**WeareEngineers --- Training on Embedded Systems**

**http://www.weareengineers.in/**

**Chapter 1 – Introduction to Linux and Basic Commands**

* 1. **Linux Basic Commands**

Linux operating system has a beautiful graphical interface which most of us will be using. It will be good to learn the basic commands in Linux to work interactively with the Linux operating system. Linux has a back end access known as shell. You can control and activate all the process in Linux from the shell. So it is very important to learn few basic commands to work with Linux operating system.

First we will learn how to login for shell access. There are 7 terminals for Linux. 6 terminals are non GUI and 1 terminal for GUI access. You can login to each terminal using Alt + Ctrl + F1, F2 ..F7. Each terminals will request your username and password for login. If you want to use the shell in the graphical interface (GUI), press Alt + F2 and type “konsole”. As a user you will have permission to access only your/home/user directory and other directories in it.

\*Note: use denotes the username. These terminal access commands vary one Linux variant to other Linx variant.

**1.2 Basic Commands in Linux**

**man -** format and display the on-line manual pages of Linux commands

**1.2.1 Navigation Commands - [pwd,cd]**

**pwd**

This command is used to find the current location or current working directory.

**Eg:**

[user@ws26~]$ pwd

/home/user

**cd**

This command id used to change the directory. You can move from one directory to another using this command. Few examples are given below. Consider you have a directory structure /home/user/test/test1/ .test and test1 are directories in user home.

**Example 1:**

Consider you have a directory “test” in /home/user. Your current working directory is /home/user. You want to change your current working directory from /home/user to/home/user/test, use the following command

[user@ws26~]$ cd test

[user@ws26~]$ pwd

/home/user/test

**Example 2:**

If you want to move back to /home/user, use the following command

[user@ws26~]$ cd ..

[user@ws26~]$ pwd

/home/user

**Example 3:**

To get back to the home directory of the user

{user@ws26~]$cd ~

[user@ws26 ~]$ pwd

/home/user.

To go back to previous dirctory

{user@ws26~]$cd -

**1.2.2 Listing contents in a directory – [ ls, ls-l, ll]**

**ls**

This command is used to list all the files and directories in the current directory.

**Eg**:

[user@ws26 ~]$ ls

1152696870.jpg book OperaDownloads snapshot52.png

Image300.jpg Desktop test spiderman2.jpg

**ls – or ll**

This command is also used to list all the files and directories. Here you will get more details about the files and directories present in the current directory. You will see the permission set, creation date, file /directory size etc.

**Eg:**

[user@ws26 ~]$ ls –l

total 5

-rw-rw-r--1 user user 33188 Dec 22 02:56 1152696870.jpg

-rw-rw-r-- l user user 17647 Aug 19 2006 534458.gif

-rw-rw-r-- 1 user user 26817 Aug 19 2006 534477.gif

drwxrwxr – x 2 user user 4096 Jan 13:04:20 book

drwx------- 3 user user 4096 Feb 4 02:34 Desktop

**1.2.3 Reading files in Linux - [ cat, more, less ]**

**cat**

This command is used to display the contents of a file. You can read the file contents using the cat command.

**Example:**

[user@ws26 ~]$ cat testfile

This is a test file… You can read me…

**less and more**

Both the less and more commands serve similar function. They are used to display file one screen at a time. You can press spacebar to continue reading the file. These commands are main used while reading large files.

**Example 1:**

[user@ws26 ~] less testfile

This is a test file… You can read me..

**Example 2:**

[user@ws26 ~]$ more testfile

This is a testfile.. You can read me..

**Example 2:**

[user@ws26 ~]$ cat testfile | more

This is a testfile.. You can read me..

**1.2.4 Manipulating files – [cp, mv, rm, mkdir]**

**cp**

This command is used to copy files/directory.

**Example:**

[user@ws26 ~]$ cp file1 directory1

This will copy the file1 to the directory 1

**cp - r**

This command is used to copy directories recursively (copy all the files and folders inside the directory).

**Example:**

[user@ws26 ~]$ cp -r directory2 directory1

This will copy the whole directory2 into directory1

**mv**

This command is used to move the file or directory.

**Example:**

[user@26 ~]$ mv directory2 directory1

This will move whole directory2 into directory1.

**rm**

This command is used to remove or delete files and directories.

**Example:**

[user@ws26 ~]$ rm directory2

This will delete the directory – directory2

* rm –rf deletes the folders and sub folders. Please care ful using this command. Once you deleted you cant back the files unless your server has snapshot backup

**mkdir**

This command is used to create a new directory

**Example:**

[user@ws26 ~]$ mkdir directory3

This will create a new directory – directory3.

**1.2.5 The vi editor**

The Linux vi Editor is a text based editor used Linux, useful for editing configuration files or creating plan text documents. Vi is started by simply entering “vi” (it is case sensitive) at the linux prompt, and once started is exited using ESC;q! To abandon changes or SHIFT+ZZ (case sensitive) to save changes. If you want to edit (or create) a specific file uses the command “vi/path\_to\_file/name\_of\_file”. “path\_to\_file” must already exit as vi will not make any required directories.

|  |  |
| --- | --- |
| **Command** | **Description** |
| I | Insert mode, (ESC to exit insert mode) allows text to be entered on the screen |
| A | Append to right mode |
| /word | Move to the occurrence of “word” |
| N | Locate the next occurrence |
| W | Advance to the next word |
| E | Advance to the next end of a word |
| B | Move to the previous word |
| 3b | Move backward 3 words |
| Yy | Copy line (then move cursor and use p to paste after current cursor line) |
| Dd | Delete line |
| 3dd | Delete 3 lines |
| D | Delete the remainder of a line |
| Dw | Delete word |
| X | Delete Character |
| O | Open space for new line below the cursor line |
| O | Open a line above the cursor |
| CTRL – w | Move back a word in append mode |
| U | Undo last |
| U | Undo all changes to current line |
| . | In command mode, repeat the last text changing the command on the current line |
| :w newfilename | Save the file to newfilename from the command mode |
| :wq | Save and quit |
| :q | Quit without saving |
| R | Replace then type a character to be replaced with r then return to break up a line |
| J | Join 2 lines |
| S | Substitute (sentence) typed text over a character, ESC when done |
| Cw | Change word |
| C | Change part of a line from the cursor to the end of the line |
| Cc | Substitute new text for aline, ESC when done |
| H | Move to the cursor back one space |
| H | Move the cursor to the highest line on the screen |
| L | Move the cursor to the lowest line on the screen |
| M | Position the cursor at the midpoint on the screen |
| G | Last line in the file |
| 0 (zero) | Move the cursor to the beginning of the line it is on |
| view filename | Open a file for viewing only |
| set number | Turn on line numbers |
| set nonumber | Turn off line numbers |
| **Options** |  |
| Autoindent | (ai ctrl-d to move to the left |
| showmatch | Match brackets and parenthesis |
| tabstop = 4 |  |
| wrapmargin(wm) | When line is wrapped (80-value) |
| :2 copy 4 | Place a copy of the line 2 after line 4 |
| :1,4 copy 7 | (lines 1-4) |
| :n | Access the next file for editing |

ps –aef : Display curent running process

wc : Gives no of lines, words, characters in the file

**Example:**

[user@ws26 ~]$ wc test

5 20 100

**Exercises:**

1. Practice following Linux commands?

pwd

ls

ls -l

mkdir

cd

cp

rm

mv

cat

2. How to create a new C program file by using vi editor?

3. Difference between ls and ls -lrt. When we use ls -lrt in real scenario.

4. What is the difference between GUI application and CLI application?

5. How to use USB flash disk (pen drive) in Linux? How to copy files to and from USB to linux. Try in both Virtual machine and dual boot Linux.

6. How to print Number of current running process (count)

Hint: use ps –aef and wc commands

**Chapter 2 - Structure of C Program**

**2.1 History of C**

Dennis Ritchie developed the C language in 1970’s at Bell Laboratories. Initially it was designed for programming in the operating system called UNIX. After the advent of C, the whole UNIX operating system was rewritten using it. Now almost the entire UNIX operating system and the tools supplied with it including the C compiler itself are written in C.

**2.2 Characteristics of C**

It is a middle level language. It has the simplicity of a high level language as well as the power of a low level language. This aspect of C makes it suitable for writing both application programs and system programs. Hence it is an excellent, efficient and general-purpose language for most of the applications such as mathematical, scientific, business and system software applications.

C is small language, consisting of only 32 English words known as keywords (if else, for, break, etc.). The power of C is augmented by the library functions provided with it. Moreover, the language is extendible since it allows the users to add their own library functions to the library.

C contains control constructs needed to write a structured program hence it is considered a structured programming language. It includes structures for selection (if…else, switch), reception (while, for, do…while) and exit (break).

The programs written in C are portable i.e. programs written for one type of computer or operating system can be run on another type of computer or operating system.

**2.3 Structure of a C program**

C program is a collection of one or more functions. Every function is a collection of statements and performs some specific task. The general structure of C program is

**Comments**

**Preprocessor directives**

**Global variables**

**int main( ) function**

**{**

**local variables**

**statements …..**

**return 0;**

**}**

**func1( )**

**{**

**local variables**

**statements**

**………….**

**}**

Comments can be placed anywhere in a program and are enclosed between the delimiters /\* and \*/. Comments are generally used for documentation purposes.

Preprocessor directives are processed through preprocessor before the C source code passes through compiler. The commonly used preprocessor directives are # include and #define. #include is used for including header files. #define is used to define symbolic constants and macros.

Every C program has one or more functions. If a program has only one function then it must be main ( ). Execution of every C program starts with main ( ) function. It has two parts, declaration of local variables and statements. The scope of the local variable is local to that function only. Statements in the main( ) function are executed one by one. Other functions are the user-defined functions, which also have local variables and C statements. They can be defined before or after main( ). It may be possible that some variables have to be used in many functions. So it is necessary to declare them globally. These variables are called global variables.

\*\*Nested comments are not allowed in C programming

**2.4 Structure of C Program with multiple C files**

The following figure represents a program with multiple C files. Some C files contain onlyfunctions, 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.

draw boxes in one box mainprogram and in other box sub functions.

**2.5 C-Program compilation & Execution in Linux Environment**

The steps for the execution of C program are as

1. Program creation
2. Program compilation
3. Program execution

Generally a command line C compiler is provided with UNIX operating system. This compiler is named as cc or gcc.

1. **Program creation**

File can be created with vi editor as

**$ vi filename.c**

Here $ is the unix prompt

1. **Program compilation**

After creation of C program, it can be compiled as

**$ cc filename.c or $ gcc filename.c or $ gcc filename.c -o out**

1. **Program execution**

After the compilation of program, it can be executed as

**$ ./a.out or $ ./out**

**Assignment:**

1. Write C program to print Hello world in Linux, compile and execute.

**Chapter 3-Elements of C**

Every language has some basic elements and grammatical rules. Before understanding programming, it is must to know the basic elements of C language. These basic elements are character set, variables, datatypes, constants, keywords, variable declaration, expressions, statements etc. All of these are used to construct a C program.

**3.1 C Character Set**

The characters that are used in C program are given below

**3.1.1 Alphabets**

**A, B, C…………Z**

**a, b, c…………..z**

**3.1.2 Digits 1, 2, 3, 4, 5, 6, 7, 8, 9**

**3.1.3 Escape Sequences**

Characters are printed on the screen through the keyboard but some characteristics such as newline, tab, and backspace cannot be printed like other normal characters. C supports the combination of backslash (\) and some chararcters from the C chararcter set to print these chararcters. These character combinations are known as escape sequences and are represented by two characters the first chararcter is “\” and second chararcter is from the C character set.

|  |  |  |
| --- | --- | --- |
| Escape Sequence | Meaning | Purpose |
| \b | Backspace | Moves the cursor to the previous position of the current line. |
| \a | Bell(alert) | Produces a beep sound for alert |
| \r | Carriage return | Moves the cursor to beginning of the current line |
| \n | Newline | Moves the cursor to the beginning of the current line. |
| \o | Null | Null |
| \t | Horizontal tab | Moves the cursor to the next horizontal tab position |
| \\ | Backslash | Presents a character with backslash(\) |

**3.2 Reserved Words / keywords**

There are certain words that are reserved for doing specific tasks. These words are known as keywords and they have standard, predefined meaning in C. They are always written in lowercase. There are only (32 keywords) available in C which are given below.

auto break case char const continue

default do double else enum extern

float for goto if int long

register return short signed sizeof static

struct switch typedef union unsigned void

volatile while

**3.3 Identifiers**

All the words that we’ll use in our C program will be either keywords or identifiers. Keywords are predefined and can’t be changed by the user while identifiers are user defined words and are used to give names to entities like variables, arrays, functions, structures etc.

**3.3.1 Rules for naming identifiers**

(1) The name should consist of only alphabets, digits and underscope sign(\_).

(2) First character should be an alphabet or underscore.

(3) The name should not be a keyword.

(4) Since C is case sensitive, the uppercase and lowercase letters are considered different.

(5) An identifier name may be arbitrarily long. Some implementations recognize 31 characters.

The identifiers are generally given meaningful names. Some examples of valid identifier names-

Value a net\_pay rec1 \_data MARKS

Some examples of invalid identifier names are

**5bc first character should be an alphabet or underscore**

**int int is a keyword**

**rec# # is a special characteristics**

**avg no blank space is not permitted**

**3.4 Data Types**

C supports different types of data. Storage representation of these data types is different in memory. There are four fundamental datatypes in C, which are int, char, float and double.

‘char’ is used to store any single any single character, ‘int’ is used to store integer value, ‘float’ is used for storing precision floating point number and ‘double’ is used for storing double precision floating point number.

There are two types of type qualifiers

1. **Size qualifiers – short, long**
2. **Sign qualifiers – signed, unsigned**

When the qualifier unsigned is used the number is always positive, and when signed is used number may be positive or negative. If the sign qualifier is not mentioned, then by default signed qualifier is assumed. The range of values for signed data types is less than that of unsigned type. This is because in signed type, the leftmost bit is used to represent the value.

The size and range of different data types on a 32-bit machine is given in the following table. The size and range may vary on machines with different word sizes.

|  |  |  |  |
| --- | --- | --- | --- |
| **Basic data types** | **Data types with type qualifiers** | **Size(bytes)** | **Range** |
| Char | char or signed char | 1 | -128 to 127 |
|  | unsigned char | 1 | 0 to 255 |
| Int | int or signed int | 4 | -2147483648 to 2147483647 |
|  | unsigned int | 4 | 0 to 4294967295 |
|  | short int or signed short int | 2 | -32768 to 32767 |
|  | unsigned short int | 2 | 0 to 65535 |
|  | long int or signed long int | 4 | -2147483648 to 2147483647 |
|  | unsigned long int | 4 | 0 to 4294967295 |
| float | float | 4 | 3.4E-38 to 3.4E+38 |
| double | double | 8 | 1.7E-308 to 1.7E + 308 |
|  | long double | 10 | 3.4D+4932 to 1.1E+4932 |

**3.5 Constants**

Constant is a value that cannot be changed during execution of the program. There are 3 types of constants



**3.5.1 Numeric Constants**

Numeric constants consists of numeric digits, they may or may not have decimal point(.).

These are the rules for defining numeric constants-

1. Numeric constant should have at least one digit.
2. No comma or space is allowed within the numeric constant.
3. Numeric constants can either be positive or negative but default sign is always positive.

There are two types of numeric constants.

* + - 1. **Integer Constants**

Integer constants are whole numbers which have no decimal point(.).

Decimal constants – 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 (base 10)

Octal Constant – 0, 1, 2, 3, 4, 5, 6, 7 (base 8)

Hex decimal constants – 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F (base 16)

Some valid decimal integer constants are –

0

123

3705

Some invalid decimal integer constants are

* 1. illegal character (.)

3#5 illegal character (#)

98 5 No blank space allowed

6,784 Comma is not allowed

* + - 1. **Real (floating point) constants**

Floating point constants are numeric constants that contain decimal point. Some valid floating-point constants are

0.5

0.0034

34.435

**3.5.2 Character Constants**

A character constant is a single character that is enclosed within single quotes. Some valid character constants are –

‘9’ ‘D’ ‘$’ ‘ ‘ ‘#’

Some **invalid character constants** are

**‘four’ There should be only one character within quotes**

**“d” Double quotes are not allowed**

**“ No character between single quotes**

**Y Single quotes missing**

Every character constant has a unique integer value associated with it. This integer is the numeric value of the character in the machine’s character code. If the machine is using ASCII(American Standard Code for Information Interchange), then the character ‘G’ represents integer value 71 and the character ‘5’ represents value 53. Some **ASCII values** are-

**A – Z ASCII value (65 – 90)**

**a- z ASCII value (97 – 122)**

**0-9 ASCII value (48 – 57)**

**; ASCII value (59)**

* + 1. **String Constants**

A string constant has zero, one or more than one character. A string constant is enclosed within double quotes (“ “). At the end of the string, the compiler automatically places \0. Note that “A” and ‘A’ are different, the first one is a string constant which consists of character A and \0 while the second one is a character constant which represents integer value 65.

* 1. **Variables**

Variable is a name that can be used to store values. Variables can take different values but one at a time. These values can be changed during execution of the program. A data type is associated with each variable. The data type of the variable decides what value it can take. The rules for naming variables are same as that for naming identifiers.

**3.6.1 Declaration of Variables**

It is must to declare a variable before it is used in the program. Declaration of a variable specifies its name and datatype. The type and range of values that a variable can store depends upon its datatype.

The synatax of declaration of a variable is –

Datatype variablename;

Here datatype may be int, float, char, double etc. Some examples are

int x;

float salary;

char grade;

We can also declare more than one variable in a single declaration. For example –

Int x, y, z, total;

**3.6.2 Initialization of variables**

When a variable is declared it contains undefined value commonly known as garbage value. If we want we can assign some initial value to the variable during the declaration itself, this is called initialization of the variable. For example –

int a = 5;

float x = 8.9, y = 10.5;

char ch = ‘y’;

int l, m, n, total = 0;

In the last declaration only variable total has been initialized.

* 1. **Expressions**

An expression is a combination of operators, constants, variables and function calls. The expression can be arithmetic, logical or relational.

Some examples are as

x+y - arithmetic operation

a = b + c - uses two operators (=) and (+)

a > b - relational expression

a==b - logical expression

func(a, b) - fuction call

* 1. **Statements**

In a C program, instructions are written in the form of statements. A statement is an executable part of the progam and causes the computer to carry out some action. Statements can be categorized as –

1. Expression statements

(ii) Compound Statements

(iii) Selection statements (if, if…else, switch)

(iv) Iterative statements (for, while, do….while)

(v) Jump statements (goto, continue, break,return)

(vi) Label statements (case, default, label statement used in goto)

**3.8.1 Expression Statement**

Expression statement consists of an expression followed by a semicolon. For example

x = 5;

x = y-z;

func(a,b);

A statement that has only a semicolon is also known as null statement. For example –

; /\*null statement\*/

**3.8.2 Compound Statement**

A compound statement consists of several statements enclosed within a pair of curly braces {}.

Compound statement is also known as block of statements. Note that there is no semiclon after the closing brace. For example

{

int i = 4, b = 2, h = 3;

int area, volume;

area = 2\*(l\*b + b\*h + h\*l);

volume = l\*b\*h;

}

If any variable is to be declared inside the block then it can be declared only at the beginning of the block. The variables that declared inside a block can be used only inside that block.

* 1. **Comments**

Comments are used for increasing readability of the program. They explain the purpose of the program and are helpful in understanding the program. Comments are written inside /\* and \*/. There can be sigle line or multiple line comments. We can write comments anywhere in a program except inside a string constant or a character constant. Comments can’t be nested i.e. We can’t write a comment inside another comment.

**Chapter 4 – Input – Output in C**

There are three main functions of any programs. It takes data as input, processes this data and gives the output. The input operation involves movement of data from an input device (generally keyword) to computer memory, while in output operation the data moves from computer memory to the output device (generally screen). The input output is performed through a set of library functions that are supplied with every C compiler.

There are several header files that provide necessary information in support of the various library functions. These header files are entered in the program using the #include directive at the beginning of the program. For example if a program uses any function from, the standard I/O library, then it should include the header file <stdio.h> as –

**#include <stdio.h>**

Similarly there are other header files like string.h,stdlib.h that should be included when certain library functions are used.

A simple method for taking the data as input is to give the value to the variables by assignment statement. For example –

**int basic = 2000;**

**char ch = ‘y’;**

But in this way we can give only particular data to the variables. The second method is to use the input function scanf(), which takes the input data from the keyboard. In this method we can give any value to the variables at run time. For output,we use the function printf().

**4.1 Conversion Specifications**

The functions scanf() and printf() make use of conversion specifications to specify the type and size of data. Each conversion specification must begin with a percent sign(%). Some conversion specification are as given below –

**%c - a single character**

**%d - a decimal integer**

**%f - a floating point number**

**%o - an octal integer**

**%x - a hexadecimal integer**

**%s - a string**

**%u - an unsigned decimal integer**

**4.2 Reading Input Data**

Input data can be entered into the memory from a standard input device (key board). C provides the scanf() library function for entering input data. This function can take all types of values as input. The scanf() function can be written as

**Scanf(“control string”, address1, address2, ……);**

This function should have at least two parameters. First parameter is a control string, which contains conversion specification characters. It should be within double quotes. The conversion specification characters may be one or more; it depends on the number of variables we want to input. The other parameters are addresses of variables. In the scanf() function at least one address should be present. The address of a variable is found by preceding the variable name by an ampersand (&) sign. A string variable is not preceded by & sign to get the address.

Some examples of scanf() function –

**#include <stdio.h>**

**main( )**

**{**

**int marks;**

**………...**

**scanf(“%d”, &marks);**

**……….....**

**}**

In this example, it contains only one conversion specification character %d, which implies that one integer value should be entered as input.

**#include<stdio.h>**

**main( )**

**{**

**int basic;**

**float hra;**

**char grade;**

**…….**

**Scanf(“%d %f %c”, &basic, &hra, &grade);**

**…….**

**}**

Here, it contains three conversion specifications characters %d, %f and %c, means that one integer value, one floating point value and one single character can be entered as –

**1500 200.50 A**

**4.3 Writing Output Data**

Output data can be written from computer memory to the standard output device using printf() library function. With this function all type of values can be written as output. The printf() function can be written as –

**Printf(“control string”, variable1, variable2,…..);**

In this function the control string contains conversion specification characters and text. It should be enclosed within double quotes. The name of variables should not be preceded by an ampersand(&) sign. Some examples of printf() function –

**#include<stdio.h>**

**main( )**

**{**

**printf(“C is excellent\n”);**

**}**

**Output:**

**C is excellent**

Here control string has only text and no conversion specification character, hence the output is only text.

**#include<stdio.h>**

**main( )**

**{**

**int age;**

**printf(“enter your age”);**

**scanf(“%d”,&age);**

**printf(“age = %d”, age);**

**}**

**Output:**

**enter your age:45**

**age = 45**

Here the first printf also does not contain any conversion specification and is used to display a message that tells the user to enter his age. Scanf will read the data from keyboard and store into the address of age variable. The second printf has the conversion specification %d which specifies the integer prints age value on the monitor.

**4.4 getchar( ) and putchar( )**

These macros getchar( ) and putchar( ) can be used for character I/O.getchar( ) reads a single character from the standard input, putchar( ) outputs one character at a time to the standard output.

**#include<stdio.h>**

**main( )**

**{**

**char ch;**

**printf(“enter a character:”);**

**ch = getchar( );**

**printf(“The entered character is :”);**

**putchar(ch);**

**}**

**Output:**

**Enter a character:B**

**The entered character is:B**

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.

**#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();**

**}**

**4.5 Reading small size integer variables**

We use “%d” as format specification in scanf( )function to read integer variables. But when wewant to read small integer variables like char, and short variables we must use differentformat specification. This is illustrated in the following program. Note that char variable iscapable of storing from –128 to 127. Where as unsigned char variable can store from 0 to255. The char variable occupies 8 bits or 1 byte of space. The unsigned short variables takevalues from 0 to 65535 where as short (signed) variables can hold from –32768 to 32767.

**#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);**

**}**

**4.6 Reading and Printing text characters**

So far we studied about storing integer and float (numbers with decimal point) numbers invariables. But another use of variables is for storing the text characters. Text characters consistsof 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 smallletters ‘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 control characters. These control characters represent nonprintablecharacters. 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 ASCIIcodes range from 0 to 127, so these codes can be easily fit in char variables.

**#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 avariable, we can assign the ASCII code to that variable. In the above program we are assigningASCII code 65 (code of letter ‘A’) directly. But easy way is to assign the character with singlequotes. 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 asformat. 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 variableby using scanf()function. The following program is printing the character it has read, in bothcharacter form and integer form. This program will be useful to find the ASCII code of anycharacter.

**#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);**

**}**

**4.7 Printing and reading integers in other notations**

We normally define integer variables by using ‘int’ type. However when we define variables byusing ‘char’ or ‘short’ type, they are also integer variables. Only difference is that these are smallintegers. 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 asformat. The %u prints integer as unsigned value. However we can print integers in Octal or Hexformat by using %o or %x as format. This is illustrated in the following program. It is assumedthat the student will know the differences between octal, decimal and hex (hexadecimal)notations.

**#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, octaland 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.

**#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);**

**}**

**4.8 Printing and reading float variables**

Similar to integer variables, which are available in three sizes (8 bit, 16 bit and 32 bit) floatingpoint variables are also available in two sizes. These are 32 bit and 64 bit. Remember that, allsizes of integer variables can be printed by using any of %d, %u, %x, and %o formats. But whilereading integer variables of 16 bit and 8 bits, you have to prefix ‘h’ and ‘hh’ respectively, befored, u, x, or o.Similarly, both the sizes of floating point variables can be printed by using %f, %e or %gformats. However while reading double type floating variables, we need to prefix ‘l’ before f, eor 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 thisway we can control the precision of printing floating point numbers.

**#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, f1f2);**

**printf(“Enter two double float numbers\n”);**

**scanf(“%lf%lf”,&d1,&d2);**

**printf(“You entered %.8f and %.8f\n”);**

**}**

**4.9 Sizeof Operator**

The sizeofoperator is useful to find the memory size of any variable. The size is always given innumber of bytes. The sizeof operator is also used to get the size of any typeby specifying typename.

**#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);**

**scanf(“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));**

**}**

**Assignment**

**1.int main()**

**{**

char ch1=2,ch2=3;

printf("Result : %c \n", ch1+ch2);

printf("Result : %d \n", ch1+ch2);

}

**2. int main()**

**{**

unsigned int x=034;

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

}

**3. int main()**

{

printf(“%d”,printf(“hellow world\n”));

}

**4. int main()**

{

Int a, b, c;

printf(“%d”,scanf(“%d%d%%d”,&a,&b,&c));

}

**Chapter 5 - Operators And Expressions**

An operator specifies an operation to be performed that yields a value. Various operators to form an expression can join the variables, constants. An operand is a data item on which an operator acts. Some operators require two operands, while others act upon only one operand, C includes a large number of operators that fall under several different categories, which are

Arithmetic operators

Assignment operator

Increment and Decrement operators

Relational operators

Logical operators

Conditional operator

Comma operator

sizeof operator

Bitwise operators

Other operators

## 5.1 Arithmatic Operators

Arithmatic operators are used for numeric calculations. They are of twp types

1. Unary arithmetic operators
2. Binary arithmetic operators

### 5.1.1 Unary arithmetic operators

Unary operators require only one operand. For example-

**+X -Y**

Here ‘-‘ changes the sign of the operand Y

### 5.1.2 Binary arithmetic operators

Binary operators require two operands. There are five binary arithmetic operators

|  |  |
| --- | --- |
| **Operator** | **Purpose** |
| + | Addition |
| - | Substraction |
| \* | Multiplication |
| / | Division |
| % | Gives the remainder in integer division |

% (Modulus operator) cannot be applied with floating point operands. There is no exponent operator in C.

Note that unary plus and unary minus are different from the addition and subtraction operators.

# 5.2 Assignment Operator

A value can be stored in a variable with the use of assignment operator. This assignment operator " = " is used in assignment expressions and assignment statements.(the operand on the left hand side should be a variable, while the operand on the right hand side can be any variable, constant or expression.). The value of right hand operand is assigned to the left hand operand. Here are seme examples of assignment expressions

**x=8 /\* 8 is assigned to x \*/**

**s=x+y-2 /\* value of expression x+y-2 is assigned to s \*/**

**y=x /\* value of x is assigned to x \*/**

We can have multiple assignment expressions also, for example-

**x = y = z = 20;**

When the variable on the left hand side of assignment operator also occurs on right hand side then we can also be written as

**x += 5**

Here += is a compound assignment operator.

**x -= 5 is equivalent to x=x-S**

**y \*= 5 is equivalent to y=y\*5**

**sum /= 5 is equivalent to sum = sum/5**

**k%= 5 is equivalent to k=k%5**

# 5.3 Increment and Decrement Operators

C two useful operators increment(++) and decrement (--). These are unary operators because they operate on a single operand. The increment operator (++) increments the value of the variable by 1 and decrement operator (--)decrements the value of the variable by 1

**++x is equivalent to x=x+1**

**--x is equivalent to x=x-1**

These operators should be used only with variables; they can't be used with constants or expressions. For example the expressions + + 5 or + + (x+y+z) are invalid.

These operators are of two types-

1. Prefix increment/decrement - operator is 'written before the operand,
2. Postfix increment/decrement - operator is written after the operand

**/\* prog3.c : pre- or post-increment (decrement) operators \*/**

#include <stdio.h>

main()

{

int w, x, y, z, result;

w = x = y = z = 1; /\* initialize x and y \*/

printf(“Given w = %d, x = %d, y = %d, and z = %d, \n”, w, x, y, z);

result = ++w;

printf(“++w gives %d \n”, result);

result = x++;

printf(“x++ gives: %d \n”, result);

result = --y;

printf(“--y gives %d \n”, result);

result = z--;

printf(“z-- gives %d \n”, result);

}

Let us have another example

**#include <stdio.h>**

**main()**

**{**

**int i=5;**

**printf("%d %d %d %d %d\n",i,i++,++i,i++,i);**

**}**

If the above example is executed in TurboC compiler then the output will be

**8 7 7 5 5**

If you execute the same program in GCC compiler the output will be different like,

**8 7 8 5 8**

**5.4 Relational Operators**

Relational operators are used to compare values of two expressions depending on their relations. An expression that contains relational operators is called relational expressions. If the relation is true then the value of relational expression is 1 and if the relation is false the the value of expression is 0. The relational operators are

|  |  |
| --- | --- |
| **Operator** | **Meaning** |
| < | Less than |
| <= | Less than or equal to |
| == | Equal to |
| != | Not equal to |
| > | Greater than |
| >= | Greater than or equal to |

Let us take two variables a=9 and b=5, and from simple relational expressions with them

|  |  |  |
| --- | --- | --- |
| **Expression** | **Relation** | **Value of expression** |
| a<b | False | 0 |
| a<=b | False | 0 |
| a==b | False | 0 |
| a!=b | True | 1 |
| a>8 | True | 1 |
| b!=0 | True | 1 |
| 2>4 | False | 0 |

It is important to note that the assignment operator (=) and equality operator(==) are entirely different. Assignment operator is used for assigning values while equality operator is used to compare two expressions. Beginners generally confuse between the two and use one in the place of another, and this eads to an error difficult to find out. For example

#include <stdio.h>

main()

{

int a=10,b=20;

printf("%d\n",a==b);

printf("%d\n",a=b);

getch();

}

Output:

0

20

The first printf statement will compare a and b variables, and will give the output as zero(0) as they are not same. The second printf statement will assign the value of b to variable a and will give the output as 20.

#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);

}

#include <stdio.h>

int main()

{

int a;

int b;

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

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

100,200

printf("Result of a < b is %d\n", a < b); 1

printf("Result of a > b is %d\n", a > b); 0

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);

}

#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);

}

**5.5 Logical Or Boolean Operators**

An expression that combines two or more expressions is termed as a logical expression. For combining these expressions we use logical operators. These operators return 0 for false and 1 for true. The operands may be constants,variables or expressions. C has three logical operators

|  |  |
| --- | --- |
| **Operator** | **Meaning** |
| && | AND |
| || | OR |
| ! | NOT |

Here logical NOT is a unary operator while the other two are binary operators. In C any non-zero value is regarded as true and zero is regarded as false.

**5.5.1 AND (&&) Operator**

This operator gives the net result true if both the conditions are true, otherwise the result is false.

**Boolean Table**

|  |  |  |
| --- | --- | --- |
| **Condition 1** | **Condition 2** | **Result** |
| False(0) | False(0) | False(0) |
| False(0) | True(1) | False(0) |
| True(1) | False(0) | False(0) |
| True(1) | True(1) | True(1) |

Let us take three variables a=10,b=5,c=0

Suppose we have a logical expression

(a==10)&&(b<a)

Here both the conditions a == 10 and b<a are true, and hence this whole expression is true. Since the logical operators return 1 for true hence the value of this expression is 1.

**5.5.2 OR (||) Operator**

This operator gives the net result false, if both the conditions have the value false, otherwise the result is true.

**Boolean Table**

|  |  |  |
| --- | --- | --- |
| **Condition 1** | **Condition 2** | **Result** |
| False(0) | False(0) | False(0) |
| False(0) | True(1) | True(1) |
| True(1) | False(0) | True(1) |
| True(1) | True(1) | True(1) |

Let us take three variables a=10, b=5, c=0

Consider the logical expression-

(a>=10) || (b>15)

This gives the result true because one condition is true.

**5.5.3 NOT (!) Operator**

This is a unary operator and it negates the value the value of the condition. If the value of the condition is false then it gives the result true. If the value of the condition is true then it gives the result false.

**Boolean Table**

|  |  |
| --- | --- |
| **Condition** | **Result** |
| False(0) | True(1) |
| True(1) | False(0) |

Let us take a variable a=10

Suppose we have this logical expression-

!(a == 10)

The value of the condition (a == 10). NOT operator negates the value of the condition. Hence the result is false.

**5.6 Conditional Operator**

Conditional operator is a terneryoperator( ? AND : ) which requires three expressions as operands.

This is written as –

***TestExpression ?Expression1 : expression2***

Firstly the TestExpression is evaluated.

If TestExpression is true(nonzero), then expression1 is evaluated and it becomes the value of the overall conditional expression.

If TestExpression is false(zero), then expression2 is evaluated and it becomes the value of the overall conditional expression.

For Example, Consider this conditional expression –

a >b ?a : b

Here first the expression a>b is evaluated, if the value is true then the value of variable a becomes the value of unconditional expression otherwise the value of b becomes the value of variable a becomes the value of conditional expression otherwise the value of b becomes the value of conditional expression. Suppose a = 5 and b = 8, and we use the above conditional expression in a statement as

Max = a >b ?a : b;

First the expression a>b is evaluated, since it is false so the value of b becomes the value of conditional expression and it is assigned to variable max.

In our next example we have written a conditional statement by putting a semicolon after the conditional expression.

a <b ?printf(“a is smaller”) : printf(“b is smaller”);

since the expression a < b is true, so the first printf function is executed.

**5.7 Comma Operator**

The comma operator (,) is used to permit different expressions to appear in situations where only one expression would be used. The comma operator seperates the expressions. The separated expressions are evaluated from left to right and the type and value of the rightmost expression is the type and value of the compound expression –

**a=8, b=7, c=9, a+b+c**

Here we have combined 4 expressions. Initially 8 is assigned to the variable a, then 7 is assigned to the variable b, 9 is assigned to the variable c and after this a+b+c is evaluated which becomes the value of whole expression. So the value of the above expression is 24. Now consider this statement-

**Sum = ( a=8, b=9, c=7, a+b+c);**

Here the value of the whole expression on right side will be assigned to variable i.e. sum will be assigned value 24. Since precedence of comma operator is lower than that of assignment operator hence the parentheses are necessary here.

#include <stdio.h>

main()

{

int a=8, b=7,temp;

printf("a=%d,b=%d\n",a,b);

temp=a,a=b,b=temp;

printf("a=%d,b=%d\n",a,b);

}

OUTPUT:

a=8,b=7

a=7,b=8

**5.8 sizeof Operator**

Sizeof is an unary operator. This operator gives the size of its operand in terms of bytes. The operand can be a variable, constant or any data type(int, float, char etc). For example sizeof(int) gives the bytes occupies by the int data type i.e. 5.

/\*Program to understand the sizeof operator\*/

#include <stdio.h>

main()

{

int var;

printf("size of int = %d\n", sizeof(int));

printf("size of float = %d\n", sizeof(float));

printf("size of var = %d\n", sizeof(var));

printf("size of integer constant = %d\n", sizeof(45));

}

OUTPUT:

size of int = 4

sizeof float = 4

sizeof var = 4

size of integer constant = 4

**5.9 Bitwise Operators**

C has the ability to support the manipulation of data at the bit level. Bitwise operators are used for operations on individual bits. Bitwise operators operate on integers only. The bitwise operators are as

|  |  |
| --- | --- |
| **Bitwise Operator** | **Meaning** |
| & | Bitwise AND |
| | | Bitwise OR |
| ~ | One’s complement |
| << | Left shift |
| >> | Right shift |
| ^ | Bitwise XOR |

It is important to distinguish between bitwise operators & and | from the logical operators && and ||

If x is 1 and y is 2

Then x & y is

**x - 0 0 0 0 0 … 0 0 0 1 32 bits 1**

**y - 0 0 0 0 0 … 0 0 1 0 32 bits 2**

**x& y = 0 (bitwise & operation)**

**x&& y is 1 (logical && operator)**

since x is true(non-zero value 1) and y is also true (non-zero value 2) resultant is true that is 1.

The bitwise operators operate on numbers (always integers) as if they were sequences of binary bits (which, ofcourse, internally to the computer they are )

These operators will make the most sense , therefore, if we consider intergers as represented in binary, octal, or hexadecimal (bases 2, 8, or 16), not decimal (base 10). Remember, you can use octal constants in c by prefixing them with an extra 0 (zero), and you can use hexadecimal constants by prefixing them with 0x (or oX).

**5.9.1 Bitwise AND (&)**

The & operator performs a bitwise AND on two integers. Each bit in the result is 1 only if both corresponding bits in the two input operands are 1. For example, 0x56 & 0x32 is 0x12, because (in binary):

**0 1 0 1 0 1 1 0**

**& 0 0 1 1 0 0 1 0**

**-------------------------------------**

**0 0 0 1 0 0 1 0**

**5.9.2 Bitwise OR (|)**

The | (vertical bar) operator performs a bitwise OR on two integers. Each bit in the result is 1 if either of the corresponding bits in the two input operands is 1. For example, 0x56 | 0x32 is 0x76, because

**0 1 0 1 0 1 1 0**

**| 0 0 1 1 0 0 1 0**

**-------------------------------------**

**0 1 1 1 0 1 1 0**

**5.9.3 Bitwise XOR (^)**

The ^ (caret) operator performs a bitwise exclusive OR on two integers. Each bit in the result is 1 if One, but not both, of the corresponding bits in the two input operands is 1. For example, 0x56 ^ 0x32 is 0x64:

**0 1 0 1 0 1 1 0**

**^ 0 0 1 1 0 0 1 0**

**-------------------------------------**

**0 1 1 0 0 1 0 0**

**5.9.4 Bitwise One’s complement (~)**

The ~ (tilde) operator performs a bitwise complement on its single integer operand. ( The ~ operator is therefore a unary operator, like ! and the unary -, &, and \* 1’s to 0’s. for example, assuming 16-bit integers, ~0x56 is 0xffa9:

**~ 0 0 0 0 0 0 0 0 0 1 0 1 0 1 1 0**

**-----------------------------------------------------------------------------------**

**1 1 1 1 1 1 1 1 1 0 1 0 1 0 0 1**

**5.9.5 Bitwise Leftshift( << ) and Rightshift ( >> )**

The << operator shifts its first operand left by a number of bits given by its second operand, filling in new 0 bits at the right. Similarly, the >> operator shifts its first operand right. If the first operand is unsigned, >> fills in 0 bits from the left, but if the first operand is signed, >> might fill in 1 bits if the high-order bit was already 1. (Uncertainty like this is one reason why it's usually a good idea to use all unsigned operands when working with the bitwise operators.)

For example, 0x56 <<2 is 0x258:

01010110<<2

---------------------

0101011000

And 0x56 >> 1 is0x3b:

**01010110 >> 1**

**----------------**

**0 101011**

For both of the shift operators, bits that scroll ''off the end" are discarded; they don't wrap around, (Therefore, 0x56 >> 3 is 0x0a.)

The bitwise operators will make more sense if we take a look at some of the ways they're typically used. We can use & to test if a certain bit is 1 or not. for example, 0x56 & 0x40 is 0x40, but 0x32 & 0x40 is 0x00:

**0 1 0 1 0 1 1 0 0 0 1 1 0 0 1 0**

**& 0 1 0 0 0 0 0 0 & 0 1 0 0 0 0 0 0 0**

**-------------------------------------------------------------**

**1 0 0 0 0 0 0 0 0 0 0 0 0 0 0**

Since any nonzero result is considered, “true” in c, we can use an expression involving directly to test some condition, for example

**If( x& 0x04)**

*Do something;*

(If we didn’t like testing against the bitwise result, we could equivalently say if ( ( x & 0x04 ) ! = 0 ). The extra parentheses are important, as well explain below.) notice that the value 0x40 has exactly one 1 bit in its binary representation, which makes it useful for testing for the presence of a certain bit. Such a value is often called a *bit mask*. Often, we’ll define a series of bit masks, all targeting different bits, and then treat a single integer value as a set of flags. A “flag” is an on-off, yes-no condition, so we only need one bit to record it, not the 16 or 32 bits (or space, it also makes it convenient to assign a set of *flags* in a single int does more than just save space, it also makes it convenient to assign a set of flags all at once from one flag variable to another, using the conventional assignment operator =. For example, if we made these global declarations:

**int DIRTY = 0x01**

**int OPEN = 0x02**

**int VERBOSE = 0X04**

**int RED = 0x08**

**int = 0x10**

We would have setup 5 different bits as keeps track of those 5 different conditions (“ dirty,” ‘ ‘open,’ ‘ etc.). If we had a variable

Unsigned int flags;

Which contained a set of these flags, we could write tests like

**if (flags & DIRTY)**

**{ /\* code for dirty case \*/}**

**if(! (flags& OPEN))**

**{ /\* code for closed case \*/}**

**if (flags & VERBOSE)**

**{ /\* code for verbose case \*/}**

**else { /\* code for quiet case \*/}**

A condition like if (flags & DIRTY) can be read as “ if the DIRTY bit is on”.

These bit masks would also be useful for *setting* the flags. To “turn on the DIRTY bit,” we’d say

**Flags = flags | DIRTY; /\*set DIRTY bit\*/**

How would we “turn off” a bit? The way to do it is to leave on every bit but the one we’re turning off, if they were on already. We do this with the & and ~ operators;

**flags = flags & ~ DIRTY; /\* clear DIRTY bit \*/**

This may be easier to see if we look at it in binary. If the DIRTY, RED, and SEASICK bits were already on, flags would be 0x19, and we’d have

**0 0 0 1 1 0 0 1**

**& 1 1 1 1 1 1 1 0**

**---------------------------------**

**0 0 0 1 1 0 0 0**

As you can see, both the | operator when turning bits on and the & (plus ~) operator when turning bits off have no effect if the targeted bit were already on or off, respectively. The definition of the exclusive-OR operator means that you can use it to toggle a bit, that is, to turn it to 1 if it was 0 and to 0 if it was one:

**flags = flags A VERBOSE; . /\* toggle VERBOSE bit \*/**

It's common to use the ' ' op = " shorthand forms when doing all of these operations:

**flags 1= DIRTY;**

**flags |= DIRTY; /\* set DIRTY bit\*/**

**flags&= ~OPEN; /\* clear OPEN bit \*/**

**flagsA = VERBOSE; /\*toggle VERBOSE bit \*/**

we can also use the bitwise operators to extract subsets of bits from the middle of an integer. For example, to extract the second-to-last hexadecimal ‘'digit," we could use

( i& 0xf0) >> 4

If i was 0x56, we have:

I 0 1 0 1 0 1 1 0

& 0x56 & 1 1 1 1 0 0 0 0

------------------------------------------------------------------------------------

0 1 0 1 0 0 0 0

andshifting this result right by 4 bits gives us 0101 or 5, as we wished.

Replacing (or overwriting) a subset of bits is a bit more complicated; we must first use & and ~ to clear all of the destination bits, then use << and | to “OR in” the new bits. For example, to replace that second-to-last hexadecimal digit with some new bits, we might use:

**( I& ~0xf0) | newbits <<4)**

If I was still 0x56 and newbits was 6, this would give us

**i 0 1 0 1 0 1 1 0**

**& ~0xf0 & 0 0 0 0 1 1 1 1**

**-------------------------**

**0 0 0 0 0 1 1 0**

**| (newbits<<4 ) | 0 1 1 0 0 0 0 0**

**--------------------**

**0 1 1 0 0 1 1 0**

Resulting in 0x66, as desired

We’ve been using extra parentheses in several of these bitwise expressions because it turns out that (for the usual, hoary sort of “historical reasons”) the precedence of the bitwise &, |, and ^ operators is low, usually lower than we’d want. (the reason that they’re low is that, once upon a time, C didn’t have the logical operators && and ||, and the bitwise operators & and | did double duty.) However, since the precedence of & and | (and ~) is lower than ==, !=, <<, and >>, expressions like

**If(a & 0x04 != 0) /\* WRONG \*?**

**and**

**i& 0xf0 >>4 /\* WRONG\*/**

would not work as desired; these last two would be equivalent to

**If(a &(0x04 != 0))**

**| & (0xf0 >>4)**

And would not do the bit test or subset extraction that we wanted.

The issue of negative numbers, by the way, explains why the right-shift operator >> is not precisely defined when the high-order bit of the value being shifted is 1, Forsigned values, if the high-order bit is a 1, the number is negative. (This is true for Ids complement, 2‘s complement, and sign-magnitude representations.) If you were using a right shift to implement division, you'd want a negative number to stay negative, so on some computers, under some compilers, when you shift a signed value right and the high-order bit Is 1, new 1 bits are shifted in at the left instead of 0s, However, you can't depend on this, because not all computers and compilers implement right shift this way. In any case, shifting negative numbers to the right (even if the high-order 1 bit propogates) gives you an incorrect answer if there’s a remainder involved : in 2’s complement, 16-bit arithmetic, -15 is 0xfff1, so -15>>1 might give you 0xfff8 shifted which is -8. But integer division is supposed to discard the remainder, so -15/2 would have given you -7.

**Examples:**

1. **To set a bit of a number, you need to use Bitwise OR ( | ) operator along with the left shift operator.**

/\* setbit.c\*/

#include <stdio.h>

main()

{

intnum,pos;

printf("enter the number and position:\n");

scanf("%d %d ", &num, &pos);

num = num | ( 1 <<pos );

printf("After setting the bit the number is : %d",num);

}

OUTPUT:

Enter the number and position:

6

3

After setting the bit the number is :14

1. **To clear a bit of a number, you need to use Bitwise AND (&) operator along with the left shift operator**

/\* clearbit.c\*/

#include <stdio.h>

main()

{

intnum,pos;

printf("enter the number and position:\n");

scanf("%d %d ", &num, &pos);

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

printf("After clearing the bit the number is : %d",num);

}

OUTPUT:

Enter the number and position:

7

2

After setting the bit the number is :3

#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);

}

**Extracting information from a given set of bits**

For example let us assume that we need to extract the value present in the four bits (bit 11 to bit of an integer. Note that always bits are measured from 0. So first bit is called as 0th bit. This problem can be done in two ways. One is first shift the bits to right, so that 11th bit comes to the

first bit position. Next mask all other bits except these four bits by using & operation.

int val; /\* this integer contains the bits to be extracted \*/

int bitVal;

bitVal = (val >> 11) & 0xf;

Second way is to first mask all other bits, and then shift right.

bitVal = (val & 0x7800) >> 11;

In the above example the mask value (0xf or 0x7800) is hard coded to the requirement. But one

should be able prepare this mask value in run time. All the bit exercise problems expect you to

prepare the mask value in run time.

**Setting the given bits of an integer with given value**

The setting of bits should be done without affecting all other bits of the integer. For this we should

do the following steps:

* Clear all the bits we are going to set, to zero first(which are going to set)
* Shift the bits position to the required position.
* Bit wise or the bit value with original value

For example. let us assume that 'bitVal' contains 5 bit value, which should be placed in the bits 21 to 25 of 'val' integer.

val = val & ~0x3E00000 /\* same as val & 0xFC1FFFFF, which clears the bits to be set \*/

bitVal = bitVal << 21; /\* shift bits to the required position \*/

val = val | bitVal;

int getBitVal(unsigned int val, int bitNumber)

{

return( (val >>bitNumber) & 1);

}

int getBitsVal(unsigned int val, int startBitNumber, int noOfBits)

{

unsigned int mask;

mask = (1 << noOfBits) – 1;

return ( (val >> startBitNumber) & mask);

}

int setBitVal(unsigned int val, int bitNumber, int bitVal)

{

if(bitVal)

return(val | (1 << bitNumber));

else

return(val & ~(1 << bitNumber));

}

int setBitsVal(unsigned int val, int startBitNumber, int noOfBits, int bitVal)

{

unsigned int mask;

mask = (1 << noOfBits) – 1;

val = val & ~(mask << startBitNumber);

val = val | (bitVal << startBitNumber);

return val;

}

Assignments

i) Every program should have only one function. That is main().

ii) Whenever asked to read some number, it is assumed that you haveto define a variable and use scanf() function to read the number.

iii) Printing should be done by using printf() function and by usingexpression 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 excusive 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. Wherechar1 and char2 are the char variables into which user entered

characters are read.

**5.10 Precedence and associativity of operators**

For evaluation of expressions having more than one operator, there are certain precedence and associativity rules defined in C, Let us see what these rules are and why are they required.

Consider the following expression -

2 + 3\*5

Here we have two operators - addition and multiplication operators. If addition is performed before multiplication then result will be 25 and if multiplication is performed before addition then the result will be 17,

In C language, operators are grouped together and each group is given a precedence level. The precedence of all the operators is given in the following table. The upper rows in the table have higher precedence and it decreases as we move down the table. Hence the operators with precedence level 1 have highest precedence and with precedence level 15 have lowest precedence. So whenever an expression contains more than one operator, the operator with a higher precedence is evaluated first, For example in the above expression, multiplication will be performed before addition since multiplication operator has higher precedence

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Description** | **Precedence Level** | **Associativity** |
| ()  []  ->  . | Function call  Array Subscript  Arrow operator  Dot operator | 1 | Left to Right |
| +  -  ++  --  |  ~  \*  &  (datatype)  Sizeof | Unary plus  Unary Minus  Increment  Decrement  Logical NOT  One’s complement  Indirection  Address  Typecast  Size in Bytes | 2 | Right to Left |
| \*  /  % | Multiplication  Division  Modulus | 3 | Left to Right |
| +  - | Addition  Subtraction | 4 | Left to Right |
| <<  >> | Left Shift  Right Shift | 5 | Left to Right |
| <  <=  >  >= | Less than  Less than or equal to  Greater than  Greater than or equal to | 6 | Left to Right |
| ==  != | Equal to  Not equal to | 7 | Left to Right |
| & | Bitwise AND | 8 | Left to Right |
| ^ | Bitwise XOR | 9 | Left to Right |
| | | Bitwise OR | 10 | Left to Right |
| && | Logical AND | 11 | Left to Right |
| || | Logical OR | 12 | Left to Right |
| ? : | Conditional operator | 13 | Right to Left |
| =  \*= /= %=  += -= &= ^= |=  <<= >>= | Assignment operators | 14 | Right to Left |
| , | Comma operator | 15 | Left to Right |

**Chapter 6 - Control statements**

In c programs, statements are excited sequentially in the order in which they appear in the program. But sometimes we may want to use a condition for executing only a part of program. Also many situations arise where we may want to execute some statements several times. Control statements enable us to specify the order in which the various instructions in the program are to be executed. This determines the flow of control. Control statements define how the control is transferred to other parts of the program. C language supports four types of control statements, which are as-

1. if.....else
2. goto
3. switch
4. loop

* while
* do....while
* for

**6.1 Compound statement or Block**

A compound statement or a block is a group of statements enclosed within a pair of curly braces {}.

The statements inside the block are executed sequentially. The general form is-

{

statements1;

statements2;

...................;

}

What is the Output of the program

int main()

{

int x=10, y=3;

{

int x=100, z=10;

printf(“%d %d”,x,z);

}

printf(“%d %d”,x,z);

}

A compound statement is syntactically equivalent to a single statement and can appear anywhere in the program where a single statement is allowed.

**6.2 if......else**

This is a bi-directional conditional control statement. This statement is used to test a condition and take one of the two possible actions. If the condition is true then a single statement or a block of statements is executed (one part of the program), otherwise another single statement or a block of statements is executed (other part of the program). Recall that in c, any nonzero value is regarded as true while zero is regarded as false.

Syntax 1:

if(condition)

statement1;

if(condition)

{

statement1;

..............

}

There can be single statement or a block of statements after if part.

Here if the condition is true (nonzero) then statement1 is executed, and if it is false (zero), then the next statement which is immediately after the if control statement is executed.

Syntax 2:

if(condition)

statement1;

else

statement2;

if(condition)

{

statement1;

.........

}

else

{

statement2;

............

}

Here if the condition is true then statement1 is executed and if it is false then statement2 is executed. After this the control transfers to the next statement which is immediately after the if....else control statement.

**/\* program to print a message if negative number is entered\*/**

main()

{

int num;

printf (“enter a number”);

scanf(“%d”,&num);

if(num<0)

{

printf(“number entered is negative\n”);

}

printf(“value of num is:%d\n”,num);

}

**/\* program to print whether the number is even or odd\*/**

#include<stdio.h>

main()

{

int num;

printf(“enter a number”);

scanf(“%d”,&num);

if(num%2==0)

{

printf(“number is even\n”);

}

else

{

printf(“number is odd\n”);

}

}

**/\*program to check out the status of a bit whether it is clear or set\*/**

#include<stdio.h>

main()

{

int num, pos;

printf(“enter the number and position:\n”);

scanf (“%d %d”,&num, &pos);

if( (num & (1<<pos) ) = = 0)

printf(“%d bit is clear “, pos);

else

printf(“%d bit is set”, pos);

}

**/\*programs if else statements\*/**

int main()

{

// int status = 0;

int status = -1

// int status = 200;

if(status)

{

printf("i am in if\n");

}

else

{

printf("i am in else\n");

}

}

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(“print b”);

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

**6.3 Nesting of if....else**

We can have another if....else statement in the if block or the else block. This is called nesting of if....else statements. Here is an example of nesting where we have if....else inside both if block and else block.

if(condition 1)

{

if(condition 2)

statement A1;

else

statement A2;

}

else

{

if(condition 3)

statement B1;

else

statement B2;

}

**/\*program to find largest number from three given numbers\*/**

#include<stdio.h>

main()

{

int a,b,c,large;

printf(“enter three numbers:”);

scanf(“%d%d%d”,&a,&b,&c);

if(a>b)

{

if(a>c)

large=a;

else

large=c;

}

else

{

if(b>c)

large=b;

else

large=c;

}

Printf(“largest number is %d\n”,large);

}

The next program finds whether a given year is leap or not. A centennial (divisible by 100) year is leap if it is divisible by 400, and a non centennial year leap if it is divisible by 4.

**/\*program to find whether a year is leap or not\*/**

#include<stdio.h>

main()

{

int year;

printf(“enter year:”);

scanf(“%d”,&year);

if(year%4= =0 && year%100!=0 | | year%400= =0)

printf(“leap year\n”);

else

printf(“not leap\n”);

}

**6.4 else if ladder**

This is a type of nesting in which there is an if.....else statement in every else part except the last else part. This type of nesting is frequently used in programs and also known as else if ladder. Here each condition is checked, and when a condition is found to be true, the statements corresponding to that are executed, and the control comes out of the nested structure without checking remaining conditions. If none of the conditions is true then the last else part is executed.

**/\*program to find out the grade of a student when the marks of 4 subjects are given. The method of assigning grade is as-**

**per>=85 grade=A**

**per<85 and per>=70 grade=B**

**per<70 and per>=55 grade=C**

**per<55 and per>=40 grade=D**

**per<40 grade=E**

**here per is percentage.**\*/

#include<stdio.h>

main()

{

float m1,m2,m3,m4,total,per;

char grade;

printf(“enter marks of 4 subjects:”);

scanf(“%f%f%f%f”,&m1,&m2,&m3,&m4);

total=m1+m2+m3+,m4;

per=total/4;

if(per>=85)

grade = ‘A’;

else if (per>=70)

grade = ‘B’;

else if(per>=55)

grade = ‘C’;

else if(per>=40)

grade = ‘D’;

else

grade =’E’;

printf(“percentage is %f\n Grade is %c \n”,per,grade);

}

**6.5 loops**

Loops are used when we want to execute a part of the program or a block of statements several times. For example, suppose we want to print “c is the best 10 times. One way to get the desired output is- we write 10 printf statements, which is not preferable. Other way out us- use loop. Using loop we can write one loop statement and only one printf statement, and this approach is definitely better than the first one. There are 3 loop statements in c

1. while
2. do while
3. for

**6.5.1 While loop**

This while statement can be written as:

while(condition)

statement;

while(condition)

{

Statement;

............

}

Like if-else statement here also we have either a single statement or a block of statements, and here it is known as the body of loop. Now let’s see how this loop works. first the condition is evaluated if it is true then the statements in the body of loop are executed. After the execution, again the condition is checked and if it is found to be true then again the statements in the body of loop are executed. this means that these statements are executed continuously till the condition is true and when it becomes false, the loop terminate and the control comes out of the loop. Each execution of the loop body is known as iteration.

**/\*program to print the numbers from 1 to 10 using while loop\*/**

#include<stdio.h>

main()

{

int i=1;

while(i<=10)

{

Printf(“%d\t”,i);

i=i+1; /\*statement that changes the values of condition\*/

}

Print(“/n”);

}

Here initially the condition (i<=10) is true. After each iteration of the loop, value of 1 increases and when the value of i equals 11 the condition becomes false and the loop terminates.

**/\* Program to print the sum of digits of any number \*/**

#include<stdio.h>

main()

{

int n,sum=0,rem;

printf(“enter the number”);

scanf(“%d”,&n);

while(n>0)

{

rem=n%10; /\*taking last digit of number\*/

sum=sum+rem;

n=n/10; /\*skipping last digit\*/

}

printf(“sum of digits = %d\n”,sum);

}

Here we are extracting the digits of the number from right to left and then those digits are added one bye one to the variable sum. Note that the variable sum is initialised to 0.this is because we are adding some numbers to it, and if not initialized then these numbers will be added to garbage value present in it. Let’s see how the loop works when the value of n is 1452.

Before loop starts rem=garbage value, sum=0 n=1452

After 1st iteration rem=1452%10=2 sum=0+2=2 n=145

After2nd iteration rem=145%10=5 sum=2+5=7 n=14

After 3rd iteration rem=14%10=4 sum=7+4=11 n=1

After 4th iteration rem=1%10=1 sum=11+1=12 n=0

Now since the value of n is equal to zero, hence the condition (n>0) becomes false and the loop stops.

**6.5.2 do...while loop**

The ‘do....while’ statement is also used for looping. The body of this loop may contain a single statement or a block of statements. The syntax for writing this loop is:

do

statement;

while(condition);

do

{

statement1;

statement2;

}while(condition);

Here firstly the statements inside loop body are executed and then the condition is evaluated. If the condition is true, then again the loop body is executed and this process continues until the condition becomes false. Note that unlike while loop, here a semicolon is placed after the condition .if initially the condition is false the while loop will not execute at all, whereas the do while loop will always execute atleast once.

**6.5.3 for loop**

The ‘for’ statement is very useful while programming in c. It has three expressions and semicolons are used for separating these expressions. The ‘for’ statement can be written as-

for(expression1;expression2;expression3)

statement;

for(expression1;expression2;expression3)

{

statement;

statement;

}

The loop body can be a single statement or block of statements. Expression1 is an initialization expression, expression2 is a test expression or condition and expression3 is an updated expression. Expression1 is executed only once when the loop starts and is used to initialize the loop variables. This expression is generally an assignment expression. Expression2 is a condition and is tested before each iteration of the loop. This condition generally uses relational and logical operators. Expression3 is an update expression and is executed each time after the body of the loop is executed.

**/\*program to find the factorial of any number \*/**

#include<stdio.h>

main()

{

int n, num, fact=1;

printf(“enter the number:”);

scanf(“%d”,&n);

num=n;

if(n<0)

printf(“no factorial of negative number\n”);

else

{

for (; n>1;n--)

{

fact=fact\*n;

}

printf(“factorial of %d=%d\n”,num,fact);

}

}

**/\*program to print the binary format of a given integer \*/**

#include<stdio.h>

main()

{

int num,pos;

printf(“enter the number:\n”);

scanf(“%d”,&num);

for(pos=31;pos>=0;pos--)

{

if( (num &(1<<pos) )

printf(“0”);

else

printf(“1”);

}

}

**6.6 Nesting of loops**

When a loop is written inside the body of another loop, then it is known as nesting of loops. Any type of loop can be nested inside any other type of loop for example a for loop may be nested inside another for loop or inside a while or do while loop similarly while and do while loops can be nested.

**/\* program to understand nesting in for loop\*/**

main()

{

int i,j;

for(i=1;i<=3;i++)

{

printf(“i=%d\n”,i);

for(j=1;j< = 4;j++)

printf(“j=%d\t”,j);

printf(“\n”);

}

}

**/\*program to find the sum of digits of a number until the sum is reduce to 1 digit\*/**

#include<stdio.h>

main()

{

int num,dig,sum;

printf(“enter a number”);

scanf(“%d”,&num);

do

{

for(sum=0;num!=0;num/=10)

{

dig=num%10;

sum=sum+dig;

}

printf(“%d\t”,sum);

num=sum;

}while(num/10!=10);

printf(“\n”);

}

int main()

{

unsigned char ch;

for(ch=5;ch>=0;ch--)

printf("%d ",ch);

}

**6.7 infinite loops**

The loops that go on executing infinitely and never terminate are called infinite loops. Sometimes we write these loops by mistake while sometimes we deliberately make use of these loops in our programs. Let us take some examples and see what type of mistakes lead to infinite loops.

1. for(i=0;i<=5;i--)

printf(“%d”\n”,i);

This loop will execute till the value of i is less than or equal to 5, i.e. the loop will terminate only when it becomes greater than 5. The initial value of i is 0 and after each iteration its value is decreasing, hence it will never become greater than 5. So the loop condition will never become false, and the loop will go on executing infinitely. For the loop to work correctly we should write i++ instead of i--.

1. A common mistake made by beginners is to use the assignment operator (=) where equality operator (==) should have been used. if this mistake is made in the loop condition then it may cause the loop to execute infinitely. For example -

while(n=2)

{

....................

}

Here we wanted the loop to execute till the value of n is equal to 2.So we should have written n= =2 but mistakenly we have written n=2. Now n=2 is an assignment expression and the value of this expression is 2, which is a nonzero (true) value and hence the loop condition is always true.

**6.8 break statement**

Break statement is used inside loops and switch statements. Sometimes it becomes necessary to come out of the loop even before the loop condition becomes false. In such a situation, break statement is used to terminate the loop. This statement causes an immediate exit from that loop in which this statement appears. The break statement is generally written along with a condition. If break is written inside a nested loop structure then it causes exit from the innermost loop.

**/\*program to understand the use of break\*/**

#include<stdio.h>

main()

{

int n;

for(n=1;n<5;n++)

{

if(n==3)

{

printf(“i am using break\n”);

break;

}

Printf(“Number=%d\n”,n);

}

Printf(“out of for loop\n”);

}

**6.9 Continue Statement**

The continue statement i used when we want to go to the next iteration of the loop after skipping some statements of the loop. It is generally used with a condition. When continue statement is encountered all the remaining statements in the current iteration are not executed and the loop continues with the next iteration.

**/\*program to find the sum and average of positive integers\*/**

#include<stdio.h>

main()

{

int i=1,n,sum=0;

printf(“enter 10 positive numbers:\n”);

while(i<=10)

{

printf(“enter number %d: “,i);

scanf(“%d”,&n);

if(n<0)

{

printf(“enter only positive numbers\n”);

continue;

}

sum=sum+n;

i++;

}

avg=sum/10.0;

printf(“sum=%d avg=%f\n”,sum,avg);

}

**6.10 goto**

This is an unconditional control statement that transfers the flow of control to another part of the program. The goto statement can be used as

goto label;

..........

..........

label:

statement;

........ ....

Here label is any valid c identifier and it is followed by a colon. Whenever the statement goto label; is encountered, the control is transferred to the statement that is immediately after the label.

**/\*program to print whether the number is even or odd\*/**

#include<stdio.h>

main()

{

int n;

printf(“enter the number:”);

scanf(“%d”,&n);

if(n%2==0)

goto even;

else

goto odd;

even:

printf(“number is even\n”);

goto end;

odd:

printf(“number is odd\n”);

goto end ;

end: printf(“\n”);

}

The label can be placed anywhere. If the label is after the goto then the control is transferred forward and it is known as forward jump or forward goto, and if the label is before the goto then the control is transferred backwards and it is known as backward jump or backward goto. In forward goto, the statements between goto and label will not be executed and in backward goto statements between goto and label will be executed repeatedly.

**6.11 Switch**

This is a multi-directional conditional control statement. Sometimes there is a need in program to make choice among number of alternatives. For making this choice, we use the switch statement. This can be written as

switch(expression)

{

Case constant1: statement1

....................

Case constant2: statement2

...............

Default: statement

...............

}

Here switch, case and default are keywords. The “expression” following the switch keyword can be any c expression that yields an integer value. It can be value of any integer or character or a function call returning an integer, or arithmetic, logical, bitwise, relational expression yielding an integer. The constants following the case keywords should be of integer or character type. These constants must be different from one another.

We can’t use floating point or string constants. Multiple constants in a single case are not allowed; Each case should be followed by any only one constant.

Each case can be followed by any number of statements. It is also possible that a case has no statement under it. Now we’ll see some valid and invalid ways of writing case constants.

**Valid**

**Case 4: case ’a’: case 2+4: case ‘a’>’b’:**

**Invalid**

**Case “second”: case 2.3 case a: case a>b: case 2,4,5:**

**/\*program to understand the switch control statement\*/**

#include<stdio.h>

main()

{

int choice;

printf(“enter your choice”);

scanf(“%d”,&choice);

switch(choice)

{

case1:

printf(“first\n”);

case2:

printf(“second\n”);

case3:

printf(“third\n”);

default:

printf(“wrong choice\n”);

}

}

Here value of choice matches with second case so all the statements after case 2 are executed sequentially. The statements of case 3 and default are also executed in addition to the statements of case2. This is known as falling through cases. Suppose we don’t want to control to fall through the statements of all the cases under the matching case, then we can use break statement. If a break statement is encountered inside a switch, then all the statements following break are not executed and the control jumps out of the switch.

**/\*program to perform arithmetic calculations on integers\*/**

#include<stdio.h>

main()

{

char op;

int a,b;

printf(“enter the number operator and another number:”);

scanf(“%d%c%d”,&a,&op,&b);

switch(op)

{

case ‘+’:

printf(“result=%d\n”,a+b);

break;

case’-‘:

printf(“result=%d\n”,a-b);

break;

case’\*’:

printf(“result=%d\n”,a\*b);

case’/’:

printf(“result=%d\n”,a/b);

break;

default:

printf(“enter valid operator\n”);

}/\*end of switch\*/

}/\*end of main()\*/

**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 that 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 char" 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".

1. 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

1. 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.

**Chapter 7 – Functions**

A function is self-contained subprogram that is meant to do some specific, well-defined task. A C-program consists of one or more functions. If a program has only one function then it must be the main( ) function.

**7.1 Advantages of Using Functions**

* Generally a difficult problem is divided into sub problems and then solved. This divided and conquer technique is implemented in C through functions. A program can be divided into functions, each of which performs some specific task. So the use of C functions modularizes and divides the work of a program.
* When some specific code is to be used more than once and at different places the use of functions avoids repetition of that code.
* The program becomes easily understandable, modifiable and easy to debug and test. It becomes simple to write the program and understand what work is done by each part of program.
* Functions can be stored in a library and reusability can be achieved.

C programs have two types of Functions

1. Library Functions.
2. User-defined Functions.

**7.2 Library Functions**

C has the facility to provide library functions for performing some operations. These functions are present in the C library and they are predefined. The functions scanf( ) and printf( ) are input output library functions. Similarly we have functions like strlen( ), strcmp( ) for string manipulations.

To use a library function we have to include corresponding header file using the preprocessor directive #include. For example to use input input output functions like printf( ), scanf( ) we have to include stdio.h, for string library string.h should be included.

**7.3 User-Defined Functions**

Users can create their own function for performing any specific task of the program. These types of functions are called user-defined functions. To create and use these functions, we should know about these three things.

1. Function definition
2. Function declaration
3. Function call

Before discussing these three points we have written one simple program that will be used for reference.

**/\*Program to find the sum of two number\*/**

#include<stdio.h>

int sum(int x,int y);

main( )

{

int a,b,s;

printf(“enter the values of a and b:”);

scanf(“%d %d”, &a,&b);

s=sum(a,b);

printf(“Sum of %d and %d is %d\n”,a,b,s);

}

int sum(int x,int y)

{

int s;

s=x+y;

return s;

}

**7.4 Function Definition**

The function definition consists of the whole description and code of a function. It tells what the function is doing and what are its inputs and outputs. A function definition consists of two parts – a function header and function body. The general syntax of a function definition is

**return\_type func\_name(type1 arg1,type2 arg2,…….)**

**{**

**local variable declaration;**

**statement;**

**……………**

**return(expression);**

**}**

The first line in the function definition is known as the function header and after this body of the function is written enclosed in curly braces.

The return\_type denotes the type of the value that will be returned by the function. The return\_type is optional and if omitted, it is assumed to be int by default. A function can return either one value or no value. If a function does not return any value then void should be written in place of return\_type.

func\_name specifies the name of the function and it can be any valid C identifier. After function name, the argument declarations are given in parentheses, which mention the type and name of the arguments. These are known as formal arguments and used to accept value. A function can take any number of arguments or even no argument at all. If there are no arguments then either the parentheses can be left empty or void can be written inside the parentheses.

The body of the function is a compund statement, which consists of declarations of variables, and C statements followed by an optional return statement. The variables declared inside the function are known as local variables, since they are local to that function only, i.e. they have existence only in the function in which they are declared, they can not be used anywhere else in the program. These can be any number of statements inside the function body. The return statement is optional.

The function definition can be placed anywhere in the program. Note that a function definition cannot be placed inside another function definition. Function definitions can also be placed in different files.

**7.5 Function Call**

The function definition describes what a function can do, but to actually use it in the program the function should be called somewhere. A function is called by simply writing its name followed by the argument list inside the parentheses.

**func\_name(arg1,arg2,arg3,……);**

These arguments arg1, arg2, arg3,…… are called actual arguments.

When a function is called, the control passes to the called function, which is executed and after this control is transferred to the statement following the function call in the calling function.

If the function is declared as void then it cannot be used to return any value from the function.

**7.6 Function Declaration**

The calling function needs information about the called function. If definition of the function is placed before the calling function, then declaration is not needed.

The function declaration is known as the function prototype, and it informs the compiler about the following three things.

1. Name of the function
2. Number and type of arguments received by the function
3. Type of value returned by the function.

Function declaration tells the compiler that a function with these features will be defined and used later in the program. The general syntax of a function declaration is

**return\_type func\_name(type1,type2,…………..);**

**/\*Program that finds whether a number is even or odd\*/**

#inlcude<stdio.h>

void find(int);

main( )

{

int num;

printf(“enter a number:”);

scanf(“%d”,&num);

find(num);

}

void find(int n)

{

If(n%2==0)

Printf(“%d is even \n”,n);

else

printf(“%d is odd \n”,n);

}

**7.7 return statement**

The return statement is used in a function to return a value to the calling function. It may also be used for immediate exit from the called function to the calling function without returning a value.

This statement can appear anywhere inside the body of the function. There are two ways in which it can be used –

**return;**

**return(expression);**

Here return is a keyword. The first form of return statement is used to terminate the function without returning any value.

**/\*Program to understand the use of return statement\*/**

#include<stdio.h>

void funct(int, float);

main( )

{

int age;

float ht;

printf(“enter age and height: “);

scanf(“%d %f”, &age, &ht);

funct(age,ht);

}

void funct(int age,float ht)

{

If(age>25)

{

printf(“Age should be less than 25 \n”);

return;

}

If(ht<5)

{

printf(“height should be more than 5\n”);

return;

}

Printf(“Selected \n”);

}

The second form of return statement is used to terminate a function and return a value to the calling function. The value returned by the return statement may be any constant, variable, expression or even any other function.

**/\*Program to find out the factorial of a number\*/**

#include<stdio.h>

int factorial(int);

main( )

{

int num;

printf(“enter a number:”);

scanf(“%d”,&num);

if(num<0)

printf(“no factorial for negative number\n”);

else

printf(“Factorial of %d is %d \n”,num,factorial(num));

}

Int factorial(int n)

{

int I,fact=1;

if(n==0)

return 1;

for(i=n;i>1;i--)

fact=fact\*i;

return fact;

}

A function can return only one value. If no value is to be returned from the function then it should be declared as void. If the function is not void type and no value is returned through return statement, then a garbage value is returned automatically.

**/\*Program of a function that returning only one value\*/**

Int func(void);

int main()

{

     k= func();

   printf("k=%d\n",k);

}

Int func()

{

   int x=3,y=4;

   return x,y;

}

**7.8 Function arguments**

The calling function sends values to the called function for communication; these values are called arguments or parameters.

**7.8.1 Actual arguments**

The arguments which are mentioned in the function call are known as actual arguments, since these are the values which are actually sent to the called function. Actual arguments can be written in the form of variables, constants or expressions or any function call that returns a value. For example-

fun(x);

func(a\*b,c\*d+k);

func(22,43);

func(1,2,sum(a,b));

**7.8.2 Formal arguments**

The name of the arguments, which are mentioned in the function definition are called formal or dummy arguments since they are used just to hold the values that are sent by the calling function.

These formal arguments are simply like other local variables of the function which are created when the function call starts and are destroyed when the function ends. However there are two differences. First is that formal arguments are declared inside parentheses while other local variables are declared at the beginning of the function block. The second diffrence is that formal arguments are automatically initialized with the values of the actual arguments passed, while local variables are assigned values through the statements written inside the function body.

The order, number and type of actual arguments in the function call should match with the order, number and type of formal arguments in the function definition.

**7.9 Types of Functions**

The functions can be classified into four categories on the basis of the arguments and return value.

1. Functions with no arguments, and no return value.
2. Function with no arguments, and a return value.
3. Function with arguments and no return value.
4. Function with arguments and a return value.

**7.9.1 Functions with no arguments and no return value**

void func(void);

main( )

{

………….

func( );

………….

}

void func(void)

{

………….

statement1;

………….

}

In the above example, the function func( ) is called by main( ) and the function definition is written after the main( ) function. As the function func( ) has no arguments, main( ) cannot send any data to func( ) and since it has no return statement, hence function can not return any value to main( ). There is no communication between the calling and the called function.

**/\*Program that uses a function with no arguments and no return value\*/**

#include<stdio.h>

void message(void);

main( )

{

message ( );

printf(“Welcome”);

}

//function definition

void message(void)

{

Printf(“This is my first program using functions \n”);

}

**7.9.2 Function with No arguments but a return value**

These types of functions do not receive any arguments but they can return a value.

Int func(void);

main( )

{

Int r;

r = func( );

…………

}

Int func(void)

{

………....

…………

return(expression);

}

**/\*Program that returns the sum of squares of all odd numbers from 1 to 25\*/**

#include<stdio.h>

Int func(void);

main( )

{

printf(“%d\n”,func( ));

}

int func(void)

{

int num,s=0;

for(num=1;num<=25;num++)

{

if(num%2!=0)

s+=num\*num;

}

return s;

}

**Output:**

**2925**

**7.9.3 Function with arguments but no return value**

These types of function have arguments, hence the calling function can be send data to the called function but the called function does not return any value. These functions can be written as-

**void func(int,int);**

main( )

{

…………..

func(a, b);

………….

}

void func(int c, int d)

{

………....

statements

………….

}

Here a and b are actual arguments which are used for sending the values c and d are the formal arguments which accept values from the actual arguments.

**7.9.4 Functions with arguments and return value**

These types of functions have arguments, so the calling function can send data to the called function, it can also return any value to the calling function using return statement. This function can be written as

int func(int, int);

main( )

{

int r,a,b;

………

r=func(a,b);

………

}

int func(int c, int d)

{

……….

……….

return (expression);

}

**/\*Program to find the sum of the digits of any number\*/**

#include<stdio.h>

int sum(int);

main( )

{

int num;

printf(“Enter the number:”);

scanf(“%d”,&num);

printf(“Sum of digits of %d is %d \n”,num,sum(num));

}

int sum(int n)

{

int i,sum=0,rem;

while(n>0)

{

rem=n%10;

sum=sum+rem;

n/=10;

}

return (sum);

}

**7.10 Order of Evaluation of Function Arguments**

When a function is called with more than one argument, then the order of evlauation of arguments is unspecified. This order of evaluation is not important in function calls like multiply(m,n) or multiply(m+n,m-n). But if we have a function call like this

int m=3,k;

k=multiply(m,m++);

Now here if the first argument is evaluated first then value of k will be 9, and if the second argument is evaluated first, the value of k will be 12. But since the order of evaluationof arguments is unspecified in C and depends on compiler, hence the result may be different compilers.

**7.11 main( ) Function**

Execution of every C programs always begins with the function main( ). Each function is called directly or indirectly in main( ) and after all functions have done their operations, control returns back to main( ). There can be only one main( ) function in a program.

The main( ) function is a user defined function but the name, number and type of arguments are predefined in the language. The operating system calls the main function and main( ) returns a value of integer type to the operating system.

**7.12 Local and Global variables**

**7.12.1 Local Variables**

The variables that are defined within the body of a function or a block, are local to that function or block only are called local variables. For example

**func( )**

**{**

**int a,b;**

**………**

**………**

**}**

Here a and b are local variables which are defined within the body of the function func( ). Local variables can be used only in those functions or blocks, in which they are declared. The same variable name may be used in different functions. For example

**func1( )**

**{**

**int a=2,b=4;**

**……………**

**……………**

**}**

**func2( )**

**{**

**int a=15,b=20;**

**………….**

**………….**

**}**

Here the values of a=2, b=4 are local to the function func1( ) and a=15,b=20 are local to the function func2( ).

**7.12.2 Global variables**

The variables that are defined outside any function are called global variables. All functions in the program can access and modify global variables. It is useful to declare global if it is to be used by many functions in the program. Global variables are automatically initialized to 0 at the time of declaration.

**/\*Program to understand the use of global variables\*/**

#include<stdio.h>

void func1(void);

void func2(void);

int a, b=6;

main( )

{

printf(“Inside main( ): a= %d, b=%d\n”,a,b);

func1( );

func2( );

}

void func1(void)

{

printf(“Inside func1( ):a=%d, b=%d\n”, a, b);

}

Void func2(void)

{

Int a=8;

printf(“Inside func2( ):a=%d, b=%d\n”, a, b);

}

**Output:**

**Inside main( ):a=0,b=6**

**Inside func1( ):a=0,b=6**

**Inside func2( ):a=8,b=6**

**/\*Program2\*/**

int x=10;

int main()

{

    int x=5;

    printf("x=%d\n",x);

    {

       int z=10;

       printf("z=%d\n",z);

   }

      printf("z=%d\n",z);

}

**7.13 Recursion**

To implement recursion technique in programming, a function should be capable of calling itself and this facility is available in C. The function that calls itself(inside function body) again and again is known as a recursive function. In recursion the calling function and the called function are same. For example-

**main( )**

**{**

**rec( );**

**……..**

**……..**

**}**

**rec( )**

**{**

**……..**

**rec( );**

**……..**

**}**

Here rec( ) is called inside the body of function rec( ). There should be a terminating condition to stop recursion, otherwise rec( ) will keep on calling itself infinitely and will never return.

Before writing a recursive function for a problem we should consider these points-

1. We should be able to define the solution of the problem in terms of similar type of smaller problem. At each step we get closer to the final solution of our original problem.
2. There should be terminating condition to stop recursion.

**/\*Program to find the factorial of a number by recursive method\*/**

#include<stdio.h>

int fact(int);

main( )

{

Int num;

printf(“Enter a number:”);

scanf(“%d”,&num);

printf(“Factorial of %d is %d\n”,num,fact(num));

}

Int fact(int n)

{

if(n==0)

return(1);

else

return(n\*fact(n-1));

}

This function returns 1 if the argument n is 0, otherwise it returns n\*fact(n-1). To return n\*fact(n-1), the value of fact(n-1) has to be calculated for which fact( ) has to be called again but this time with an argument of n-1. This process of calling fact( ) continues till it is called with an argument of 0.

Suppose we want to find out the factorial of 5.

**Initially main( ) calls fact(5)**

**Since 5>0,fact(5) calls fact(4)**

**Since 4>0,fact(5) calls fact(4)**

**Since 3>0,fact(5) calls fact(4)**

**Since 2>0,fact(5) calls fact(4)**

**Since 1>0,fact(5) calls fact(4)**

When factorial( ) is called with n=0 then the condition inside if statement becomes true, so now the recursion stops and control returns to factorial(1). Now every called function will return the value to the previous function. These values are returned in the reverse order of function calls.

**/\*Program to generate Fibonacci series using recursion\*/**

#include<stdio.h.

int fib(int);

main( )

{

int nterms,I;

printf(“Enter number of terms:”);

scanf(“%d”,&nterms);

for(i=0;i<nterms;i++)

printf(“%d”,fib(i));

printf(“\n”);

}

int fib(int n)

{

if( n==1)

return(1);

else

return(fib(n-1)+fib(n-2));

}

**7.13.1 Advantages and Disadvantages of Recursion**

The use of recursion makes the code more compact and elegant. It simplifies the logic and hence makes the program easier to understand. But the code written using recursion is less efficient since recursion is a slow process because of many function calls involved in it.

Most problems with recursive solutions also have an equivalent non-recursive solution. A non-recursive solution increases performance while a recursive solution is simpler.

**Applications of Recursion:**

1. Dynamic programming, Back tracking

2. Divide and Conquer (Merge sort, quick sort ..)

3. Towers of Hanoi, Linux search (DFS)

#include<stdio.h>

void fun(int n)

{

if(n == 0)

return;

fun(n – 1);

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

}

int main() {

fun(3);

return 0;

}

int fun(int n)

{

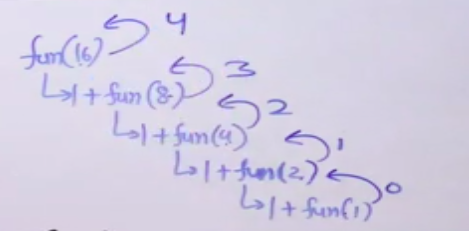
if(n == 1)

return 0;

else

return 1+fun(n/2);

}



void fun(int n)

{

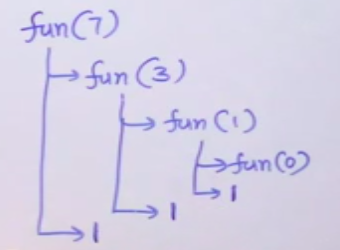
if(n == 0)

return;

fun(n/2);

printf("%d\n",n%2);

**}**



\* print n to 1 using recursion

\* print 1 to n using recursion

**7.14 Storage Classes**

In addition to data type, each variable has one more attribute known as storage class. The proper use of storage classes makes our program efficient and fast. In larger multi file programs the knowledge of storage classes is indispensable. We can specify a storage class while declaring a variable. The general syntax is-

**storage\_class datatype variable\_name;**

There are four types of storage classes-

1. Automatic
2. External
3. Static
4. Register

So we may write declaring statements like this-

auto int x, y;

static float d;

register int z;

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

A storage class decides about these four aspects of a variable-

1. Lifetime - Time between the creation and destruction of a variable.
2. Scope – Location where the variable is available for use.
3. Initial value – Default value taken by an uninitialized variable.
4. Place of storage – Place in memory where the storage is allocated for the variable.

**7.14.1 Automatic**

All the variables declared inside a block/function without any storage class specifier are called automatic variables. We may also use the keyword auto declare automatic variables, although this is generally not done.

The uninitialized automatic variables initially contain garbage value. The scope of these variables is inside the function or block in which they are declared and they can’t be used in any other function/block.

They are named automatic since storage for them is reserved automatically each time when the control enters the function/block and are released automatically when the function/block terminates. For example-

**/\*Program to understand automatic variables\*/**

#include<stdio.h>

main( )

{

func( );

func( );

func( );

}

func( )

{

int x=2,y=5;

printf(“x=%d,y=%d”,x,y);

x++;

y++;

}

**Ouput:**

**x=2,y=5**

**x=2,y=5**

**x=2,y=5**

Here when the function func( ) is called first time, the variables x and y are created and initialized, and when the control returns to main( ), these variables are destroyed. When the function func( ) is called for the second time, again these variables are created and initialized and are destroyed after execution of the function. So automatic variables come into existence each time the function is executed and are destroyed when the function terminates.

Since automatic variables are known inside a function or block only, so we can have variables of same name in different functions or blocks without any conflict. For example in the following program the variable x is used in different blocks(here blocks consists of function body) without any conflict.

**/\*Program to understand automatic variables\*/**

#include<stdio.h>

main( )

{

int x= 5;

printf(“x=%d\t”, func( ));

}

func( )

{

int x=15;

printf(“x=%d\n”, x);

}

Output:

x=5 x=15

Here the variables x declared inside main( ) is different from the variable x declared inside the function func( ).

**7.14.2 External**

The variables that have to be used by many functions and different files can be declared as external variables. The initial value of an uninitialized external variable is zero.

Before studying external variables, let us first understand the difference between their definition and declaration. The declaration of an external variable declares the type and name of the variable, while the definition reserves storage for the variables as well as behaves as a declaration. The keyword extern is specified in declaration but not in definition. For example the definition of an external variable salary will be written as-

**float salary;**

Its declaration will be written as-

**extern float salary;**

The following points will clarify the concept of definition and declaration of an external variable.

**Definition of an external variable**

1. Definition creates the variable, so memory is allocated at the time of definition.
2. There can be only one definition.
3. The variable can be initialized with the definition and initialize should be constant.
4. The keyword extern is not specified in the definition.
5. The definition can be written only outside functions.

**Declaration of an external variable**

1. The declaration does not create the variable, it only refers to a variable that has already been created somewhere, so memory is not allocated at the time of declaration.
2. There can be many declarations.
3. The variable cannot be initialized at the time of declaration.
4. The keyword extern is always specified in the declaration.
5. The declaration can be placed inside functions also.

Consider this program:

**#include<stdio.h>**

**int x=8;**

**main( )**

**{**

**…………..**

**}**

**func1( )**

**{**

**………….**

**}**

**func2( )**

**{**

**………..**

**}**

In this program the variable x will be available to all functions, since an external variable is active from the point of its definition till the end of a program.

Till now we had written our program in a single file. When the program is large it is written in different files and these files are compiled separately and linked together afterwards to form an executable program. Now we will consider a multi file program, which is written in three files viz. first.c, second.c and third.c.

**first.c second.c third.c**

**int x=8; extern int x; func4( )**

**main( ) func2( ) {**

**{ { …….**

**……. …… }**

**} } func5( )**

**func1( ) func3( ) {**

**{ { ………**

**…….. ….. }**

**} }**

Here in the file first.c an external variable x is defined and initialized. This variable can be used both in main( ) and func1( ) and it can be accessible to other files. In the file second.c to access this variable then we used the declaration in this file as extern int x;

Suppose our program consists of many files and in file first.c, we have defined many variables that may be needed by other files also. We can put an extern declaration for each variable in every file that needs it. Another better and practical approach is to collect all extern declarations in a header file and include that header file in the files, which require access to those variables.

**7.14.3 Static**

There are two types of static variables

1. Local static variables
2. Global static variables

**7.14.3.1 Local Static Variables**

The scope of a local static variable is same as that of an automatic variable i.e. it can be used only inside the function or block in which it is defined.The lifetime of a static variable is morethan that of an automatic variable. A static variable is created at the compilation time and it remains alive till the end of a program. It is not created at the compilation time and control enters a “function/block”. Hence a static variable is created only once and its value is retained between function calls. If it has been initialized, then the initialization value is placed in it only once at the time of creation. It is not initialized each time the function is called.

If a static variable is not explicitly initialized then by default it takes initial value zero.

**int x=8;**

**int y=x;/\*Valid\*/**

**static int z=x; /\*Invalid, Initializer should be constant\*/**

**/\*Program to understand the use of local static variables\*/**

#include<stdio.h>

main( )

{

func( );

func( );

func( );

}

func( )

{

static int flag=0;

if(x>9)

{ flag=1;

}

static int x=2,y=5;

printf(“x=%d,y=%d\n”,x,y);

x++,y++;

}

**Output:**

**x=2,y=5**

**x=3,y=6**

**x=4,y=7**

Note that the effect of initialization is seen only in the first call. In subsequent calls initialization is performed and variables x and y contain values left by the previous function call.

The next program uses a recursive function to find out the sum of digits of a number. The variable sum taken inside function sumd( ) should be taken as static.

**/\*Program to find out the sum of digits of a number recursively\*/**

# include<stdio.h>

int sumd(int num);

main( )

{

int num;

printf(“enter a number”);

scanf(“%d”,&num);

printf(“Sum of digits of %d is %d\n”,num,sumd(num));

}

int sumd(int num)

{

static int sum=0;

if(num>0)

{

sum=sum+(num%10);

sumd(num/10);

}

**7.14.3.2 Global Static Variables**

If a local variable is declared as static then it remains alive throughout the program. In case of global variables, the static specifier is not used to extend the lifetime since global variables have already a lifetime equal to the life of program. Here the static specifier is used foe information hiding. If an external variable is defined as static, then it can’t be used by other files of the program. So we can make an external variable private to a file by making it static.

first.c second.c third.c

int x=8; extern int x; func4( )

static int y=10 func2( ) {

main( ) { ……

{ ….. }

….. } func5( )

} func3( ) {

func1( ) { ……

{ ….. }

…. }

}

Here the variables y is defined as a static external variable, so it can be used only in the first.c. We can’t use it in other files by putting extern declaration for it.

**7.14.4 Register**

Register storage class can be applied only to local variables. The scope, lifetime and initial value of register variables are same as that of automatic. The only difference between the two is in the place where they are stored. Automatic variables are stored in memory while register variables are stored in CPU registers. Registers are small storage class for faster processing. For example the variables used as loop counters may be declared as register variables since they are frequently used.

**/\*Program to understand the use of register variable\*/**

#include<stdio.h>

main( )

{

register int i;

for(i=0;i<20000;i++)

printf(“%d\n”,i);

}

Register variables don’t have memory address so we can’t apply address operator (&) to them. There are limited number of registers in the processor hence we can declare only few variables as register. If many variables are declared as register and the CPU registers are not available then compiler will treat them as automatic variables.

The register storage class specifier can be applied to formal arguments of a function while the other three storage class specifiers can’t be used in this way.

The register storage class specifier can be applied to formal arguments of a function while the other three storage class specifiers can’t be used in this way.

**7.14.5 Storage Classes in Functions**

The storage class specifiers extern and static can be used with function definitions. The definition of a function without any storage specifier is equivalent to its definition with the keyword extern i.e. by default the definition of a function is considered external. If a function is external then it can be used by all the files of the program and if it is static then it can be used only in the file where it is defined.

first.c second.c third.c

main( ) extern float func1(int); void func4( )

{ func2( ) {

….. { ……

} ….. }

float func1(int) } void func5( )

{

…… static int func3(int ) {

} { ……

….. }

}

Here the function func1( ) is defined in file first.c. Its declaration is put in file second.c, so it can be used in this file also. The function func3( ) in file second.c is defined as static so it can’t be used by any other file. Generally declarations of all functions are collected in a header file and that header file is included in other files.

**7.14.6 Linkage**

There are three types of linkages in C

1. External linkage
2. Internal linkage
3. No linkage

Local variables have no linkage, so their scope is only within the block where they are declared. Global variables and functions have external linkage, so they can be used in any file of the program. Static global variables and static functions have internal linkage, so their scope is only in the file where they are declared.

**7.15 Memory During Program Execution**

The register variables are stored in CPU registers and rest of the variables are stored in memory. Now let us see how memory is organized when a C program is run.

**7.15.1 Code Area**

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

**7.15.2 Data Area**

This area stores static and global variables. It is further subdivided into two areas viz. initialized data area and uninitialized data area. The initialized data area stores all the initialized static and global variables while the uninitialized data area stores all the uninitialized static and global variables. The reason for division of this area is that all uninitialized variables can be collectively assigned value zero. The size of data area is also fixed and does not change during run time.

**Code Code of the program**

**Initialize Data Initialized static variables**

**Initialized global variables**

**Uninitialized Data Uninitialized static variables**

**Uninitialized global variables**

**Heap Dynamically allocated memory**

**Stack Initialized and Uninitialized automatic variable**

**7.15.3 Heap**

This area is used for dynamically allocated memory. It is the responsibility of the programmer to allocate memory from the heap. The size of this area is dynamic i.e. it may change during run time.

**7.15.4 Stack**

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

The following table summarizes all the features of storage classes-

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Keyword** | **Place of declaration** | **Lifetime** | **Scope** | **Initial Value** | **Place of storage** | **Linkage** |
| auto  register  static  (local)  (none)  Extern  Static  (global) | Inside Function/Block  Inside Function/Block  Inside Function/Block  Outside function (Definition of external variable)  Outside or inside function (Declaration of external variable)  Outside  function | Function/Block  Function/Block  Program  Program  Program  /Function  Program | Function/Block  Function/Block  Function/Block  Definition to end of file (can be shared in other files using extern declaration)  Declaration to end of file  Definition to end of file (can’t be shared in other files using extern declaration) | Garbage  Garbage  Zero  Zero  Zero | Memory  (stack)  Registers  Memory  (data area)  Memory  (data area)  Memory  (data area)  Memory  (data area) | None Variable  None  None  External  External  Internal |

**/\*Program on Extern\*/**

…..main.c

static int x=5;

static void display()

{

  printf("happy\n");

}

int main()

{

  f1();

  f2();

  display();

  printf("main:%d\n",x);

}

……..f1.c

static void display()

{

  printf("f1 is happy\n");

}

void f1()

{

  x=100;

  printf("i m in f1:%d\n",x);

  display();

}

……..f2.c

static void display()

{

  printf("f2:happy\n");

}

void f2()

{

  printf("i m in f2:%d\n",x);

  display();

}

**7.16 Functions with Variable Number of Arguments**

In the functions that we had created till now, the number and data type of arguments was fixed at the time of function definition. In some situations we may need functions that can accept variable number of arguments of different types. The library functions printf( ) and scanf( ) are examples of these type of functions. We have already used these functions that can accept variable number of arguments.

The header file stdarg.h provides the facilities needed to define functions with variable numbers of arguments. This file defines a new type called va\_list and three macros va\_start,va\_arg,va\_end that can operate on variables of this new type.

**Type**

va\_list-Used to declare argument pointer variables.

**Macros**

va\_start - Initializes the argument pointer variable.

Va\_arg - Retneves the current argument and increments the argument pointer.

Va\_end - Assigns NULL to argument pointer

A function that accepts variable number of arguments should be defined with ellipsis(….) at the end or argument list. The ellipsis should occur only at the end of argument list and there should be at least one fixed argument. For example-

func(char \*str,in num,…..)

{

……….

……….

}

Here func( ) takes two fixed arguments viz. str,num and after that it can accept any number of arguments. For example all these calls of func( ) are valid-

func(“Chennai”,40,67.89,’P’,”Madras”,23,67);

func(“Lucknow;”,35,66);

func(“Barelly”,30,’x’,20,39,12.5);

In all these calls first two arguments are always a string and an integer, rest of the arguments can be of any type. Now inside the function definition we can access the fixed arguments using their name but the remaining arguments don’t have a name so they are accessed using the macros defined in stdarg.h file. This is why fixed arguments are known as named arguments and variable number of arguments are known as unnamed arguments.

Now we’ll see how we can access these unnamed arguments inside the function body. Initially we’ll declare a variable of type va\_list.

**va\_list ap;**

This variable is conventionally named ap. Here ap is known as known as argument pointer and will be used to point to the unnamed arguments. The macro va\_start initializes ap and makes it point to the first unnamed argument passed to the function. This macro takes two arguments, first one is the argument pointer ap and second one is the name of the last fixed argument passed to the function (i.e. argument which is just before the ellipsis). For example in the above function func( ), va\_start will be called as-

**va\_start(ap,num);**

Now we can access individual variable arguments sequentially by using va\_arg. This macro takes ap and data type of the current argument.

**arg=va\_arg(ap,datatype);**

It returns the value of the current argument and increments the pointer ap so that it points to the next argument. After calling va\_start, the first call of va\_arg returns first unnamed argument, second call returns second unnamed argument and so on. If datatype of the current unnamed argument in the function call does not match with the datatype in va\_arg then the behaviour is undefined.

The macro va\_end should be called before existing from the function. This macro sets the argument pointer to NULL.

**Va\_end(ap);**

The unnamed arguments can’t be used after calling va\_end. If we wnt to use those arguments then once again we’ll have to initialize ap with va\_start.

The whole procedure is summarized in these steps

1. Include the header file stdarg.h
2. The function header should contain ellipsis to denote the variable argument list.
3. Declare a variable of type va\_list
4. Initialize this argumnent pointer using va\_start,so that it points to the first unnamed argument.
5. Use va\_arg to retrieve the value of arguments.
6. Call the macro va\_end when you have finished working with these arguments.

There is no facility to count how many arguments were passed in the function call and what was the type of each argument. It is programmer’s responsibility to pass this information to the function through fixed arguments. For example we may decide to take the first fixed argument as an integer that represents the total number of unnamed arguments. Generally a format string is passed as a fixed argument, which contains information about the type of each argument. For example printf( ) uses the format string that conatins conversion specifications, which denote the data type of each unnamed argument.

There is no facility to count how many arguments were passed in the function call and what was the type of each argument. It is programmer’s responsibility to pass this information to the function through fixed arguments. For example we may decided to take the first fixed argument as an integer that represents the total number of unnamed arguments. Generally a format string is passed as a fixed argument, which contains information about the type of each argument. For example printf( ) uses the format string that conatins conversion specifications, which denote the data type of each unnamed argument.

In the following program we will make a function sum( )which returns the sum of integers passed to it.

**/\*Program to find out the sum of Integers\*/**

#include<stdio.h>

#include<stdarg.h>

int sum(int,…);

main( )

{

printf(“Total=%d\n”,sum(2,99,68));

printf(“%d\n”,sum(3,11,79,32));

printf(“%d\n”,sum(5,23,34,45,56,78));

}

int sum(int num,….)

{

int I;

va\_list ap;

int arg,total=0;

va\_start(ap,num)

for(i=0;i<num;i++)

{

arg=va\_arg(ap,int);

printf(“%d”,arg);

total+=arg;

}

va\_end(ap);

return total;

}

**Output:**

**99 68 Total =167**

**11 79 32 Total=122**

**23 34 45 56 78 Total=236**

Here we have called the function sum with different number of arguments. We have taken only one fixed argument num that represents the number of unnamed arguments passed to the function. It is used in the loop to step through the unnamed arguments. Inside the function, firstly a variable of va\_list type is declared. Then va\_start is called and num is passed to it since it is the last fixed argument. After that each variable argument is obtained by calling va\_arg repeatedly inside a loop.

**CHAPTER 8 – ARRAYS & POINTERS**

The variables that we have used till now are capable of storing only one value at a time. Consider a situation when we want to store and display the age of 100 employees. For this we have to do the following-

1. **Declare 100 different variables to store the age of the employees.**
2. **Assign a value to each variable.**
3. **Display the value of each variable.**

Although we can perform our task by the above three steps but just imagine how difficult it would be to handle so many variables in the program and the program would become very lengthy. The concept of arrays is useful in these types of situations where we can group similar type of data items.

An array is a collection of similar type of data items and each data item is called an element of the array. The data type of the elements may be any valid data type like char,int or float. The elements of array share the same variable name but each element has a different index number known as subscript.

For the above example we can take an array variable age[100] of type int. the size of this array variable is 100 so it is capable of storing 100 integer values. The individual elements of this array are

**age[0],age[1],age[2],………..age[98],age[99]**

arrays can be single dimensional or multidimensional. The number of subscripts determines the dimension of array. A one-dimensional array has one subscript, two dimensional array has two subscripts and so on.

**8.1 One Dimensional Array**

8.1.1 Declaration of 1-D array

Like other simple variables, arrays should also be declared before they used in the program. The syntax for declaration of an array is –

***data\_type array\_name[size]***

Here array\_name denotes the name of the array and it can be any valid C identifier, data\_type is the data tye of the elements of the array. The size of the array specifies the number if elements that can be stored in the array.

Some examples of array declarations-

int age[100];

float sal[5];

char grade[20];

**8.1.2 Accessing 1-D Array Elements**

Specifying the array name followed by subscript in brackets can access the elements of an array. In C, the array subscripts starts from 0. Hence if there is an array of size 5 then the valid subscripts will be from 0 to4. The last valid subscript is one less than the size of the array.Let us take an array-

Int arr[5]; /\*size of array is 5, can hold five integer elements\*/

The elements of thus array are –

arr[0], arr[1], arr[2], arr[3], arr[4]

a subscripted array element is treated as any other variable in the program. We can store values in them, print their values or perform any operation that is valid for any simple variable of the same data type. For example if arr[5] and arr[10] are two arrays then these ara valid statements-

*int arr[5];*

*float sal[10];*

*int i=1;*

*scanf("%d",&arr[i]); /\*input value into arr[1]\*/*

*printf("%f",sal[3]); /\*Print value of sal[3]\*/*

*arr[4]=25; /\*assign a value to arr[4]\*/*

*arr[4]++; /\*increment the value of arr[4] by 1\*/*

*sal[5]+=200; /\*add 200 to sal[5]\*/*

*sum=arr[0]+arr[1]+arr[2]+arr[3]+arr[4]; /\*Add all the values of arr[5]\*/*

*i=2;*

*printf("%f",sal[i++]); /\*print value of sal[2] and increment the value of I\*/*

**8.1.3 Processing 1-D Arrays**

For processing arrays we generally use a for loop and the variable is used at the place of subscript. The initial value of loop variable is taken 0 since array subscripts from zero. The loop variable is increased to 1 each time so that we can access and process the next element in the array and in each pass we will process one element.

Suppose arr[10] is an array of int type-

1. ***Reading values in arr[10]***

***for(i=0;i<10;i++)***

***scanf(“%d”,&arr[i]);***

1. ***Displaying values of arr[10]***

***For(i=0;i<10;i++)***

***Printf(“%d”,arr[i]);***

*/\*Program to add the elements of an array\*/*

*#include<stdio.h>*

*main()*

*{*

*int arr[10],i,sum=0;*

*for(i=0;i<10;i++)*

*{*

*printf("Enter the value for arr[%d];",i);*

*scanf("%d",&arr[i]);*

*sum+=arr[i];*

*}*

*printf("Sum=%d\n",sum);*

*}*

**8.1.4 initialization of 1-D Array**

After declaration, the elements of a local array have garbage value while the elements of global and static arrays are automatically initialized to zero. We can explicitly initialize arrays at the time of declaration. The syntax for initialization of an array is –

***data\_type array\_name[size]={value1,value2,value3,…..valueN}***

Here array\_name is the name of the array variable, size of the array and value1,value2, valueN are the constant values known as initializers, which are assigned to the array elements one after another. These values are separated by commas and there is a semicolon after the ending braces. For example –

***int marks[5] = {5, 85, 45, 63, 78}***

the values of the array elements after this initialization are –

***marks[0] = 50, marks[1]= 85, marks[2]= 45, marks[3]= 63, marks[4]= 78***

while initializing 1-D arrays, it is optional to specify the size of the array. Of this size is omitted during initialization then the compiler assumes the size of the array equal to the number of initializers. For example –

***int marks[5]= {43,68}***

the value of the elements are as –

***marks[0]=43, marks[1]=68, marks[2]=0, marks[3]=0, marks[4]=0***

so if we initialize an array like this

***int arr[100]={0};***

then all the elements of arr will be initialized to zero.

*/\*Program to find the maximum and minimum numbers in an array\*/*

*#include<stdio.h>*

*main()*

*{*

*int i,j,arr[10]={2,5,4,1,8,9,11,6,3,11};*

*int min,max;*

*min=max=arr[0];*

*for(i=1;i<10;i++)*

*{*

*if (arr[i]<min)*

*min=arr[i];*

*if (arr[i]>max)*

*max=arr[i];*

*}*

*printf("Minimum =%d, Maximum=%d\n",min,max);*

*getch();*

*}*

*OUTPUT :*

*Minimum =1, Maximum = 11*

We have taken the value of first element as the initial value of min and max. inside the for loop, we’ll start comparing from second element onwards so this time we have started the loop from 1 instead of 0.

*/\* Program to reverse the elements of an array \*/*

*#include<stdio.h>*

*main()*

*{*

*int i,j,temp,arr[10]={1,2,3,4,5,6,7,8,9,10};*

*for(i=0,j=9;i<j;i++,j--)*

*{*

*temp=arr[i];*

*arr[i]=arr[j];*

*arr[j]=temp;*

*}*

*printf("After reversing the array is : ");*

*for(i=0;i<10;i++)*

*printf("%d",arr[i]);*

*printf("\n");*

*}*

*OUTPUT:*

*After reversing the array is : 10 9 8 7 6 5 4 3 2 1*

In the for loop we have used comma operator and taken two variables I and j. The variable ‘i’ is initialized with the lower bound and ‘j’ is initialized with the upper bound. After each iteration of the loop, ‘I’ is incremented while ‘j’ is decremented. Inside the loop, a[i] is exchanged to a[j]. so a[0] will be exchanged with a[9], a[1] with a[8], a[2] with a[7] and so on.

**8.2 1-D Arrays and Functions**

**8.2.1 Passing Individual Array Elements to a Function**

We can pass individual array elements as arguments to a function like other simple variables.

*/\*****Program to pass array elements to a function as arguments and find out whether even or odd*** *\*/*

*#include<stdio.h>*

*void check(int);*

*main()*

*{*

*int arr[10],i;*

*printf("Enter the array elements:");*

*for(i=0;i<10;i++)*

*{*

*scanf("%d",&arr[i]);*

*check(arr[i]);*

*}*

*}*

*void check(int num)*

*{*

*if(num%2==0) if((num &0x1)==0)*

*printf("%d is even \n",num);*

*else*

*printf("%d is odd \n",num);*

*}*

**8.2.2 Passing whole 1-D Array to a Function**

We can pass whole array as an actual argument to a function. The corresponding formal argument should be declared as an array variable of the same data type.

***main()***

***{***

***Int arr[10];***

***……………..***

***func(arr); /\*In function cal, array name is specified without brackets\*/***

***}***

***func(int val[10]);***

***{***

***-------***

***-------***

***}***

It is optional to specify the size of the array in the formal argument. We have studied that changes made in formal arguments donot affect the actual arguments, but this is not the case while passing an array to a function. The mechanism of passing an array to a function is quite different from that of passing simple variable. When an array is passed as an actual argument, the called function actually gets access to the original array and works on it, so any changes made inside the function affect the original array.

***/\*program to understand the effect of passing an array to a function\*/***

*#include<stdio.h>*

*void func(int[],int);*

*main()*

*{*

*int i,arr[6]={1,2,3,4,5,6};*

*func(arr,6);*

*printf("Contents of array are now: ");*

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

*printf("%d",arr[i]);*

*printf("\n");*

*}*

*func(int val[],int n)*

*{*

*int sum=0,i;*

*for(i=0;i<n;i++)*

*{*

*val[i]=val[i]\*val[i];*

*sum=sum+val[i];*

*}*

*printf("The sum of squares = %d\n",sum);*

*}*

*OUTPUT:*

*The sum of squares = 91*

*contents of array are now: 1 4 9 16 25 36*

Here we can see that the changes made to the array inside the called function are reflected in the calling function. The name of the formal argument is different but it refers to the original array.

**8.3 Two Dimensional Array**

**8.3.1 Declaration and Accessing Individual Elements of a 2-D array**

The syntax of declaration of a 2-D array is similar to that of 1-D arrays, but here we have two subscripts.

***data\_type array\_name[rowsize][coloumnsize];***

Here rowsize specifies the number if rows and coloumnsize represents the number of coloumns in the array. The total number of elements in the array are rowsize\*coloumnsize. For Example-

***int arr[4][5];***

Here arr is a 2-D array with 4 rows and 5 coloumns. Applying two subscripts, where the first subscript denotes the row number and the second subscript denotes the coloumn number, can access the individual elements of this array. The starting element of this array is arr[0][0] and the last element is arr[3][4]. The total number of elements in this array is 4\*5=20.

***Col 0 Col 1 Col 2 Col 3 Col 4***

***Row 0 arr[0][0] arr[0][1] arr[0][2] arr[0][3] arr[0][4]***

***Row 1 arr[1][0] arr[1][1] arr[1][2] arr[1][3] arr[1][4]***

***Row 2 arr[2][0] arr[2][1] arr[2][2] arr[2][3] arr[2][4]***

***Row 3 arr[3][0] arr[3][1] arr[3][2] arr[3][3] arr[3][4]***

**8.3.2 Processing 2-D Arrays**

For processing 2-D arrays, we use two nested for loops. The outer for loop corresponds to the row and the inner for loop corresponds to the coloumn.

***int arr[4][5];***

1. ***Reading Values in arr***

***for(i=0; i<4 ;i++)***

***for(j=0; j<5 ;j++)***

***scanf(“%d”,&arr[i][j]);***

1. ***Displaying values of arr***

***for(i=0; i<4 ;i++)***

***for(j=0; j<5 ;j++)***

***printf(“%d”,&arr[i][j]);***

***/\*Program pg no45 to display a matrix\*/***

***/\* Program to input and display a matrix – 8\_5.c\*/***

*#include<stdio.h>*

*main()*

*{*

*int mat[3][4],i,j;*

*printf("Enter the elements of matrix [3][4] row wise : \n");*

*for(i=0;i<3;i++)*

*for(j=0;j<4;j++)*

*scanf("%d",&mat[i][j]);*

*printf("The matrix that you have entered is : \n");*

*for(i=0;i<3;i++)*

*{*

*for(j=0;j<4;j++)*

*printf("%5d",mat[i][j]);*

*printf("\n");*

*}*

*printf("\n");*

*getch();*

*}*

*OUTPUT:*

*Enter the elements of matrix[3][4] row wise:*

*2 3 4 7*

*8 5 1 9*

*1 8 2 5*

*The matrix that you have entered is :*

*2 3 4 7*

*8 5 1 9*

*1 8 2 5*

**8.4 Introduction to Strings**

In C strings are treated as arrays of type char and are terminated by a null character (‘\0’). This null character has ASCII value Zero.

There are two forms of initialization of a string variable-

***char str[10] = { ‘e’, ‘m’, ‘b’, ‘e’, ‘d’, ‘d’, ‘e’, ‘d’, ‘\0’ };***

***char str[10] = “embedded”; /\*Here the null character is automatically placed at the end\*/***

**8.4.1 Input and output of strings**

***/\*Program for input and output of strings using scanf() and printf()\*/***

*#include<stdio.h>*

*main()*

*{*

*char str[10]="embedded system";*

*printf("String is : %s\n",str);*

*printf("Enter new value for string: ");*

*scanf("%s",str);*

*printf("String is : %s\n",str);*

*}*

*OUTPUT:*

*String is :* *embedded*

*Enter new value for string : embedded systems*

*String is : embedded*

**POINTERS**

C is a very powerful language and the real power of C lies in pointers. The use of pointers makes the code more efficient and compact. Some of the uses of pointers are –

1. Accessing array elements
2. Returning more than one value from a function
3. Accessing dynamically allocated memory
4. Implementing data structures like linked lists, trees.

**8.5 About Memory**

The memory in a computer is made up of bytes arranged in a sequential manner. Each byte has an index number, which is called the address of that byte.  
We have studied that it is necessary to declare a variable before using it, since compiler has to reserve space for it. The data type of the variable also has to be mentioned so that the compiler knows how much space needs to be reserved. Suppose we declare a variable age of type int –

***int age;***

The compiler reserves 4 consequtive bytes from the memory for this variable and associated the name age with it. The address of first byte from the 4 allocated bytes is known as the address of variable age.

**8.6 Address Operator**

C provides and address operator. ‘&’. which returns the address of a variable when placed before. This operator can be read as “the address of” , so &age means address of age, similarly &sal means, address of sal.

***/\*Program to print address of variables using address operator\*/***

*#include<stdio.h>*

*main()*

*{*

*int age=30;*

*float sal=1500.50;*

*printf("Value of age = %d, Address of age = %u\n",age,&age);*

*printf("Value of sal = %d, Address of sal = %u\n",sal,&sal);*

*}*

*OUTPUT:*

*Value of age = 30, Address of age = 65524*

*Value of sal = 1500.500000, Address of sal = 65520*

Here we have used %u control sequence to print the address, but this doesnot mean that addresses are unsigned integers. Adresses are just whole numbers. These addresses may be different each time you run your program, it depends on which part of the memory is allocated by operating system for this program.

**8.7 pointer Variables**

A pointer is a variable that stores memory address. Lile all other variables it also has a name, has to be declared and occupies some space in memory. It is called pointer because it points to a particular location in the memory by storing the address of that location.

**8.7.1 Declaration of Pointer Variables**

Like other variables, pointer variables should also be declared before being used. The general syntax of declaration is –

***data\_type \*pname;***

Here pname is the name of pointer variable, which should be a valid C identifier. The asterisk ‘\*’ preceding this name informs the compiler that the variable is declared as a pointer. Here data\_type is known as the base type of pointer. Let us take some pointer declarations –

***int \*iptr;***

***float \*fptr;***

***char \*cptr,ch1,ch2;***

Here iptr is a pointer that should point to variables of type int, similarly fptr and cptr should point to variables of float and char type respectively. We can also combine the declaration of simple variables and pointer variables as we have done in the third declararion statement where ch1 and ch2 are declared as variables of type char.

Pointers are also variables so compiler will reserve space for the and they will also have some address. All the pointers irrespective of their base type will occupy same space in memory since all of them contain address only. Generally 4 byte are used to store an address (may vary in different compilers)

**8.7.2 Assigning Address to Pointer Variables**

When we declare a pointer variable it contains garbage value i.e. it may be pointing anywhere in the memory. So we should always assign an address before using it in the program. The use of an unassigned pointer may give unpredictable results and even cause the program to crash.

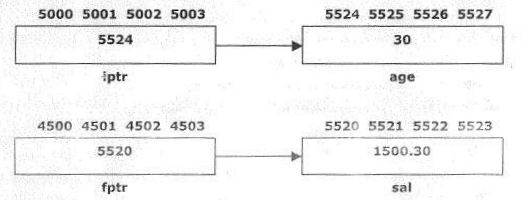
***int \*iptr, age=30;***

***float \*fptr, sal=1500.50***

***iptr=&age;***

***fptr=&sal;***

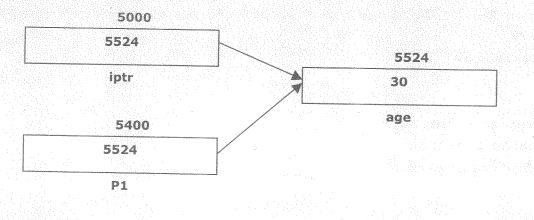
Now iptr contains the address of the variable age i.e. it points to variable age, similarly fptr points to variable sal. Since iptr is declared as a pointer of type int, we should assign address of only integer variables to it. If we assign address of some other data type then compiler won’t show any error but the output will be incorrect.



It is also possible to assign the value of one pointer variable to the other, provided their base type is same. For example if we have an integer pointer p1 then we can assign the value of iptr to it as

***p1=iptr;***

Now both pointer variables iptr and p1 contain the address of variable age and point to the same variable age.



We can assign constant zero to a pointer of any type. A symbolic constant NULL is defined in stdio.h, which denotes the value zero. The assignment of NULL to a pointer guarantees that it does not point to any valid memory location. This can be done as –

***ptr = NULL;***

**8.7.3 Dereferencing Pointer Varibales**

We can also access a variable indirectly using pointers. For this we will use the indirection operator (\*). By placing the indirection operator before a pointer variable, we can access the variable whose access is stored in the pointer. Let us take an example –

***int a=87;***

***float b=7.5;***

***int \*p1=&a;***

***float \*p2=&b;***

***\*p1=9 is equivalent to a=9;***

***(\*p1)++ is equivalent to a++;***

***x=\*p2+10; is equivalent to x=b+10;***

***printf(“%d %f”,\*p1,\*p2) is equivalent to printf(“%d %f”, a,b);***

The indirection operator can be read as ‘value at the address’. For example \*p1 can be read as value at he address p1’. This indirection operator (\*) is different from the asterisk that was used while declaring the pointer variable.

We have already seen that while declaring a pointer variable, we have to mention the data type. The reason is that when we use the indirection operator, the number of bytes received will be different for different data types. The value of the pointer only tells the address of starting byte. If the base type is int/float then 4 bytes information will be retrieved and if base type is char 1 byte information will be retrieved.

***/\*Program to print the size of pointer variable and size of value dereferenced by that pointer\*/***

*#include<stdio.h>*

*main()*

*{*

*char a='x', \*p1=&a;*

*int b=12, \*p2=&b;*

*float c=12, \*p3=&c;*

*double d=18.3, \*p4=&d;*

*printf("sizeof(p1)= %d,sizeof(\*p1)=%d\n",sizeof(p1),sizeof(\*p1));*

*printf("sizeof(p2)= %d,sizeof(\*p2)=%d\n",sizeof(p2),sizeof(\*p2));*

*printf("sizeof(p3)= %d,sizeof(\*p3)=%d\n",sizeof(p3),sizeof(\*p3));*

*printf("sizeof(p4)= %d,sizeof(\*p4)=%d\n",sizeof(p4),sizeof(\*p4));*

*}*

*OUTPUT:*

*sizeof(p1)= 4, sizeof(\*p1)=1*

*sizeof(p2)= 4, sizeof(\*p2)=4*

*sizeof(p3)= 4, sizeof(\*p3)=4*

*sizeof(p4)= 4, sizeof(\*p4)=8*

**8.8 Pointer Arithmetic**

All types of arithmetic operations are not possible with pointers. The only valid operations that can be performed are as –

1. Addition of an integer to a pointer and increment operation.
2. Subtraction of an integer from a pointer and decrement operation.
3. Subtraction of a pointer from another pointer of same type.

Pointer arithmetic is somewhat different from ordinary arithmetic. Here all arithmetic is performed relative to the size of base type of the pointer. For example if we have an integer pointer p1 which contains address 1000 then on incrementing we get 1004 instead 1001. This is because the size of int data type is 4. Let us see some pointer arithmetic for int and char pointers.

***int a=5, \*pi=&a;***

***char c=’x’, \*pc=&c;***

Suppose the address of variables a,c are 1000,5000 respectively, so initially values of pi,pc will be 1000 and 5000.

***pi++; or ++pi; pi=1000+4=1004(since int is 4 bytes)***

***pi=pi-3; pi=1004-3\*4=992***

***pi=pi+5; pi=992+5\*4=1012***

***pi--; or –pi; pi=1012-4=1008***

***pc++; or ++pc; pc=5000+1=5001(since char is 1 byte)***

***pc=pc-3; pc=5001-3=4998***

***pc=pc+5; pc=4998+5=5003***

***pc--; or –pc; pc=5003-1=5002***

**8.9 Precedence of Dereferencing Operator and Increment/ Decrement Operators**

The precedence level of \* operator and increment/decrement operators is same and their associativity is from right to left. There can be confusion while combining these operators in pointer expression. So we should use them carefully.

Suppose ptr is an integer pointer and x is an integer variable.

1. ***x=\*ptr++;***

The expression \*ptr++ is equivalent to \*(ptr++), since these operators associate them from right to left. Hence the increment operator will be applied to ptr, and not to \*ptr. The increment operator is postfix,so first the value of ptr will be used in the expression and then it will be incremented.

***x=\*ptr;***

***ptr=ptr+1;***

1. ***x= \*++ptr;***

The expression \*++ptr is equivalent to \*(++ptr). Here also the increment operator is applied to ptr. The increment operator is prefix, so first ptr will be incremented and then its new value will be used in the expression.

***ptr=ptr+1;***

***x=\*ptr;***

1. ***x=++\*ptr;***

The expression ++\*ptr is equivalent to ++(\*ptr). Here the increment operator is applied over \*ptr and not ptr. So here the value of pointer will not change but the value pointed to by the pointer will change. i.e. \*ptr will increment. Since the increment operator is prefix hence first the value of \*ptr will increment and then this value will be assigned to x.

***\*ptr=\*ptr+1;***

***x=\*ptr;***

1. ***x=(\*ptr)++; x=\*ptr++;***

Here also the increment operator is applied over \*ptr and since it is postfix increment hence first the value of \*ptr will be assigned to x and then it will be incremented.

***x=\*ptr;***

***\*ptr=\*ptr+1;***

***// PROGRAM 1***

***#include <stdio.h>***

***int main(void)***

***{***

***int arr[] = {10, 20};***

***int \*p = arr;***

***++\*p;***

***printf("arr[0] = %d, arr[1] = %d, \*p = %d", arr[0], arr[1], \*p);***

***return 0;***

***}***

“arr[0] = 11, arr[1] = 20, \*p = 11“.

***// PROGRAM 2***

***#include <stdio.h>***

***int main(void)***

***{***

***int arr[] = {10, 20};***

***int \*p = arr;***

***\*p++;***

***printf("arr[0] = %d, arr[1] = %d, \*p = %d", arr[0], arr[1], \*p);***

***return 0;***

***}***

**arr[0] = 10, arr[1] = 20, \*p = 20*“.***

***// PROGRAM 3***

***#include <stdio.h>***

***int main(void)***

***{***

***int arr[] = {10, 20};***

***int \*p = arr;***

***\*++p;***

***printf("arr[0] = %d, arr[1] = %d, \*p = %d", arr[0], arr[1], \*p);***

***return 0;***

***}***

***“*arr[0] = 10, arr[1] = 20, \*p = 20*“.***

**8.10 Pointer to Pointer**

We know that pointer is a variable that can contain memory address. This pointer variable takes some space in memory and hence it also has an address. We can store the address of a pointer variable in some other variable, which is known as a pointer-to=pointer variable. Similarly we can have a pointer to pointer to pointer variable, which is known as pointer-to-pointer variable. Similarly we can have a pointer to pointer variable and this concept can be extended to any limit, but in practice onlt pointer-to-pointer is used. Pointer to pointer is generally used while passing pointer variables to functions.

The syntax of declaring a pointer to pointer is as –

***data\_type\*\*pptr;***

***For example***

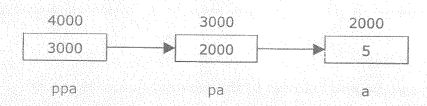
***int a=5;***

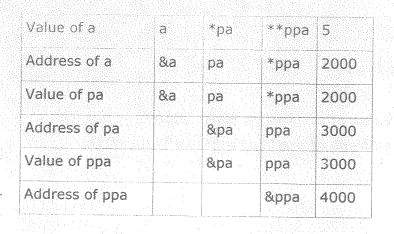
***int \*pa=&a;***

***int \*\*ppa=&pa;***

Here type of variable a is int, type of variable pa is (int \*) or pointer to int, and type of variable ppa is (int \*\*) or pointer to pointer to int.

`

Here pa is a pointer variable, which contains the address of the variab;e a and ppa is a pointer to pointer variable, which contains the address of the pointer variable pa.

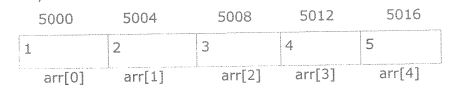


**8.11 Pointer and One Dimensional Arrays**

The elements of an array are stored in contiguous memory locations. Suppose we have an array arr[5] of type int.

***int arr[5] = {1, 2, 3, 4, 5 };***

This is stored in the memory as –



Here 5000 is the address of first element. And since each element ( type int) make 4 bytes so address of next element is 5004, and so on. The address of first element of the array is also known as the base address of the array.

We can access the array elements using pointer expressions. Actually the compiler also accesses the array elements by converting subscript notation to pointer notation. Following are the main points for understanding the relationship of pointer with arrays.

1. Elements of an array are stored in consecutive memory locations.
2. The name of an array is a constant pointer that points to the first element of the array, i.e. it stores the address of the first element, also known as the base address of array.
3. According to pointer arithmetic, when a pointer variable is incremented, it points to the next location of its base type.

***/\*Program to print the value and address of the elements of an array – 8\_10.c\*/***

*#include<stdio.h>*

*main()*

*{*

*int arr[5]={5,10,15,20,25};*

*int i;*

*for(i=0;i<5;i++)*

*{*

*printf("Value of arr[%d]=%d\t",i,arr[i]);*

*printf("Address of arr[%d]=%u\n",i,&arr[i]);*

*}*

*}*

***OUTPUT:***

*Value of arr[0]=5 Address of arr[0]=2000*

*Value of arr[1]=10 Address of arr[1]=2004*

*Value of arr[2]=15 Address of arr[2]=2008*

*Value of arr[3]=20 Address of arr[3]=2012*

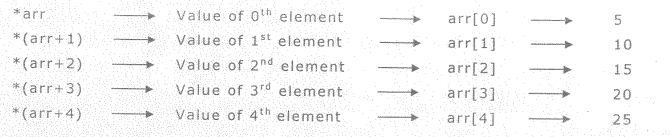
*Value of arr[4]=25 Address of arr[4]=2016*

The name of the array ‘arr’ denotes the address of 0th element of array which is 2000. The address of 0th element can also be given by &arr[0], so arr and &arr[0] represent the same address. The name of an array is a constant pointer, ans according to pointer arithmetic when an integer is added to a pointer then we get the address of next element of same base type. Hence (arr+1) will denote the address of the next element arr[1]. Similarly (arr+2) denotes the address of arr[2] and so on.

In general we can write –

The pointer expression (arr+i) dehnote the same address &arr[i]

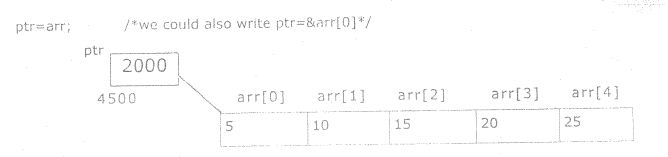
Now if we dereference arr, then we get the 0th element of array, i.e. expression \*arr or \*(arr+0) represents 0th element of array. Similarly on dereferencing (arr+1) we get the 1st element and so on.

In general we can write –

**\*(arr+i) ------------- arr[i]**

**8.12 Subscripting Pointer Variables**

Suppose we take a pointer variable ptr, and initialize it with the address of the 0th element of the array

 int \*ptr;

On applying pointer arithmetic an dereferencing we can see that the expression (ptr+i) denotes the address of ith element of array and the expression \*( ptr+i) denotes the value of ith element of an array. According to the equivalence of pointer and subscript notations, \*( ptr+i) can be written as ptr[i]. So if we have a pointer variable pointing to the 0th element of an array, then we can access the elements of array by subscripting that pointer variable. This equivalence of pointer and subscript notations is used in dynamic arrays and while sending arrays to functions.

**8.13 Pointer to an Array**

In previous section, we had a pointer that pointed to the 0th element of array. We an also declare a pointer that can point to the whole array instead of only one element of array. This pointer is useful when talking about multidimensional arrays. Declaration of a pointer to an array is –  ***int( \*ptr)[10];***

Here ptr is pointer that can point to an array of 10 integers, note that it is necessary to enclose the pointer name inside parenthesis. Here the type of ptr is ‘Pointer to an array of 10 integers’.

Note that the pointer that points to the 0th element of an array and the pointer that points to the whole array are totally different.

***/\*Program to understand difference between pointer to an integer and pointer to an array of integers 8\_11.c\*/***

*#include<stdio.h>*

*main()*

*{*

*int \*p; //can point to an integer*

*int(\*ptr)[5]; //can point to an array of 5 integers*

*int arr[5];*

*p=ptr=arr; //points to 0th element of arr*

*printf("p=%u,ptr=%u\n",p,ptr);*

*p++;*

*ptr++;*

*printf("p=%u,ptr=%u\n",p,ptr);*

*}*

***OUTPUT:***

*p=3000,ptr=3000*

*p=3004,ptr=3020*

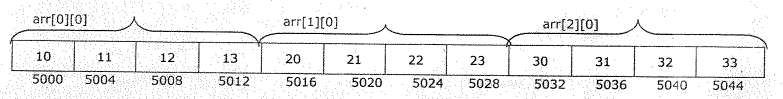
**8.14 Pointers And Two Dimensional Arrays**

In a two dimensional array we can access each element by using two subscripts, where first subscript repreents row number and second subscript represents the coloumn number. The element of 2-D arry can be accessed with the help of pointer notation also. Suppose arr is a 2-D array, then we can access any element arr[i][j] of this array using the pointer expression \*(\*(arr+i)+j).

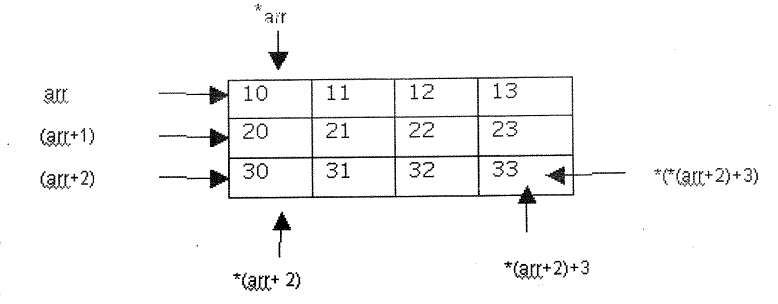
Let us take a two dimension array arr[3][4] –

int arr[3][4] = { {10,11,12,13},{20,21,22,23},{30,31,32,33}};

we have been talking about 2-d arrays in terms of rows and coloumns, but since memory in computer is organized linearly it is not possible to store the 2-D array in rows and coloumns. The concept of rows and coloumns is only theoretical, actually a 2-D array is stored in row major order i.e. rows are placed next to each other. The following figure shows how the above 2-D array will be stored in memory

Each row can be considered as a 1-D array, so a two-dimensional array can be considered as a collection of 1-D arrays that are placed one after another. In other words we can say that a 2-D array is an array of arrays. So here arr is an array of 3 elements where each element is a 1-D array of 4 integers.

So we can say that arr points to the 0th 1-D array, (arr+1) points to the 1st 1-D array and (arr+2) points to the 2nd 1-D array.



*/\*Program to print the values and address of elements of a 2-D array 8\_12.c\*/*

*#include<stdio.h>*

*main()*

*{*

*int arr[3][4]={*

*{10,11,12,13},*

*{20,21,22,23},*

*{30,31,32,33}*

*};*

*int i,j;*

*for(i=0;i<3;i++)*

*{*

*printf("Address of %dth array = %u %u\n",i,arr[i],\*(arr+i));*

*for(j=0;j<4;j++)*

*printf("%d %d",arr[i][j], \*(\*(arr+i)+j));*

*printf("\n");*

*}*

*}*

*OUTPUT:*

*Address of 0th 1-D array = 65000 65000*

*10 10 11 11 12 12 13 13*

*Address of 1th 1-D array = 65016 65016*

*20 20 21 21 22 22 23 23*

*Address of 2th 1-D array = 65032 65032*

*30 30 31 31 32 32 33 33*

**8.15 Pointers And Functions**

The arguments to the functions can be passed in two ways –

1. Call by Value
2. Call b y Address

In Call by Value, only the values of arguments are sent to the function while in call by reference, addresses of arguments are sent to the function. In call by value method, any changes made to the formal arguments will not reflect on the actual arguments. C uses only call by value when passing arguments to a function, but we can simulate call by address by using pointers.

***/\*Program to explain call by value 8\_13.c\*/***

*#include<stdio.h>*

*main()*

*{*

*int a=5,b=8;*

*printf("before calling the function, a and b are %d, %d\n",a,b);*

*value(a,b);*

*printf("after calling the function, a and b are %d, %d\n",a,b);*

*}*

*value(int x int y)*

*{*

*x++;*

*y++;*

*printf("In function changes are %d, %d\n",x,y);*

*}*

***OUTPUT:***

*before calling the function, a and b are 5,8*

*In function changes are 6,9*

*after calling function a and b are 5,8*

Here a and b are variables declared in the function main() while x ans y are declared in the function value(). These variables reside at different addresses in memory. Whenever the function value() is called, two variables are created named x and y and are initialized with the variables a and b. This type of paramenter passing is called call by value since we are only supplying the values of actual arguments to the calling function. Any operation performed on variables x and y in the function value(), will not affect variables a and b.

although C does not use call by address, but we can simulate it by passing addresses of variables as arguments to the function. To accept the addresses inside the function, we will need pointer variables.

***/\*Program to explain call by reference - 8\_14.c\*/***

*#include<stdio.h>*

*main()*

*{*

*int a=5,b=8;*

*printf("before calling the function, a and b are %d, %d\n",a,b);*

*ref(&a,&b);*

*printf("after calling the function, a and b are %d, %d\n",a,b);*

*}*

*ref(int \*p, int \*q)*

*{*

*(\*p)++;*

*(\*q)++;*

*printf("In function changes are %d, %d\n",\*p,\*q);*

*}*

***OUTPUT:***

*before calling the function, a and b are 5,8*

*In function changes are 6,9*

*after calling function a and b are 6,9*

Here we are passing addresses of variables a and b in the function call. So the receiving formal arguments in the function declaration should be declared of pointer type. Whenever function ref() is called, two pointer to int variables, named p and q, we will be able to access variables a and b from function ref().

**8.16 Function returning Pointer**

We have a function that returns a pointer. The syntax of declaration of such type of function is –

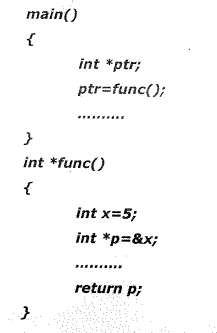
***type \*func( type1,type2,…);***

For example –

***float \*fun(int,char); /\* This function returns a pointer to float\*/***

***int \*func(int,int); /\* This function returns a pointer to int\*/***

When returning a pointer, make sure that the memory address returned by the pointer will exist even after the termination of function. For example a function of this form is invalid –



Printf(“%d”,\*ptr);

Here we are returning a pointer which points to a local variable. We know that a local variable exists only inside the function. Suppose the variable x is stored at address 2500, so the value of p will be 2500 and this value will be returned by the function func(). As soon as func() terminated, The local variable x will cause to exist.

The address returned by func() is assigned to pointer variable ptr inside main(), so now ptr will contain address 2500. When we dereference ptr, we are trying to access the value of a variable that no longer exists. So never return a pointer that points to a local variable.

***/\*Program to show the use of a function that returns pointer 8\_15.c\*/***

*#include<stdio.h>*

*main()*

*{*

*int arr[10]={1,2,3,4,5,6,7,8,9,10},n,\*ptr;*

*n=5;*

*ptr=fun(arr,n);*

*printf("value of arr = %u, value of ptr = %u, value of \*ptr = %d\n",arr,ptr,\*ptr);*

*}*

*int \*fun(int \*p,int n)*

*{*

*p=p+n;*

*return p;*

*}*

***OUTPUT:***

*value of arr = 65104, value of ptr = 65124, value of \*ptr = 6*

**8.17 Passing a 1-D array to a Function**

In the previous chapter we had studied that when an array is passed to afunction, the changes made inside the function affect the original array. This is because the function gets access to the original array.

***/\*Program to show that changes to the array made inside the function affect the original array 8\_16.c\*/***

*#include<stdio.h>*

*main()*

*{*

*int i,arr[5]={3,6,2,7,1};*

*func(arr);*

*printf("Inside main() :");*

*for(i=0;i<5;i++)*

*printf("%d",arr[i]);*

*printf("\n");*

*}*

*func(int a[])*

*{*

*int i;*

*printf("Inside func() :");*

*for(i=0;i<5;i++)*

*{*

*a[i]=a[i]+2;*

*printf("%d",a[i]);*

*}*

*printf("\n");*

*}*

***OUTPUT:***

*Inside func() :5 8 4 9 3*

*Inside main() :5 8 4 9 3*

There are 3 ways of declaring the formal parameter, which has to receive the array. We can declare it as an un-sized or sized array or we can declare it as a pointer.

***func(int a[])***

***{***

***-------------***

***}***

***func(int a[5])***

***{***

***-------------***

***}***

***func(int \*a)***

***{***

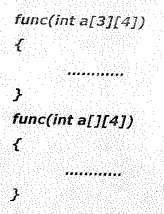
***-------------***

***}***

In all the three cases the compiler reserves space only for a pointer variable inside the function. In the function call, the array name is passed without any subscript or address operator. Since array name represents the address of first element of array, hence this address is assigned to the pointer variable in the function. So inside the function we have a pointer that contains the base address of the array. In the above program, the argument “a” is declared as a pointer variable whose base type is int, and it is initialized with the base address of array arr.

**8.18 Passing a 2-D array to a Function**

Whenever a multidimensional array is passed to a function, then it is optional to specify the leftmost dimension but all other dimensions must be specified. So if we have a 2-D array with 3 rows and 4 columns, then the definition of a function that accespts it can be written in these two ways –



Any changes made to the array in the function will be reflected in the calling function, whenever a 2-D array is passed to a function, the function actually receives a pointer to a 1-D array, where the size of 1-D array is equal to the number of columns. For example in the above case the function receives a pointer to an array of 4 integers. So we may write the function definition in this form also –

***func(int (\*a)[4])***

***{***

***-------------***

***}***

Here a is declared as a pointer to an array of 4 integers, and it initialized with the base address of the original 2-D array.

**8.19 Array of Pointers**

We can declare an array that contains pointers as its elements. Every element of this array is a pointer variable that can hold address of any variable of appropriate type. The syntax of declaring an array of pointers is similar to that of declaring arrays except that an asterisk is places before the array name.

***Datatype \*arrayname[ size ];***

For example to declare an array of size 10 that contains integer pointers we can write –

***int \*arrp[10];***

***/\*Program for understanding the concept of array of pointers 8\_17.c\*/***

*#include<stdio.h>*

*main()*

*{*

*int \*pa[3];*

*int i,a=5,b=10,c=15;*

*pa[0]=&a;*

*pa[1]=&b;*

*pa[2]=&c; \*(pa+i)*

*for(i=0;i<3;i++)*

*{*

*printf("pa[%d}=%u\t",i,pa[i]);*

*printf("pa[%d}=%d\t",i,\*pa[i]); \*(pa+i)*

*}*

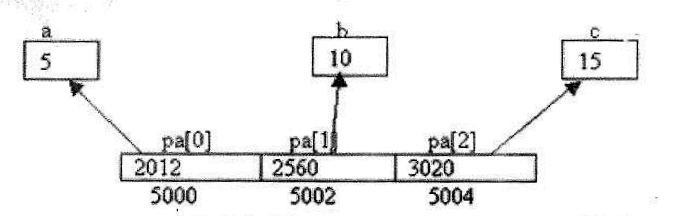
*}*

***OUTPUT:v***

*p[0]=2012 \*pa[0]=5*

*p[1]=2560 \*pa[1]=10*

*p[2]=3020 \*pa[2]=15*

\*(pa+2) pa[2]

\*(\*(pa+2) \*pa[2]

Here pa is declared as an array of pointers. Every element of this array is apointer to an integer. pa[i[ gives the values of the ith element of ‘pa’ which is an address of any int variable and \*pa[i] gives the value of that int variable.

The array of pointers can also contain addresses of elements of another array.

**8.20 Void Pointers**

A pointer to void is a generic pointer that can point to any data type. The syntax of declaration of a void pointer is –

***void \*vpt;***

Here void is a keyword and vpt is declared as apointer of void type. For example –

**int i=2, \*ip=&i;**

**float f=2.3, \*fp=&f;**

**double d;**

**void \*vp;**

**ip=fp; /\*incorrect\*/**

**vp=ip; /\*correct\*/**

**vp=fp; /\*correct\*/**

**vp=&d; /\*correct\*/**

A void pointer can’t be dereferenced simply by using indirection operator. Before dereferencing, it should be type cast to appropriate pointer data type. For example if vp is a void pointer and it holds the address of an integer variable then we can’t dereference it just by writing \*vp. We’ll have to write \*(int \*)vp, whether leftmost asterisk is the indirection operator and d (int \*) is used for typecasting. Similarly pointer arithmetic can’t be performed on void pointers without typecasting.

***/\*pg 17 Program to understand the dereferencing of void pointer\*/***

**8.21 Const Qualifier**

If any variable is declared with *const* qualifier, then the program can’t change the value of that variable. The const qualifier can occur in the declaration before or after the data type. For example –

const int x = 9;

int const x = 9;

Both these declarations are equivalent and declare x as a const variable. Any attempt to change the value of this variable in the program will result in an error. For example these statements are invalid –

x = 10; /\*invalid\*/

x = func( ); /\*invalid\*/

x++; /\*invalid\*/

The const qualifier informs the compiler that the variable can be stored in read only memory. A const variable can be given a value only through initialization or by some hardware devices. Note that the program can’t modify the value of a const variable, but any external event outside the program can change its value.

If an array, structure or union is declared as const then each memner in it becomes constant. For example –

***const int arr[5] = { 10,11,12,13,14 };***

***const struct***

***{***

***char x;***

***int y;***

***float z;***

***} var = { ‘A’, 12, 29.5 };***

***arr[2] = 22; /\*invalid\*/***

***var.x = ‘B’; /\*invalid\*/***

***Read-only shared-memory buffer***

***int const volatile comm\_flag;uint8\_t const volatile comm\_buffer[BUFFER\_SIZE];***

*const and volatile is where you have two or more processors communicating via a shared memory area and you're coding the side of this communications that will only be reading from a shared memory buffer*.

Now we’ll see how to use const in pointer declarations. We can declare three types of pointers using the qualifier const.

1. ***Pointer to const data***
2. ***const pointer***
3. ***const pointer to const data***

Consider these declarations-

*const int a =2, b = 6;*

*const int \*p1 = &a; /\* or int const \*p1 = &a; \*/*

Here p1 is declared as a pointer to const integer. We can change the pointer p1 but we can’t change the variable pointed to by p1.

*\*p1 = 9;*  /\*invalid\*/

*p1 = &b;* /\*valid since p1 is not a constant itself\*/

Now consider these declarations-

***int a = 2, b = 6;***

***int \*const p2 = &a;***

Here p2 is declared as a const pointer. We can’t change the pointer variable p2, but we can change the variable pointed to by p2.

***\*p2 = 9; /\*Valid\*/***

***p2 = &b; /\*Invalid since p2 is a constant\*/***

Constant addresses of hardware registers

uint8\_t volatile \* const p\_led\_reg = (uint8\_t \*) 0x80000;

p\_led\_reg = LED1\_ON; //Error

\*p\_led\_reg = LED1\_ON;

Now consider these declarations –

***const int a = 2, b = 6;***

***cont int const \*p3 = &a;***

Here p3 is declared as a const pointer to const integer. We can neither change the pointer variable p3 nor the variable pointed to by it.

***\*p3 = 9;***  ***/\*invalid\*/***

***p3 = &b; /\*invalid\*/***

So the three different types of declarations of pointers using const are –

***int const \*ptr ; or const int \*ptr; /\* Pointer to const integer\*/***

***int \*const ptr; /\*const pointer to an integer\*/***

***const int \*const ptr; /\*const pointer to a const integer\*/***

***Read-only hardware register***

***uint8\_t const volatile \* const p\_latch\_reg = (uint8\_t \*) 0x10000000;***

*Declaring your variable IS A (constant) pointer TO A constant and volatile memory location you request all of the appropriate protections*

**8.22 Dynamic Memory Allocation**

The memory allocation that we have done till now was static memory allocation. The memory that could be used by the program was fixed i.e. we could not increase or decrease the size of the memory during the execution of the program. In many applications it is not possible to predict how much memory the program would need at runtime . For example, if we declare an array of integers-

***in temp\_no[200];***

If an array, it must to specify the size of the array while declaring, so the size of this array will be fixed during runtime. Now two types of problems may occur. The first case is that the number of values to be stored is less that the size of the array and hence there is wastage of memory. For example if we have to store only 50 values in the above array, the space for 150 values (300 bytes) is wasted. In second case our program fails if we want to store more values than that of the size of array, for example if there is need to store 205 values in the above array

To overcome these problems we should be able to allocate memory at runtime. The allocation and release of this ,memory space can be done with the help of some built-in-functions whose prototypes are found in stdlib.h and alloc.h header files. These functions take memory from a memory area called heap and release this memory whenever not required, so that it can be used again for some other purpose.

Pointers play an important role in dynamic memory allocation because we can access the dynamically allocated memory only through pointers.

**8.22.1 malloc( )**

***Declaration : void \*malloc(size\_t size);***

This function is used to allocate memory dynamically. The argument size specifies the number of bytes to be allocated. The type size\_t is defined in stdlib.h as unsigned int. On success, malloc() returns a pointer to the first byte of allocated memory. The returned pointer is of type void, which can be type cast to appropriate type of pointer. It is generally used as –

***ptr=(datatype \*)malloc(specified size);***

Here ptr is a pointer of type datatype, and specified size is the size in the bytes required to be reserved in memory. The expression (datatype \*) is used to typecast the pointer returned by malloc(). For example –

***int \*ptr;***

***ptr=(int \*)malloc(20);***

This allocated 20 contiguous bytes of memory space and the address of first byte is stored in the pointer variable ptr. This space can hold 5 integers. The allocated memory contains garbage value. We can use sizeof operator to make the program portable and more readable

***ptr=(int \*) malloc(5 \* sizeof(int));***

This allocates the memory space to hold five integer values.

If there is not sufficient memory available in heap then malloc() returns NULL. So we should always check the return value by malloc()

***ptr=(float \*)malloc(10\*sizeof(float));***

***if(ptr == NULL)***

***printf(“Suficient memory not available”);***

***/\*Pg 21 program to understand dynamic memory allocation of memory\*/***

**8.22.2 calloc()**

***Declaration : void \*calloc(size\_t n, size\_t size);***

The calloc() function is used to allocate multiple blocks of memory. It is somewhat similar to malloc() function except for two differences. The first one is that it takes two arguments. The first argument specifies the number of blocks and the second one specifies the size of each block. for example

***ptr = (int \*) calloc(5, sizeof(int) );***

This allocate 5 blocks of memory, each block contains 4 bytes and the starting address is stored in the pointer variable ptr, which is of type int.

The other difference beteen calloc() and malloc() is that the memory allocated by malloc() contains garbage value while the memory allocated by calloc() is initialized to zero. But this initialization by calloc() is not very reliable, so it is better to explixitly initialize the elements whenevwr there is need to do so.

Like malloc(), calloc() also returns NULL if there is not sufficient memory available in the heap.

**8.22.3 realloc()**

***Declaration : void \*realloc(void \*ptr, size\_t newsize);***

We may want to increase or decrease the memory allocated by malloc() or calloc(). The function realloc() is used to change the size of the memory block. It alters the size of the memory block without losing the old data. This is known as reallocation of memory.

The function takes two arguments, first is a pointer to the block of memory that was previously allocated by malloc() or calloc() and a second one is the new size for that block. for example –

***ptr = (int \*) malloc(size);***

This statement allocates the memory of the specified size and the starting address of this memory block is stored in the pointer variable ptr. If we want to change the size of this memory block, then we can use realloc() as –

***ptr=(int \*) realloc(ptr, newsize);***

This statement allocates the memory space of newsize bytes, and the starting address of this memory block is stored in the pointr variable ptr. The newsize may be smaller or larger than the old size. If the newsize is larger, then the old data is not lost and the newly allocated bytes are uninitialized. The starting address contained in ptr may change if there is not sufficient memory at the old address to store all the bytes consecutively. This function moves the contents of old block into the newblock and the data of the old block is not lost. On failure, realloc() returns NULL.

***/\*Pg23 program to understand the use of realloc() function\*/***

**8.22.4 free()**

***Declaration: void free(void \*p);***

The dynamically allocated memory is not automatically released. It will exist till the end of the program. If we have finished working with the memory allocated dynamically, it is our responsibility to release that memory so that it can be reused. The function free() is used to release the memory space allocated dynamically. The memory released by free() is made available to the heap again and can be used for some other purpose. For example –

***free(ptr);***

Here ptr is a pointer variable that contains the base address of a memory block created by malloc() or calloc(). Once a memory location is freed it should not be used. We should not try to free any memory location that was not allocated by malloc(), calloc(),realloc().

When the program terminate all the memory is released automatically by the operating system but it is a good practose to free whatever has been allocated dynamically. We won’t get any errors if we don’t free the dynamically allocated memory, but this would lead to memory leak. i.r. memory is slowly leaking away and can be reused only after the termination of the program.

**8.23 Dynamic Arrays**

The memory allocated by malloc(), calloc() and realloc() is always madeup of contiguous bytes. Moreover in C there is an equivalence between pointer notation and subscript notation i.e. we can apply subscripts to a pointer variable. So we can access the dynamically allocated memory through subscript notation also. We can utilize these features to create dynamica arrays whose size can vary during runtime.

***Using single pointer***

***#include <stdio.h>***

***#include <stdlib.h>***

***int main()***

***{***

***int r = 3, c = 4;***

***int \*arr = (int \*)malloc(r \* c \* sizeof(int));***

***int i, j, count = 0;***

***for (i = 0; i < r; i++)***

***for (j = 0; j < c; j++)***

***\*(arr + i\*c + j) = ++count;***

***for (i = 0; i < r; i++)***

***for (j = 0; j < c; j++)***

***printf("%d ", \*(arr + i\*c + j));***

***/\* Code for further processing and free the***

***dynamically allocated memory \*/***

***return 0;***

***}***

***using array of pointers***

***#include <stdio.h>***

***#include <stdlib.h>***

***int main()***

***{***

***int r = 3, c = 4, i, j, count;***

***int \*arr[r];***

***for (i=0; i<r; i++)***

***arr[i] = (int \*)malloc(c \* sizeof(int));***

***// Note that arr[i][j] is same as \*(\*(arr+i)+j)***

***count = 0;***

***for (i = 0; i < r; i++)***

***for (j = 0; j < c; j++)***

***arr[i][j] = ++count; // Or \*(\*(arr+i)+j) = ++count***

***for (i = 0; i < r; i++)***

***for (j = 0; j < c; j++)***

***printf("%d ", arr[i][j]);***

***/\* Code for further processing and free the***

***dynamically allocated memory \*/***

***return 0;***

***}***

***Using pointer to pointer***

***#include <stdio.h>***

***#include <stdlib.h>***

***int main()***

***{***

***int r = 3, c = 4, i, j, count;***

***int \*\*arr = (int \*\*)malloc(r \* sizeof(int \*));***

***for (i=0; i<r; i++)***

***arr[i] = (int \*)malloc(c \* sizeof(int));***

***// Note that arr[i][j] is same as \*(\*(arr+i)+j)***

***count = 0;***

***for (i = 0; i < r; i++)***

***for (j = 0; j < c; j++)***

***arr[i][j] = ++count; // OR \*(\*(arr+i)+j) = ++count***

***for (i = 0; i < r; i++)***

***for (j = 0; j < c; j++)***

***printf("%d ", arr[i][j]);***

***for (i = 0; i < r; i++){***

***free(arr [i]);***

***}***

***free(array);***

***return 0;***

***}***

Here the rows are not contiguous, but we can access each element with subscripts. The size of any row can be easily increased or decreased by realloc().

**8.24 Pointers to Functions**

The code of a function resides in memory hence every function has an address like all other variables in the program. We can get the address of a function by just writing the function’s name without parentheses.

**8.24.1 Declaring A pointer to a Function**

The syntax for declaration of a pointer to a function is as –

***return-type (\*ptr\_name)(type1,type2,…);***

For example –

***float (\*fp)(int);***

***char (\*func\_p)(float,char);***

Here fp is a pointer that can point to any function that returns a float value and accepts an int value as argument. Similarly func\_p is a pointer that can point to functions returning char and accepting float and char as arguments.

Now we have learnt how to declare a pointer to function, the next step is to assign a function’s address to it.

***float (\*fp)(int,int); /\*Declaring a function pointer\*/***

***float func(int,int); /\*Declaring a function\*/***

***fp=func; /\*Assigning address of function func() to pointer fp\*/***

After the above assignments fp contains the address of function func(). Declaring a function is necessary before using its address anywhere in the program because without declaration the compiler will not know about this function and will generate an error.

**Chapter 9 –STRINGS**

There is no separate data type for strings in C.They are treated as array of type char.A character array is a string if it ends with null character(‘\0’). This null character is an escape sequence with ASCII value 0.Strings are generally used to store and manipulate data in text form like words or sentences.

**9.1 String Constant or String Literal**

A string constant is a sequence of characters enclosed in double quotes .It is sometimes called a literal.The double quoted are not a part of the string.Some examples of string constants are

**“Taj Maha” “2345”**

**“Subhash Chandra Bose was a great leader”**

**“”(Null string, contains only ‘\0’)**

**“My age is %d and height id %f\n”(Control string used in printf)**

Whenever a string constant is written anywhere in a program , it is stored somewhere in memory as an array of characters terminated by a null character(‘\0’). The string constant itself becomes a pointer to the first character in the array. For example the string “Taj mahal” will be stored in memory as

1000 1001 1002 1003 1004 1005 1006 1007 1008 1009

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| T | a | J |  | M | a | h | a | L | \0 |

Each character occupies one byte and compiler automatically inserts the null character at the end. The string constant “ Taj Mahal” is actually a pointer to the character ‘T’. SO whenever a string constant is used in the program it is replaced by a pointer pointing to the string.

If we have a pointer variable of type char \* ,then we can assign the address of those, string constant to it as

**char \*p = “Taj Mahal”;**

Similarly when we write –

**printf(“Institute\n”);**

Then actually a pointer to character(char \*) is passed to the printf() function.

If identical string constants are used in a program, they will be stored seperately at different memory locations . For example if a string constant “India” appears three times in a program then it will be stored thrice in memory.

**\\*program to show that identical string constants are stored separately\*/**

**#include<stdio.h>**

**main()**

**{**

**printf(“%u\n”,”good”);**

**printf(“%u\n”,”good”);**

**if (“bad”==”bad”)**

**printf(“same\n”);**

**else**

**printf(“Not same\n”);**

**}**

**Output**

**174**

**183**

**Not same (output according to the TC)**

From the above output we can see that the two strings “good” are identical but they are stores at different places. When we compare two identical strings using equality operator then we are actually comparing the addresses and not the strings.

We have studied that a string constant gives the address of first character in it , but there is an exception to this rule; when the string constant is used as an intializer for a character array then it does represent any address and it is stored anywhere in memory. For example –

**char arr[5] = “Deep”;**

Here the string constant “Deep” is not stored in memory and hence does not represent any address. Note that the ‘b’ and “b” are different . ‘b’ is a character constant which represents the ASCII value of character ‘b’ while “b” is a string constant which consists of character ‘b’ and null character ‘\0’ .

Inside a string constant, the backslash is considered as an escape character, so if there is a need to include backslash character within a string constant then it should be preseded by another backslash. If we want to include double quotes inside string constant, then it should also be preseded by a backslash. Foe example consider these two printf statements-

**printf(“good\\bad”);**

**printf(“I love \”c\” programming”);**

The output of these 2 statements would be

**Good\bad**

**I love “C” programming**

The length of a string is not limited to a line only,it can be continued by adding a backslash at the end of the line.

**9.2 String Variables**

To create a string variable we need to declare a character array with sufficient size to hold all the characters of the string including null character.

**char str[] = {‘N’,’e’,’w’,’y’,’o’,’r’,’k’,’\0’};**

We may also initialize it as

**char str[] =”New York”;**

This initialization is same as the previous one and in this case the compiler automatically inserts the null character at the end. Note thet here the string constant does not represent an address. The array str will be stored in memory as

1000 1001 1002 1003 1004 1005 1006 1007 1008

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| N | e | W |  | Y | o | r | k | \0 |

str[0] str[1] str[2] str[3] str[4] str[5] str[6] str[7] str[8]

Here we have not specified the size of the array , but if we specify it then we should take care that array should be large enough to hold all the characters including the null character.

**/\* program to print characters of a string and address of each character .\*/**

**#include<stdio.h>**

**int i;**

**for(i=0;str[i]!=’\0’;i++)**

**{**

**printf(“Character = %c \t” ,str[i]);**

**printf(“Address = %u \n”, &str[i]);**

**}**

**}**

**Output:**

**Character =I Address = 1000**

**Character =n Address = 1001**

**Character =d Address = 1002**

**Character =i Address = 1003**

**Character =a Address = 1004**

**/\* Program to print the address and characters (If the string using ponter \*/**

**#include<stdio.h>**

**main()**

{

**char str[] = “India”;**

**char \*p;**

**p = str;**

**while(\*p!= ‘\0’)**

**{**

**printf(“character %c \t”, \*p);**

**printf(“Address = %u \n”, p);**

**p++;**

**}**

**}**

The output of this program is same as that of previous program

Here ‘p’ is pointer variable which holds the base address of array str[]. Incrementg this pointer by 1 gives the address of next element of character array str[], so on increamenting and dereferncing this pointer we can print all the elements of the string .This procedure is similar to that applied in other arrays , except that here the loop terminates when the character ‘\0’ is encountered which signifiec the end of the string.

There is a shortcut way for entering and printing strings , using %s specifications in the control string of printf() and scanf().

**/\* program to input and output a string variable using scanf()\*/**

**#include<stdio.h>**

**main()**

**{**

**char name[20];**

**printf(“Enter name:”);**

**scanf(“%s”, name);**

**printf(“%s”, name);**

**printf(“%s\n”, “Vector”);**

**}**

**1st run:**

**Enter a name : Institute**

**Institute Vector**

**2nd run:**

**Enter a name: Embedded systems**

**Embedded vector**

The printf () takes the base address of string and continues to display the characters until it encounters the character ‘\0’.

When we enter string using %s , the null character is automatically stored at the end of the array. We haven’t used & sign in the scanf () since the name of the array is itself address of the array. In the 2nd run when we enterd a string with space we could not get the required result. This is because scanf () stops reading as soon as it encounters a whitespace. So far entering strings with whitespaces we can use the function gets(). It stops reading only when it encounters a newline and replaces this newline by the null character . We have another function puts () Which can output a string and replaces the null character by a newline.

**/\* Program to understand the use of gets() and puts()\*/**

**#include<stdio.h>**

**main()**

**{**

**char name[20];**

**printf(“Enter name”);**

**gets(name);**

**printf(“Entered name is”);**

**puts(name);**

**}**

**Output:**

**Enter name : Embedded Systems**

**Entered name : Embedded Systems**

**9.3 String Library Functions**

There are several library functions used to manupulate strings. The prototypes for these functions are in header file string .h .We’ll discuss some of them below.

**9.3.1 strlen()**

This function returns the length of the string i.e. the number of characters in the string excluding the terminating null character. It accepts a single argument, which is pointer to the first character or the string. For example strlen(“vector”) returns the value 6. Similarly if s1 is an array that contains the name “india” then strlen(s1) returns the value 5.

**/\* program to understand the work of strlen() function\*/**

**#include<stdio.h>**

**#include<string.h>**

**main()**

**{**

**char str[20];**

**int length;**

**printf(“ Enter the string:\n”);**

**scanf(“%s”, str);**

**length=strlen(str);**

**printf(“Length of the string is :%d\n”,length);**

**}**

**Output**

**Enter the string : VectorIndia**

**Length of the string :11.**

**Creation of this Function**

**int astrlen (char str[])**

**{**

**int i=0;**

**while(str[i]!= ‘\0’)**

**i++;**

**return i;**

**}**

**9.3.2 strcmp()**

This function is used for comaprison of two strings. If the two strings match,strcmp() returns a value 0, otherwise it returns a non-zero value.This function comapres the strings character by character.The comparasion stops when either the end of string is reached or the corresponding characters in the two strings are not same.The non-sero value returned on mismatch is the difference of the ASCII values of the non-matching characters of the two strings-

**strcmp(s1,s2) returns a value –**

**< 0 when s1 < s2**

**= 0 when s1 == s2**

**> 0 when s1 > s2**

Generally we don’t use the exact non –zero value returned in case of mismatch .We only need to know its sign to compare the alphabetical positions of the two strings .We can use this function to sort the strings alphabetically.

**/\* program to understand the work of strcmp () function\*/**

**#include<stdio.h>**

**#include<string.h>**

**main()**

**{**

**char str1[10],str2[10];**

**printf(“Enter the first string”);**

**scanf(“%s”, str1);**

**printf(“Enter the second string”);**

**scanf(“%s”, str2);**

**if(strcmp(str1,str2)==0)**

**printf(“Strings are same \n”);**

**else**

**printf(“strings are not same\n”);**

**}**

**Output:**

**Enter the first string :bangalore**

**Enter the second string :mangalore**

**String are not same.**

**9.3.3 strcpy():**

This function is used for copying one string to another string, strcpy(str1,str2) copies str2 to str1.Here str2 is the source string and str1 is destination string .If str2 =”Vector” then this function copies “Vector” into str1. This function takes pointers to two strings as arguments and returns the pointer to first string.

**/\* Program to understand the work of strcpy() function\*/**

**#include<stdio.h>**

**#include<string.h>**

**main()**

**{**

**char str1[10],str2[10];**

**printf(“Enter the first string”);**

**scanf(“%s”, str1);**

**printf(“Enter the second string”);**

**scanf(“%s”, str2);**

**strcpy(str1,str2);**

**printf(“ First string :%s \t \t second string :%s \n”,str1,str2);**

**strcpy(str1,”Delhi”);**

**strcpy(str2,”Calcutta”);**

**printf(“ First string :%s \t \t second string :%s \n”,str1,str2);**

**}**

**Output:**

**Enter the first string :bombay**

**Enter the second string :mumbai**

**First string :mumbai second string :mumbai**

**First string :delhi second string :calcutta**

The programmer should take care that the first string has enough space to hold the second string .The function calls like strcpy(“New”,str1) or strcpy(“New”,”York”) are invalid because “New” is a string constant which is stored in read only memory and so we can’t overwrite it.

**9.3.4 strcat()**

This function is used for concatenation of two strings. If first string is “abcd” and second string is “efgh” then using this function the first string becomes “abcdefgh”.

**strcat(str1,str2); /\* concatenates str2 at the end of str1\*/.**

The null character from the frist string is removed , and the second , string is added at the end of first string .The second string remains unaffected.

This function takes pointer to two strings as arguments and returns a pointer to the first (concatenated) string.

**/\* Program to understand the work of strcat() function\*/**

**#include<stdio.h>**

**#include<string.h>**

**main()**

**{**

**char str1[20],str2[20];**

**printf(“Enter the first string”);**

**scanf(“%s”, str1);**

**printf(“Enter the second string”);**

**scanf(“%s”, str2);**

**strcat(str1,str2);**

**printf(“ First string :%s \t \t second string :%s \n”,str1,str2);**

**strcat(str1,”\_one”);**

**printf(“ Now First string :%s \n”,str1);**

**}**

**Output:**

**Enter the first string :data**

**Enter the second string :base**

**First string :database Second string :base**

**Now first string is : database\_one**

The function strcat() returns a pointer to the first string ,hense it can be nested .The following program illustrates this –

**/\* Program to understand the work of strcat() function\*/**

**#include<stdio.h>**

**#include<string.h>**

**main()**

**{**

**char str1[20] =”Subhash”;**

**char str2[10]= “ Chandra”;**

**strcat( strcat(str1,str2),”Bose”);**

**}**

**Output:**

**str1 – Subhash Chandra Bose**

**9.4 String Pointers**

We can take a char pointer and initialize it with a string constant .For example ,char \*ptr = “Chennai”; here ptr is a char pointer which points to the first character of the string constant “Chennai” i.e. ptr contains the base address of this string constant.

Now we’ll compare the strings defined as array and strings defined as pointers.

**char str[] = “Mumbai”;**

**char \*ptr = “Chennai”;**

These two forms may look similar but there are some differences in them. The initialization itself has different meaning in both forms.In the array form ,initialization is a short form for –

**char str[] ={‘M’,’u’,’m’,’b’,’a’,’i’,’\0’};**

While in pointer form ,address of string constant is assigned to the pointer variable .Now let us see how they are represented in memory.

1000 1001 1002 1003 1004 1005 1006

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| M | U | M | B | A | I | \0 |

Str[0] str[1] str[2] str[3] str[4] str[5] str[6]

5220 5224

|  |
| --- |
| 200 |

Ptr

Here str is an array of characters and 7 bytes are reserved for it. Since str is the name of an array hence it is a constant pointer which will alwys point to the first element of array.The elements of array are initialized with the characters of string.Note that we had mentioned before that when a string constant appears as array initializer ,then it does not return an address.

In the second case, the string constant “Chennai” is stored somewhere in memory with 8 consecutive bytes reserved for it .The string constant returns the address of the first character of the string that is assigned to the pointer variable ptr. So in this case total 12 bytes are reserved , 4 bytes for the pointer variable and 8 bytes for the string.

The main difference is that str is a constant pointer and will always contain address 1000 while ptr is pointer variable and may contain any other address. So string assignments are valid for pointers while they are invalid for strings defined as arrays.

**str = “Bombay” ; /\*Invalid\*/**

**ptr = “Delhi” ; /\*Valid\*/**

We can assign string of any length to ptr .That string constant will be stored somewhere and its address will be assigned to ptr. We can assign a different string to str by scanf (), strcpy or by assigning characters .strcpy (str.”Bombay”);

We have studied earlier that string constants are stored in read only area by some compilers and so thay can’t be changed . SO these operations are invalid –

**char \*ptr = “India”;**

**ptr[0] = ‘D’/\*Invalid\*/**

**strcpy(ptr, “Bareilly”); /\*Invalid\*/**

For this first we should allocate memory through malloc (). And let this pointer point to that region of memory.

**ptr = (char\*)malloc(20);**

**scanf(“%s”,ptr); /\*Valid\*/**

**9.5 Array of Strings or Two Dimensional Array of Characters**

Strings are character arrays so array of strings means array of character type array i.e. a two dimensional array of characters.

Suppose we declare and initialize a two-dimensional array of characters as –

**char arr[5] [10] ={**

**“white”,**

**“red”,**

**“green”,**

**“yellow”,**

**“blue”,**

**};**

This initialization is equivalent to

**char arr[5] [10] ={**

**{‘w’,’h’,’i’,’t’,’e’,’\0’,},**

**{‘R’,’e’,’d’,’\0’,},**

**{‘G’,’r’,’e’,’e’,’n’,’\0’,},**

**{‘Y’,’e’,’l’,’l’,’o’,’w’,’\0’,},**

**{‘B’,’l’,’u’,’e’,’\0’,}**

**};**

Here first subscipt of array denotes number of strings in the array and second subscript denotes the maximum length that each string can have . The space reserved for this two –d array is 50 bytes.

Here first name of 2-D array is arr it gives the base address of the array i.e. it gives the address of first string.

**arr[0] represents 0th string , points to 0th character of 0th string**

**arr[1] represents 1st string , points to 0th character of 1st string**

**arr[i] represents ith string , points to 0th character of ith string**

**arr[i][j] represents jth character i ith string**

**/\* program to print the strings of the two-dimensional character array\*/**

**#include<stdio.h>**

**main()**

**{**

**char arr[5][10]={“white”,”red”,”green”,”yellow”,”blue”};**

**int i;**

**for(i=0;i<5;i++){**

**printf(“String %s\t”,arr[i]);**

**printf(“Address of string =%u \n”, arr[i]);**

**}**

**Output:**

**String =white Address of string =2000**

**String =red Address of string =2010**

**String =green Address of string =2020**

**String =yellow Address of string =2030**

**String =blue Address of string =2040**

Now we’ll take a program to sort the strings alphabetically using the selection sort technique.

**/\* program to sort the array of strings \*/**

**#include<stdio.h>**

**main()**

**{**

**char arr[5][10]={“white”,”red”,”green”,”yellow”,”blue”};**

**char temp [10];**

**int i,j;**

**printf(“Before sorting :\n”);**

**for(i=0;i<5;i++)**

**printf(“%s “,arr[i]);**

**printf(“\n”);**

**for(i=0;i<5;i++)**

**for(j=i+1;j<5;J++)**

**if(strcmp(arr[i],arr[j])>0)**

**{**

**strcpy(temp,arr[i]);**

**strcpy(arr[i],arr[j]);**

**strcpy(arr[j],temp);**

**}**

**printf(“After sorting . \n);**

**for(i=0;i<5;i++)**

**printf(“%s”,arr[i]);**

**}**

**Output :**

**Before sorting :**

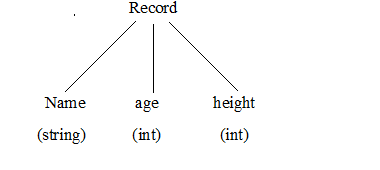
**White red green yellow blue**

**After sorting**

**Blue green red white yellow**

10. Structures and unions

Array is a collection of same type of elements but in many real life applications we may need to group different types of logically related data.For if we want to create a record of a person that contains name, age and height of that personm then we cant use array because all the three data elements are of different types.



To store these related fields of different data types we can use a structure,which is capable of storing heterogeneous data. Data of different types can be grouped together under a single name using structures. The data elements of a structure are referred to as members.

**10.1 Defining a structure**

Definition of a structure creates a template or format that describes the characteristics of its members. All the variables that would be declared of theis structure type, will take the form of this template. The general syntax of a structure definition is:

**Struct tagname{**

**Datatype member1;**

**Datatype member2;**

**……………….**

**Datatype memberN;**

**};**

Here struct is a keyword, which tells the compiler that a structure is being defined. Member1, member2,……………. member are known as members of the structure and are declared inside curlybraces. There should be a semicolon at the end of te curly braces. These members can bo of any data type like int, char, float, array, pointers or another structure type. Tagname is the name of the structure and it is used further in the programto declare variables of this structure type.

Definition of a structure provides one more data tpe in addition to the built in data types. We can declare variables of this new data type that will have the format of te defined structure. It is important to note that definition of a structure template does not reserve any space in memory for the members; space is reserved only when actual variables of this structure type are declared. Although the syntax of declaration of members nside the template is identical to the syntax we use in declaring variables but these members are not variables, they don’t have any existence until they are attached with a structure variable. The member names inside a structure should be different from one another but these names can be similar to any other variable name declared outside the structure. The member names of two different structures may also be same.

Let us take an example of defining a structure template.

**Struct student{**

**Char name[20];**

**int rollno;**

**float marks;**

**};**

Here student is the structure tag and there are three members of this structure viz name, rollno and marks. Structure template can be defined globally or locally i.e. it can be placed before all functions in the program or it can be locally present in a function. If the template is global then it can be used by all functions while if it is local then only the function containing it can use it.

**10.2 declaring structure variables**

By defining a structure we have only created a format, the actual use of structures will be when we declare variables based on this format, we can declare structure variables in two ways

1. **With structure definition**
2. **Using the structure tag**

**10.2.1 with structure definition**

**Struct student{**

**Char name[20];**

**int rollno;**

**float marks;**

**}stu1,stu2,stu3;**

Here stu 1, stu2 and stu3 are variables of type struct student. When we declare a variable while defining the structure template, the tagname is optional.

**Using structure tag**

We can alse decare structure variables using structure tag. This can be written as-

**Struct student{**

**Char name[20];**

**int rollno;**

**float marks;**

**};**

**Struct student stu1,stu2;**

**Struct student stu3;**

Here stu1,stu2,stu3 are structure variables that are declared using the structure tag student. Declaring a structure variable reserves space in memory. Each structure variable declared to be of type struct student has three members vix.name, rollno and marks. The compiler will reserve space for each variable sufficient to hold all the members. For example each variable of type struct student will occupy 28(20+4+4)bytes.

**10.3 initialization of structure variables**

The syntax of initializing structure variables is similar to that of arrays. All the values are given in curly braces and the number, order and type of these values should be same as in the structure template definition. The initializing values can only be constant expressions.

**Struct student {**

**Char name[20];**

**int rollno;**

**float marks;**

**}stu1={“mary”,25,98};**

**Struct student stu2={“john”,24,67.5};**

Here value of members of stu1 will be “mary” for name, 25 for rollno, 98 for marks. The values of members of stu2 will be “john” for name, 24 for rollno,67.5 for marks.

**Note : we cannot initialize members while defining the structure.**

If the number of initializers is less than the number of members then the remaining members are initialized with zero. For example if we have this initialization:

**Struct student stu1={“mary”};**

here the members rollno and marks of stu1 will be initialized to zero. This is equivalent to the initialization.

**Struct student stu1={“mary”,0,0};**

**10.4 accessing member of a structure**

For accessing any member of a structure variable, we use the dot(.) operator which is also known as the period or membership operator. The format for accessing a structure member is

**Structvariable. Member**

Here on the left side of the dot there should be a variable of structure type and on right hand side there should be the name of a member of that structure. For example consider the following structure

**Struct student{**

**Char name[20];**

**int rollno;**

**float marks;**

**};**

**Struct student stu,stu2;**

**Name of stu1 is given by –stu1.name**

**Rollno of stu1 is given by-stu1.rollno**

**Marks of stu1 is given by-stu1.marks**

**Name of stu2 is given by- stu2.name**

**Rollno of stu2 is given by- stu2.rollno**

**Marks of stu2 is given by- stu2.marks**

We can use stu1.name, stu1.marks, stu2.marks etc like any other ordinary variables in the program. They can be read, displayed, processed, assigned values or can be send to functions as arguments. We cant use student.name or student.rollno because student is not a structure variable, it is a structure tag.

**/\*program to display the values of structure members\*/**

**#include<stdio.h>**

**#include<string.h>**

**Struct student{**

**Char name[20];**

**int rollno;**

**float marks;**

**};**

**main(){**

**struct student stu1={“mary”,25,68};**

**struct student stu2,stu3;**

**strcpy(stu2.name,”john”);**

**stu2.rollno=26;**

**stu2.marks=98;**

**printf(“enter name,rollno and marks for stu3: “);**

**scanf(“%s%d%f”,stu3.name,&stu3.rollno,&stu3.marks);**

**printf(“stu1: %s %d%.2f\n”,stu1.name,stu1.rollno,stu1.marks);**

**printf(“stu2:%s %d%.2f\n”,stu2.name,stu2.rollno,stu2.marks);**

**printf(“stu3:%s%d%.2f\n”,stu3.name,stu3.rollno,stu3.marks);**

**}**

**Output:**

**Enter name, rollno and marks for stu3 :Tom 27 79.5**

**Stu1: mary 25 68.00**

**Stu2: john 26 98.00**

**Stu3: tom 27 79.50**

In this program we have declared three variables of type struct student. The first variable stu1 has been initialized,the members of second variable stu2 are given values using separate statements and the values for third variable stu3 are input by the user. Note that since stu2.name is an array so we cant assign a string to it using assignment operator,hence we have used the strcpy() function.

The dot operator is one of the highest precedence operators, its associativity is from left to right.

Hence it will take precedence over all other unary,relational, logical , arithmentic and assignment operators. So in an expression like ++stu.marks, first stu.marks will be accessed and then its value will be increased by 1.

**10.5 assignment of structure variables**

We can assign values of a structure variable to another structure variable if both variables are defined of the same structure type. For example-

**/\*program to assign a structure variables to another structre variable\*/**

**Struct student{**

**Char name[20];**

**int rollno;**

**float marks;**

**};**

**Main(){**

**Struct student stu1={“vector”,12,98};**

**Struct student stu2;**

**Stu2=stu1;**

**Printf(“stu1 :%s %d %.2f\n”,stu1.name,stu1.rollno,stu1.marks);**

**Printf(“stu2 :%s %d %.2f\n”,stu2.name,stu2.rollno,stu2.marks);**

**}**

**Output:**

**Stu1 : vector 12 98.00**

**Stu2 : vector12 98.00**

**10.6 storage of structure in memory**

The members of structures are stored in consecutive memory locations.

**/\*program to show that members of structure are stored in consecutive memory locations\*/**

**#**include<stdio.h>

Main()

{

Struct student{

Char name[5];

Int rollno;

Float marks;

}stu;

Printf(“address of name =%u\n”,stu.name);

Printf(“address of rollno = %u\n”,&stu.rollno);

Printf(“address of marks = %u\n”,&stu.marks);

}

Output:

Address of stu.name=65514

Address of stu.rollno=65519

Address of stu.marks=65523

The output may be different on different machines, and the number of bytes occupied may also vary because of the reasons explained in next section, but the main point to be noted here is that structure members are stored in consecutive memory locations.

**10.7 size of structure**

We may need to find out the size of structure in some situations like reading or wrinting to files. To find out the size of a structure b size of operator, we can either use the structure variable name or the tagname with the struct keyword.for example-

**Sizeof(struct student);**

**Sizeof(stu1);**

**Sizeof(stu2);**

Here if stu1 and stu2 are variables of type struct student, then all the three expressions will give the same result.

Size of structures may be different on different machines. This is because of certain memory alignment restrictions on some computers. For example some machines store integers only at even addresses and long ints only at addresses which are multiple of 4. This is called aligning of data. Consider this structure-

**Struct**

**{**

**Char ch;**

**Int num;**

**}var;**

Here instead of occupying 5 bytes this structure variable will occupy 8 bytes with 3 holes of unused bytes in between. Due to these reasons, size of whole structure may not be equal to the sum of sizes of its members. So it is always better to find the size of structure variable by using sizeof operator rather than using the sum of sizes of its members.

**10.8 array of structures**

We know that array is a collection of elements of same datatype. We can declare array of structures where each element of array is of structure type. Array of structures can be declared as

**Struct student str[10];**

Here stu is an array of 10 elements, each of which is a structure to type struct student, means each element of stu has 3 members, which are name, rollno and marks. These structures can be accessed through subscript notation. To access the individual members of these structures we will use the dot operator as usual.

Stu[0].name stu[0].rollno stu[0].marks

Stu[1].name stu[1].rollno stu[1].marks

Stu[2].name stu[2].rollno stu[2].marks

…………. ………. ………….

Stu[9].name stu[9].rollno stu[9].marks

All the structures of an array are stored in consecutive memory locations.

**/\*program to understand array of structures\*/**

**#**include<stdio.h>

Struct student{

Char name[20];

int rollno;

float marks;

};

main(){

Int i;

Struct student stuarr[10];

for(i=0;i<10;i++)

{ Printf(“enter name,rollno and marks”);

Scanf(“%s%d%f”,stuarr[i].name,stuarr[i].rollno,stuarr[i].marks);

}

The arrayof structures may be initialized using the same syntax as in arrays. For example-

Struct student stuarr[3]={{“mary”,12,98.5},{“john”,11,97},{“tom”,12,89.5}};

The inner pairs of braces are optional if all the initializers are present in the list.

**10.9 arrays within structures**

We can have an array as a member of structure. In structure student, we have taken the member name as an array of characters. Now we’ll declare another array inside the structure student.

**Struct student{**

**Char name[20];**

**Int rollno;**

**Int submarks[4];**

**};**

The array sub marks denotes the marks of students in 4 subjents.

It stu is a variable of type struct student then-

Stu.submarks[0] – denotes the marks of the student in first subject

Stu submarks[1]- denotes the marks in second subject.

Stu.name[0]- denotes the first character of the name member.

Stu.name[4]- denotes the fifth character of the name member.

If stuarr[10] is an array of type struct student then-

Stu.submarks[0]- denotes the marks of the student in first subject

Stu.submarks[1]- denotes the marks in second subject

Stu.name[0]- denotes the first character of the name member.

Stu.name[4]- denotes the fifthe character of the name member.

if stuarr[10] is an array of type struct student then-

stuarr[0].submarks[0] – denotes the marks of first student in first subject

stuarr[4].submaks[3] – denotes the marks of fifth student in fourth subject

stuarr[0].name[0] – denotes the first character of name member of first student

stuarr[5].name[7] – denotes the fourth character of name member of sixth student

**/\*program to understand arrays within structures\*/**

#include<stdio.h>

Struct student{

Char name[20];

int rollno;

int submarks[4];

};

Main()

{

Int i,j;

Struct student stuarr[3];

For(i=0;i<3;i++)

{

Printf(“enter data for student%d\n”,i+1);

Printf(“enter name”);

Scanf(“%s”,stuarr[i].name);

Printf(“enter roll number”);

Scanf(“%d”,&stuarr[i].rollno);

For(j=0;j<4;j++)

{

Printf(“%d”,stuarr[i].submarks[j]);

Printf(“\n”);

}

}

**10.10 nested structures (structures within structure)**

The members of a structure can be of any data type including another structure type i.e. we can include a structure within another structure. A structure variable can be a member of another structure. This is called nesting of structures.

Struct tag1{

Member1;

Member2;

………

Struct tag2{

Member1;

Member2;

…………

Member m;

}var1;

……………….

Member n;

}var2;

For accessing member1 of inner structure we’ll write-

**Var2.var1.member1**

Here is an example of nested structures-

Struct student{

Char name[20];

int rollno;

struct date{

int day;

int month;

int year;

}birthdate;

Float marks;

}stu1,stu2;

Here we have defined a structure date inside the structure student. This structure date has three

Members day, month, year and birthdate is a variable of type struct date. We can access the members of inner structure as-

**Stu1.birthdate.day - day of birthdate of stu1**

**Stu1.birthdate.month - month of birthdate of stu1**

**Stu1.birthdate.year - year of birthdate of stu1**

Here we have defined the template of structure date inside the structure student, we could have defined it outside and declared its variables inside the structure student using the tag. But remember if we define the inner structure outside, then this definition should always be before the definition of outer structure. Here in this case the date structure should be defied before the student structure.

Struct date{

Int day;

Int month;

Int year;

};

Struct student{

Char name[20];

Int rollno;

Float marks;

Struct date birthdate;

}stu1,stu2;

The advantage of defining date structure outside is that we can declare variables of data type anywhere else also

**10.11 pointers to structures**

We have studied that pointer is a variable which holds the starting address of another variable of any data type like int, float or char. Similarly we can have pointer to structure, which can point to the starting address of a structutructurere variable. These pointers are called structure pointers and can be declared as-

**Struct student{**

**Char name[20];**

**Int rollno;**

**Int marks;**

**};**

**Struct student stu,\*ptr;**

Here ptr is a pointer variable that can point to a variable of type struct student. We will use the & operator to access the starting address of a structure variable,so ptr can point to stu by writing

Ptr=&stu;

There are two ways of accessing the members of structure through the structure pointer.

As we know ptr is a pointer to a structure, so by dereferencing it we can get the contents of structure variable. Hence \*ptr will give the contents of stu. So to access members of a structure variable stu we can write-

(\*ptr).name

(\*ptr).rollno

(\*ptr).marks

Here parentheses are necessary because dot operator has higher precedence than the \*operator. This syntax is confusing so c has provided another facility of accessing structure members through pointers. We can use the arrow operator(->)which is formed by hyphen symbol and greater than symbol.so we can access the members as-

**Ptr->name**

**Ptr->rollno**

**Ptr->marks**

The arrow operator has same precedence as that of dot operator and it also associates from left to right.

**/\*program to understand pointers to structures\*/**

**#include<**stdio.h>

Struct student{

Char name[20]

Int rollno;

Int marks;

};

Main(){

Struct student stu={“mary”,25,68};

Struct student\*ptr=&stu;

Printf(“name %s\t”,ptr->name);

Printf(“rollno %d\t”,ptr->rollno);

Printf(“marks %d\n”,ptr->marks);

We can also have pointers that point to individual members of a structure variable. For example-

Int \*p=&stu.rollno;

Float\*ptr=&stu.marks;

The expression &stu.rollno is equivalent to &(stu.rollno)because the precedence of dot operator is more than that of address operator.

**10.12 pointers within structures**

A pointer can also be used as a member of structure. For example we can define a structure like this-

**Struct student{**

**Char name[20];**

**int \*ptrmem;**

**};**

**Struct student stu,\*stuptr=&stu;**

Here ptrmem is pointer to int and is a member of the structure student. To acess the value of ptrmem,we’ll write

**Stu.ptrmem or stuptr->ptrmem**

To access the value pointed to by stu.ptrmem,we’ll write

**\*stu.ptrmem or \*stuptr->ptrmem**

Since the priority of dot and arrow operators is more than that of dereference operator, hence the expression \*stu.ptrmem is equivalent to \*(stu.ptrmem),and the expression \*stuptr->ptrmem is equivalent to \*(stuptr->ptrmem).

**10.13 structures and functions**

Structures may be passed as arguments to function in different ways. We can pass individual members, whole structure variable or structure pointers to the function. Similarly a function can return either a structure member or whole structure variable or a pointer to structure.

**10.13.1 passing structure members as arguments**

We can pass individual structure members as arguments to functions like any other ordinary variable.

**/\*program to understand how structure members are sent to a function\*/**

#include<stdio.h>

#include<string.h>

Struct student{

Char name [20];

int rollno;

int marks;

};

Display(char\*,int,int);

Main()

{

Struct student stu1={“john”,12,87};

Struct student stu2;

Strcpy(stu2.name,”mary”);

Stu2.rollno=18;

Stu2.marks=90;

Display(stu1.name,stu1.rollno,stu1.marks);

Display(stu2.name,stu2.rollno,stu2.marks);

}

Display(char\*name,int rollno,int marks)

{

Printf(“name- %s\t”,name);

Printf(“rollno-%d\t”,rollno);

Printf(“marks-%d\n”,marks);

**}**

**Output:**

**Name –john rollno-12marks-87**

**Name –maryrollno-18 marks-90**

Here we have passed members of the variables stu1 and stu2 to the function display(). The names of the formal arguments can be similar to the names of the members.we can pass the arguments using call by reference also so that the changes made in the called function will be reflected in the calling function. In that case we’ll have to send the addresses of the members. It is also possible to return a single member from a function.

**10.13.2 passing structure variable as argument**

We can pass the whole structure as an argument

**/\*program to understand how a structure variable is sent to a function\*/**

**#include<stdio.h>**

**Struct student{**

**Char name[20];**

**int rollno;**

**int marks;**

**};**

**Display(struct student);**

**Main()**

**{**

**Struct student stu1={“john”,12,87};**

**Struct student stu2={“mary”,18,90};**

**Display(stu1);**

**Display(stu2);**

**}**

**Display(struct student stu)**

**{**

**Printf(“name -%s\t”,stu.name);**

**Printf(“rollno-%d\t”,stu.rollno);**

**Printf(“marks-%d\n”,stu.marks);**

**}**

**Output:**

**Name-john rollno- 12 marks-87**

**Name-mary rollno-18 marks-90**

Here it is necessary to define the structure template globally because it is used by bothe functions to declare variables.

The name of a structure variable is not a pointer unlike arrays, so when we send a structure variable as an argument to a function,a copy of the whole structure is made inside the called function and all the work is done on that copy. Any changes made inside the called function are not visible in the calling function since we are only working on a copy of the structure variable, not aon the actual structure variable.

**10.13.3 passing pointers tostructures as arguments**

We can access the members of the structure variable inside the calling function using arrow operator. In this case any changes made to the structure variable inside the called function, will be visible in the calling function since we are actually working on the original structure variable.

**/\*program tounderstand how a pointer to structure variable is sent to a function \*/**

**#include<stdio.h>**

**Struct student{**

**Char name[20];**

**int rollno;**

**int marks;**

**};**

display(struct student \*);

inc\_marks(struct student\*);

main()

{

Struct student stu1={“john”,12,87};

Struct student stu2={“mary”,18,90};

Inc\_marks(&stu1);

Inc\_marks(&stu2);

Display(&stu1);

Display(&stu2);

}

Inc\_marks(struct student \*stuptr)

{

(stuptr->marks)++;

}

Display(struct student \*stuptr)

{

Printf(“name- %s\t”,stuptr->name);

Printf("rollno-%d\t",stuptr->rollno);

Printf("marks-%d\n",stuptr->marks);

}

Output:

Name - john rollno-12marks-88

Name- mary rollno-18 marks-91

**10.13.4 Returning a structure variable is returned from function**

Structure variables can be returned from functions as any other variable. The returned value can be assigned to a structure of the appropriate type.

***/\*programe to undersand how a structure variable is returned from a function\*/***

*#include<stdio.h>*

*Struct student{*

*Char name[20];*

*Int rollno;*

*Int marks;*

*};*

*Void display (struct student);*

*Struct student change(struct student stu); main()*

*{*

*Struct student stu1={"john",12,87};*

*Struct student stu2={"mary",18,90};*

*Stu1=change(stu1);*

*Stu2=change(stu2);*

*Display(stu1);*

*Display(stu2);*

*}*

*Struct student change(struct student stu)*

*{*

*Stu.marks=stu.marks+5;*

*Stu.rollno=stu.rollno-10;*

*Return stu;*

*}*

*Void display(struct student stu)*

*{*

*Printf('name-%t",stu.name);*

*Printf("rolln0-%d\t",stu.rollno);*

*Printf("marks-%d\n",,stu.marks);*

*}*

***Out put:***

*Name - john rollno-2marks-92*

*Name - mary rollno-8 marks-95*

**10.14 Self Referential Structure**

A structure that contains pointer to structures of its own type is known as self referential

Structure. For example

Struct tag

{

Data type member1;

Data type member2;

......................

Struct tag \*ptr1;

Struct tag \*ptr2;

};

Here ptr1 and ptr2 are structure pointers that can point to structure variables of type struct tag, so struct tag is a self-referential structure. These types of structures are helpful in implementing data structures like linked and trees.

**10.5 Unions**

Union is a user defined data type like structure and it can also contain members of different data types. The syntax used for definition of a union, declaration of union variables and for accessing members is similar to that used in structures, but here keyword union is used instead of srtuct. The main difference between unions in structures is in the way memory is allocated for the members. In a structure each member has its own memory location, whereas members of union share the same memory location. When a variable of type union is declared, compiler allocates sufficient memory to hold the largest member in the union. Since all members share the same memory location hence we can use only one member at a time. Thus union is used for saving memory. The concept of union is useful when it is not necessary to use all members ofthe union at a time.

The syntax of definition of a union is-

***Union union\_ name***

***{***

***Datatype member1;***

***Datatype member2;***

***..............................***

***};***

Like structure variables, the union variables can be declared along with the definitionor seperately. For example

***Union union\_ name{***

***Datatype member1;***

***Datatype member2;***

***..............................***

***};variable\_name*;**

This can also be declared as-

**Union union\_name variable\_name;**

We can access theunion members using the same syntax used for structures. If we have a union variable then the members can be accessed using dot(.) Operator, and if we have a pointer to union then the members can be accessed using the arrow (->) operator.

***/\*programe for accessing union members\*/***

*#include<stdio.h>*

*Main()*

*{*

*Union results {*

*Int marks;*

*Char grade;*

*Float per;*

*}res;*

*Res.marks=90;*

*Pritf("marks : %d\n",res.marks);*

*Res.grade='a';*

*Printf("grade : %c\n",res.grade);*

*Res.per=85.5;*

*Printf("percentage %f\n",res.per);*

*Printf("grade : %c\n",res.grade);*

*}*

***Output:***

***Marks:90***

***Grade:a***

***Percentage:85.500000***

Before the first printf, the value 90 is assigned to the union member marks, so other members grade and per contain garbage value. After first printf, "the value 'a' is assigned to the union members grade. So now the other two members marks and per contain garbage value. Only one member of union can hold value at a time, don't try to use all the members simultaneously. So a union variable of type result can be treated as either an int variable or a float variable. It is the responsibility of the programmer to keep track of member that currently holds the value.

Union variables can also be initialized, but there is a limitation. We know that due to sharing of memory, all the members can't hold values simultaneously. So during initialization also only one member can be given an initial value, and this privilege is given to the first member. Hence only the first member of a union can be given an initial value. The type of the initializer should match with the type of the first member. For example, we can initiate the above union variable as-

**Union result res ={78};**

*Now we'll take a program and compare the memory allocated for a union and strcture variable.*

*/\*****program to compare the memory allocated for a union and structure variable\*/***

*#include<stdio.h>*

*Struct stag{*

*Char c;*

*Int i;*

*Float f;*

*}*

*Union tag {*

*Char c;*

*Int i;*

*Float f;*

*};*

*Main()*

*{*

*Union utag uvar;*

*Struct stag svar;*

*Printf("size of svar = %u\n",size of (svar));*

*Printf("address of svar : %u\t",&svar);*

*Printf("address of members : %u %u %u\n",&svar.c,&svar.i,&svar.f);*

*Printf("size of uvar=%u\n", size of (uvar));*

*Printf("address of uvar : %u\t",&uvar);*

*Printf("address of members : %u %u %u\n",&uvar.c,&uvar.i,&uvar.f);*

*}*

***Output:***

*Size of svar = 7*

*Address os svar :65514*

*Address of members : 65514 65515 65519*

*Size of uvar = 4*

*Address of uvar : 65523*

*Address of members : 65523 65523 65523*

The address of members of a union are same while the address of members of a structure are different. The difference in the size of variables svar also indicates that union is very economical in the use of memory.

A structure may be a member of union may bea member of structure. For example

*{*

*Int marks;*

*Char grade;*

*};*

*Struct res*

*{*

*Char name[15];*

*Int age;*

*Union result performance;*

*}data;*

Here data is the structure variable of type struct res. It has three members, an array of characters name, an integer age and a union member performance.union will take only one value ata atime, either an integer value marks or a character value grade. This can also be written as.

*Struct res*

*{*

*Char name[15];*

*Int age;*

*Union result*

*{*

*Int marks;*

*Char grade;*

*}performance;*

*}data;*

If we want to use the member grade then we can write-

Data.performance.grade

Similarly to use the member marks we can write-

Data.performance.marks

**10.15.1 some other features of union are**

**(1) array of unions can be declared.**

**(2) functions can take union variable as arguments and can return union variables.**

**(3) pointers t unions can be declared.**

**(4) unions can be nested.**

**10.15.2 endian concept**

Unions are also helpful in low level programming. We may manipulate the individual bytes in a data type using union. For example we can find whether a given machine's byte order is little-endian or big-endian. First we'll see what i a machine's byte order and then we'llwrite a program to determine the same.

The byte orderof a machine specifies the order in which a multibyte data item is stored in memory or disk. There are two common byte orders.

**Big-endian - most significant byte is stored at the lowest address**

**Lowest-endian - least significant byte is stored at the lowest address.**

Intel family of processors use little-endian byte order, motorola family of processors use big-endian byte order.

The binary representation of integer 5193 in 2 byte is-

**000i 0i00 0i00 i00i**

**Msbyte lsbyte**

The following figure shows how this integer is stored in different byte orders.

**Little-endian**

**2000 0100 1001 lsb**

**2001 0001 0100 msb**

In little- endian,the least significant bye is stored at the lower address (2000), while the most significant byte is stored at the higher address (2000). In big-endian the msb is stored at the lower address ( 2000),while the lsb is stored at thehigher address (200i).

The following program determines the byte order of a machine

***/\*program to determines the byte order of a machine\*/***

*#include<stdio.h>*

*main()*

*{*

*union{*

*int x;*

*char c [2];*

*}*

*var;*

*var.x=1;*

*If(var.c[0]==1)*

*Printf("little endian\n");*

*else*

*Printf("big endian\n");*

*}*

**10.16 typedef**

The type definition facility allows us to define a new name for an existing data type. The general syntax is-

**Typedef data\_type new\_name**

Here typedef is a keyword, data\_type is any exixting data type that may be a standard data type or a user defined type, new\_name is an idenifier, which is name for this new name for this data type. Note that we are not creating any new data type but we are only creating a new name for the existing data type. For example we can define anew name for int type by writing-

**Typedef int marks;**

Now mrks is a synonym for int andwe can use marks instead of int anywhere in the program, for example-

**Marks sub1,sub2;**

Here sub1, sub2 are actually int variablesand are similar to any variable declared ising int keyword. The above declaration is equivalent to -

**Int sub1, sub2;**

Some more examples are-

**Data def float real;**

Here real is another name for float. The typedef declaration can be written wherever other declarations are allowed. We can give more than one name to a single data type using only one typedef statement.for example-

**Typedef int age, marks, units;**

In the above type statement, we have defined three names for the data type int.

Since typedef is syntactically considered as a storage class, so we can't include a storage class in typedef statement. Foe example statements of these types are invlid-

**Typedef static char achar; /\*invalid\*/**

**Typedef extern int marks; /\*invalid\*/**

Now we'll see how typedef can be used to define new names for pointers, arrays, functions and structures.

**1. Pointers**

**Typedeffloat \*fptr;**

After this statement, fptr is a synonym for float \* or pointer to float.now consider this declaration-

**Fptr p,q,\*r;**

Here p and q are declared as pointer to float, while r is declared as a pointer to pointer to float.

**2. Array**

**Typedef int intarr[10];**

After this statement intarr is another name for integer arraysof size 10.now consider his declaration statement-

**Intarr a,b,c[15]; (equivalent to -> inta[10], b[10], c[5][10];)**

Here a, b are declared as 1-d arrays of size 10, and c is defined as 2-d array of size 15x10.

**3. Functions**

**Typedef float funct(flot,int);**

Here function is any function that takes two values, one float and one int value and return a float value. The above statement is equivalent to thefollowing declaration statements-

**Float add(float, int);**

**Float sub(float, int);**

**Float mul(float, int);**

**Float div(float, int);**

**4. Structures**

Similarly we can also use typedef for defining a new name for structures. Suppose we have this structure definition-

**Struct studentrec{**

**Char name[20];**

**Int marks;**

**};**

Now whenever we want to use this structure we have to write atruct studentrec. We can give a short and meaningful name to this structure by typedef.

**Typedef struct studentrec student;**

Now we can declare variable like this-

**Student stu 1, stu 2;(equivalent to -> struct studentrec stu 1, stu 2;)**

We can also combine typedef and structure definition. The syntax is as-

***Typedef struct student{***

***Char name[20];***

***Int age;***

***}person;***

**Person student, teacher, emp;**

Here person is a name for their structure and we have defined three structure variables, which have the format of the above definition.

**10.16.1 advantages of using typedef**

1. It makes our programs more readable and understandable since we can document our program by giving meaningful and descriptive names for existing types.

2. In structures it is important since we can give a single name to the structure, we need not write keyword repeatedly.

3. It makes our programs more portable. When program is run on a different machine on which standard data types are representedby different numberof bytes, only typedef statement has to be changed.

The typedef declarations may seem similar to #define directives, but they are different. The basic difference is that the typedef declarations are interpreted by the compiler while #define directives are processed by the preprocessor. In #define directive we can define an expansion for any text while in typedef we can only define new names for the data types only.

**10.17 enumeration**

Words speak more than numbers and this is the reason for the inclusion of enumerated data types in c. Sometimes the replacement of integer constants like 1,2, 3 by some meaningful and respectful and descriptive names, enhance the readability of the code and makes it self documenting. For example suppose we are making a data related program, then it would be better if we could use names like jan, feb, mar, apr instead of the numbers 1,2,3,4.

**An enumeration type is user defined data type, which can take values only from a user defined list of named integer constants called enumerators. The syntax of defining an enumeration data type is same as the that of structure or union. The general format of definition is-**

***Enum tag{***

***Member1;***

***Member2;***

***.................***

***};***

Here enum is a keyword, tag is an identifier that specifies the name of the new enumeration type being defined, and member1, member2 are identifiers, which represent integer constants and are called enumerator constants or enumerators. The list of these enumerators is called enumerator list. Note that unlike structure and union, here the members inside the braces are not variables, they are named integer constants.

After the definition, we can declare variables of this new dta type as-

**Enumtag var1, var2, var3;**

Here var1, var2, var3 are variables of type enum tag. These variables can take values only from the enumerator list.

The variables can also be declared with the definition as-

***Enum tag {***

***Member1;***

***Member2;***

***.................***

***}var1,var2,var3;***

Here the tag is optional. Letus take an example-

**Enum month{ jan, feb, mar, apr, may,jun };**

Here a new data type month is defined and the enumerator list contains six enumertors.

Internally the compiler treats these enumerators as integer constants. These are automatically assigned integer values beginning from 0,1,2 ....etc till the last member of the enumeration. In the above example, these enumerators will take following values-

**Jan 0**

**Feb 1**

**Mar 2**

**Apr 3**

**May 4**

**Jun 5**

These are the default values assigned to the enumerators. It is also possible to explicity assign any value to enumerators but in the case, the successive unassigned enumerators will take values one greater than the value of the previous enumerators. For example

**Enum month{ jan, feb = 4, mar, apr, may = 11, jun};**

Now the enumerators will take following values

**Jan 0**

**Feb 4**

**Mar 5**

**Apr 6**

**May 11**

**Jun 12**

We can assign any signed integer value to enumerators, provided the value is within the range of int. It is also possible to assign same value to more than one enumerator.

**Enum month{ jan, january =0,feb =1,february =1,mar=2,march =2};**

The enumerated variables can be processed like other integer variables. We can assign value to them from the enumeratorlist or they can be compared to enter variables and values of teh same type. For example

**Enum month{ jan, feb, mar, apr, may, jun}m1,m2;**

**M2=mar;**

**M2=may;**

Now m1has integer value 2 and m2 will take the value 4.

Any variableof type enum month can take values only from the 6 enumerators specified in the list.for example this is vald

**M1=dec; /\*invalid\*/**

Some other examples of enum data type definitions are-

**Enum suit{spades, hearts, clubs, diamonds};**

**Enum position{ace =1,king,queen, jack, ten, nine, eight, seven, six, five, three, two};**

**Enum month{ jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec};**

**Enum day{sunday, monday, tuesday, wednesday, thursday, friday, saturday};**

**Enum color{white, black, red, green, blue, yellow, pink, brown};**

**Enum boolean{true, false};enum switch{off, on};**

**Enum subject{hindi, english, maths, physics, chemistry, biology, history};**

**Enum base{binary=2,octal=8,decimal=10,hexadecimal=16};**

Here are some examples of code using these enum definitions-

1. Enum color walls, floor;

If(walls==pink)

Floor=blue;

Else

Floor=white;

*2. Enum day today;*

*Today=sunday;*

*If(today==sunday)*

*Printf("holiday\nh);*

*Else if(today==saturday)*

*Printf("half working day\nh);*

*Else*

*Printf("full working day\nh);*

*3. enum subjects s;*

*Int passmarks;*

*Switch(s)*

*{*

*Case hindi;*

*Case english: passmarks=25;*

*Break;*

*Case physics;*

*Case chemistry: passmarks=35;*

*Break;*

*Default: passmarks=33;*

*}*

***4. Suppose we need a function that returns number of days in a mmonth. It can be written as***

*Enum month m;*

*Int days;*

*If(m==apr ii m==jun ii m=sep ii m==nov)*

*Days=30;*

*If(m==jan ii m==mar ii m==may ii m jul ii m==aug ii m oct ii m==dec)*

*Days=31;*

*If(m==feb)*

*Days=28*

***/\*program to print the value of enum variables\*/***

*#include<stdio.h>*

*Main()*

*{*

*Enum month{jan,feb,mar,apr,may,jun}m1,m2;*

*M1=mar;*

*Printf("m1 %d \n",m1);*

*Printf("enter value for m2");*

*Scanf(%d",&m2);*

*Printf("m2 %d \n",m2);*

*}*

***Output:***

*M1=2*

*Enter the value for m2:5*

*M2=5*

It is not possible to perform input and output in terms of enumerator names. The input and output is only in the form of their integer values. You may be tempted to use %s to output and input enumerated variables but this is invalid since enumerators are not strings.

***Printf("%s",m1);/\*invalid\*/***

Since enumerators are identifiers so their name should be different from other identifiers in the same scope. The following code is wrong because the identifier chemistry has benn used at two different places.

***Enum group 1{physics, chemistry, maths};***

***Enum group2{zoology, botany, chemistry}; /\*invalid\*/***

***Float maths; /\*invalid\*/***

We can use typedef in the definitions of enum, for example

***Typedef enum { false, true } boolean;***

Now we can define variables like this-

***Boolean flag = true;***

We can always write our programs by using integer variables instead of enumerated variables, but use of enum in complicated programs makes the program more understandable.

**10.18 Bit fields**

In some applications it may be desirable to work with data items that contains only of bits (e.g.,a single bit flag to indicate true/false condition, a 3-bit integer whose values can range from 0 through 7. Or a 7 bit ascii character). Several such data items can be packed into an individual word of memory. To do so, the word is subdivided into individual bit fields. These bit fields are defined as members of structure. Each bit field can be accessed individually. Like any other member of a structure.

In general terms, the decomposition of a word into bit fields can be writen as

***Struct tag***

***{***

***Member1:n;***

***Member2:n;***

***Membern:n;***

***};***

N indicates no of bits.

Where, the individual elements have the same meaning as in a structure. Each member declaration must now order the bit fields from right-to-left, whereas other c compilers will order them left-to-right. We will assume right-to-left ordering in the examples shown below.

***Example: a cprogram contains the following declarations.***

***Struct sample***

***{***

***Unsigned a:1;***

***Unsigned b:3;***

***Unsigned c:2;***

***Unsigned d:1;***

***};***

***Struct sample temp;***

The first declaration defines a structure, which is subdivided into four bit fields, called a, b, c, and d, these bit fields have widths of 1, 3, 2 and i bit respectively. Hence, the bit fields occupy a total of 7 bits within a word of memory. Any additional bits within the word the word will remain uncommitted.

The figure below illustrates the layout of the bit fields within the word, assuming a 16- bit word with the fields ordered from right-to-left.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Uncommitted bits d c b a

Bitno:15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

The second declaration states that temp is a structure variable of type sample. Thus, temp a is a field within temp whose width is i bit. Similarly temp.b is a fieldwhose width is 3 bits and so on.

A bit field can only be defined as a portion of an integer to an integer or an unsigned word (some compilers also permit a bit field to be a portion of a char or a long word). In all other respects, however, the rules for defining bit fields are the same as the rules that govern other kind of structure.

***Example: the declaration in the above examples can be combined to read struct sample***

***{***

***Unsigned int a:1;***

***Unsigned*** ***int b:3;***

***Unsigned int c:2;***

***Unsigned int d:1;***

***}temp;***

The interpretation of the variables temp is the same as that given in above example. Moreover, the tag can be omitted, so that the above declaration can be further shortened to

***Include<stdio.h>***

***Main()***

***{***

***Struct***

***{***

***Unsigned int a:5;***

***Unsigned*** ***int b:5;***

***Unsigned int c:5;***

***Unsigned int d:5***

***}str={1,2,3,4};***

***Printf("\n str.a = %d =%d, str.d = %d\n", str.a, str.b, str.c, str.d);***

***Printf( "the required %d bytes",sizeof(str));***

***}***

The four fields within str require total of 20 bits. If the computer allows only 16 bits for an unsigned integer quantity, this structure declaration will require 2 words of memory. The firs three fields will be stored in the first word. Since the last fields straddle the word boundary, it is automatically forced to the beginning of the second word.

Executing this program will produce the following output:

Str.a=1,str.b=2,str.c=3 require 2 bytes.

Let us alter this program by adding an unnamed field whose width is 6 bits. I.e,

***Main()***

***{***

***Static struct***

***{***

***Unsigned int a:5;***

***Unsigned*** ***int b:5;***

***Unsigned int c:5;***

***}str{1,2,3};***

***Printf("\n str.a = %d =%d, str.d = %d\n", str.a, str.b, str.c, str.d);***

***Printf( "the required %d bytes",sizeof(str));***

***}***

This program is similar to the second program shown in the above example. Now, however, the structure declaration includes as unnamed bit field width is 0. This will automatically force the last field to tie beginning of a new word. Executing this program would result into

***Str.a=1,str.b=2,str.c=3 requires 3 bytes.***

The reader is again reminded that some compilers order bit fields from right-to-left (i.e., low-order-bits to high-order-bit to low-order-bit).

*#include<stdio.h>*

*Main()*

*{*

*Struct date*

*{*

*Unsigned int day:4;*

*Unsigned int month:4;*

*Unsigned int year:8;*

*};*

*Struct date dob={6,7,75);*

*Printf("\n date of birth:%d %d\n", dob.day, dob.month, dob.year);*

*}*

***The output would be : date of birth:6 7 75***

Here day has 4 bits so the number can be represented by member variable day is up to (2 power 4)-1=15.. Member variable month can also and member variable year can take up to (2 power 8)-1=255.

If sum of the fields is more than the size of word, it occupies next word.

#include <stdio.h>

#include "search.h"

struct node

{

struct record rec;

struct node \*pnxt;

};

#define MAX\_HASH\_SIZE 17

struct node \*hashArray[MAX\_HASH\_SIZE];

int myHashFun(int phone)

{

return phone % MAX\_HASH\_SIZE;

}

int searchRec(int phone, struct record \*prec);

{

int hix;

struct node \*ptmp;

hix = myHashFun(phone);

ptmp = hashArray[hix];

while(ptmp)

{

if(ptmp->rec.phone == phone)

{

\*prec = ptmp->rec;

return SUCCESS;

}

ptmp = ptmp->pnxt;

}

return NOT\_FOUND\_ERR;

}

int addRec(const struct record \*prec)

{

int hix;

struct node \*pn;

pn = malloc(sizeof(\*pn));

if(pn==NULL)

return NO\_MEMORY\_ERR;

pn->rec = \*prec;

hix = myHashFun(prec->phone);

pn->pnxt = hashArray[hix];

hashArray[hix] = pn;

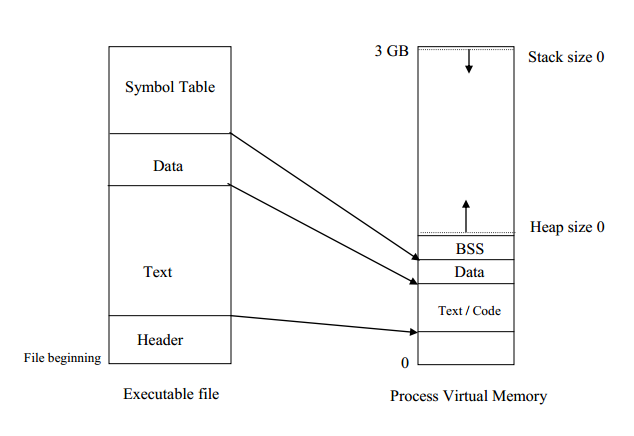
return SUCCESS;

}

int delRec(int phone)

{

int hix;



Commands to get information about executable (binary) file:

 size

 nm

 objdump

 strip

#size a.out

text data bss dec hex filename

1175 260 12 1447 5a7 a.out

#nm a.out

080496b4 A \_\_bss\_start

080496a4 D \_\_data\_start

080496b4 A \_edata

080496c0 A \_end

08048538 T \_fini

08048554 R \_fp\_hw

08048278 T \_init

080482c0 T \_start

080482e4 t call\_gmon\_start

080496b4 b completed.1

080496a4 W data\_start

0804833c t frame\_dummy

080496b8 B g1

080496bc B g2

080496b0 D g3

08048368 T getLargestNum

08048395 T getSmallestNum

080483c2 T main

**Static Library vs Dyanmic Library:**

Static Library:

1. gcc -Wall -c \*.c

2. ar cvq libstr.a \*.o

3. gcc main.c ./libstr.a -o output --- lib is in current directory

OR

copy file to /usr/local/lib --- lib is in /usr/lib

gcc main.c -lstr -o out

OR

gcc main.c -L /path/ -lstr --- lib is in some other folder

Dynamic library:

1. gcc -Wall -fPIC -c \*.c

2. gcc -shared -Wl,-soname,libctest.so -o libctest.so \*.o

copy the libctest.so to /lib folder

OR

3. export LD\_LIBRARY\_PATH=$LD\_LIBRARY\_PATH:/path/to/lib

4. gcc main.c -lctest

g++ test.cpp -lpthread -I/usr/local/include/libusb-1.0 -lusb-1.0

**Make utility**

Typical project consists of tens or hundreds of C program files. So if you want to compile all these

files, you have to give a command some thing like below:

$gcc -o program cfile1.c cfile2.c cfile3.c . . . cfile24.c

If you have used any IDE tool for software development, you will find project concept. Every IDE

provides a method to create a project with certain name and add files to the project. When you

give a command to „build‟ project, all the files in the project will get compiled and linked to

generate executable file.

The „make‟ command will do the samething in command line environment. To use make

command first we have to prepare a file with name “makefile”. In that file we have to give names

of all our object files. The following is the sample “makefile” to build your project.

CC = gcc

CFLAGS = -c

OBJS = mn.o myslen.o myscpy.o myscat.o

myprog: $(OBJS)

$(CC) $(OBJS) -o myprog

mn.o: mn.c

$(CC) $(CFLAGS) mn.c

myslen.o: myslen.c

$(CC) $(CFLAGS) myslen.c

myscpy.o: myscpy.c

$(CC) $(CFLAGS) myscpy.c

myscat.o: myscat.c

$(CC) $(CFLAGS) myscat.c

install:

cp myprog /usr/local/bin

clean:

rm \*.o myprog

static inline functions

volatile constant