill the ‘name’ and ‘address’ fields of structure, we used *gets()* function, instead of *scanf()* function. The *gets()* function allow us to take name containing multiple words. Where as *scanf()* accepts only a single word.

prg9_3.c

```
#include <stdio.h>

struct student
{
  char name[40];
  int  phone;
  char addrs[50];
};

struct student students[ ] =
{
  { "DEPIK Tech", 23818683, "MHR House, SR Nagar"},
  { "Suresh", 23818681, "F1, ABC Apartments, Ameerpet"},
  { "Ramesh", 23808682, "F202, Krish Apartments, Begumpet"},
  { "Rajesh", 23708682, "#103, Parimala Apartments, Balanagar"},
  { "Swathi", 23728682, "#204, Sri Apartments, Srinagar colony"},
  { "Jyothi", 23718682, "F4, Hitech Apartments, Madhapur"}
};

int main()
{
```

```
  int ii, pnum, npnum;

  for(ii=0; ii < sizeof(students)/sizeof(struct student); ii++)
  {
    printf("%-10s   %d    %s\n",students[ii].name, students[ii].phone,
                                students[ii].addrs);
  }

  printf("Enter the phone number to modify : ");
  scanf("%d", &pnum);
  printf("Enter the new phone number       : ");
  scanf("%d", &npnum);

  for(ii=0; ii < 6; ii++)
  {
    if(students[ii].phone == pnum)
    {
      students[ii].phone = npnum;
      break;
    }
  }
  if(ii==6)
    printf("Sorry phone number not found\n");

  for(ii=0; ii < sizeof(students)/sizeof(struct student); ii++)
  {
    printf("%-10s   %d    %s\n",students[ii].name, students[ii].phone,
                                students[ii].addrs);
  }
}
```

prg9_4.c

```
#include <stdio.h>

#define MAX_STUDENTS  10

struct student
{
  char name[40];
  int  phone;
  char addrs[50];
};

struct student students[MAX_STUDENTS];

int main()
{
  int ii, choice;

  while(1)
  {
    printf("\n\tEnter 1 to add student record\n");
    printf("\tEnter 2 to display all student records\n");
    printf("\tEnter 3 to exit\n\n");
    printf("Enter your choice now: ");
```

```
    scanf("%d", &choice);
    switch(choice)
    {
      case 1:
        for(ii=0;ii<MAX_STUDENTS; ii++)
        {
          if(students[ii].phone==0)
          {
            printf("Enter name: ");
            __fpurge(stdin);
            scanf("%[^\n]",students[ii].name);
            printf("Enter phone number: ");
            scanf("%d", &students[ii].phone);
            printf("Enter address: ");
            __fpurge(stdin);
            scanf("%[^\n]",students[ii].addrs);
            break;
          }
        }
        if(ii==MAX_STUDENTS)
          printf("No sapce to add a new record\n");
        break;

      case 2:
        for(ii=0;ii<MAX_STUDENTS; ii++)
        {
          if(students[ii].phone)
            printf("Name: %s,Phone: %d, Addrs: %s\n",students[ii].name,
                   students[ii].phone, students[ii].addrs);
        }
        break;

      case 3:
        return 0;

      default:
        printf("Invalied choice\n");
    }
  }
}
```

Assigning a structure variable

We can copy one structure variable into another structure variable by using assignment statement. To use structure assignment, both the variables must be structures of identical type. If s1 and s2 are structure variables of type ‘struct student’. Assume that all fields of s1 are initialized with some values. Now we can copy content of s1 into s2 with the following assignment statement.

```
s2 = s1;
```

prg9_5.c

```
#include <stdio.h>

struct student
{
  char name[40];
  int  phone;
  char addrs[50];
};

int main()
{
  struct student s1, s2;

  // Following statements will fill the structure variable ‘s1’
  printf("Enter student’s name:\n”);
  scanf(“%[^\n]”, s1.name);

  printf(“Enter student’s phone:\n”);
  scanf(“%d”, &s1.phone);

  __fpurge(stdin);
  printf(“Enter student’s address:\n”);
  scanf(“%[^\n]”, s1.addrs);

  s2 = s1;

  // Print the content of structure variable s2
  printf(“Name:%s, Phone:%d, Address:%s\n”,s2.name, s2.phone,s2.addrs);
}
```

```
Review Questions
----------------
01. How to initialize the structure variables?
02. Is it possible to assign one structure variable to another structure
    variable of same type?
03. If there are two identical arrays of same type like 'int a[10], b[10];'
    then is it possible to assign one array to other?
04. If same integer array is present in side a structure variable then
    can we assign that structure variable to another structure variable of
    same type?
05. What is the structure template?

Assignments
-----------
 1. Define a structure variable of 'struct book' type given below.
    Fill this structure by taking the data from the user. Finally
    print all the fields in the structure.

    struct book
    {
      int   bookId;
      char  title[64];
      char  author[64];
      char  publisher[64];
      short yearOfEdition;
      int   price;
    };

 2. This is similar to above program. Define an array of 5 book
    structures. Read all the five structures from the user. Finally
```

display all the five structures.

3. This is similar to sample program prg9_4.c given in your material. Instead of student records this program maintains the book records. This program provides the following options:

    1 - Add a book record
    2 - display all the books
    3 - Delete a book record
    4 - Display a book record of given ID
    5 - Display books whose price is lies in between given price.

   When user selects 'Delete' option, the program asks the Book Id and just sets that bookId to zero. So note that any book structure with bookId zero is considered empty record. So when adding a new book record, you may put that in this empty record.

   The option 5 asks user to enter low price and high price and displays all the books whose price is in between.

4. Define two variables of type 'struct vehicleEntry'. Fill one of them by reading data from the user. Copy this variable to second variable by using assignment statement and print the second variable. If user enters any invalid data, the program should display error and exit.
   Following are the valid data.

```
seconds: 0 to 59
minutes: 0 to 59
hours  : 0 to 23
date   : 1 to 31
month  : 1 to 12
```

   You may also validate date based on the month, if you feel you are a good programmer.

```
struct mytime
{
  char secs;
  char mins;
  char hours;
};

struct mydate
{
  short date;
  short month;
  short year;
};

struct datetime
{
  struct mydate  dt;
  struct mytime  tm;
};

struct vehicleEntry
{
  char vehicalNumber[20];
  struct datetime entryTime;
};
```

10. Pointer variables

Understanding pointers very well is the most important requirement for good C programmers. In fact understanding pointers is very easy if someone teaches them in right way. If not understood them properly, pointers will cause a lot of confusion to the students. Because of this confusion, lot of students will afraid of pointers as well as C programming. But believe me, pointers are so simple to understand and very interesting to use. In fact the real power and pleasure of C language is due to its pointers.

Direct address and Address operator

When we are learning about variables, we learned that every variable occupies a certain location in the memory. Each such location will have certain address and size. The following program will print the size, content and address of the various locations associated with various types of variables. The address operator &, is used to print the address and *sizeof* operator to print the size of variable.

prg10_1.c

```
#include <stdio.h>

char   c = 100;
short  s = 2000;
int    I = 30000;
float  f = 1.2345;
double d = 2.3456789;

int main()
{
  printf("Size,content and address of c: %d, %d, %x", sizeof(c),c,&c);
  printf("Size,content and address of s: %d, %d, %x", sizeof(s),s,&s);
  printf("Size,content and address of i: %d, %d, %x", sizeof(i),i,&i);
  printf("Size,content and address of f: %d, %f, %x", sizeof(f),f,&f);
  printf("Size,content and address of d: %d, %f, %x", sizeof(d),d,&d);
}
```

Let us call the address of a variable as 'Direct Address'. So every variable will have some 'Direct address'. In the above program we are printing the direct addresses of all the variables.

Direct addressing

When we use the following statement to store 25 in variable 'a',

```
a = 25;
```

We are instructing the computer to store a value of 25 in the location starting at the direct address of variable 'a'. This kind of addressing is called 'Direct Addressing'. We read or write into variables by doing direct addressing. So all variables support direct addressing. I am stressing this direct addressing because; there is another addressing called 'Indirect addressing'. This 'indirect addressing' is supported only by pointer variables. But note that pointer variables support both direct addressing and indirect addressing.

Pointer variables

Remember that when we are discussing about variables in chapter 3, we classified them into three groups. These are Integers, Floating type and Pointer type. So far we used only Integer and Float type variables. In this chapter we will learn about Pointer type variables.

Pointer variables are used to store addresses of memory locations. Next by using these pointers we can read or write into those memory locations. The following is the data definition statement, defining a pointer variable.

```
int  *p;
```

The above definition tells about two things. One is about ‘p’ and second is about *p. It says ‘p’ is a pointer variable in which we can store address of integer memory location. The name after * is the name of pointer variable. In 32 bit computers, addresses are 32 bit numbers. So size of pointer variable will be 32 bits or 4 bytes. The *p is the integer memory location, whose address is in ‘p’. So the ‘int’ type in the data definition statement refers to the *p.

We can get address to store in ‘p’ in two possible ways. One way is to get the address of some integer variable by using address operator &. This is shown below:

```
p = &a;
```

In the above statement ‘a’ is an integer variable. Now ‘p’ contains the address of ‘a’ and *p represents ‘a’ location. This is illustrated with the following program.

prg10_2.c

```
#include <stdio.h>

int main()
{
  int  a = 100;
  int  *p;

  printf(“Size of p is %d\n”, sizeof(p) );
  printf(“Size of *p is %d\n”, sizeof(*p) );
  p = &a;
  printf(“Address of a is %x\n”, &a);
  printf(“Content of p is %x\n”,  p);
  printf(“Value of a is %d\n”, a);
  printf(“ *p is %d\n”, *p);
  *p = 500;
  printf(“ Value of a is %d\n”, a);
}
```

In the above program the ‘p’ is called an integer pointer, because *p represents integer location. We can define pointers of any type. The following program shows the usage of char pointer.

prg10_3.c

```
#include <stdio.h>

int main()
{
  char  c = 10;
  char  *p;

  printf(“Size of p is %d\n”, sizeof(p) );
  printf(“Size of *p is %d\n”, sizeof(*p) );
  p = &c;
  printf(“Address of c is %x\n”, &c);
  printf(“Content of p is %x\n”,  p);
  printf(“Value of c is %d\n”, c);
  printf(“ *p is %d\n”, *p);
  *p = 50;
  printf(“ Value of c is %d\n”, c);
}
```

Every pointer variable contains the address. But what location *p represents depends on the pointer type.

Indirect Addressing

Pointer variable ‘p’ will have some direct address and it supports direct addressing. The following statement shows the direct addressing with pointer variable ‘p’.

```
p = &a;
```

In the above statement we are storing the address of ‘a’ in pointer variable ‘p’. This is direct addressing similar to a = 25; discussed above. The content of ‘p’ is also address. We call this address as indirect address. It is very important to distinguish these two addresses. So let me say it gain. Pointer variable will have direct address, similar to any variable. We can print the direct address of ‘p’ by using address operator &. What we store in ‘p’ , that is content of ‘p’ is also called address. Let us call this address indirect address.

Writing or reading some thing from ‘p’ is called direct addressing. Where as writing or reading from *p (location pointed by indirect address) is called indirect addressing.

```
p = &a;    // direct addressing. Writing to ‘p’
*p = 125; // indirect addressing. Writing to ‘a’.
```

prg10_4.c

```
#include <stdio.h>

int     val = 300;

int main()
{
  int    *p;

  p = &val;
  printf(“Direct address of p is %x\n”, &p);
  printf(“Indirect address of p is %x\n”, p);
```

```
  printf("Value at indirect address is %d\n", *p);
}
```

Accessing arrays using pointers

In the above sections we saw how to access a variable by using a pointer variable. This we have done by storing the address of variable in pointer variable. Similarly we can store the address of array in a pointer variable and by using this pointer variable we can access the elements of array.

In C language there exist a lot of similarity between pointer variables and arrays. Every array name acts like a constant pointer, initialized with the address of beginning of array. So every array name is a pointer with fixed address in it.

```
int *p;
int a[10];
```

In the above, we are defining integer pointer variable ‘p’ and an integer array ‘a’ of size 10 elements. Now we can store address of array in pointer ‘p’ in two possible syntaxes as shown below:

```
p = &a[0];
p = a;
```

The first statement is easy to understand, as we are storing the address of first element of array in ‘p’. But as we already learned above that, array name is a pointer, initialized to array’s start address; the second statement is identical to the first statement. Now we can access the elements of array ‘a’ through pointer variable ‘p’ in two syntaxes. First one is called pointer syntax and second is called array syntax.

Pointer Syntax:

```
*p     = 1;  // same as a[0] = 1;
*(p+1) = 2;  // same as a[1] = 2;
*(p+9) = 3;  // same as a[9] = 3;
```

Array Syntax:

```
p[0] = 1;  // same as a[0] = 1;
p[1] = 2;  // same as a[1] = 2;
p[9] = 3;  // same as a[9] = 3;
```

As shown above once pointer variable is initialized with the address, we can access the successive elements at that address by using that pointer variable (with pointer syntax or array syntax).

Also note that, as array name ‘a’ represents a pointer, we can access array elements by using array name in pointer syntax. This is illustrated below:

```
*a     = 1;  // same as a[0] = 1;
*(a+1) = 2;  // same as a[1] = 2;
*(a+9) = 3;  // same as a[9] = 3;
```

The following sample program illustrates the concepts learned above. First you guess the output of following *printf()* statements, and then run this program and verify with your result.

prg10_5.c

```
#include <stdio.h>

int ar[10] = {10, 5, 21, 32, 76, 32, 20, 3, 7, 11};
int *p;

int main()
{
  p = a;
  printf("%d", ar[0]);
  printf("%d", *(ar+1));
  printf("%d\n", p[2]);
  printf("%d\n", *(p+3));

  p[0]   = 123;
  *(p+1) = 234;
  a[2]   = 321;
  *(a+3) = 456;

  printf("%d", ar[0]);
  printf("%d", *(ar+1));
  printf("%d\n", p[2]);
  printf("%d\n", *(p+3));
}
```

Sample Program 6

prg10_6.c

```
#include <stdio.h>

int    array[10] = { 23, 45, 61, 22, 58, 92, 60, 71, 99, 200};

int main()
{
  int    *p;

  p = array;

  for(ii=0; ii<10; ii++)
    printf("%d  ", *(p+ii) );
  printf("\n\n");

  for(ii=0; ii<10; ii++)
    printf("%d  ", p[ii] );
  printf("\n\n");


  for(ii=0; ii<10; ii++)
    printf("%d  ", *(array+ii) );
  printf("\n\n");

```

```
  for(ii=0; ii<10; ii++)
    printf(“%d  “, array[ii] );
  printf(“\n\n”);
}
```

In the above program we are initializing pointer ‘p’ with ‘array’. Now we can use ‘p’ to access elements of array in two notations. One is with pointer notation, i.e. `*p, *(p+1), *(p+2), *(p+3),` etc.. Another is array notation i.e `p[0], p[1], p[2]` etc.. The above program is illustrating both the methods.

Also note that in C language, the array name itself works as a constant pointer. So we can use array name in pointer notation to access its elements like `*array, *(array+1), *(array+2),` etc..

Sample Program 7

The following program shows how to fill the array by using pointer variable. This program also shows two ways of using pointers. One way is to pass `&p[ii]`. In this method we are using p in array notation. So we are passing address of ii element of array. In the second method we are passing `p+ii`. Here `p` represents the address of fist element, when added `ii` it gives address of `ii` element.

prg10_7.c

```
#include <stdio.h>

int    array[10];

int main()
{
  int    *p;

  p = array;

  for(ii=0; ii<10; ii++)
    scanf(“%d“, (p+ii) );

  for(ii=0; ii<10; ii++)
    scanf(“%d  “, &p[ii] );

  for(ii=0; ii<10; ii++)
    printf(“%d  “, array[ii] );

  printf(“\n\n”);
}
```

Accessing structure variables using pointers

We can define pointer variables of any specific structure type. Next we can store the address of similar structure variable in this pointer variable. Now with the help of this pointer we can read or

write to that structure variable. The following statements define structure and structure pointer variables.

```
struct student
{
  char name[40];
  int  id;
  int  phone;
};

struct student s;   // s is a structure variable
struct student *ps; // ps is a pointer variable

ps = &s;  // ps points to memory of structure variable s
```

We can use structure pointer to access the fields the structure pointed by the pointer, as shown below:

```
*ps.phone   = 23818683;  // is wrong because it is equal to *(ps.phone)
(*ps).phone = 23818683;  // This is correct way of accessing fields
                         //   fields with structure pointer
```

But there is a short form notation to access structure fields with a structure pointer as shown below:

```
  ps->phone = 23818683;  // short notation
  ps->id    = 123;       // short notation
```

The following sample program shows how to display the contents of structure variable by using a structure pointer variable.

prg10_8.c

```
#include <stdio.h>

struct student
{
  char name[40];
  int  phone;
  char addrs[50];
};

struct student s = { "DEPIK Tech", 23818683, "MHR House, SR Nagar"};

int main()
{
  struct student *ps;

  ps = &s;

  printf("Name: %s, Phone: %d, Addrs: %s\n", (*ps).name, (*ps).phone,
                                                        (*ps).addrs);

  printf("Name: %s, Phone: %d, Addrs: %s\n",ps->name,ps->phone,
                                                        ps->addrs;

}
```

The following program illustrates the using of structure pointer variable to read the data from user.

prg10_9.c

```
#include <stdio.h>

struct student
{
  char name[40];
  int  phone;
  char addrs[50];
};

struct student s;

int main()
{
  struct student *ps;

  ps = &s;

  printf(“Enter name, phone and address\n”);
  scanf(“%s%d%s”, ps->name, &ps->phone, ps->addrs);
  printf(“%s, %d, %s\n”, s.name, s.phone, s.addrs);

}
```

Sample Program 10

prg10_10.c

```
#include <stdio.h>

struct student
{
  char name[40];
  int  phone;
  char addr[50];
};

int main()
{
  struct student s;
  struct student *ps;

  // Following statements will fill the structure variable ‘s’
  printf("Enter student’s name:\n”);
  gets(s.name);

  printf(“Enter student’s phone:\n”);
  scanf(“%d”, &s.phone);

  printf(“Enter student’s address:\n”);
  gets(s.addr);

  // Following statement will print the content of structure variables
```

```
  printf(“Name: %s, Phone: %d, Address: %s\n”, s.name,s.phone,s.addr);

  ps = &s;
  // printing same structure by using pointer
  printf(“Name: %s, Ph: %d, Addr: %s\n”, p->name, p->phone, p->addr);

  // Fill the structure using structure pointer
  printf("Enter student’s name:\n”);
  gets(p->name);

  printf(“Enter student’s phone:\n”);
  scanf(“%d”, &p->phone);

  printf(“Enter student’s address:\n”);
  gets(p->addr);
  printf(“Name: %s, Phone: %d, Address: %s\n”, s.name,s.phone,s.addr);
}
```

Applications of Pointers

Pointers have many applications, but there are two important uses (applications) for the pointers, these are:

- Passing references of variables to the functions
- Using Dynamic memory

We will discuss the application pointers in passing parameters to functions in the next chapter covering Functions. Now we study the application pointers in using dynamic memory.

Pointers to access Dynamic memory

So far we used pointers to access already existing variables. We are defining the pointer variables of appropriate type and storing the addresses of existing variables in these pointers. With the help of these pointers we could able to read or write to these existing variables. This is not the way we really use pointers. The main use of pointer is for accessing newly allocated memory during run time of program. We refer this new memory as dynamic memory or dynamically allocated memory. In this method, newly allocated memory address is kept in the pointer variables. Now by doing indirect addressing with the pointer we can read or write to this new memory.

For all the variables the memory is allocated during compile time. While program is running, if we need additional memory we can allocate additional memory by calling a library function *malloc()*. This function returns the address of newly allocated memory. We can store this address in a pointer variable and can write into this location and as well as read from this new memory location by using the pointer variable. Important thing here is that, we should allocate memory enough to hold the variable we are accessing. For example we want to access integer location, then the memory allocated should be 4 bytes. If we want access an array of 100 integers, then the allocated memory should be 100 x 4 i.e. 400 bytes.

prg10_11.c

```
#include <stdio.h>

int main()
{
  int *p;

  p = malloc(sizeof(int));

  printf(“Enter some number\n”);
  scanf(“%d”, p);
  printf(“Number stored in new memory is %d\n”, *p);
}
```

prg10_12.c

```
#include <stdio.h>

int main()
{
  int *pa, ii;

  p = malloc(sizeof(int) * 10);

  for(ii=0; ii< 10; ii++)
  {
    printf(“Enter number %d\n”, ii+1);
    scanf(“%d”, p+ii);
  }
  for(ii=0; ii< 10; ii++)
  {
    printf(“%d \n”, *(p+ii));
  }
}
```

prg10_13.c

```
#include <stdio.h>

struct student
{
  char name[40];
  int  phone;
  char addrs[50];
};

int main()
{
  struct student *ps;

  ps = malloc(sizeof (struct student));

  printf(“Enter name, phone and address\n”);
  scanf(“%s%d%s”, ps->name, &ps->phone, ps->addrs);
  printf(“%s, %d, %s\n”, ps->name, ps->phone, ps->addrs);

}
```

Pointer Arithmetic

When we define a pointer variable. It is not initialized. So pointer is not holding any valid address, so we cannot do indirect addressing (i.e accessing *p). So to keep valid address in pointer variable, we got two ways. One is to assign address of some other variable to the pointer. Second is to allocate new memory and get its address by using dynamic memory allocation function *malloc()*. Both are shown below:

```
int *p;
int a;

p  = &a;        // method one
*p = 10;       // 10 will get stored in a
p  = malloc(4); // method two
*p = 10;  // 10 will get stored in 4 bytes of new memory allocated
```

Once pointer variable is initialized, we can increment the pointer variable or we can add integer to this pointer variable. The following statement will illustrate that:

```
p  = malloc(100);
p++;
p = p + 5;
```

Let us assume that *malloc()* returned a valid address of 1000. So now p is filled with 1000 as indirect address. When we increment p (with p++), p is incremented to point to next location. As p points to integer location of 4 bytes (32 bits), it will get incremented to 1004. If p happens to be a pointer to char, it increments by size of char location, that is 1. Similarly if p is short pointer, it will increment by two to 1002. Similarly p = p +5; will increment p by 20 (assuming p is integer pointer).

The following sample program illustrates the concepts learned above. First you guess the output of following *printf()* statements, and then run this program and verify with your result.

prg10_14.c

```
#include <stdio.h>

int  *pi;
char *pc;
short *ps;
double *pd;

int main()
{
  pi = (int *) 1000;
  pc = (char *) 1000;
  ps = (short *) 1000;
  pd = (double *) 1000;

  printf(“%d, %d, %d, %d”\n, pc, ps, pi, pd);
  printf(“%d, %d, %d, %d”\n, pc+1, ps+1, pi+1, pd+1);
  printf(“%d, %d, %d, %d”\n, pc+5, ps+5, pi+5, pd+5);
```

```
}
```

```
Review Questions
----------------
01. What is the pointer variable?
02. What is meant by Direct address and Indirect address?
03. What is Direct Addressing and Indirect Addressing?
04. In C, array name acts like a pointer. Then what is the difference
    between array name pointer and true pointer?
05. What are the two different notations possible to access fileds of
    a structure by using structure pointer?
06. What are the two important uses of a pointer?
07. What is the library function to allocate dynamic memory?
08. How pointer arithmetic is different from integer arithmetic?

Assignments
-----------
 1.  Define one variable each of type char, short, int and float. Also
     define one pointer variable each of type char, short, int and float.
     Initialize char, short, int and float variables with different values.
     Initialize each pointer variable by storing the address of corresponding
     variable. Now print all the variables directly by using variable names.
     Next print the variables through the pointer variables. Next
     write different values into the variables through the pointers.
     Next display the variables. Next use scanf statement to read the
     values into variables. In the scanf() function you must use only
     pointer variables. Finally print the variables directly by using
     printf statement.

 2.  Define an array of 10 floats and also define a pointer to a float. Fill
     this array by taking 10 floats from user. Fill the pointer variable with
     the address of array. Now print the contents of array by using pointer
     variable. Next fill the array again by taking 10 float numbers from the
     user. In the scanf() statement use only pointer variable. Next print
     the 10 number by using array name.

 3.  Define a structure variable of type book(this 'struct book' is given in
     assignments of previous chapter). Also define a pointer variable
     to a book structure. Fill the book structure variable by reading data
     from the user. Fill the pointer with book structure variable address
     and print the contents using pointer variable. Next fill the structue
     again by taking data from the user, but this time use pointer variable.
     Finally print using book structrure variable.

 4   Define four pointer variables, one each of type char, short, int, float.
     Fill these pointers by allocate memory of required size by calling
     malloc() function. Read data from the user and fill in the memory
     just allocated using malloc(). Finally display the data

5.  Define an integer pointer variable. Ask you user how many integers she/he
     would like to store. Read that count from the user. Next allocate
     memory by using malloc() function of size count multiplied by 4.
     Store this allocated address in pointer variables. Now this pointer acts
     like dynamically allocated array. Read count number of integers from the
     user and fill this dynamic array. Finally print the integers present
     in dynamic array.

 6   Define a pointer variable to a 'struct book'. Call malloc() function
     with sizeof(struct book) and assign the return address to the pointer
     variable. Now fill the book structure by reading data from the user.
     Finally display the data.
```

7 **Define a pointer variable to a 'struct book'. Ask how many book structures user would like to enter. Read this count into a count variable. Next allocate memory equal to count multiplied by the sizeof book structure. Store this address in pointer variable. Now pointer variable acts like a dynamic array of book structures. Read book structures from the user and fill the dynamic array. Finally display all the book structures.**

8. **Define different types of pointers as shown below:**

```
char   *pc;
short  *ps;
int    *pi;
float  *pf;
double *pd;
struct book *pb;
```

Initialize all these pointer variables with a value of 100. Following is one example.

```
pb = (struct book *) 100;
```

After initializing all the pointers print the content of all these pointer variables. Next increment all these pointers by one and again print the content of pointers. Next add 9 to all the pointers and again print the content of pointers. Observe the results and understand the results.

11. Functions

All the skills you learned so far should be useful to you in developing the functions. Every useful software application we develop contains tens or hundreds or even thousands of functions. The functions required for the application are identified during the design phase of software development. The descriptions of all these required functions are described in the software design document, which is prepared during design phase. The programmers need to develop (i.e. write) these functions according to the given description in the design document. So as a budding programmer your job is to write and test functions according to the given description. This chapter just teaches that.

A function is a group of statements doing some specific job and having some name. A function may take some input and may return some output. To execute a function, we need to call the function from some other function. Only *main()* function is exception to this rule, which will get called automatically.

Following are the three important aspects of a function:

- Function Definition
- Function Declaration also called Function Prototype
- Function Invocation

Function Definition

Function definition is the one, where we define a function by writing function return type, function name, parameters to the function and finally the function body. Following is the function definition for *getAverage()* function.

```
int getAverage(int a, int b, int c)
{
  int total, avg;

  total = a + b + c;
  avg   = total / 3;
  return total;
}
```

The first line of function definition contains the return type of the function. For this function, return type is '*int*'. After return type, it contains the name of the function, i.e. '*getAverage*'. Next it contains parameters to the function. All the parameters are specified inside the parenthesis *( )*. Each parameter is specified by giving its type and name. This function is taking three parameters with names a, b, and c. All are integer parameters. The last and bigger part of a function definition is its body. The body of the function is enclosed with braces *{ }*. A function body contains a set of statements. Some statements are data definition statements, which are used to define local variables. Other statements are executable statements.

Variables defined inside a function are called local variables. In the above function, variables 'total' and 'avg' are local variables. These local variables can be used only inside the function.

Where as variable defined outside the function, are called global variables and all functions can use the global variables.

Important Note:

The parameters to a function are also can be considered as local variables. So these parameters variables can be used only within the function. But the main difference is that, the parameter variables are initialized by the calling function. The calling function passes some values to these parameters. So even before function is executed, these parameter variables will get initialized with those passed values. Where as the initial values of un-initialized local variables will be unknown.

Following is the example for some more function definitions:

```
int readNumber()
{
  int num;

  do
  {
    printf(“Enter some number between 0 and 100 : “);
    scanf(“%d”, &num);
  }while ((num <0) || (num > 100));

  return num;
}

void printHexNumber(int num)
{
  printf(“0x%x”, num);
}
```

If you compare the above two function definitions with *getAvarage()* function definition, you will observe the following differences:

The *readNumber()* function is not taking any parameters. But retuning ‘int’ similar to *getAverage()* functions. The *printHexNumber()* function is not returning any type. Such functions which are not returning any value are called void functions, and their return type is written as ‘void’. This *printHexNumber()* function is taking a single parameter of type ‘int’.

Function Declaration

Function declaration (also called function prototype) is a statement declaring the information about a function. This is exactly similar to the first line of function definition, but with a semicolon at the end. Note that function declaration will never contain function body. So following is the function declaration for the *getAverage()* function defined above. Note the semicolon at the end.

```
int getAvareage(int a, int b, int c);
```

We can also write function declarations without parameter names. The purpose of function declaration is to provide information about the function. This information is return type of the

function, number of parameters and type of each parameter. So names of parameters are not important or must. So following is also a valid declaration for the *getAverage()* function.

```
int getAvareage(int, int, int);
```

Function Invocation

To execute any function we need to call or invoke that function from some other function. The statement we use to call a function is called function invocation statement. The important thing while invoking (i.e. calling) a function is to pass the arguments properly. The arguments are the variables or constants we pass to the function in the place of parameters of the function. In the function definition we specify the parameter name and their types. While invoking the function we pass arguments in the place of each parameter. The calling of a function can be done in slightly different ways. Look at the following function invocations:

```
av = getAverage(10, 20, 30);
av = getAverage(v1, v2, v3);
av = getAverage(10, v1, v2);

result =  r1 + r2 * getAverage(v1, v2, v3);
printf(" Average = %d \n", getAverage(v1, v2, 30));

av = getAverage (readNumber(), readNumber(), readNumber());

printHexNumber(100);

printHexNumber(readNumber());

getAverage(v1, v2, v3);
```

Functions that return some value are called non-void functions. The *getAverage()* and *readNumber()* are non-void functions as both are returning integer values. Where as the *printHexNumber()* is a void function.

When we are invoking a non-void function, we usually save its return value by assigning it to some variable. This is shown in the first three statements above. In these three statements, the return value of *getAverage()* function is assigned to 'av' variable.

For some functions, even though it is returning some value, that value may not be important for the caller of that function. In such cases, we just call that function, but we ignore its return value. The last statement in the above, we are calling *getAverage()* function but ignoring its return value. That is we are not assigning return value to any variable.

While calling the *getAverage()* function we are passing some values as arguments. Note that these values will get copied to the parameter variables of the function definition. Look at the function definition of *getAverage()* function, it is using three parameter variables a, b, and c.
When the functions is invoked as *getAverage(10, 20, 30),* the 10, 20 and 30 will get copied to the a, b, and c of that function. Similarly when we call as *getAverage(v1, v2, v3),* the values of v1, v2 and v3 will get copied to a, b and c of the *getAverage()* function. The following invocation statement, which is not correct, but shows what is happening when with function invocation statement.

```
av = getAverage(a=v1, b=v2, c=12); // av = getAverage(a, b, 12);
```

Non-void functions can be used inside an expression just like any variable or constant. Here the function represents the value returned by the function. That value is used in the expression. This is illustrated in the 4th, 5th and 6th statements of above program.

Function declaration must be present before the function invocation statement. It is common to keep all the function declaration statements at the beginning of the file. For all the library functions, the function declarations are present in the header files such as stdio.h.

Classification of Functions

For the convenience of learning, we can classify the functions into the following types.

1. Functions that will not take any arguments and also returns none
2. Functions that will not take any arguments but returns some value
3. Functions that takes one or more arguments but returns none
4. Functions that takes one or more arguments and returns some value

Students should learn using (i.e. calling from their functions) these types of functions and as well as writing these types of functions.

Functions with No parameters no return values

These functions are simple to use and understand. But they may not be much useful as they are not flexible to use. These functions are called void functions because they are not returning any value.

In general, functions process input data and output the result through return value. So we typically pass the data through the parameters to the function. Function gives back the result through return value. So functions that will not take any parameters and will not return any value are not much useful as generic functions. However see the following examples of such functions.

prg11_1.c

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

void computeAreaOfCircle();

int main()
{
  printf("I am calling a function to compute area\n");
  computeAreaOfCircle();
  printf("I called area function\n");
}

void computeAreaOfCircle()
{
  float r;

  printf("Enter radius of circle\n");
  scanf("%f",&r);
```

```
  printf("Area of circle with radius %f is %f\n", rad, M_PI*r*r);
}
```

The above program contains two functions. One is *main()* function and second is *computeAreaOfCircle()* function. The return type of *computeAreaOfCircle()* is specified as *void*. The function with *void* type will not return any values. We kept the prototype of the function in the beginning. We are calling the function without passing any arguments and without using its return value as this function does not take any argument and does not return any value. In the above program we kept *main()* as first function and *computeAreaOfCircle()* as second function. However we can keep any function at an position, there are no such restrictions.

Functions which take value arguments but returns none (void)

These type of functions, which takes arguments but returns none, are slightly more useful than previous functions. One common application of these functions is to output the given arguments to some IO device (such as Display monitor) or a file. These functions may output data after doing some processing or without any processing.

prg11_2.c

```
#include <stdio.h>

void printNumber(int a);

void printNumber(int a)
{
  printf("I am printing given number in octal, decimal and Hex\n");
  printf("0%d, %d, 0x%x\n", a, a, a);
}

int main()
{
  int x;

  printf("Enter a number\n");
  scanf("%d", &x);
  printNumber(x);
  printf("I called simple to print number\n");
}
```

In the above program, *printNumber()* is taking a single parameter and outputting that number to display in three formats. This function is a void function, as it is not returning any value. While calling this function from the *main()*, x is passed as parameter. That means ‘x’ will get copied to parameter ‘a’ of *printNumber()*. In the above program we kept *main()* function as a second function. Wherever we keep *main()*, it only will get executed first.

prg11_3.c

```
#include <stdio.h>

struct student
{
  int    id;
  char   name[40];
  int    phone;
};

void printStudentInfo(struct student s);

int main()
{
  struct student st;

  printf("Enter Student id, Name and Phone number\n");
  scanf("%d%s%d", &st.id, st.name, &st.phone);
  printStudentInfo(st);
}

void printStudentInfo(struct student s)
{
  printf("Student Id = %d\n", s.id);
  printf("Student name = %s\n", s.name);
  printf("Student phone = %d\n", s.phone);
}
```

In the above program, *printStudentInfo()* function is taking structure variable as parameter. So note that, we can pass structure variables to a function just like integer or float variables. When main function is calling the *printStudentInfo()* function, it is passing 'st' structure variable as parameter. Note that this 'st' will get copied into the 's' parameter of *printStudentInfo()* function.

Functions with no arguments but returns some

This type of functions, which takes no arguments but returns some value, are also slightly useful similar to previous functions. One common application of these functions is to read data from input device or file and return them after some validation or processing.

prg11_4.c

```
#include <stdio.h>

int getAge();

int main()
{
  int a;

  printf("I am going to get age by calling getAge function\n");
  a = getAge();
  printf("The age returned by getAge function is %d\n", a);
}

int getAge()
{
```

```
  int age;

  while(1)
  {
    printf("Enter your Age in years\n);
    scanf("%d", &age);
    if((age < 2) || (age > 150))
    {
      printf("Please enter your age correctly\n");
    }
    else
      return age;
  }
}
```

In the above program, *getAge()* is not taking any parameter but returning an integer. This function is reading age from the input device (keyboard) and validating and finally returning if age appears to be a valid one.

Any function that is returning some value must use 'return' statement to return value. In the *getAge()* function, the return statement is returning 'age' variable. The 'return' statement is also used in void functions to return from the middle of a function.

prg11_5.c

```
#include <stdio.h>

struct student
{
  int    id;
  char   name[40];
  int    phone;
};

struct student getStudentInfo();

int main()
{
  struct student st;

  st = getStudentInfo();
  printf("Student Id = %d\n", st.id);
  printf("Student name = %s\n", st.name);
  printf("Student phone = %d\n", st.phone);
}

struct student getStudentInfo()
{
  struct student s;

  printf("Enter Student id, Name and Phone number\n");
  scanf("%d%s%d", &s.id, s.name, &s.phone);
  return s;
}
```

The above program shows the function definition, declaration and invocation of function *getStudentInfo()*. This function is not taking any arguments, but it is returning a structure. So note that, a function can return a structure variable just like how returns a simple integer or float variable. In the function invocation we are assigning the return value to a structure variable.

Functions with arguments and return values

These are the most common types of functions. A lot of library functions we use, and a lot of functions we need to write are of this kind of functions. The main purpose of every function is to process the data it received through the parameters, and return the output through return statement.

As there are so many functions of this type, let us divide these into different types based on the types of arguments:

- Functions that take simple types such as char, short, int, float types of parameters
- Functions that take structure variables, may be along with other arguments
- Functions that take pointers as arguments, may be along with other types

In the same way, functions may return a simple type variable, or a structure or a pointer.

Functions that take simple types of parameters

prg11_6.c

```
#include <stdio.h>

int getBiggestNumber(int a, int b, int c);

int main()
{
  int x, y, z;
  int biggest;

  printf(“Enter three numbers\n”);
  printf(“Enter character whose count to be found in above line\n”);
  scanf(“%d%d%d”,&x, &y, &z);
  biggest = getBiggestNumber(x, y, z);
  printf(“The biggest of three is %d\n”, biggest;
}

int getBiggestNumber(int a, int b, int c)
{
  if( (a > b) && (a > c))
    return a;
  if( (b > c) && (b > a))
    return b;
  else
    return c;
}
```

Functions that take pointer variables as parameters

A lot of functions take one or more pointer parameters, optionally along with other types of parameters. A function requires pointer variables as parameters for the following purposes:

- When a function wants to receive an array or string (string is nothing but an array of characters) as parameter, it uses a pointer parameter, into which it will receive the address of array. Now the function can use this pointer just like an array.
- When a function wants receive structure variable as parameter, it can use structure variable itself as parameter. But passing structure variables to function is inefficient. So function can take address of a structure variable as parameter. In this case the parameter should be a pointer to a structure.
- One most common reason for a function to take pointer parameter is to modify or write to the variable of calling function. In this case the calling function will pass the address of its variable to this function. This function receives the address through a pointer parameter. Now by using this pointer parameter, this function can write to the variable of the calling function. The best example for this kind of function is *scanf()* function. Remember that we always pass addresses of our variables (&a, &b) to the *scanf()* function.

Passing addresses

When a function definition contains a pointer (or array) as parameter, while calling that function we must ensure that we are passing valid address for that parameter. We can pass addresses in various ways. One way is to pass the address of our variable by using address operator &. Second possibility is to pass the array or string, as array or string represents the address. Third method is to pass a pointer variable, which is already initialized with a valid address.

const keyword

When a function receives a pointer as parameter, the function can use this pointer to write to the address present in the pointer, or to read from the address present in the pointer. So when a function takes pointer (address), we want to be sure that, whether it is taking address fro reading purpose or writing purpose. This can specified by using a ‘const’ keyword in font of a pointer parameter in the function definition.

```
int getAverageMark(const int *marks, int count);
```

In the above function declaration, the ‘const’ keyword indicates that, the marks pointer can be used only for reading the marks array, but not for writing into it. If const keyword is not present in front of the pointer, then pointer can be used for both reading and writing to that address.

prg11_7.c

```
#include <stdio.h>

int getAverageMark(const int *marks, int count);
int getVowelCount(const char *str);

int main()
{
  int stu_marks[10];
```

```
  char mystring[] = "This is my sample string\n";
  char *mystring2 = "This is my second string\n");
  int ii;
  int avgmark;
  int vowels;

  for(ii=0; ii<10; ii++)
  {
    printf("Enter marks of student %d\n", ii+1);
    scanf("%d", &stu_marks[ii]);
    // Following is another way of writing above scanf
    // scanf("%d", stu_marks+ii);
  }
  avgmark = getAverageMark(stu_marks, 10);
  printf("Average mark is %d\n", avgmark);
  vowels  = getVowelCount(mystring);
  printf("Number of vowels in first string are %d\n", vowels);
  printf("Number of vowels in second string are %d\n",
                                             getVowelCount(mystring2);
}

int getAvarageMark(const int *marks, int count)
{
  int ii, total

  total = 0;
  for(ii=0; ii<count; ii++)
  {
    total = total + marks[ii];
    // the above statement can also be written as below:
    // total = total + *(marks+ii);
  }
  return (total/count);
}

int getVowelCount(const char *str)
{
  int cnt = 0;

  while(*str)
  {
    switch(*str)
    {
      case 'a': case 'A':
      case 'e': case 'E':
      case 'i': case 'I':
      case 'o': case 'O':
      case 'u': case 'U';
        cnt++;
    }
    str++;
  }
  return cnt;
}
```

In the above program, we defined and used two functions. These are *getAverageMark()* and *getVowelCount()* both are taking pointer parameters and both are returning integer. The first function is taking pointer to receive the address of marks array. It is also taking second parameter 'count'. As the first parameter gives the address only, the second parameter is required to know how many marks are present in the array. That is the size of the array.

The second function is taking only address of the character array, i.e. string. A string has got a special property that; the last character in the string is always a NULL character. So there is no need to specify the count of characters present in the string. So this function takes only single parameter.

prg11_8.c

```
#include <stdio.h>

struct student
{
  int    id;
  char   name[40];
  int    phone;
};

void printStudentInfo(const struct student *ps);

int main()
{
  struct student st;

  printf("Enter Student id, Name and Phone number\n");
  scanf("%d%s%d", &st.id, st.name, &st.phone);
  printStudentInfo(&st);
}

void printStudentInfo(const struct student *ps)
{
  printf("Student Id = %d\n", ps->id);
  printf("Student name = %s\n", ps->name);
  printf("Student phone = %d\n", ps->phone);
}
```

In the above program, we defined a function that takes structure pointer as parameter. Using the structure pointer, this function prints the fields of the structure. We already wrote similar function, that takes same student structure as parameter. The difference is that one takes structure variable as parameter, another takes pointer to structure as parameter. There is an advantage in passing structure pointer, instead of structure variable. When we pass structure variable, entire structure is get copied to the structure parameter. If structure size is huge, it may inefficient to pass structure variable, as it takes more time to copy. Where as when we pass pointer to a structure, only pointer (4 bytes) will get copied to pointer parameter. Using this pointer the function can print the fields of structure.

One of the primary purposes of pointer parameter is to allow the *called* function to write in to the variables of *calling* function. Note that the *called* function is the one, which we are calling using function invocation statement. The *calling* function is the one, which contains the function invocation statement. In all the above programs, *main()* function is the calling function.

So when a calling function wants its variables (like char, short, int, float or any structure variables) to be written by the called function, then calling function must pass the addresses of its variables. The called function receives their addresses through the pointer parameters. Using these pointers the called function can write to the variables of calling function.

Simple variables (like char, short, int, float and double) and any type of structure variables can be passed either by value or by address. When we pass by value (that is by copying into the parameter variables of called function) the called function cannot write into calling function variables. But if we pass the address, then called function can read or write to variables of calling function.

However array variables are always passed by address only, it is not possible to pass arrays by value. Once we pass address of array, the called function can read or write the array elements.

prg11_9.c

```
#include <stdio.h>

struct stuRecord
{
  int    class;   // 0 is fail, 1 is First class, 2 second, 3 ordinary
  int    totalMarks;
  int    average
};

void getMarksOfStudent(int *marks);
void getStuRecord(const int *marks, struct stuRecord *psr);
void getStuInfo(const int *marks, int *class, int *total, int *avg);

int main()
{
  int ii;
  struct stuRecord sr;
  int    stumarks[5];
  int    stuClass, stuTotal, stuAvg;

  getMarksOfStudent(stumarks);
  printf(“Marks in Telugu, English, Maths, Science & Social: “);
  for(ii=0; ii<5; ii++)
    printf(“%d “, stumarks[ii]);

  getStuRecord(stumarks, &sr);
  printf(“class: %d, total: %d, Avg: %d\n”,sr->class,sr->totalMarks,
                                           sr->average);
  getStuInfo(stumarks, &stuClass, &stuTotal, &stuAvg);
  printf(“class: %d, total: %d, Avg: %d\n”, stuClass,stuTotal,stuAvg);
}

void getMarksOfStudent(int *marks)
{
  int ii;

  printf(“Enter marks in Telugu, English, Mathes, Science & Social\n”);
  for(ii=0; ii<5; ii++)
```

```
    scanf("%d", marks+ii);
}

void getStuRecord(const int *marks, struct stuRecord *psr)
{
  int ii, minmarks;

  psr->totalMarks = 0;
  minmarks = marks[0];

  for(ii=0; ii<5; ii++)
  {
    psr->totalMarks = psr->totalMarks + marks[ii];
    if(minmarks > marks[ii])
      minmarks = marks[ii];
  }
  psr->average = psr->totalMarks / 5;
  if(minmarks < 35)
    psr->class = 0;
  else if(psr->average >= 60)
    psr->class = 1;
  else if(psr->average >= 50)
    psr->class = 2;
  else
    psr->class = 3;
}

void getStuInfo(const int *marks, int *class, int *total, int *avg)
{
  int ii, t, a, c, min;

  t = 0;
  min = marks[0];

  for(ii=0; ii<5; ii++)
  {
    t = t + marks[ii];
    if(min > marks[ii])
      min = marks[ii];
  }
  a = t / 5;
  if(min < 35)
    *class = 0;
  else if(a >= 60)
    *class = 1;
  else if(a >= 50)
    *class = 2;
  else
    *class = 3;

  *total = t;
  *avg   = a;
}
```

In the above program we defined three functions. These are *getMarksOfStudent*(), *getStuRecord*() and *getStuInfo*().

The *getMarksOfStudent*() is taking integer pointer as parameter. The *main*() function while calling this function, passing the address of its array '*stumarks*'. Using the pointer the *getMarksOfStudent*() function is filling the given array. Once this function returns, the '*stumarks*' array is in filled state. The main function is displaying the contents of this array to prove this point.

The *getStuRecord*() function is taking two pointer parameters, one is integer pointer and other is pointer to a structure. While calling this function from *main*(), *main*() function is passing address of the array '*stumarks*' and address of its structure '*sr*'. This function by using array pointer it is reading the marks and by using structure pointer it is filling the fields of the structure. In this way *main*() function's structure variable '*sr*' is written by the *getStuRecord*() function. After calling this function, *main*() is displaying the contents of '*sr*'.

The *getStuInfo*() function is taking total four pointer parameters. All the four pointers are integer pointers only. However we are assuming that the first pointer is the starting address of array of integers. So using this single pointer, we can access all the successive elements present at that address. Where as remaining three pointers are assumed as addresses of individual integer variables. The *main()* function is passing address of array and addresses of its three variables '*stuClass*', '*stuTotal*' and '*stuAvg*'. This *getStuInfo*() function by using these addresses it is placing the results. After calling this function, *main*() function is displaying the values of these variables, which are filled by the *getStuInfo*() function.

The main() function's parameter

Now is the time to look at the parameters to the *main()* function. So far we have written *main()* function without specifying parameters. But now learn that, the *main()* function receives two parameters. Following is the function prototype or declaration for the *main()* function.

```
int main(int argc, char **argv);
```

The second parameter may look little bit confusing. But what we are receiving through second parameter is an array of character pointers. Each character pointer in this array will point to one string. When we receive an array of characters, we define parameter as '`char *p`', because `*p` represents element of array, which is a *char* variable. But here the element of array is not a *char*; it is a *char pointer*. So parameter looks like '`char * *argv`'. That is `*argv` is a *char pointer*. The first parameter *argc* gives the number of elements in the *argv* array. That is size of *argv* array.

We run our program, by typing the program name at shell prompt. We can always type some strings (arguments) after the program name. All these strings along with program name are collected as an array of strings and this array is passed as second parameter to the *main()* function. The first parameter to the *main() function* is the number of strings in that array.

In the following program the main() function prints all the strings it is receives through the *argv* array.

prg11_10.c

```
#include <stdio.h>

int main(int argc, char **argv)
{
  int ii;

  printf("The number of strings in argv array are %d\n", argc);
  printf("Following are the strings\n");
  for(ii=0; ii<argc; ii++)
    printf("string %d is %s\n", ii+1, argv[ii]);
}
```

Run the above program by giving command line arguments differently as shown below and observe the output of the program each time.

```
$ a.out
$ a.out  hello How are you
$ a.out  123    567  hello   abc  890
```

The following program takes two command line arguments from the user. If user did not enter the command line arguments, the program reports error and returns. Next it uses the getVowelCount() function given in *prg11_7.c* to compute the vowels in the argument. Finally it prints this vowels count.

prg11_10.c

```
#include <stdio.h>

int main(int argc, char **argv)
{
  int ii;

  if(argc != 3)
  {
    printf("Error: You must enter two arguments after command name\n");
    return 1;
  }
  printf("Number of vowels in first argument are %d\n",
                                        getVowelCount(argv[1]));

  printf("Number of vowels in secnd argument are %d\n",
                                        getVowelCount(argv[2]));
}
```

Review Questions

```
01. Distinguish between Function definition, Function declaration and
    Function Invocation.
02. In a function declaration the parameter names are must. (True/False)
03. Classify the functions based on return values and parameters.
04. What are the reasons or uses for a function to take pointer parameter?
05. When a function takes a pointer as parameter, sometimes this pointer
    parameter is defined with 'const' keyword. What this 'const' key will
    do?
```

Assignments

```
i)  Each program should contain only two functions. The first one should be
    the function described in the problem. The second function should be
    the main() function calling the function written above.

1.   Write the following programs. Each program should contain the following
     function and a main() function calling the above function. All these
     functions does not take parameters and does not return value.

1.1  void printBigOfThree();

     This function reads three numbers into three integer local variables
     and prints the biggest of three.

1.2  void printAreaOfTriangle();

     This function reads base and height of a trinagle as float numbers and
     prints the area.

1.3  void printVolumeOfCuboid();

     This function reads length, breadth and height of a cuboid from the user
     into three float variables. Next computes the volume and prints it.

1.4  void printSurfaceAreaOfCuboid();

     This function reads length, breadth and height of a cuboid from the user
     into three float variables. Next computes the surface area and prints it.


1.5  void printAreaOfCircle();

     This function reads the radius of a circle from the user into a float
     variable. Next it computes the area and prints it. Include math.h file
     and you can use M_PI to represent the value of PI.

1.6  void printInterest();

     This function reads the principle, time in years and rate of interest for
     year. Take all the three in float variables. It computes the interest and
     prints it.


2.  Modify all the above functions such that, instead of printing the result
    the functions should return the result. Following are the new
    prototypes and names for the above functions. Modify the main()
    function such that it prints the return value returned by the function.

2.1  int   getBigOfThree();
2.2  float getAreaOfTriangle();
2.3  float getVolumentOfCuboid();
```

```
2.4  float getSurfaceAreaOfCuboid();
2.5  float getAreaOfCircle();
2.6  float getInterest();
```

3. Modify all the above functions such that they take input from the parameters and computes the result and prints the result. Following are the new prototypes for the above functions. Modify the main() function such that it reads input from the user, passes these inputs to the function.

```
3.1  void printBigOfThree(int a, int b, int c);
3.2  void printAreaOfTriangle( float base, float height);
3.3  void printVolumentOfCuboid(float l, float b, float h);
3.4  void printSurfaceAreaOfCuboid( float l, float b, float h);
3.5  void printAreaOfCircle(float r);
3.6  void printInterest(float p, float r, float t);
```

4. Modify all the above functions such that they take input from the parameters and computes the result and return the result. Following are the new prototypes for the above functions. Modify the main() function such that it reads input from the user, passes these inputs to the function and finally prints the output returned by the function.

```
4.1  int   getBigOfThree(int a, int b, int c);
4.2  float getAreaOfTriangle( float base, float height);
4.3  float getVolumentOfCuboid(float l, float b, float h);
4.4  float getSurfaceAreaOfCuboid( float l, float b, float h);
4.5  float getAreaOfCircle(float r);
4.6  float getInterest(float p, float r, float t);
```

5. The following programs are already learned or written by you. Here you have to implement then as functions. And add a main() function to call that function.

5.1 `int countBitOnes(unsigned int val);`

This function counts the number of 1s in total 32 bits and returns.

5.2 `unsigned int rotateLeft(unsigned int val);`

This function rotates the given number left by one bit and returns it.

5.3 `unsigned int rotateRight(unsigned int val);`

This function rotates the given number right by one bit and returns it.

5.4 `void printBinary(int val);`

This function prints the given number in binary format.

6. All the following functions takes pointer as parameter. Through the pointer these functions are getting the address of arrays. When we receive address, we can read from that address or write to that address. When address is given only for reading, the pointer is prefixed with 'const' keyword.

6.1 `int getVowelCount(const char *str);`

This function counts the number of vowels in the string (character array) and returns that.

```
6.2  int getWordCount(const char *str);

     This function counts the number of words in the given string
     (character array) and returns that.

6.3  int getStringLen(const char *str);

     This function counts the number of characters in the given string
     and returns it.

6.4  int getCapitalLetterCount(const char *str);

     This function counts the number of capital letters in the given string
     and returns it.

6.5  int getMaxMark(int *marks, int noOfStudents);

     This function finds the maximum mark in the given array of marks
     and returns it. The 'noOfStudents' gives the number of elements in
     marks array. The main() function defines an array of marks of certain
     size. Fills the marks array by taking marks from the user. Next it
     calls the above function by passing marks array and its size. Finally
     main() function prints the value returned by this function.

6.6  int getAverageMark(const int *marks, int noOfStudents);

     This function finds the average mark and returns it.
     The main() function of this program should be similar to above program's
     main() function.

6.7  int getNoOfStudentsWhoGotMark(const int *marks,int noOfStudents, int mark);

     This function finds the number of students who got the given mark and
     returns it.

6.8  int getStudentPassStatus(const int *marks);

     This function assumes that the size of marks array is 6. It returns
     0 for Fail, 1 for Pass, 2 for Second class and 3 for First class.
     Pass mark is considered as 35.

7.   All these functions also receiving array address through pointer
     parameters. But these functions are modifying the given array. Because
     of this reason the 'const' keyword is not present here.

7.1  void convertToUpper(char *str);

     This function converts the small letters present in the string to
     the upper letters. The main() function reads a string from the user
     and calls this function by passing that string. Finally it prints the
     string.

7.2  void convertToLower(char *str);

     This function converts the capital letters present in the string to
     the lower case letters. The main() function reads a string from the user
     and calls this function by passing that string. Finally it prints the
     string.

7.3  void reverseString(char *str);

     This function reverses the given string. The main() function reads a
     string from the user and calls this function by passing that string.
```

Finally it prints the string.

7.4 void addGraceMarks(int *marks);

This function assumes that the 'marks' is an array of 6. This function adds grace marks to the failed subjects to make it 35. Only 5 grace marks are given. These grace marks can be added to any number of subjects as long as total grace marks are less than or equal to 5.
The main() function defines an array of 6 integers, fills that array by taking marks from the user. Finally it prints the marks array.

8.1 void convertToUpper(const char *srcStr, char *dstStr);

This function is similar to above function(7.1), but instead of modifying the original string, while copying to dstStr it modifies small letter to upper case letters. The main() function should define two character arrays. Read a string from the user into one character array and call the above function. Finally print the second string.

8.2 void convertToLower(const char *srcStr, char *dstStr);

This function is same as above function and copies the converted string into destination string.

8.3 void reverseString(const char *srcStr, char *dstStr);

This function reverses the string by copying it into the second string.

8.4 void addGraceMarks(const int *srcMarks, char *dstMarks);

The marks after adding the grace marks are copied to 'dstMarks'.

8.5 void stringCopy(const int *srcStr, char *dstStr);

This function copies the srcStr to the dstStr. The main() function defines two character arrays of size 100 each. Next it reads a string from the user and calls this function. Finally main() function prints the second string.

8.6 void stringAppend(const int *srcStr, char *dstStr);

This function appends the srcStr to the end of dstStr. The main() function defines two character arrays of size 100 each. Next it reads a string from the user into first string and calls this function. Finally main() function prints the second string.

```
struct stuMarks
{
  int  english;
  int  hindi;
  int  telugu;
  int  mathes;
  int  science;
  int  social;
};

struct stuStatus
{
  int  passStatus;
```

```
  int  totalMarks;
  int  avgMark;
  int  higestMark;
  int  lowestMark;
}

9.1  void displayStudentMarks(const struct stuMarks *psm);

     This function displays the marks of 6 subjects present in the
     'struct stuMarks'. The main() should define the structure variable
     fill it by reading data from the user and should call this function
     to display it.

9.2  void fillStudentMarks(struct stuMarks *psm);

     This function reads the 6 subject marks from the user and fills
     the structure. The main() function should define a structure
     variable and should call this function to get it filled. Finally
     main() function should display the contents.

9.3  void displayStudentMarks(struct stuMarks sm);

     This is same as 9.1. Only difference is that it taking structure
     as parameter instead of pointer to structure.

9.4  struct stuStatus  getStudentStatus(const struct stuMarks *psm);

     This function returns the student status structure. This function
     defines a local structure variable of type 'struct stuStatus'. Next
     it fills this structure by computing the values from the input structure
     '*psm'. Finally it returns this structure variable.

9.5  void getStudentStatus(const struct stuMarks *psm, struct stuStatus *pss);

     This is same as above function, but it fills the *pss structure by
     computing status from the values available in *psm.

9.6  void getStudentStatus(struct stuMarks sm, struct stuStatus *pss);

     This function is same as above function, only difference is that, it
     receives stuMarks structure instead of pointer.

      Important Note:
        all the above three functions are doing the same job.
        That is giving back the 'stuStatus' structure to the
        calling function. But the same thing is done with
        different interfaces. Similarly the problems given in
        1.x, 2.x, 3.x, 4.x and the following 10.x are all
        identical functions. But same functions are written
        with different interfaces. So it is important to learn
        how to call these functions from the main().

10.  You have already written these functions many times and really got bored
     with them. But i can't help. In the following functions the return value
     is returned through the output parameters. The calling function passes
     address (i.e. reference), this function keeps the result at that address.
     Write a main() function to call these functions in an appropriate manner.

10.1  void  getBigOfThree(int a, int b, int c, int *bignum);
10.2  void  getAreaOfTriangle( float base, float height, float *area);
10.3  void  getVolumentOfCuboid(float l, float b, float h, float *vol);
10.4  void  getSurfaceAreaOfCuboid( float l, float b, float h, float *sa);
10.5  void  getAreaOfCircle(float r, float *area);
```

```
10.6  void  getInterest(float p, float r, float t, float *interest);
10.7  void  getSurfaceAreaAndVolumeOfCuboid( float l, float b, float h,
                                           float *sa, float *vol);
```

12. Library Functions

So far we have used only two library functions, printf() and scanf(). But there exist so many library functions. All these functions are referred as 'Standard C library' functions. Every C programmer should get familiar with these functions. If you know the name of any library function you can get all the information about that function by using *man* command as shown below:

```
$ man strncpy
```

The man command stands for manual page. It displays the manual page associated with the given library function. This manual page of that function will have sufficient information about the function. This information describes the parameters to the function, return value of that function and names of any other related functions.

By reading the manual page of a function, you should be able to use that function from your program. The main purpose of this chapter is to make you aware of so many library functions already available, and to show you, how to use those functions from the programs. We will show you how to use some functions, you should be able to try the remaining functions on your own.

Once you understood and use some library function, then it will be good idea to write that function your self. This will improve your programming skills significantly and make you a very good programmer. Especially most of the string functions described in section 12.4 are suitable for writing on your own.

12.1 Character Type Functions

Following are very simple library functions to use and write. Each of the following function takes one character as input argument and returns TRUE (non zero) or FALSE (zero) indicating whether it belongs that class or not.

```
int isalnum(int c); // returns TRUE if c is any alpha numeric character
int isalpha(int c); // returns TRUE if c is alpha char i.e a-z or A-Z
int iscntrl(int c); // TRUE if c is control character i.e < 32
int isdigit(int c); // True if c is a numeric digit
int isgraph(int c); //
int islower(int c);
int isupper(int c);
int isprint(int c);
int ispunct(int c);
int isspace(int c);
int isxdigit(int c);
int toupper(int c);
int tolower(int c);
```

Following is the sample program that shows the usage of library function *isupper()* as well as our own implementation of same library function *my_isupper().*

```
#include <stdio.h>

int my_isupper(int c);
```

```
int main()
{
  int ch;
  char ch1;

  printf("Enter two characters and then press 'Enter' key\n");
  ch = getchar();
  ch1 = getchar();

  if(isupper(ch))
    printf("%c is upper case character\n");
  if(my_isupper(ch1))
    printf("%c is upper case character\n");
}

int my_isupper(int c)
{
  if ((c >= 'A') && (c <= 'Z'))
    return 1;
  else
    return 0;
}
```

12.2. IO and Format IO Functions

You already used *printf()* and *scanf()*. Develop programs to use other functions. Observe the usage of 'const' keyword. All the parameters with 'const' are pointer parameters. The 'const' keyword says that the pointer should be used only to read the contents pointed pointer. You should not write in those locations.

```
#include <stdio.h>

int putchar(int c);
int puts(const char *s)
int printf(const char *format, ..);
int sprintf(char *str, const char *format, ..);

int getchar(void)
char *gets(char *gets)
int scanf(const char *format, ...);
int sscanf(const char *str, const char *format, ...);
```

12.3. File Access Functions

```
#include <stdio.h>

FILE *fopen(const char *filename, const char *mode);
int fprintf(FILE *stream, const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int fgetc(FILE *stream);
char *fgets(char *s, int n, FILE *stream);
int fputc(int c, FILE *stream);
int fputs(const char *s, FILE *stream);
```

```
int fclose(FILE *stream)
```

12.4. String Library Functions

These string functions are the most widely used functions. And also best function to write on your own, because they neither simple nor complex.

```
#include <string.h>

char   *strcat(char *dest, const char *src);
char   *strchr(const char *s, int c);
int    strcmp(const char *s1, const char *s2);
char   *strcpy(char *dest, const char *src);
size_t strcspn(const char *s, const char *reject);
size_t strlen(const char *s);
char   *strncat(char *dest, const char *src, size_t n);
int    strncmp(const char *s1, const char *s2, size_t n);
char   *strncpy(char *dest, const char *src, size_t n);
char   *strpbrk(const *s, const char *accept);
char   *strrchr(const char *s, int c);
size_t strspn(const char *s, const char *accept);
char   *strstr(const char *haystack, const char *needle);
char   *strtok(char *s, const char *delim);

void *memchr(const void *s, int c, size_t n);
void *memrchr(const void *s, int c, size_t n);
int  memcmp(const void *s1, const void *s2, size_t n);
void *memcpy(void *dest, const void *src, size_t n);
void *memmove(void *dest, const void *src, size_t n);
void *memset(void *s, int c, size_t n);
```

12.5. Date and Time Library Functions

Learn how to use these functions. These are interesting functions. You may not need to write these functions.

```
#include <time.h>

time_t     time(time_t *t);
char       *asctime(const struct tm *tm);
char       *ctime(const time_t *timep);
struct tm  *gmtime(const time_t *timep);
struct tm  *localtime(const time_t *timep);
time_t     mktime(struct tm *tm);
```

12.6. Memory Allocation and Free functions

The following are the very important functions to learn. You need not write these function on your own. But you must master the usage of these functions.

```
#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);
```

12.7. Math Library Functions

Write the program to use the following mathematical functions. You need not write them.

```
double exp(double x);
double log(double x);
double log10(double x);
double pow(double x, double y);
double sqrt(double x);

double floor(double x);
double modf(double x, double *iptr);
double fmod(double x, double y);

double sin(double x);
double cos(double x);
double tan(double x);

double asin(double x);
double acos(double x);
double atan(double x);
```

Appendix-B ASCII codes

Dec	Hex	Char	Dec	Hex	Char
0	00	NUL '\0'	64	40	@
1	01	SOH (start of heading)	65	41	A
2	02	STX (start of text)	66	42	B
3	03	ETX (end of text)	67	43	C
4	04	EOT (end of transmission)	68	44	D
5	05	ENQ (enquiry)	69	45	E
6	06	ACK (acknowledge)	70	46	F
7	07	BEL '\a' (bell)	71	47	G
8	08	BS '\b' (backspace)	72	48	H
9	09	HT '\t' (horizontal tab)	73	49	I
10	0A	LF '\n' (new line)	74	4A	J
11	0B	VT '\v' (vertical tab)	75	4B	K
12	0C	FF '\f' (form feed)	76	4C	L
13	0D	CR '\r' (carriage ret)	77	4D	M
14	0E	SO (shift out)	78	4E	N
15	0F	SI (shift in)	79	4F	O
16	10	DLE (data link escape)	80	50	P
17	11	DC1 (device control 1)	81	51	Q
18	12	DC2 (device control 2)	82	52	R
19	13	DC3 (device control 3)	83	53	S
20	14	DC4 (device control 4)	84	54	T
21	15	NAK (negative ack.)	85	55	U
22	16	SYN (synchronous idle)	86	56	V
23	17	ETB (end of trans. blk)	87	57	W
24	18	CAN (cancel)	88	58	X
25	19	EM (end of medium)	89	59	Y
26	1A	SUB (substitute)	90	5A	Z
27	1B	ESC (escape)	91	5B	[
28	1C	FS (file separator)	92	5C	\
29	1D	GS (group separator)	93	5D	]
30	1E	RS (record separator)	94	5E	^
31	1F	US (unit separator)	95	5F	_
32	20	SPACE	96	60	`
33	21	!	97	61	a
34	22	"	98	62	b
35	23	#	99	63	c
36	24	$	100	64	d
37	25	%	101	65	e
38	26	&	102	66	f
39	27	´	103	67	g
40	28	(	104	68	h
41	29	)	105	69	i
42	2A	*	106	6A	j
43	2B	+	107	6B	k
44	2C	,	108	6C	l
45	2D	-	109	6D	m
46	2E	.	110	6E	n
47	2F	/	111	6F	o
48	30	0	112	70	p

49	31	1	113	71	q
50	32	2	114	72	r
51	33	3	115	73	s
52	34	4	116	74	t
53	35	5	117	75	u
54	36	6	118	76	v
55	37	7	119	77	w
56	38	8	120	78	x
57	39	9	121	79	y
58	3A	:	122	7A	z
59	3B	;	123	7B	{
60	3C	<	124	7C	|
61	3D	=	125	7D	}
62	3E	>	126	7E	~
63	3F	?	127	7F	DEL