C-Programming

Ashok Shankar Das

April 14, 2008

Hello! This material is free. You are encouraged to use it. But if something bad i mean any thing bad happened to you like you lost your keys or even blown your computer then the author is not responsible. But it will help the author a lot if you can send errata to his e-mail: ashok.s.das@gmail.com for future modifications. The intent of author is to help beginers. author is no way a pro because he thinks life is a learning process. Hence your suggestions and modifications will teach author a lot. The plan author has is to add some project codes. Those projects should be intended for beginers. Any thing like a small database management system, text based game tic-tac-toe will be great.

Lastly author reserves the copyright.

Contents

Chapter 1

Introduction

1.1 History Of Computer

Gone were the days when the human race started counting numbers, with objects, like fingers, pebbles, sticks etc. With the advancement of civilization, they devised methods to add, subtract, multiply and divide. Gradually those methods of representing them were modified too. In the process of modification they came across machines from Abacus to calculators. In the process of development Blaize Pascal had invented an incredible machine known as Calculating machine.

The calculating machine was remarkable in certain way that it had features to store data for future use. It had a memory unit. The whole of calculating machine was made of mechanical part and Blaize Pascal gave the idea of mechanical memory.

Based upon the Calculating machine mechanical desk calculators were designed. In those machines there was a keyboard similar to typewriter but had number of function (operation) keys. You had to key-in the numbers send them to accumulator and use function-keys to perform operation on the numbers you entered.

Then comes the era of Electrical machines. Now the electric energy is being used to drive those machines. When Samuel Morse developed his Telegraph machine and his Morse code, it gave rise to a state represented character. This idea gave birth to Binary representation. In the time of world war-II electro-magnetic switches (relays) were used to control different machineries. As we all know, a switch has 2-states, namely ON and OFF, so relays have also 2-states. Using relays some scientists developed a decision-making machine for war strategy. It was quite efficient in those days. When vacuum tubes (diodes, triodes) were developed in the era of electronics the old relays were replaced with them. Now scientists got an electronic version of a decision making machine. They started adding features and upgrading it. Ultimately they made a machine, named ENIAC. It was a huge machine having switch for input data and bulbs for output. At that time scientists assumed if they could have four such machines then the computing need of the world would have been solved. As technology grew, with them grew the Electronic industry and new electronic devices were made. The transistor was developed when scientists used semiconductors. The use of semiconductor devices like

transistors and diodes reduced the size of computer. Then came the Integrated Circuits (ICs). One IC could accommodate several transistors. The size of the computer is again reduced. Then LSI (Large Scale Integration) and VLSI (Very Large Scale Integration) technology were developed. Hence an IC can now accommodate several thousands of transistors on itself. Now computer made its way to Desktop and became affordable.

As the computer became smaller and smaller, and HLLs were introduced its cost decreased and more people started taking interest in it. As it was difficult to operate a computer by making a combination of switches On or Off, some scientists started developing alternative methods. They added a keyboard, (which is like typewriter keyboard) a suitable output device that can print the output on paper, etc.& also software to control them. In this state even they used to program the computer in 0 and 1. Programming in 0s and 1s are pretty difficult. Hence they developed some easier means. The outcome of this effort is assembly language. Now they are free from punching 0s and 1s. But programming in Assembly Language is not so easy task. It had its won drawbacks too. So they started developing high-level languages.

The first programming language was developed for scientific community only. It is FORTRAN, which stands for Formula Translation. Then the large business houses showed some interest in computer. To satisfy their need COBOL (Common Business Oriented Language) was developed. As the cost of computer slashed with every new technology, Common man got interested in computer hence the computer found its way to home. Now layman is not going to program in either of the languages mentioned above, because he does not need those things. So a separate easy to understand normal English like programming language was developed which is known as BASIC. BASIC stands for (Beginners All-purpose Symbolic Instruction Code). It is a very good language we can say from the point of ease of programming and simplicity. Still it lacks some features like modularity. To implement modularity in computer language scientist developed another language known as PASCAL. It is a very good modular and structured language. Still it lacks features for system programmer. Hence to satisfy their need C-language was developed.

1.2 History and Features of C

C language was developed by a research group at AT&T Bell labs, which demonstrated the feature to program system software in a high level language. The designers of this language were Denise Ritchie and Brian Kernighan. They developed this language on a PDP-7 machine made by DEC corporation. The whole translator (compiler) for this new language was written in assembly language of PDP-7. When they had a working compiler for their new language C, they rewrote the compiler in C again. When DEC announced the PDP-11, by only changing the code generator section they got a C compiler for the new machine.

C is designed in such a way that everyone can use it, from a student to a professional; from a layman to a programmer. Capability of this language, one can say is unlimited, as 90% of the software spectrum can be covered or developed in C. It is portable that is if a program is written for some platform (system) it can be easily taken to other platforms. As it is a High-level language, it is similar to normal English.

Besides its feature of being a high level language it has several feature of low-level language. So it is treated as a mid-level language or hybrid language.

1.3 Course direction

The approach I intend to use is to learn programming in C. This is not intended to teach theoretical aspects of C. So you may notice some broken links in topics. Those are because I think it is important that you the reader should learn to write program first. Then we should discuss the aspects when we feel comfortable. Below is a list of the things we will see in this course.

- Overview of programming.
- A basic C program to demonstrate the different parts and start the fire of baptism in you.
- Fundamental data types, and there representation.
- Simple arithmetic operators and simple logical operators,
- Discussion of branching statements. (If...else, nested if, switch...case)
- Loops.
- Compound data types. (Arrays, string, Structures and unions)
- Functions definition and use.
- Pointers. (Relation between arrays and pointers, pass by reference, pointer to structures, function returning pointers)
- Advanced concept in C

Chapter 2

Overview Of Programming

2.1 Computer based Problem solving

2.1.1 Introduction

Computers are mere machines. They cannot do any thing of there own, hence must be instructed what to do. This set of instruction is known as program. The programs are written in different programming languages. Imagine while talking to a shopkeeper if you talk physical specification and the internal working of a computer then will he understand. So you the first person you are intelligent enough to switch the context. The same thing happens with the computer. For different purpose different programming languages are used.

2.1.2 Problem definition

Real life computer programs are not written for demonstrating the algebraic functions or how a logical expression is evaluated. They generally developed for the requirement of client. As an example, the client is a big manufacturing unit, then the requirement maybe to develop an application which can be helpful to track the raw material in hand, market requirement of finished product, cost analysis, supply tracking, collection of payments and pending payments.

Here the requirement stated above is a broad problem statement. Generally this kind of problem statement is broken into sub statements and modules are written to implement them.

2.1.3 Use of example for problem solving

When we are in junior school our teacher used to give us example like we have 5 apples and we sold 3 at rupees 4.00 each and the rest 2 apples are eaten by us. At the end we saw we have no profit no loss so we were instructed to find out what is the cost price. Similarly here in computer programming we have to consider examples. For the problem of the large production unit you can take example of a small countryside hotel

or even a tea stall near you. The nature of requirement is same if you watch carefully you can understand that. So basically if you develop an application for a tea stall then that application can work for a large production unit with minimal modifications. When ever a problem looks critical then search for the similarity with a small problem then the approach to the solution will be easier. This method is known as prototyping.

2.1.4 Similarities between problems

In above example the application for tea stall is similar to the application needed by the large production unit the basic logic is same but the program output will be definitely bigger in case of production unit. Like a paper boat and a real boat uses same principle to float but they cannot be same. Using matchbox you design nice flats but you cannot live inside that.

2.1.5 Problem solving strategies

You got some idea in previous sections. We discussed about modules and binding them in to a single program. This method, which involves breaking up into modules and combines them to make a finished product, is known as 'divide and conquer'. Considering the approach needed to solve a problem there are 2-types of ways top-down and bottom-up. In top-down approach one goes from the beginning using predefined set of rules, whereas in bottom-up approach the result is known first then we come to the beginning.

2.2 Program design and implementation issues

We already know that writing real life application is not a easy job though it is not impossible. For this there are certain well-defined ways and approach present which one has to follow.

2.2.1 Programs and algorithms

Programs are nothing but a set of executable instructions. Does it mean a set of instruction of any combination? NO so we can modify the definition to it is a set of instruction but having a goal. This means the program does something useful or fruitful. The way any goal is achieved from an initial condition is known as algorithm. In other words algorithm is the systematic way or set of rules which when followed at initial condition leads to the result.

2.2.2 Top-down design and stepwise refinement

The top-down design is approaching the problem from the data given and reaching at the solution. Recall your high school algebra, in the assignment you start a problem from the left hand side and arrive at result on right hand side. Top-down design is the same. Just think of a situation, you have to reach at Tajmahal. It is obvious you

cannot reach there by staying inside; hence you have to get out of your apartment. Tajmahal is not outside your apartment, so you have to take a cab. If you are a resident of Agra in India then this is sufficient. If you are from Texas, USA then the process is different. You have to reach the airport take a flight to India then arrive at Indiragandhi International Airport, Delhi. Go to a hotel, book a room get refreshed, ask for a travel agent and go to Agra and see the Tajmahal.

Well the above example is actually not meant for your travel plan. It shows the way the top-down approach works. And it also shows how logically we are heading our way to the solution. In every step we are closer to the goal and we make a positive advance.

2.2.3 Construction of loops-basic programming construct

Programs are not always simple or straightforward. Some time it is needed, some thing is to be repeated. This repetition is known as loop or in other words the repeated execution of a block of statements is known as loops.

Every loop must satisfy some conditions, those conditions are known as loop-condition. If those conditions are not satisfied then either the loop will not be executed or the loop will be an infinite loop.

A loop has 3 parts; they are loop initialization terminal condition and lastly the re-initialization. The repeating statements are placed in-between these three parts.

We will see an example of loop. In the example of visiting the Tajmahal, if the flight is delayed then what can be done? In that situation we can repeat some of our activities like go to snacks bar and bring a cup of coffee, comeback wait for the flight if flight has not arrived and coffee is finished repeat the activity again. Here in the above statement "if flight has not arrived" is the check for looping. The second part "and coffee is finished" is also determines the looping but it is auxiliary. The initial condition in this example is "delay in flight arrival". The terminating condition is the "arrival of flight".

2.2.4 Implementation

Modular design is the key to successful program implementation. The modular design means breaking up into smaller parts. Each part is now a module, which can be developed independently. And at the end they can be combined to make a finished product. We have already discussed this in problem solving strategy.

In a program there will be several variables. Hence the variable names should be given in such a way that one could understand the purpose of that particular variable. If a variable is needed to store the roll number of student it could be named "studroll or nroll" instead of "cat or dog".

It is a practical experience, that if a program is written and tested ok, and after few days the source code is examined again; it becomes difficult to understand what the code is doing. Whenever a program is written it should be sufficiently commented and documented, else it is very difficult to maintain or upgrade the program latter.

The last stage is the program testing. This is a quality testing part too. The testing of finished program reveals several things as the consistency or in-consistent parts.

Preliminary test at development site reveals only logical and programming errors. The client site tests reveal the shortcomings and inadequacies in the program.

2.3 Programming environment

In above paragraphs we have discussed about programs and how to design them. Considering that we have already designed a program, so how to put it it in to the computer? Well, there are programming languages and their translators to computer readable formats available. We will discuss briefly here.

2.3.1 Programming Language Classification

Programming languages are broadly classified in to 2 types.

- Low level
- High level

Low-level languages are machine language and assembly language. High-level languages are BASIC, FORTRAN, PASCAL, C and there are several to mention. Machine language is just numbers, one has to code in binary numbers if he/she needs to program in machine language. For all other language there are translators available.

2.3.1.1 Assemblers

Translator for assembly language is known as assembler. This translates the assembly language into machine language (binary).

2.3.1.2 Compilers

Compilers are those, which translate high-level language into assembly or machine language. They have some diagnostic features to help rectify errors. Compilers generate an executable or machine understandable code and store it in a file.

2.3.1.3 Interpreters

Interpreters are like compilers but they do process one line of source code at a time if any error exists then it will display that error and wait for user interaction. The interpreters don't generate an executable file. They translate the source code into machine-readable instructions in memory and execute them from memory.

2.4 Summing up

This chapter gives a brief idea on what programs are, how they are designed and how they are executed. We will discuss it in later chapter too.

Chapter 3

C Programs

3.1 A small program

Every programming language has some provision for 3 things. Those are Input, output, processing. These are provided as library functions. In C, also these are present. The basic input statement in C is scanf (), for output printf () and for processing we use several constructs or library function. Let us now write a very small C program and analyze it.

```
#include<stdio.h>
main()
{
   printf("Hey U...\n");
}
```

Enter this above program in a text editor, save it as hey.c, and then depending upon your compiler installation compile it, link it, and then execute it. Let me describe the process for TURBO-C and gcc under Linux.

3.2 Using your compiler

3.2.1 For turbo C

- At dos prompt type tc.
- Open File menu select New.
- Enter the above program.
- Again open File menu select Save
- In the dialog box shown enter hey.c
- Now choose Compile menu select compile.
- If there is any error then it will appear in message window.

Remember the sequence. It is valid for all programs

- Correct those.
- If no error then choose Run menu select Run press enter.
- Then press ALT+F5 this will display you the executed result

3.2.2 For gcc (Linux/djgpp)

- At $ prompt type vi
- Press I to enter insert mode
- Enter the program
- Press ESC key
- Press: press w then type hey.c
- Press: press q
- Now you will be in $ prompt
- Type gcc hey.c -o hey it will compile and link the file.
- Now at $ prompt type. /hey
- This will display the result.

Enough said regarding compiling this little program. If you need more about your compiler then you should refer to the documentation available with your compiler. Now let us analyze the program.

3.3 Dissection of hey.c

In the first line we have written #include<stdio.h>, this is a preprocessor directive. For preprocessor directives consult your compiler manual. Preprocessor directives are not executable statements; rather they direct the compiler to do something. Here in this case this instructs compiler to include something called stdio.h. The stdio.h is known as header file, stands for standard input output header. As the description suggests this is responsible for all functions doing standard input and output.

The second line contains main (). This is the main entry point to the program. Every C program has to have this main () if they are designed to be executable. If they are designed for library then main () is not needed. Here main () is a function, as C only has functions.

The third line contains a curly bracket '{'. This is the start of a statement block. C is a block structured language so a block of executable statements are enclosed with '{' and '}'.

The fourth line is printf("Hey U...\n"); this represents a call to a library function printf(), with parameter "Hey U...\n". You can guess the call is an out put statement to print some thing to 'stdout', yes it is.

The fifth line is to close the statement block. As this is our first program the block contains only one statement but it can contain as many lines as you can put.

Well you might be thinking to print a simple message why you will do so many things? Instead of taking pain you can do this in BASIC in one line, right? Yes you are right here too. But in BASIC you have to take pain to write an operating system, a compiler or even a dirty and nasty virus. Well as the designer of C say, "C is designed to be the best tool for system software".

We will now discuss a skeleton of c program. You might be thinking after you wrote a C program why to bother about a skeleton? Yes, It is necessary, because we have left out so many things of a C program, That is many parts of C program we have over looked.

3.4 Skeleton in the cupboard...

```
/*This is a skeleton of a C program.
   This is a comment block. You need to write
   Comments in your C programs as much as you can
   Comments are necessary for understanding the
   program */
   /*After comments you should include header files*/
   #include<stdio.h>
   #include<myheader.h> /*I have a header known as myheader.h*/
   /*Here define the macros and constants*/
   #define PI 3.14285 /*Constant PI */
   #define eat(x) {(x)*(x)} /* a macro function to do my things*/
   /*Define global and external variables here */
   int ram, hari; /* global integer type variable */
   float demi_moore; /* global float type variable */
   char abdul_seikh; /* global char type variable */
   /* here you have to define function prototypes or the functions,
   which you will be using in your program */
   void print_msg (char *msg); /* this is a prototype of a function which
   is defined later in the program */
   int time2(int n) /* this is a function defination*/
   {
      return (n*2);
   }
   /* now it is time for the main() */
   main()
   {
      /* define local variables for main */
      int k,l;
      float area;
      float peri;
      /* do processing things here */
```

```
    printf("Hello this is a skeleton\n");
    printf("Enter the radius of a circle : ");
    scanf("%d",&k);
    area=PI*eat(k); /* PI we defined, macro eat calculate square*/
    printf("area of circle with radius %d is %f\n",area);
    l=time2(k); /* time2 multiplies with 2 */
    peri=PI*l; /* PI multiplied by 2 times radius */
    print_msg("Perimeter of circle : ");
    printf("%d for radius %d\n", peri,k);
    /*our processing ends here ...suppose... */
} / main() ends here */
/* we have to define the function bodies for which we have prototyped
in the beginning */
void print_msg(char *msg)
{
    printf("%s",msg);
}
/* this ends the skeleton */
```

The skeleton given above is just for reference and I don't say this is complete. This is generic for many small programs. But you can see certain variation of this one and also there may be certain addition too. So do not mistake this as the only template available. With this skeleton you should be able to distinguish different parts of C program, also can write your small C programs too.

As we have seen the skeleton and a small program to see how things work, we have encountered certain new things, which seems to be unknown to us. We will discuss them in the next chapter.

3.5 Summing Up

In this chapter we saw only 3 things.

1. A small program to print a message.
2. How to use TURBO-C and gcc compiler.
3. A skeleton of C program to show different parts.

Chapter 4

Data Types

In the last chapter we have seen a skeleton a small program with some analysis too. But that analysis does not clarify much. We have seen in the skeleton something as 'int k, l' and 'float area'. So what do they mean? Why they are written like that? Well, we will see them here and some part in a latter chapter.

4.1 Data

Before you understand that int float etc. you must know something else. So now you ask yourself what are those, which follows int or float? I mean what are k, l and area? Which are defined in skeleton. Well they are variables. If they are variables what they do? Good... as the name suggests they can have values. You can change their values too. But in contrast PI, which is defined in the beginning, cannot be changed. So now we encounter another thing, which cannot be changed, it is known as constant. Well the PI, k,l and area are the names. In those spaces you can put your name, your pet's name, your friends name etc. These names can store some values. Those values are DATA. Now any sort of data can be stored in any variable? NO, NO, NEVER... Like you cannot keep your car or bike in your bedroom and sleep in the garage. In the same way there are some distinctions or types. These types are known as 'Data types'.

4.1.1 Classification of data

Commonly there are 2 types or class of data. They are Pre-defined and User defined. We will discuss user-defined types somewhere else in latter chapter but pre defined type now.

Pre defined data type has five classes namely 'character char', 'integer int', 'real numbers float', 'double precision real double' and 'void'. As the name suggests 'char' type can store an ASCII character. ANSI standards recommend a single byte (8-bit) for its representation. Type 'int' can store any integer value, it is proposed to have two bytes (16-bits) but it should not be assumed that 'int' type is always 2-bytes (16-bits), depending upon compiler installation it may vary. Real number type values are

stored in 'float' type variables, the proposed size of 'float' is 4-bytes (32-bits), but it can change. Double floating point 'double' can store values of real numbers of size 8-bytes, it may also change depending upon the installation. Type 'void' is special type used in generic casting of pointer or used for functions, which do not return anything.

4.1.2 Type Modifiers

These basic types again have some subclasses to modify their range. Namely:

- *signed*
- *unsigned*
- *short*
- *long*

The modifier 'signed' and 'unsigned' applies to 'char' ,'int' and 'double' type, 'short' and 'long' to 'int' type, 'unsigned' and 'long' are applicable to 'double'.

Depending on the installation type the range also varies. That is for a 16-bit platform the size is different from a 32-bit platform. To know the range of your compiler consult your compiler manuals. Bellow is given a table of ranges for my installation of compiler.

Type	Length	Range
Unsigned char		0 to 255
char	8 bits	-127 to 128
enum	16 bits	-32,768 to 32,767
Unsigned int	16 bits	0 to 65,535
short int	16 bits	-32,768 to 32,767
int	16 bits	-32,768 to 32,767
unsigned long	32 bits	0 to 4,294,967,295
long	32 bits	-2,147,483,648 to 2,147,483,647
float	32 bits	3.4 * (10**-38) to 3.4 * (10**+38)
double	64 bits	1.7 * (10**-308) to 1.7 * (10**+308)
long double	80 bits	3.4 * (10**-4932) to 1.1 * (10**+4932)

The Datatype Table.

4.2 Identifiers

The lexical meaning of identifier is who identifies. The variable names, function names, labels and other user-defined things are identifiers. The rule to define an identifier is, it should start from lower case alphabet or underscore and subsequent characters may be underscore, alphabets or digits. If any other things are used then it is incorrect. Secondly keywords cannot be used as identifiers. Lastly as C is case sensitive case plays a greater role. Hence XYZ, Xyz, xYz, xyZ, xyz are different.

4.3 Variables

We now know the data-types; we have used them in the skeleton. So now we will see how to use those data-types. In previous section we have discussed what is the meaning of variables that is their value can change. Now we will see how to define them and different types of variable

<Data type> variable_name;
<Data type> variable_name1, variable_name2,...;

Above two syntaxes are used to define variables. Data type can be any data type, variable names are those, which we will be using as the replacements for data values. The convention for variable name follows the convention of identifiers.

Examples of proper variable naming:
int a,ab,muler;
char Ac, cA;
float A,_a,_A,_muler;
int muler1,muler_1,muler_,_muler_;

Wrong way to define variable names:
int 1,1a;
int a@, b#, c+a;
float if, auto;

4.3.1 Variable Types

Depending on data types variables can be of 5 types, which we have seen earlier. But depending upon the position it can be of 3 different types.
These are:

- Local
- Global
- Formal parameters

Following is a discussion of these types.

4.3.1.1 Local Variables

Local variables are those, which are defined in a function or block. Their effect is limited within the block only. They are also termed as 'automatic' variables. They come into picture only when the block of code where they are declared is executed. (A block of code is enclosed by '{','}').After execution of the block they are removed from memory.

void do_it()

```
{
   int x;
   ....
   ....
   ....
}
```

Here 'x', which is integer type, is local to do_it() function. It cannot be realized outside the block.

4.3.1.2 Global Variables

Global variables are those, which are available throughout the program's execution. They are not defined inside any block. They exist in memory until the program is in memory.

```
#include <stdio.h>
int k;
void do_less()
{
   printf("\n k= %d\n",k);
   k=19;
}
main()
{
   k=91;
   do less();
   printf("\n now k= %d\n",k);
}
```

In the above example k is not defined in any block but in the beginning. So it is available throughout the program.

4.3.1.3 Formal Parameters

Formal parameters are those, which are passed to functions. They are only available in that function body. They behave like any other local variable declared in that function. You can change their values they get from calling function too. They will be discussed in a later chapter.

4.4 Access Modifier

There are two access modifiers. They are *'const'* and *'volatile'*. These must precede the type specifier. Access modifier or qualifier changes the way a variable behaves.

4.4.1 const

The 'const' modifier says the compiler that it may not be modified by the program. But it can be modified by system functions like timer interrupt. A const variable can get its value from an explicit assignment too. For this type compiler is free to place the variable in read only memory (ROM).

const int my_var=10;

In the above expression my_var is a variable of type integer and protected from being changed by the program and has been initialized to 10.

4.4.2 volatile

This modifier changes the way the variable is changed. That is it can not be changed by the program like 'const', and can be changed by some external occurring like interrupt or can be modified by the lower level function which interact with hardware or operating system. It is also possible to add 'const' before 'volatile'.

const volatile int tick;

4.5 Storage Class Specifier

These are to modify the properties of the variables defined. There are four such specifiers in C language. They are:

- *extern*
- *static*
- *register*
- *auto*

Storage class specifiers are used in the following manner:
<storage class specifier> <data type> var_name;

4.5.1 extern

The 'extern' specifies, the variable used here is defined in some other file, which is a part of the project (program). Take an example of a program, which has more than one file of C code. We need some variables in both the files. So we can compile those files separately and link to a single executable. If we define the same variable in two files as globals then this will solve our purpose but at the link time there will be an error, which says the symbol, is defined more than once. Ultimately the solution is to use the variable but do not define it. Now how to achieve it is the question. The way out is use

extern in other file and defines it in one file. The general use of 'extern' is:

extern char x; / defined in subsequent files where x is used*/*

There is another way 'extern is used. It can be used to refer a global variable in a function but it is not needed.

4.5.2 static

static int k;

'Static' variables are permanent like global variables. Except they are not known to outside of their block or file. Still their values are preserved between successive calls.

'Static' variables are of two types:

- *'local static'*
- *'global static'*

4.5.2.1 Local static variables

Local static variables are those, which are defined inside a block or function. They are necessary in certain conditions where it is required to keep track of previous value of a variable.

4.5.2.2 Global static variables

Global static variables are which are defined out side any block like global variables. The difference is the global variable is visible to outside the file but static global variable is not. It also retains the value when the module is used subsequently.

4.5.3 register

This keyword instructs the compiler that the variable is transient hence directly put it in to the CPU register for manipulation. But modern compiler assumes it as '*process as first as possible*'. Traditionally the register specifier is used with 'char' or 'int' type but practically it can be used with any type of variable.

register int sum;

Here sum is a variable of type integer and storage class is register.

4.6 Summing up

In this chapter we have discussed:

Data types
Variables

Storage class

We also discussed types of variables depending uponThe position of declaration,Modifiers and Storage classes.

In the next chapter we will see the operators, which can be applied to these variable types.

Chapter 5

Operators

All programming languages have operators. These operators operate on constants or variables. If there would be no operators then there is no need for programming. Operators are broadly classified in two categories.

They are:

- Arithmetic
- Relational and Logical

As there names suggests the arithmetic operators are meant for arithmetical calculations and logical operations are meant for logical calculations.

5.1 Arithmetic operators

Arithmetic operators are those, which you use, in simple arithmetical calculations. They are addition '+', subtraction '-', multiplication '*' and division '/'. Arithmetic operators can be used with almost any type of predefined data types. Besides these operators C has modulus '%', increment '++' and decrement '–' operator in arithmetic operator category.

You all well versed with all but last 3 operators. Still There is a point to discuss them in brief. Addition operator adds 2 numeric values or numbers. But it also can add 2 characters too. Output of adding 2 characters is not the concatenation of the operands but a number. If that number is within 255 then it can be stored and represented as a character else as an integer. Same thing happens to all other arithmetic operators too.

As an example:

```
char x, y, z;
x='A';      /* 'A' means character constant A =65 in decimal*/
y='z';      /* 'z' is 122 decimal ascii */
```

```
z=x+y;    /* z= 65+122 that is 187 */
```

Now for the last 3 operators. The modulus operator finds the remainder in a division. It can be only applied to integers and character type. Let us see an example.

```
int i,j,k;
i=10;
j=3;
k=i%j;
```

Here value of k will be the remainder of the division 10 and 3 that is 1. This operator is used to check the divisibility.

The increment and decrement operator increases or decreases the value of a variable. depending on the occurrence they can be pre or post increment or decrement operators. That means if '++' or '–' happens to precede the variable then it is pre, if they happen to follow the variable name then they are post. There is an interesting thing to remember the post increment is done after the execution of the statement.

Example:

```
int x, y, z;
x=10; /*assigned 10 to x*/
y=x++; /* assigned x++ to y ?? */
z=x; /* assigned x to z ?? */
```

Guess the values of y and z. did you guessed y=11 and z=11? If so then you are WRONG. y=10 and z=11 is correct. Let us analyze the example. First we declared and assigned 10 to x. Then in the third line we wrote y=x++ that means assign x++ to y. Yes here it is post incremented. So evaluation of x++ is done later. So y is assigned 10. Then x is incremented to 11 and in the last line we assign z with the value of x that is 11 now, so z is 11. The above holds good for decrement too.

Assume in the above example if instead of post increment it is pre increment then what would be y and z? the would be 11 both. Because pre increment is calculated first then the statement is executed.

5.2 Relational and Logical Operators

This category includes 2 things relational operators and logical operators. Relational operators are greater '>', greater or equal '>=', equality '==', lesser '<', less or equal '<=' and not equal '! ='. These operators are used to compare two operands. Hence these are binary operators. They compare operands on both the sides of them and return a truth-value. A truth-value is either TRUE=1 or FALSE=0.

s1	s2	>	>=	<	<=	==	!=
a=10	b=5	T	T	F	F	F	T

a=5	b=5	F	T	F	T	T	F
a=3	b=6	F	F	T	T	F	T

T=TRUE F=FALSE

Logical operators are those, which evaluates the logical Truth-values. They are logical and '&&', logical or '||' logical not '!'. Let us see some examples.

a=10 b=200
hence a>5 is true, a>b is false
(a>5)&&(a>b) false
(a>5)||(a>b) true
(a>5)&&(!(a>b)) true

s1	s2	&&	||
T	T	T	T
T	F	F	T
F	T	F	T
F	F	F	F

s1 and s2 are 2 statements
T=TRUE F=FALSE

The logical not operator changes the truth-value of the statement.

s	!s
T	F
F	T

The relational and logical operators are used in conditional statements and also in loops which we will be dealing soon.

5.3 Miscellaneous other operators

Besides these arithmetic and logical operators there are other operators too. They are conditional evaluation **'?'**, coma **','**, address of **'&'**, value at **'*'** and element access dot **'.'**, **'->'**. There is also another category of operators known as bit wise operator.

We will discuss the address of '**&**' and value at '*****' operators in chapter dealing with pointers. The dot '**.**' and '**->**' operators in user defined data types. But the rest of the operators here.

The '**?**' operator, evaluates a condition then executes statements depending on the truth-value of the first statement. Generally a colon accompanies it '**:**'. The colon separates 2 other statements, which are dependant on the truth-value of the first statement. Syntax for '**?**' operator is:

expr1?expr2:expr3

If we expand the above statement in english it will be " If expr1 is true then do expr2 else do expr3". Here expr2 and expr3 are expressions, which can be evaluated. That is if expr1 is true then value of expr2 is the value of the expression and if expr1 is false then value of expression is value of expr3.

```
y=(x>9) ? 10 : 20;
```

This expands to if x>9 is true then y=10 if x>9 is false then y=20.

The coma ',' operator joins several operations together. The left hand side is always evaluated as void. The right side expression is always value of expression.

```
x=(x=3,y=3,x+y); /* x will be 6 here*/
```

5.4 The () and [] Operators

These two operators are generally overlooked as they are used whenever needed. The () operator is used to increase the precedence of an expression to be evaluated. Like in this expression a+b*c-d/e, d/e will be evaluated first, then b*c will be evaluated then a+(b*c) and finally a+(b*c)-(d/e) will be calculated. But if we place brackets like in the following expression (a+b)*(c-d)/e then a+b will be evaluated first. Hence () increased the precedence of the addition.

The [] operator is used in arrays only. So we will not discuss it here. It will be discussed in detail when we will discuss arrays.

5.5 Bit wise operators

These are a set of operators, which are intended for low-level operations. By low-level operation here we mean is setting, testing or shifting of bits in a byte. These operators are only used in character type or integer type. It cannot be used for any other type.

Bit wise operators include AND '&', OR '|', NOT(one's complement) '~', Exclusive OR (XOR) '^', left shift '<<' and right shift '>>'. AND operator here finds the ANDed value of its operand.

```
char ch, ch1;
ch1=ch & 0x0f; /* mask high nibble */
```

It chopped off high nibble of ch and returns the lower nibble value in ch1. The OR operator, similarly finds the ORed value. In the above example if we replace '&' with '|' this will make lower nibble to be always 0xf. Let us check XOR '^' operator. For this we have to take actual values for clarity.

```
ch=0x69;      /* 0x69=   01101001 */
ch1=0x50;     /* 0x50=   01010000 */
              /* XOR '^' ———————— */
ch2=ch^ch1;   /* 0x39=   00111001 */
```

So in the end ch2 will have the value 0x39. The One's complement operator is used to complement the bits present in the operand byte.

```
ch=0x59; /* 0x59 = 01011001 */
ch2=~ch; /* ~(0x59)= 10100110 = 0xA6 */
```

Bellow is a table to illustrate these operations.

a	b	a&b	alb	a^b
1	1	1	1	0
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

Truth table for AND OR XOR

a	~a
1	0
0	1

Truth table for One's complement NOT

Shift operators shift the bits towards left or right depending whether it is left or right shift. The number of shift is on the right hand side of the shift operator. The syntax of shift operator is:

<Variable or constant> shift operator <number of bits to shift>
variable is character or integer type shift operator is left shift '<<' or right shift '>>' number of bits to shift is an integer less than 32.

```
0x05<<1 = 0x0A          00000101<<1=000001010
————————————————————————^This      ^This is added
————————————————————————is cutoff
0x05<<2 = 0x14          00000101<<2=00010100
0x04>>1 = 0x02          00000100>>1=00000010
0x0C>>2 = 0x03          00001100>>2=00000011
```

If you have watched carefully the examples above then you must have noticed shifting left by one bit multiplies the number by two. For 2 bits shift the result is

4 times the number. Similarly right shifting one bit result in half the number being shifted and if shift left by 2 bits the result is one fourth of the original number.

5.6 Type casting

Type casting is done to force a type to change into other type. The casting comes handy when a particular type is needed as result. The syntax is:

(type to force to) expression
type to force is any valid type.
Example:
char a;
—
—
(int) a = 65;

5.7 Precedence table

Highest
() [ ] -> .
! ~ ++ – - type-cast value at address of size of
* / %
+ -
<< >>
< <= > >=
== ! =
& bit wise
^
|
&&
||
?
= += -= *= /=
Lowest

5.8 Summing Up

In this chapter we read about operators. And how they are used. We also discussed the low level operators, which are known as bit wise operators. The address of '&' and value at '*' operators are to be discussed in pointer chapter and the dot '.' and arrow '->' operators will be discussed in user defined data types.

Chapter 6

Branching

Branching means changing the flow of the programs. By changing the flow of program it means skipping some executable statements. This skipping is of 2-types, unconditional and conditional.

6.1 goto

The unconditional branching construct skips the executable statements what ever the case may be. This is achieved by abruptly jumping to another statement leaving aside the next statements. This can be done by go to statement without any condition.

statement_1;
statement_2;
statement_3;
go to label_1;
statement_4;
statement_5;
label_1: statement_6;
statement_7;
...
...

In the above case the program flow jumps to statement_6 without executeting statement_4 and statement_5. This is not necessary in normal programming and it is advised **NOT TO USE goto** at all.

6.2 If...else

Now let us see the conditional branching. Before we do that we should check what is a condition. A condition is a truth-value of an expression. Of course the expression is a

logical one. To use this conditional construct. We use **'if...else'** statements.

```
if(condition)
{
    statements
}
else
{
    statements
}
```

In the above syntax the else part is optional. But it cannot be avoided always. Let us see some examples.

```
int a, b;
printf("Enter 2 integers\n");
scanf ("%d%d", &a, &b);
if(a<10)
{
    b=20;
}
printf("a=%d b=%d\n");
```

The above program fragment prints the value of **b** depending on the value of **a**. For any value of **a** less than 10, **b**=20 is printed. Yes the value we input to **b** is lost. But for any value of **a** above 10 the value of **b** we have supplied is printed.

Now let us check for the *else* part. In above case *else* is hidden. By that it is meant, *if **a** is less than 10 then **b** is 20 else **b** is whatever value we have given*. Let us check another example.

```
int a;
printf("Enter an integer ");
scanf ("%d", &a);
if (a%2)
{
    printf("%d is an even number\n");
}
else
{
    printf("%d is an odd number\n");
}
```

Here *else* is compulsory. Because if we omit *else* and **a** happens to be an even number then both the printf statements will be executed. So we get an ambiguous answer.

6.3 Nested if...else statement

The *if else* statements can be nested, that is the *if* statements can be chained. To achieve this we uses *if...else if...* construct.

```
if(condition)
{
    statement block
}
else if(condition2)
{
    statement block2
}
```

6.4 switch...case...

This is another type of branching statement. One can say it is analogous to 'nested if' statement or a group of if statements.

```
switch (condition)
{
    case value_1: action to be performed
            break;
    case value_2: action to be performed
            break;
...
...
    default: default action
}
```

We can write the above using if as following:

```
If (condition==value_1)
{
    action to be performed fro value_1
}
else if(condition==value_2)
{
    action to be performed for value_2
}
...
```

But mark the presence of break, it is a key word in c. This is used in Switch statement to exit out of the block when a matching case is found. Suppose by mistake

we omit the break in one case what will happen?? Well the execution of actions will continue until a break is encountered. Remember to use break judiciously.

6.5 Summing Up

In this chapter we read:

1. Conditional and unconditional branching
2. Go to statement and label;
3. if...else, nested if...else statements
4. switch (..) case statement.

In the next chapter we will see how loops are made.

Chapter 7

Loops

Till now we have seen conditional branching, now we will see how to execute a group of statements or a single statement repeatedly. Let us now recall the 'go to label' statement and use it to form a loop.

```
main ()
{
    int i=0;
    label_1:
    printf("Hello world\n");
    i++;
    if (i<5)
    go to label_1;
}
```

In the above example the printf () statement will be executed 5 times. This implants a simple loop. Let us analyze how this loop works. If you mark the code then we are declaring a variable 'i' and initializing it to be 0. Then we put a label to jump to. Then an 'if' statement, which checks the value of the variable 'i' with 5, then we put a go to statement. Here if the value is 5 or more the go to statement will not be executed. Otherwise it will loop through the label. In the above code 'i' is known as loop counter, the if statement is known as loop condition, i++ is loop reinitialization. CAUTION: Never use back jump using go to. Besides this crude looping C language has keywords to implement looping. They are: *for, while,* and *do...while*. We will discuss them one by one.

7.1 for loop

The for loop is the most commonly used in C. the syntax of for loop is for(counter initialization; condition; reinitialization of counter)Unlike the above code all the things needed by a loop is present in the line where the loop is started. If everything is present then where is loop body? Ok in case of for loop the loop body follows the loop declaration. And it is enclosed in a block.

Let us rewrite the above loop:

```
main ()
{
    int i;
    for(i=0;i<5;i++)
    {
        printf("Hello world\n");
    }
}
```

The output of both the program fragments are same. So the general structure of a for loop is:

```
for(initialization; condition; reinitialization)
{
    statement block;
}
```

Note here the number of times loop will execute is dependant on the initialization, condition and re initialization.

7.2 while loop

In contrast, while loop is less common and more or less similar to the go to implementation of loop. In case of while loop, the loop counter is initialized before the loop body and the condition is checked in the beginning, then the loop body starts. The reinitialization of loop counter is done inside the loop body. The structure of while loop is:

```
initialization of condition;
while(condition)
{
    statements to be executed;
    reinitialization of condition;
}
```

Let us re write the go to example using while loop.

```
main()
{
    int i=0;
    while(i<5)
    {
       printf("Hello world\n");
    }
}
```

This gives the same output as that of above go to and for examples.

7.3 do...while loop

The last looping structure is do...while loop. It is basically the while loop but starts with 'do' and ends with 'while'. Other things are same as that of while loop. the syntax for 'do...while' loop is:

```
do{
    loop body;
    reinitialization of loop counter;
}while(condition);
```

Let us rewrite above loop example using 'do...while' loop

```
main()
{
    int i=0;
    do{
        printf("Hello world\n");
        i++;
    } while(i<5);
}
```

If we carefully judge the code the above loop will execute at least once. This is because we check the condition at the loop end.

7.4 Nested loops

Loops can be nested. This means a loop can contain another loop. Nesting facility is available in all the three loops. We will see them now. In a nested loop the inner loop executes faster the outer loop will continue when inner loop is executed.

7.4.1 Nested for loop

For example complete this code and execute:

```
main()
{
    int i,j;
    for(i=0;i<3;i++)
    {
        for(j=0;j<4;j++)
        {
            printf("inner counter=%d outer counter=%d\n", i,j);
        }
    }
```

```
}
```

See the output of this code and verify what is happening. If you are careful enough then you could have found out when 'i' remains same 'j' varies from 0 to 3. So this implies the inner loop executes faster.

7.4.2 Nested while loop

A nested while loop will be like this:

```
main()
{
    int i=0,j=0;
    while(i<3)
    {
        while(j<4)
        {
            printf("inner counter=%d outer counter=%d\n", i,j);
            J++;
        }
        i++;
    }
}
```

7.4.3 Nested do...while loop

A nested do loop will be:

```
main()
{
    int i=0,j=0;
    do
    {
       do
       {
            printf ("inner counter=%d outer counter=%d\n", i,j);
            j++;
        }while(j<4);
        i++;
    }while(i<3);
}
```

7.5 Infinite loop

An infinite loop is a loop, which never ends. This is an error in loop. So you must be very careful to write loops. If a loop became infinite then it hangs the system. As an

example let us check the following code

```
for(;;)
{
    printf("Infinite loop\n");
}

while(1)
{
    printf("Infinite loop\n");
}

do {
    printf("infinite loop\n");
}while(1);
```

All the above case are infinite loops. An infinite loop can be result from irresponsibly placed condition or no condition at all. But note all condition less loops are not infinite. Like if you hesitate to put a condition in its default place, then you can place the condition inside loop body. Also you have to put a 'break' after the condition to exit from loop.

7.6 Break and continue

We have seen the 'break' key word previously in 'switch...case' construct. We will discuss it here. A 'break' statement takes control to the end of the block in which it is used. It means if a 'break' statement is encountered then all the executable statements following it are skipped until the end of that block. Let us check it now:

```
main()
{
    while(1)
    {
        printf("this is not an infinite loop\n");
        break;
        printf("This will not print\n");
    }
printf("Now out of loop\n");
}
```

In contrast 'continue' takes control to the beginning of the loop

```
main()
{
    int i,j;
```

```
    for(i=0;i<10;i++)
    {
        j=i%2;
        if(j!=0)
        continue;
        printf("%d\n",i);
    }
}
```

In above case the loop will be executed 10 times but it will print only 5 times. That is it will print only the numbers, which are even. Because if 'j' which stores the remainder, if not 0 then the continue statement will be executed. Hence the control will go to the loop beginning.

7.7 Summing Up

In this chapter we discussed about loops. We also discussed 'break' and 'continue' statements. So the points are:

- Loops are used to execute a statement or a group of statements repeatedly.
- A mall formed loop may hang the system.
- A do loop executes at least once.
- Loops can be nested. That is a loop can contain several other loops.
- 'break' statement exits a loop.
- 'continue' statement shifts control to the beginning of loop.

Chapter 8

Compound Data-Types

Till now we have dealt with simple data-types. Like char, int, float, etc.. now we will discuss something about compound data-types. Compound data-types include arrays, structures and unions. We will discuss them one by one.

8.1 Arrays

Array means a group of similar type of things. In C programming it means a collection of similar type of data in consecutive memory locations. as an example array of integers, array of floats etc... The syntax to define or declare an array is:

<data-type> <array name>[size];
Where :
data-type is any valid data-type.
array name is identifier to denote array.
size is number of elements we want to put in array.

Note here the presence of '[' and ']', if you recall what we have read in operators you can say these are nothing but *'[ ]' operator*. The *'[ ]' operator is used heavily in arrays*. Let us now define an array.

int my_array [20];

This defines an identifier my_array array of 20 integers. This means my_array is an array, which can contain 20 integer type values.

8.1.1 Accessing array element

Now we can define an array. So next we will put values in to it. To put a value in an array cell we have to use the '[ ]' operator again. As an example:

```
int int_array[5];
int_array[0]=10;
int_array[1]=30;
...
int_array[5]=100;
```

Notice here the array cell starts from 0 and go up to the 1 less size specified. So the maximum cell number in int_array above is 4 (0,1,2,3,4). We can assign values to an array like we have seen above. Else we can assign values at the time of declaration.

```
float flt_arr[4]={10.0,20.0,25.3,11.9};
```

Both these ways are easier for smaller array but these methods are not fit for larger arrays. This is because; if there are 100 elements in an array, then think of the situation. So there must be some way to deal with. See above when we were writing int_array[1]=30, this means we are putting 30 into int_array at first cell. Here 1 is known as array index. So using any loop we can fill the array.

```
int i;          /*we are defining one integer type variable we
                will use it as loop counter as well as array index */
int int_array[5]; /*an array of integer with 5 cells*/
for(i=0;i<5;i++) /* loop to fill the array*/
{
    int_array[i]=i*10; /*assign the array elements */
}
```

From above code it should be clear that using loops to fill the array is easiest and convenient.

We have entered the value to array cells, now we should be manipulating them else there is no meaning of array type. Well the array cell can be used as normal variables, the only difference is we have to specify array index and must use '[ ]' operator. Similarly to get the values from cells of an array we have to use loop with array as we have used for entering values.

```
for(i=0;i<5;i++)
{
    printf("%d\n", int_array[i]);
}
```

The above code fragment will print the values stored in cells of int_array.

8.1.2 Multi dimensional arrays

Up to this point what we have discussed is single dimensional arrays. But arrays can have more than one dimension. Consider a matrix or a determinant. They are 2-dimensional arrays. Consider a vector in 3-D plane. It is an array of 3-dimensions.

Think of Einstein he took another dimension time. So to represent vector in his calculations a 4-D array is needed. More than 4-D arrays are rarely used. They are used in calculating atomic energy power plants and research institute.The general definition of a multi dimensional array is:
<data-type>array name[size1][size2][size3]...[size n];

As an example *int abc[3][4][2]* defines *abc* to be a *3-dimensional array* of *integer*. To access cells of this array we need 3 loops one for each dimension. Let us see how we can access the cells.

```
int i, j, k, temp;
int pqr[5][6][9];
/*entering data into array*/
for(i=0;i<5;i++)
{
    for(j=0;j<6;j++)
    {
        for(k=0;k<9;k++)
        {
            printf("Enter a number ");
            scanf("%d", &temp);
            pqr[i][j][k]=temp;
        }
    }
}
/*getting data out of array*/
for(i=0;i<5;i++)
{
    for(j=0;j<6;j++)
    {
        for(k=0;k<9;k++)
        {
            printf("value at cell %d %d %d = %d\n",pqr[i][j][k]);
        }
    }
}
```

he above code fragment is capable of reading a number from user store it in a multidimensional array, and display it.

Here is a point to note. Watch the above code carefully, you will see pqr is an array of array of arrays. Just watch array is repeated 3-times. Similarly a 2-D array is a single dimensional array of single dimensional arrays. Hence the following declaration is valid.

```
int xyz[3][2]={{1,2},{3,4},{5.6}};
```

8.1.3 Array of characters

We have discussed, an array can have any data type as its element and it is homogeneous. This implies an array of character is possible and it is a normal case. So what is the importance to discuss the array of character?? Well there is a special case. If a character array has null character (\0) in the end then it is known as a string. A string can be said as an array of character and the last character is a null character. Example of string is as common as your name. We will discuss strings in some more detail in some future chapter.

8.2 Summing up

In this chapter we have read regarding arrays.

- Arrays are homogeneous collection of data.
- Elements of an array are stored consecutively in memory.
- Array can be single dimensional or multi dimensional.
- A multidimensional array can be seen as a single dimensional array of arrays.
- A string is a special case of character array.

In the next chapter we will discuss regarding *user defined data types*.

Chapter 9

User defined data-types

We have read about the simple data types. Those are also known as pre-defined data types or simple data types. Those are of course fundamental data-types but are inadequate in some respect. Like if we need to group a set of data for database application then they cannot be used. Even if they are used it will be difficult to handle, hence we need a separate way to handle a group of data. The first attempt to do that is using arrays. As we have seen array groups data, but array can only group similar data. But in case of database application we need to group several unrelated fields. Take an example of your college. Your college maintains a record for all the students reading. A single record may contain Name, Address, Batch, roll number, Subjects, Marks obtained in exams, etc... So if we try to maintain the record for every student then it will be very difficult for us to do using the simple data types. To ease our work C has a way out. It gives user to define his/her won data types, these are known as *User-deined Data Types* (UDT).

Under user defined data type we have two key word struct (structure) and union. We will discuss them one by one here.

9.1 Structure

It is, as the name suggests, a compound data type. It groups several different data types, which are related to one another. Take the example of student record. It has strings integers floats etc... It is the structure, which binds these different types together as a single data type.The syntax for structure declaration is:

struct <structure name>{
type <data>;
type <data>;
...
}[<optional name>];

Let us see an example a very simple one, which has only 2 fields, name a string type and percentage of mark.

```
struct st_ach{
        char name[40]; /* Student name*/
        float pc; /* Percentage of marks obtained*/
};
```

Here we have defined a structure *st_ach* which contains a *student name* and *percent of marks*. As we have stated earlier it is a data-type we can define a variable of this type too. It is done as bellow.

```
struct st_ach ram, pearson, abdullah , harrison;
```

In the above statement we are defining 4 variables of struct st_ach type. Well we have defined those variables now so we should give values to those variables. Here we can not simply assign the variables with their corresponding values. Because there are two fields and we should assign 2 values to these variables. The assignment of values in case of structure is done as bellow.

```
strcpy(ram.name,"Ramesh");
ram.pc=76.54;
strcpy(pearson.name,"david");
pearson.pc=74.95;
strcpy(abdullah.name,"Ramesh");
abdullah.pc=76.54;
strcpy(harrison.name,"david");
harrison.pc=74.95;
```

Recall in the operator chapter we have read about dot "." operator. The dot operator is used here in structures to access the fields. The other operator used with structure is arrow "->" . We will discuss the arrow operator in pointers chapter.

This is a small example, we will see how to store data in to a structure type variable and how to retrieve data from a structure type variable. Consider a record which stores a studentâs name, address and grade point. We will put data for some students and see the result by printing them.

```
#include<stdio.h>
main()
{
    struct student
        {
            char name[20]; /*studentâs name will be within 20 chars*/
            char address[20];/*address of student*/
            float grade;
        };
```

```
    struct student a, b, c;
    /*assigning values to structures*/
    strcpy(a.name,"hari");
    strcpy(a.address,"cuttack");
    a.grade=7.5;
    strcpy(b.name,"peter");
    strcpy(b.address,"newyork");
    b.grade=6.5;
    strcpy(c.name,"ramirez");
    strcpy(a.address,"atlanta");
    a.grade=6.45;
    /*reading values from structure*/
    printf("%s %s %f\n",a.name,a.address,a.grade);
    printf("%s %s %f\n",b.name,b.address,b.grade);
    printf("%s %s %f\n",c.name,c.address,c.grade);
}
```

Now it must be clear how the structure works. And you should have noticed the work of dot "." operator. Remember the dot operator is used to access the fields of a structure. Some times it is also called element access operator.

9.1.1 Array of structure

We have already seen array. Array contains a uniform data for all its cell or elements. We cannot put dissimilar objects in to array. Like an array can contain all integers or all floats, that is it should be homogeneous. We have seen array of different data type. Now an obvious question come can there be array of structures?? Yes definitely. Let us see that now. And also we will see their advantage too. Consider the following program fragment, which describes the array of structure and how it can be implemented.

```
struct person
{
    char name[20];
    char sex[7];
    int age;
};
/*now we can define array of structures*/
struct person p[100];    /*p is an array of 100 structures of type person*/
/*now to input data we should start a loop*/
for(i=0;i<100;i++)
{
    ...
    ...
    strcpy(p[i].name,person_name);
    strcpy(p[i].sex, person_sex);
    p[i].age=person_age;
```

```
}
/*now to display we have to start another loop*/
for(j=0;j<100;j++)
{
    printf("%s %s %d\n",p[i].name,p[i].sex,p[i].age);
}
```

From above code you could have guessed the advantage of array of structures. Is not it?? Well see in earlier example we have instantiated 3 variables of same structure. Here if for 100 students we have declared only one array so our code got simpler and smaller too.

9.1.2 Structure within a structure

We have seen a structure, array of structures and arrays as elements of a structure. So why cannot a structure be an element of another structure? Yes a structure can also be an element of another structure. Even a union that we will be discussing next can be an element of a structure. Consider a case like maintaining information of an employee in an organization. Every employee has a name, designation, department, employee serial number, and address. An address contains street, post office, province, country and pin code. Here we can separate out this address information in a separate structure. And make it an element of the main employee structure. Let us see how it can be done.

```
struct address
{
    char street[40];
    char post_office[40];
    char province[40];
    char pin[10];
};
struct employee
{
    char name[40];
    char desig[30];
    char dept[30];
    int emp_ser_no;
    struct address emp_address;
};
```

In the above example we define a structure address first as we will be using it in our main employee structure. Then we wrote the employee structure. Well how to access the elements in this case? It is simple with the same dot operator ".". Now we will instanciate an instance of this structure and see.

```
struct employee emp[100]; /*we have defined an array of 100 employes*/
/*here we will consider only one employee*/
```

```
strcpy(emp[i].name,"Philippe kahn");
strcpy(emp[i].desig,"Chief executive officer");
strcpy(emp[i].dept,"adminstration");
emp[i].emp_ser_no=1;
strcpy(emp[i].emp_address.street,"lakestreet");
strcpy(emp[i].emp_address.post,"santaclara");
strcpy(emp[i].emp_address.state,"California");
strcpy(emp[i].emp_address.pin,"632559");
```

In the above example see how we access the element of address structure. Watch the two dot operators. The first dot operator for employee structure and the second dot operator for address structure. So there is basically no difference. We can use as many dot or arrow operators as we can for nested structures.

9.2 Union

Unions are very similar to structures. But they differ in properties. Structures hold all the fields they have but unions can hold only one field at any time. Just think of a structure which has an integer and a character. So we can put 300 in integer field and 'Z' in character field. But in case of unions we can put either 300 or 'Z' at any time not both. The syntax of declareing union is as follows:

```
union <union name>
{
    <type> field1;
    <type> field2;
    ...
    ...
};
```

Example:

```
union my_union
{
    int I;
    char ch;
};
```

The values are stored in the union in the same way as that of structures, by using dot operator or by arrow operator. If we consider the above example of union then it will be as follows:

```
union my_union test;
test.i=300; /*if we store an integer*/
```

Or

*test.i='Z'; /*if we store a character*/*

Like structures unions can contain other unions or structures as their elements. And we also can make an array of unions.

9.3 Enumerated data type

This type associates numbers to identifiers. The enumerated identifiers are always integers. Let us see how we can define enumerated variable.

enum bool {false, true};

Here enum is the keyword, bool is variable and false and true are the possible values of bool. Here the false is assigned 0 as default and true the next value 1. If we donât specify a value to the first then automatically it is assigned a value of 0 and next values are given in 1 difference. Let us check some other examples.

enum colour {black, white, red, green, blue};

In the above case values will be black=0, white=1, red=2, green=3 and blue=4. Suppose we modify that and write black=1 instead of only black then the other values in this enum will be in 1 difference so they will be 2, 3, 4 and 5 respectively. Suppose we wrote black=-3 then the values for white and others will be â2, â1, 0 and 1. Suppose we are writing the whole thing as :

enum colour{ black =-3, white, red=1,green, blue=4};

Then the different values are -3 ,-2,1,2 and 4 respectively.Enum are used in programs where more redability is required.

9.4 Typedef

The type def comes handy in many situations. It can be used to shorten the identifier name or to name the identifier in a meaning full way. Suppose we have a structure named address. When ever we use this structure we have to write struct address which is lengthy hence we can write :

typedef struct address addrs;

From this point onwards we can refere the same structure with only addrs so the struct address is no more needed. Sometimes it is needed to make identifiers some what more meaningful, hence the typedef comes in handy in those situation. Typedef never creates a new data type, it just assign a new name to a existing datatype.

9.5 Summing up

In this chapter we read about user defined data types, namely structures, unions and enumerated datatype. Lastly we saw the typedef which creates a new name for user convenience. We certainly left out some more interesting things on user defined data types. We will see them in following chapters.

Chapter 10

Functions

Functions are major building blocks in C programs. The definition of function can be, it is a part of program, which works separately but cannot execute separately. The functions are separate entities, they can exist separately they have there won worlds but they cannot execute separately. This means they are helpers in a program and some thing is needed to execute them.

Functions are not alien to you. You have used then in your earlier programs. It sounds interesting is not it? Well the printf () and scanf (), which you have used, are also functions. They are known as library functions. Functions are two types library functions and user written functions. We will discuss functions in general and how to write user defined functions in particular.

Any C program has at least one function. The default function is main () we have seen that several times in the previous chapters. If you recall we have some other functions in skeletal program. Use of functions make the program modular, which in turn make it easier to modify or in case of error it is easy to spot the culprit. So it is always advisable to write different working part as different functions.

Every function has a name which identifies it, a set of parameters which is passed to it, which are written within parentheses and a function body which is enclosed within a pair of curly brackets. The function body contains variable declarations, which are known as local variables and executable statements. The scopes of the local variables are limited within that function only.

Occasionally functions return a value. This is known as return value of a function. In this case additionally a return data type precedes the function name and a return statement should be present in side the function body. If a function does not return any thing then it is better to write void in place of return type. Now let us see the syntax of function definition.

```
int add(int a, int b)
{
    int c;
    c=a+b;
    return ( c );
```

```
}
```

In the above code we are defining a function named add which takes two integer type variables a and b, which are known as formal parameters, we have discussed them in data types. Inside the function body we are defining a local variable c, we then put an executable statement c=a+b then return statement, which returns c an integer.

From above discussion we can discus function in 4 ways that is based upon the fact that input parameter and return parameter. Function doesnât take and doesnât return, takes but doesnât return, returns but doesnât take and returns as well as takes. We earlier discussed if a function doesnât return any value then we should put void as return type. Now if a function doesnât take any value then we can write void as input parameter or formal parameter or leave it blank. Writing void is a good practice. The forms of the 4 different types are:

```
void xyz (void) or xyz ()
void xyz (<type> var,...) or xyz(<type> var,...)
<type> xyz (void) or <type> xyz ()
```

10.1 Function call

Functions cannot execute themselves with exception to main (). In a C program only main is executable. All other functions are not. If this is true then there must be some way by which the functions executes. This is known as function call. A function can be called from another function with exception to main() function. This means main() functions can call other functions but other functions can not call main(). If by curiosity you happen to call main() function and it compiled well then you may end up in infinite loop.

A function is called from within another function by just putting its name inside that function with proper parameters and if it returns any thing then you must supply a proper variable to catch the value. Before you can call the function it should be defined or at least prototyped. See the example bellow.

```
#include<stdio.h>
/*this function takes 2 integer and return their sum*/
int add(int a, int b)
{
    int c;
    c=a+b;
    return ( c);
}
main()
{
    int k;
    printf("Main function calling add()\n");
    k=add(3,4);
```

```
    printf("Return value from called function is %d\n",k);
}
```

In the above code we first define a function add () , then we call it from function main() with 2 arguments 3 and 4. We catch the returned value in k which is an integer type, next we print the value stored in k. From above code it is clear that the calling function should supply the arguments to called function and store the returned value if any from the called function. Just like main () function in above program any other function can call function add. To test that just patch the above code as follows.

```
#include<stdio.h>
int add(int a, int b)
{
    int c;
    c=a+b;
}
/*this is just the prototype of the function calculate*/
float calculate(int d, int e, int f);
main()
{
    int k,l,m;
    float p;
    printf("Enter 3 integers k l m");
    scanf("%d%d%d",&k&l&m);
    printf("Now calling calculate ()\n");
    p=calculate(k, l, m);
    printf("The return value from calculate () is %f\n",p);
}
/*the function definition is to be done here*/
float calculate(int d, int e, int f)
{
    int g;
    float h;
    /*we now call add() function*/
    g=add(d, e);
    h=1.0*g/e; /*just promoting type of g by multiplying 1.0*/
    return (h);
}
```

10.1.1 Parameter passing technique

Parameters are the values upon which we want the function should operate. These are passed to functions within parenthesis. The mechanism by which parameters are passed to function is known as parameter passiing. There are basically 3-types of techniques.

- Call by value
- Call by referance
- Call by name

What ever we saw in the previous section above is known as *call by value*. Modern compiler no longer supports *call by name*. Earlier it was available in some language. We will discuss the *call by reference in the pointers chapter*.

10.1.2 Call by value

Formal parameter
Their Scope

Function-call mechanism

Recall the variables. We have discussed formal parameter there. The variables declared inside the parentheses are known as formal parameters. They are local to the function itself they donot exist outside the function. The formal parameters act like interface between the calling function and the called function. Calling function passes the actual variables and the called function accepts them in formal parameters. The actual parameters are mapped one to one on to the formal parameters. This means the formal parameters are assigned the values of actual parameters when the execution of the function takes place. Due to this reason this method is known as call by value.

In call by value method if the formal parameters are changed then the values in calling function will not change. This is because formal parameters are local to the function. and actual parameters are local to the calling function. But there are certain cases we need to change the value of the variable from inside a function. This is usually done in *pass by reference* methode or else it can be done only incase of global variable. Only global variables can be changed from within function. Here an interesting thing comes to mind. If we have defined a global variable and a local variable with same name then if we change the value then whose value will be changed, let us see it. Consider the following example.

```
#include<stdio.h>
int v1,v2;_/*we define two global variables*/
void test(void);
void main()
{
    v1=10;_/*we assign 10 and 20 to these variables*/
    v2=20;
    printf("value of v1=%d and v2=%d\n",v1,v2);
    test();
    printf("value of v1=%d and v2=%d after the function call\n");
}
void test()
{
    int v2;
    printf("assigning 100 to v1 and 200 to v2 within function\n");
    v1=100;
    v2=200;
```

```
}
```

Try to compile and run this program the output screen will look something like bellow. See for yourself to get convinced.

value of v1=10 and v2=20
assigning 100 to v1 and 200 to v2 within function
value of v1=100 and v2=20 after the function call

The above example shows, if we have two variables one global and one local in the same name and if the value is changed then the value of local variable will be changed not the global variable.

10.2 Structures and functions

10.2.1 Passing elements of a structure

We can pass individual elements to structure or the whole structure as parameter to a function we will see how it is done. Elements of a structure can be passed to a function as normal parameters. That is the pass by value method parameter pasing is done. If compound data type member is passed then pass by reference is done. As an example if a string is an element of a structure then that element's address is passed. See the example bellow.

```
struct test
{
    int a,b;
    float c,d;
};
prn_int(int k)
{
    printf("%d\n", k);
}
main()
{
    struct test t;
    t.a=10;
    t.b=20;
    t.c=1.2;
    t.d=2.9;
    prn_int(t.a);
}
```

This example will take the element as a normal integer value and print it.

10.2.2 Passing the entire structure

The entire structure can be passed to a function too. This is also can be done by pass by value. Care must be taken the passed parameter and formal parameters are of same type. This means *if two structures are identical then also we can not interchange them in function calls*. See the example bellow to get a beter understanding.

```
/*the structures bellow are identical*/
struct test{
    int a;
    char c;
};
struct test2{
    int a;
    char c;
};
/*Let us write a function to take entire structure*/
my_func(struct test t)
{
    printf("integer=%d character=%c\n",t.a,t.c);
}
main()
{
    struct test t1;
    struct test2 t2;
    t1.a=10;
    t1.c='A';
    t2.a=20;
    t2.c='X';
    my_func(t1); /*this is alright*/
    my_func(t2); /*this is wrong*/
}
```

Identical structures are not same

10.3 Recursion

Recursion is the phenomenon of function calling itself repeatedly. Consider a function calls itself from within itself. You may think it is a looping function. As in case of loop care must be taken to stop the it, here too care must be taken to stop the recursion. This is achieved by using a check inside the function to stop recurring infinitely. Whenever there is a recurring function the compiler generate such code to trace back the stack. This is useful for the system to wind up when the recursion stops. Recursion is useful in the situation, where some series of similar and dependant calculations are needed. The best example is factorial. *Factorial of n(n!)* can be written as *n*(n-1)*(n-2)*(n-3)*.....*2*1*. So it is a continued product from n to 1. Hence we can rewrite it

as *n*factorial of(n-1)*. From this we can write a function to calculate the factorial of a number.

```
int fact(int n)
{
    if(n<=1) /*if n is 1 or less return as 1!=1 and 0!=1*/
    return(1);
    return(n*fact(n-1)); /*else call fact with n-1 */
}
```

Recursive functions have some disadvantages. They do not optimize memory usage nor they generate significantly small code. Even in some cases they execute slower than there non recursive counterpart. When ever there is a function call the program sets up a stack to store the local variables. Hence for recursive functions some time the stack gets over run. The advantage of using recursive function call is to write clear and easier to understand routine. In some cases recursion is unavoidable.

10.4 Summing Up

In this chapter we saw how to write functions and how to call them. We also discussed pass by value method in details. We also saw how fields of structure and also the entire structure can be passed. If a field or entire structure is passed to the function then what ever modification we did to the structure inside the function will not be reflected if this structure is not a global structure.

The function call mechanism we saw here is pass by value or call by value. The other way to pass parameter is pass by reference or call by reference will be discussed in pointer chapter. There also we will see passing variable numbers of parameters and another interesting thing passing parameter to main function which is also known as command line parameter passing.

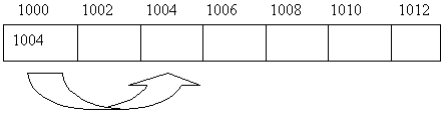
Chapter 11

Pointer

It is said if you don't know pointers then you don't know C. It is right. Pointers give us unlimited power in C programming. We can access any system resource any par of memory and practically any thing. We should be very careful in handling pointers else pointers are very dangerous.

11.1 What is pointer?

Pointer is a variable, which contains memory address. This address is the location of another variable in memory. Consider 2 variables. If the first variable contains the address of second variable then it is said the first variable points to second variable.



In the above picture the variable at 1000 is a pointer to the variable at 1004.

11.1.1 Pointer variables

Here comes the operator '*'(asterisk). This operator is used to declare a pointer type variable. The declaration of such a variable consists of type (any valid type), '*' and variable name. The syntax for this purpose is as follows.

*<type> * <variable name>;*

The type defines here which type of data the pointer is going to point. This means which type of variableâs address this pointer will point to.

11.2 The operators

Recall the chapter on operators. We have seen 2 operators and left them for pointer chapter. One is address of operator unary &) and the other is value at operator (unary *). Do not confuse these operators with "bit-wise and" and "multiplication" operator.

The "&" operator gets the address of any variable and the '*' operator counters the '&' operator. That is gets the value at that address taking into consideration the base type. Let us see an example.

```
Int a, b;
Int *ptr1, *ptr2;
/*Here ptr1 and ptr2 are two pointer type variables,
which can contain the address of 2 integer variables*/
ptr1=&a;
ptr2=&b;
```

In the above example **ptr1** contains the **address of a** and **ptr2** contains the **address of b**. By address it is meant the memory location of the variable at the run time.

Suppose at run time **a** is at **location 1004** and it has a value of say 50, then **ptr1** will contain **1004 not 50**.

Let us now check the * operator. This operator gets the value. So to get the value stored at any location we can write

```
m=*ptr1;
```

Here **type of m and base type of ptr1 must be same**. Now **m** contains the **value** stored **at address** pointed by **ptr1**, that is value of **a**.

11.3 Pointer expressions

Expressions in pointer are similar to any other data-type. You can add subtract compare pointers. By add or subtract it is meant you can add a constant. 2pointers can not be added or subtracted. Similarly multiplication or division of any sort is not possible. Of course one can multiply with the value pointed by a pointer.

11.3.1 Assignment

A pointer can be assigned to another pointer type variable of same base type. Consider the following code.

```
int a;          /*a is a normal variable*/
int *ptr1, *ptr2;   /* ptr1 and ptr2 are 2 pointer type variable*/
ptr1=&a;        /* ptr1 is assigned the address of a */
ptr2=ptr1;      /* ptr2 also stores the address of a */
```

```
printf("%p %p\n",ptr1,ptr2); /* will print address of a 2 times */
```

The above code shows assign ing of the address of a variable in to a pointer type variable. Then we store the address stored in one pointer type variable in another pointer type variable. Then in last statement we are printing the address stored in those pointers. You can notice we are using %p here to print the address.

11.3.2 Addition and subtraction

Addition and subtraction in pointers deal with constants, as 2 pointers cannot be added or subtracted. The obvious question in your mind is why, is not it?? Well think in this way, pointer is an address of a variable. So adding two addresses has no meaning and it wont help in any circumstances. Like if you add your address with your friends address then what will happen? It is now clear I think.

You can add a constant to a pointer the constant is some times known as offset. You can increment or decrement a pointer too. We will see these in detail in pointer and array relationship shortly. But for the time being remember the above restriction.

```
int *p;
int *q;
int *r;
r=p+q;      /* this is wrong */
r=p-q;      /*this is wrong too*/
r=p+1;      /* right */
q+=1;      /*right */
q++; q–;    /* right */
++q; –q;    /*is right too*/
```

An interesting thing to mark here suppose in above example code âpâ points to 1000, then âp++â points to 1002 not 1001. This is because address if added with a constant gives the address of another variable of same type as that of base type of pointer. Here base type of pointer is âintâ, each integer takes 2 bytes adding one means the address of next integer which is 2 bytes away.

11.3.3 Comparison

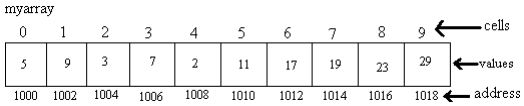
Two pointers can be compared using **if** statement. If **p** and **q** are two pointer type variables then they can be compared like any other data-type.

```
if(p<q)
printf("p is in lower memory than q\n");
```

11.4 Relation between array and pointer

Pointers are cousins to arrays. They have many things in common and can be some times used interchangeably. The first point to remember here is array name is a pointer.

Hence we can apply pointer arithmetic to array. See the picture bellow.



The addresses are assumed in the above picture. When you create an array the program reserves memory for it. The amount of memory reserved is dependent on the type of each element of array and size of array. In the above case we assumed elements are integer type and the size of array is 10. Thus the corresponding program declaration is like bellow.

```
int myarray[10];
```

Here our program reserved 20 bytes in consecutive memory location starting at 1000. From the above picture you can guess the address of 0th element is 1000, address of 1st element is 1002 and so on. Now let us access the elements. For this we will use â[ ]â operator. If we print myarray[0] it will print 5, for myarray[1] it will output 9 and so on. Here instead of writing the index directly if we use a variable letâs say âiâ then we can write myarray[i]. Consider the following code.

```
main()
{
    int i=0;
    int myarray[10];
...
...
    while(i<10)
    {
        printf("%d th element = %d \n", i, myarray[i]);
        i++;
    }
...
...
}
```

This will print the array elements. Now we will change some thing in the while loop. Rewrite the while loop as follows. We now know the array name is the address of the first element of array. Hence we will use the value at operator *.

```
...
int *a; /* a is a pointer to integer */
a=myarray; /* assign myarray to a */
while(i<10)
{
    printf("%d th element = %d \n", i, *a); /* print i and value */
    i++;
```

```
    a++;
}
```

See the result, they are same, is not it? From the above example you could see the pointer assignment, value at address and pointer arithmetic. In the above example instead of writing **a++** you can write **a+=1** too. More of array and pointer relation we will se latter in this chapter. We just see the relation between pointer and single dimensional array. Can there be a pointer to multi dimensional array? Yes, certainly. But incase of multi dimensional array the declaration is a bit different but similar. In case of one-dimensional array we put a single * in front of the pointer type variable. In this case we have to put number of stars depending upon the number of dimension. For example, in case of 2-dimensional array **, in three-dimensional array *** and so on. The number of stars should be equal to number of dimensions.

```
int b[2][3];
int c[2][4][2];
int **k;
int ***l;
k=b; /* as k and b are identical*/
l=c; /*as l and c are identical*/
```

11.5 Pointer to pointer

If there can be a pointer to any data-type then why can not be for a pointer? Yes it is possible. In case of multi dimensional array we are defining them. As we know a two-dimensional array is a single dimensional array of single dimensional arrays. Hence a pointer to pointer defines a 2 dimensional array. Some times this is called **multiple indirections**.

11.6 Array of pointers

If array of any data type is possible then array of pointers is also possible. In this case each element of the array is a pointer to any data type. Suppose we want to create an array of integer pointer then we should write in following manner.

```
int k,l,m,n;
int *a[4]; /*here a is an array of pointer to integer*/
/*hence we can write*/
a[0]=&k;
a[1]=&l;
a[2]=&m;
a[3]=&n;
```

11.7 Multidimensional array

We read earlier how to represent multi dimensional arrays using pointers. Now we will look into how to access the elements of multi dimensional array. Follow the example bellow.

```
Int a[3][3][3]; /* a is a multi dimensional array*/
Int p,x,y,z,;
Int ***k; /* k is a pointer to pointer to pointer to integer */
/* hence k can hold the address of a 3-dimensional array*/
k=a;
/*now we will store values into the array*/
for(x=0;x<3;x++)
{
   for(y=0;y<3;y++)
   {
      for(z=0;z<3;z++)
      {
         printf("Enter element %d %d %d\n", x, y,z );
         scanf("%d",&p);          /*read the element*/
         *(*(*(k+x)+y)+z)=p;      /*store the element in array*/
      }
   }
}
/*now to print the values */
for(x=0;x<3;x++)
{
   for(y=0;y<3;y++)
   {
      for(z=0;z<3;z++)
      {
         p=*(*(*(k+x)+y)+z);
         printf("The element %d %d %d=\n", x, y,z , p);
      }
   }
}
```

11.8 Pointer to structure

In the structure chapter we learnt about the operator associated with structure. There are 2 operators one is dot and the other is arrow. We discussed about dot operator. Now in pointer chapter we will see how the arrow operator works. The arrow operator is associated with pointer to structure. That means if we have a pointer, pointing to a structure type variable, then by using arrow operator the elements can be accessed.

Now let us see an example. Consider a structure having fields name, age, and sex.

```
struct my_struct
{
    char name[40];
    int age;
    char sex[7];
};
main()
{
    struct my_struct *frnd;      /*define a pointer to structures*/
    char temp[40];
    int k;
    printf("Enter name=>");
    gets(temp);
    strcpy(frnd->name, temp);
    printf("Enter sex=>");
    gets(temp);
    strcpy(frnd->sex, temp);
    printf(âEnter age=>);
    scanf(â%dâ, &k);
    frnd->age=k;
/*we put values to structure now we will print the values*/
    printf("%s %d %s\n",frnd->name,frnd->age,frnd->sex);
}
```

See here the program will look like any other program on structure but have arrow operator instead of dot operator. Remember the arrow replaces dot incase of pointer to structure. We also can make an array of pointer to structure. And use them in a similar fashion as above. Incase of array of structure we are using dot operator to access the fields of each cell. Here we have to use arrow operator.

11.9 Function and pointer

We have seen in function chapter how a function takes any data-type and returns different values. Here we will see how a function takes and how it can return pointers.

11.9.1 Pass by reference (passing pointer as argument)

We have discussed pass by value method in functions. In that case the value is passed to the function. But in case of pass by reference we pass the address of variables to the called function. As the address of variable is passed, any modification of value at that address will be reflected in the calling function. Hence this should be considered seriously.

The pass by reference is advantageous when we pass array or we need the variableâs value should be changed by the function. Consider the case of a swapping function we need to alter the values and that should be available in calling function. We know a function can return only one value hence pass by value is not possible. So it is done using pass by reference. Consider the following swapping example.

```
#include<stdio.h>
void swap (int *p, int *q) /* we take two pointers */
{
    int * temp;          /* temporary pointer variable */
    *temp= *p;           /*store the value stored in p*/
    *p= *q;             /* store value stored in p into q */
    *q= *temp;           /* transfer the stored value in temp to q*/
}
main()
{
    int a=10, b=30;
    printf("Value of a=%d and of b=%d before swap()\n", a, b);
    swap (&a, &b);      /* passing addresses*/
    printf("Value of a=%d and of b=%d after swap()\n", a, b);
}
```

11.9.2 Function returning pointer

Function can return pointer. Like any other type of data a pointer can be returned from a pointer, by putting the address of the returned variable in return statement. It should be remembered, when a function finishes its execution, its variables are wiped out. The following example shows how to return a pointer.

```
#include<stdio.h>
Int *calc (int k)
{
    int p;
    p=k*2;
    return(&p); /*we are returning address of p an integer type variable*/
}
main()
{
    int k,*l;
    k=5;
    l=calc(k); /*
    printf("twice %d is %d\n"; k, *l);
}
```

A string or an array can also be returned but the methods are somewhat more complex. See bellow for an example.

```
char *stringadd (char * st1, char *st2)
{
    char * retval=st1;/* save the original address of first string */
    while(*st1!='\0') /*find the end of the string */
        st1++;
    while(*st2!='\0')
        st1++=st2++; /*get character from 2nd and put in 1st*/
    *st1 = *st2; /* put the null character at the end*/
return(retval);
}
```

Take it as a challenge and try to replicate all string functions and make a string library.

The above function in the code for string manipulation function **strcat()** function. Similarly other string functions can be written.

11.9.3 Pointer to function

Well we now know a pointer is an address. As the function also stays in memory at the time of execution, it has an address too. If we know the address of a function what can we do?? Because a function is not a variable we cannot put value to it so how it is useful to know address of a function?? Yes it is very important. The function pointers are comes handy if we need to pass function to another function as a parameter or make it an element of a structure or union. Application of this is numerous. Even using this concept we can write programs in Object Oriented manner. Let us see a small example to check how we can write and use pointer to function.

```
#include<stdio.h>
void my_func(void) /*declare a function */
{
    printf("Inside my_func()\n");
}
/*declare a function to take another function as parameter*/
int second_fun(void (*p)(),int k)
{
    printf("passed integer is %d\n",k);
    printf("Executing function passed\n");
    p();            /*execute the passed function indirectly*/
    return(k);
}
main()
{
    /*define a structure with a function pointer as an element*/
    struct
    {
        int i;
        void (*f)(); /*element is a function pointer*/
    }pqr;
```

A function can be passed to another function as a parameter

A function can be a member of a structure or union

```
    void (*test)();     /*define test as a function pointer which points
            points to a function that takes nothing and returns nothing*/
    test=&my_func;   /*assign address of my_func() to test*/
    pqr.i=5;
    pqr.f=test;      /*store the address of my_func() in the structure*/
    test();          /*execute the function indirectly through pointer */
    pqr.f();         /*execute again as structure member */
    second_fun(test,5);  /*pass to a function as parameter */
}
```

11.9.4 Variable numbers of argument passing to function

In the previous chapter we have seen passing parameter by value and in this chapter you have seen passing parameter by reference. In both the ways the number of parameter passed to the function is known beforehand. So we just mention the types inside the parenthesis. But if the number of arguments to be passed to a function is not known then we cannot write a function. Well no, we can write. The way is known as variable number of arguments passing to function.

Variable argument passing is done by including **stdargs.h**. This stands for standard arguments passing. We will see the mechanism latter but a program to demonstrate the fact follows.

```
/*
    EXAMPLE of variable number of arguments passing to a function
*/
#include<stdio.h>
#include<stdarg.h>
/*
    The function bellow accepts variable number of integer arguments
    and returns sum of them
*/
int summer(int count,...)
{
    int result=0; /* to the sum */
    va_list args; /* args is the list */
    int arg; /* to store indivisual integer*/
    va_start(args,count); /* create the list */
    while(count!=0)
    {
        arg=va_arg(args,int); /* separate out the integers */
        result+=arg; /* find sum */
        count–;
    }
    va_end(args);
    return (result); /* return the result */
```

```
}
main()
{
    int res;
    printf("passing 3 integers 1 2 3 to summer()\n");
    res=summer(3,1,2,3);
    printf("sum=%d \n",res);
    printf("passing 5 integers 3 4 5 6 7to summer()\n");
    res=summer(5,3,4,5,6,7);
    printf("sum=%d \n",res);
}
```

Examine the above code. The function summer will sum the arguments passed. The function declaration contains **count** integer type, which denotes the number of arguments to be passed. Then **'...'** the ellipsis. The ellipsis denotes the variable number of arguments follows.

In the function we have used âva_listâ which is an array defined in the header file **stdarg.h**. The **va_list** type array contains the argument list passed to function. Then we encounter **va_start** this is a macro, which is defined in the header file. This is responsible for initializing the list of arguments. This means this should be called to initialize the **va_list** type variable. This macro takes 2 arguments; they are **va_list** type array and the last fixed variable. In our example **args** is **va_list** type and the last fixed variable is **count**. The next new thing we encounter is **va_arg**. This macro also takes 2 arguments. The **va_list** type array and the type of arguments present in the array. In our example we pass integers. After the list is processed we stop the process or clear the list by calling **va_stop**. This macro takes only one argument of type **va_list** and clears it.

The sequence of use of the macros stated above is fixed. This implies the **va_start** should be used first, and then to separate out the arguments **va_arg** and to finish up **va_stop** should be used. If the sequence is changed then the result is unpredictable.

11.10 Command line Parameter passing

We have seen functions take parameters. So can main() take parameters?.

Yes it can. This is known as command line parameter passing. To understand command line parameter passing we should know how a program is executed. We already have discussed the execution of a program starts at main(). So main () is the starting function and it calls other user written or library functions. But how main is executed? We have already stated until another process or function does not call a function it cannot be executed. So there must be some process or function that calls the main function. Yes this is the shell or command interpreter which calls the main function. So the command interpreter should pass the parameter to the main(). Now let us see one example,but prior to that we should see what information the Operating System or Command interpreter passes. Generally the operating system passes the parameters written on the command line through the PSP (Program Segment Prefix). This is an

array of strings. There is also another type of information available to the program; that is known as environment parameter, an array of string too. See the following example.

```
/*
    EXAMPLE of command line parameter passing
*/
#include<stdio.h>
main(int argc, char *argv[],char *envp[])
{
    int i=0,n=0;
    printf("number of command line parameters = %d\n",argc);
    while(argc>n)
    {
        printf("%s\n",argv[n]);
        n++;
    }
    while(*envp!=NULL)
    {
        printf("%s\n",*envp);
        envp++;
    }
}
```

Compile the program and execute from command prompt as follows. Suppose the name of the program is **cmdln.c** then the executable name will be cmdln.exe. now at command prompt type **cmdln 1 2 3 4** and press enter. It will display you some information as follows.

```
number of command line parameters = 5
F:\ZIPS\C-BOOK\CMDLN.EXE
1
2
3
4
TMP=C:\WIN98\TEMP
TEMP=C:\WIN98\TEMP
PROMPT=$p$g
winbootdir=C:\WIN98
COMSPEC=C:\WIN98\COMMAND.COM
PATH=C:\WIN98;C:\WIN98\COMMAND;G:\BC3\BIN;
windir=C:\WIN98
BLASTER=A220 I5 D1 T4 P330
CMDLINE=cmdln 1 2 3 4
```

In the above output the first line is from the programâs first printf() statement. Next line is the executed programs name with path this is argv[0]. Then the program outputs 1, 2, 3, 4 that we have supplied in the command line after that all are the environmental

variables the operating system and shell need for their processing.

11.11 Summing up

It was a long chapter is not it ?? well we read all aspects of pointer(no no not all). We left out certain thing too. We left some exercise in string manipulation and data structure. The data structure part will be dealt in next chapter. But the strings are left as exercise to you the reader.

Here you should remember every thing we write in a program and which generates a code has an address so we can have a pointer to that. **But one type of variable has no address they are register variables**.

Chapter 12

Dynamic Data structures

Dynamic data structure?? Sounds odd?? Yep!! It sounds odd. Well well cool down. This chapter is really really big. Many books are available on this subject. It is really a vast topic. We cannot deal with all aspects of data structure. What we will deal is fundamental of this subject. In this section and stipulated space we will try to grasp some knowledge of dynamic allocation of memory. Some algorithm related to data structure namely singly linked list, stack and queue. This is not all in data structure. This subject also includes doubly linked list, tree (binary tree), rings, graph and several theorems dealing with them.

12.1 Dynamic memory allocation

The repetition of dynamic must be irritating you. When ever we declare a variable some memory is reserved for it. This is static allocation. This allocated memory cannot be de-allocated by the program. This means this allocation is valid until the scope of the variable exists. In contrast dynamic allocation means we can assign memory for a variable and we can de-allocate that memory when we feel to do so. This allocation and de-allocation is done by library functions in C-language. We will see it in the following example.

```
#include<stdio.h>
#include<alloc.h>
void main()
{
    int i,temp;
    int *a; /*declareing a pointer to integer*/
    /*now we will create an array dynamically*/
    /*we will create an array of 10 integer*/
    a=(int *)malloc(10*sizeof(int));
    /*now we have created an array of 10 integer*/
    printf("prompt 10 times to input an integer\n");
```

```
    for(i=0;i<10;i++)
    {
        printf("enter an integer ");
        scanf("%d",&temp);
        a[i]=temp;
    }
    /* now we will print them*/
    printf("You entered: ");
    for(i=0;i<10;i++)
    {
        printf("%d ",a[i]);
    }
    printf("\nAre those same??\n");
}
```

In the above program we have seen a new function malloc (). This function is responsible for allocating memory. The syntax for malloc () function is

*Void * malloc (size_t n).*

In the above syntax malloc function allocates n bytes of memory from heap and return a void pointer. A void pointer is a generic pointer, which can be casted to any type. See the above example, we have written a=(int *) malloc (10 * sizeof (int)); here we cast the pointer returned by malloc to integer pointer then store that in a. So ultimately we are reserving 20 bytes (10 *2 bytes) of memory. Now consider if there is not sufficient memory to allocate then our malloc function will fail. If it fails how can we know?? Yes if it fails then it returns a null value. Well we have not checked whether our malloc function is successful or not. We assumed it to be successful. A real programmer would have checked it in th following manner.

```
a=(int *) malloc(10*sizeof(int));
if(a==NULL)
{
    printf ("Not enough memory \n");
    exit (-1); /*exit with code -1*/
}
```

We also have not freed the memory at the end of our program. It is not advisable to leave allocated memory even after it is used. So a **free ()** function should be called. A free function frees a block of memory, and takes the pointer which points to the memory block as argument. So we should write f**ree (a);** before we close the program. Another thing to remember we should free the allocated memory immediately after the use and if we don't need it any more. Again if we have allocated more than one block of memory then they should be freed in the reversed order of their allocation. There are several other memory allocation routines but malloc is used often and we need malloc in this chapter. For those of you are interested in knowing other memory allocation

routines, they should refer to the compiler documentation. We now move to some real thing of data structure.

12.2 Linked list

Linked list is another method of storing values like array. In case of array the number of element must be known before hand, but here it is dynamic. You can add as much element you can until your systems memory is exhausted.

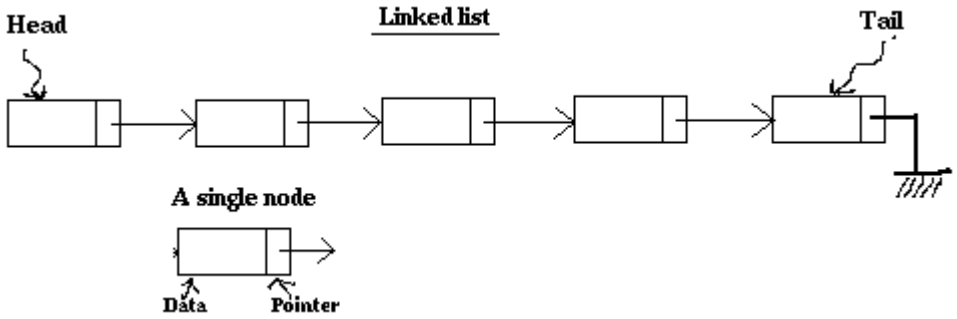
12.2.1 Creation

This linked list is based upon two things one every link is a self-referring structure and the other is we allocate the link at runtime using malloc. If you don't understand these two then re-read and try some example.

Every link in a linked list is known as a node. Every node has a data part and at least one pointer part to point to nearby node. Now we will see the algorithm to create a linked list.

Algorithm for singly linked list

- **Declare a pointer to node assume it as list.**
- **Allocate memory for a single node and store the address in list.**
- **Put the data in data part.**
- **Put NULL in pointer part.**
- **For subsequent nodes:**
- **Allocate memory for a node.**
- **Put the data in to data part.**
- **In pointer part put address stored in list.**
- **Assign current node address to list.**



single linked list.

The above algorithm makes a linked list. Here observe the recent node is at beginning of list. This is actually the algorithm of stack, which we will see later. We could have done in other way that is first element is at beginning. Now let us see code examples for both the method.

```
/*example of linklist first element is at end of linked list*/
#include<stdio.h>
#include<alloc.h>
/*define a self-reeferring structure for use as node*/
struct my_struct
{
    int num;
    struct my_struct *next;
};
typedef struct my_struct node;
node *list;             /*declare a global variable list*/
/*initialize the list or creates a list*/
void init(int val)
{
    list=(node *)malloc(sizeof(node));/*Allocate a block of memory*/
    list->num=val; /*put the data*/
    list->next=NULL; /*put null in pointer*/
}
/*add to list*/
void add(int val)
{
    node *temp;
    temp=(node *)malloc(sizeof(node)); /*allocate a block*/
    temp->num=val; /*put the data*/
    temp->next=list; /*put address in list into pointer part*/
    list=temp; /*store the new address in list*/
}
/*prints the list*/
void show()
{
    node *temp;
    temp=list; /*assign the address in list to temp*/
    while(temp) /*while temp!=NULL */
    {
        printf("%d ",temp->num); /*print the data stored in data part*/
        temp=temp->next; /*shift the pointer to next node*/
    };
    printf("\n");
}
/*the main program*/
main()
```

```
{
    init(1); /*initialize the list*/
    add(2); /*add an element*/
    add(3);
    add(4);
    add(5);
    show(); /*show the list element*/
}
```

Second method

```
/*example of linked list first element is at begining*/
#include<stdio.h>
#include<alloc.h>
/*define a self-reeferring structure for use as node*/
struct my_struct
{
    int num;
    struct my_struct *next;
};
typedef struct my_struct node;
node *list,*cur; /*declare two global variables list and cur*/
/*initialize the list or creates a list*/
void init(int val)
{
    list=(node *)malloc(sizeof(node)); /*Allocate a block of memory*/
    cur=list; /*make current is list*/
    list->num=val; /*put the data*/
    list->next=NULL; /*put null in pointer*/
}
/*add to list*/
void add(int val)
{
    node *temp;
    temp=(node *)malloc(sizeof(node)); /*allocate a block*/
    cur->next=temp; /*add the new node after current*/
    temp->num=val; /*put the data*/
    temp->next=NULL; /*put address in list into pointer part*/
    cur=temp; /*store the new address in list*/
}
/*prints the list*/
void show()
{
    node *temp;
    temp=list; /*assign the address in list to temp*/
    while(temp) /*while temp!=NULL */
    {
```

```
        printf("%d ",temp->num);/*print the data stored*/
        temp=temp->next; /*shift the pointer to next node*/
    };
    printf("\n");
}
/*the main program proper*/
main()
{
    init(1); /*initialize the list*/
    add(2); /*add an element*/
    add(3);
    add(4);
    add(5);
    show(); /*show the list element*/
}
```

Note in above examples the first example shows adding at beginning and the second adding at end. We will use those in two different situations first one in stacks and second one in queue. The show functions are same they just traverse the list and print the values stored in different nodes.

We now saw how to make a linked list. Next we will see how to search for a particular data in a list, how to add a new node before and after a node, lastly how to delete a particular node.

12.2.2 Searching

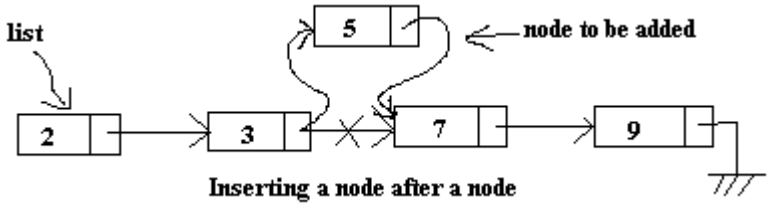
Searching a data in a list is done sequentially. That is we have to start at the head and move up to tail, checking the data in every node. The way one follows is first store the address of head of list. Compare the data, if found end searching else go to next node by doing temp=temp->next and repeat comparing until we reach the last node. See bellow for a search function on the above created list.

```
int search(int val) /*we need to pass the value*/
{
    node *temp;
    int nc=0; /*node counter*/
    temp=list; /*store the list head in temp*/
    while(temp) /*while not last node */
    {
        if(temp->num==val) /*compare data*/
        return (nc);/*if matches return node number*/
        temp=temp->next; /*else go to next node*/
        nc++; /*increment node count*/
    } /*loop*/
    return(-1); /*loop exhausted so not found*/
    /*return -1*/
```

```
}
```

12.2.3 Inserting a node

We can add an element to a list in 2 ways. Add after and add before. Add before is tricky but add after is somewhat easier. The logic for adding after a data value is, first search the data, get the address of node holding it, allocate memory for new node to be inserted after, assign the data value, store the next node address of current node in new nodeâs next node address, store the new nodeâs address in current nodeâs next node field. Well this logic is confusing, is not it? See the picture bellow.



In the above picture we have a list, which has 2,3,7 and 9 as its element. Now we want to add a new node whose data value is 5 and it should be inserted after the node containing 3. We first browse the list to find the address of node containing 3. then we allocated the new node, which contains 5, we now store the address contained in the next field of node containing 3 in the new nodeâs next node field. Now we store the address of new node in the next node field of node containing 3. Now we will write a function to do the same.

```
void insert_after(int which, int what)
{
    node *cur, *temp;
    cur=list;
    while(cur->num!=which)
    {
        cur=cur->next;
    }
    /*we found the node now*/
    temp=(node *)malloc(sizeof(node));
    temp->num=what;
    temp->next=cur->next;
    cur->next=temp;
}
```

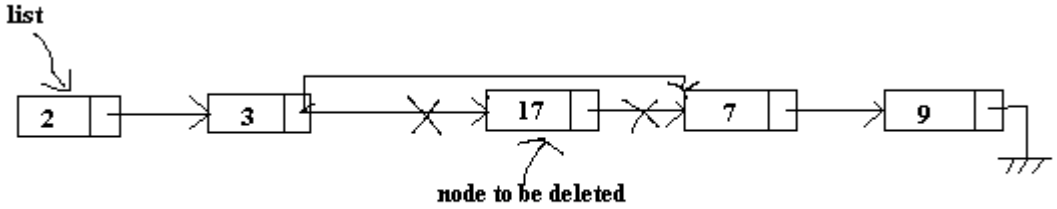
Write this above function into list program and modify the main function to call this function after the show() , again call show function to see what happens.

In case of add before we have to store the previous node address and call use the technique of add after. See the above picture for add after if we say we are adding 5 before 7 we are right too. For adding before we need to determine the address of node

before the node to which we are interested to add before, then we have to perform add after. Try to write a function to do that.

12.2.4 Deleting

When we find a node, which has no use in the list anymore, we have to delete it. When a node is deleted the memory occupied by it is freed. Hence we get more memory to work with. We will see how we can delete or remove a node from a list. The logic for deletion is first search the node to be deleted and store its previous nodeâs address, store the next node address in previous nodeâs next field, then free the node. We will implement this logic in a function.



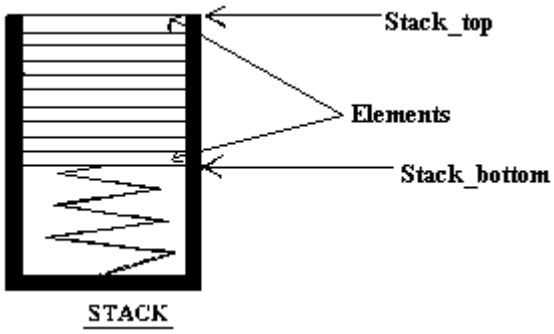
Deleting a node

```
void remove_node(int what)
{
    node *temp,*prev; /*prev is used to store the previous
    nodes address*/
    temp=list; /*store the list begining in temp*/
    /*check if first node is to be deleted*/
    if(temp->num==what)
    {
        list=list->next;
        free(temp);
    }
    else
    {
        while(temp)
        {
            prev=temp;
            temp=temp->next;
            if(temp->num==what)
            {
                prev->next=temp->next;
                free(temp);
            }
        }
    }
}
```

We saw how to create, insert and remove a node in a list. Here we have assumed the data values are present in list. If data value not present in the list then what will happen. The program will not complain but nothing will happen and we will not know why nothing happened. It is better to write some code, which will print some messages regarding what has happened. Try to implement some verbose output or return some error values and check those in calling program. These are left to you for practice.

12.3 Stacks

Stack is a special case. It is said to be Last In First Out(LIFO). That is when ever an element is put in to the stack the order of retrieval is opposite. That is last element put on to the stack is taken out first. It is analogous to a single ended pipe.



The stack has two functions associated with it. **Push** and **pop**. Push **pushes elements to stack** and **pop gets elements from stack**. Pop is a destructive function when ever a pop is done the element is taken out of stack and the node is deleted. The next element is now at the stack top. Hence another pop retrieves the next. As an example first we pushed 3 values as follows, 100, 200, 350, when we pop first 350 will be popped next 200 and then 100. Let us write a code for stack.

```
/*example of linked list implementation of stack*/
#include<stdio.h>
#include<alloc.h>
/*self referring structure*/
struct my_struct
{
    int num;
    struct my_struct *next;
};
typedef struct my_struct node;
/*bottom of the stack is null.*/
node *list=NULL;
/*push to stack*/
void push(int val)
{
```

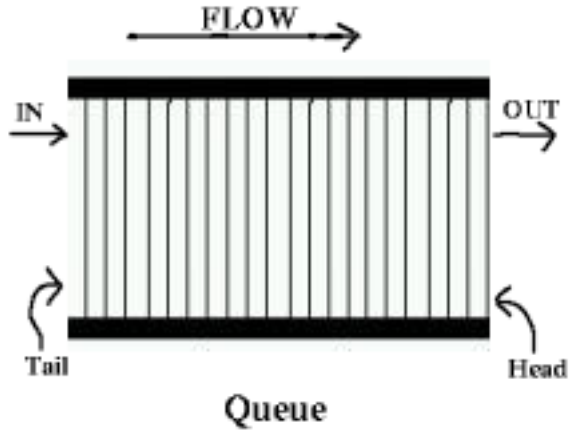
```
    node *temp;
    temp=(node *)malloc(sizeof(node)); /*allocate a block*/
    temp->num=val; /*put the data*/
    temp->next=list; /*put address in list into pointer part*/
    list=temp; /*store the new address in list*/
}
int pop() /*pop will pop a value and returned it*/
{
    node *temp;
    int t;
    if(list==NULL)
        return -1; /*stack is empty*/
    temp=list;
    list=list->next;
    t=temp->num;
    free(temp);
    return t;
}
main()
{
    push(1);
    push(2); /*push an element*/
    push(3);
    push(4);
    push(5);
    push(6);
    push(7);
    printf("%d\n",pop()); /*pop element 7*/
    printf("%d\n",pop()); /*6*/
    printf("%d\n",pop()); /*5*/
    printf("%d\n",pop()); /*4*/
    printf("%d\n",pop()); /*3*/
    printf("%d\n",pop()); /*2*/
    printf("%d\n",pop()); /*1*/
    printf("%d\n",pop()); /*we are already at bottom of stack*/
}
```

Remember here stack has only one end. Push puts an element to the stack at its opened end. Pop pops the element from the open end of the stack.

Application of stack is very enormous. In real computing stack is used everywhere. Whenever there is a function call the system pushes its content on to the stack. On returning from function it pops the content and restore the state. Your Operating system also uses stack to track its operation and state. The compiler you use uses stack to calculate expressions. The text editor you use uses stack to track and record the undo information. So uses of stack is many, it depends on you how to use it.

12.4 Queue

This is another use of linked list. This is also used to store data dynamically at the run time. Data stored in queue is accessed **First In First Out (FIFO)** manner. The queue is a open pipe line. Data comes into queue at one end and goes out at the other end. The receiving end is known as tail and the other end from which data is got is known as head.



Queue

The queue has two function associated with it. One is put and the other is get. Put function puts an element at tail and the get function gets one element from the head and deletes it. So whatever things are put to queue is retrieved from queue in the same manner. Let us see an example now to test it.

```
/*queue example using linked list*/
#include<stdio.h>
#include<alloc.h>
/*define our self referring structure*/
struct my_struct
{
    int num;
    struct my_struct *next;
};
typedef struct my_struct node; /*name it as node */
/*declare global variables head and tail*/
node *head,*tail;
/*declare a flag to check if queue is empty*/
int q_empty=1;
/*the put function puts at tail */
void put(int val)
{
    node *temp;
```

```
    if(q_empty) /*if queue is empty*/
    {
        head=tail=(node *)malloc(sizeof(node)); /*create it*/
        tail->num=val; /*put the value*/
        tail->next=NULL;
        q_empty=0; /*queue empty is false*/
    }
    else /*if queue is not empty then*/
    {
        temp=(node *)malloc(sizeof(node));/*allocate a node*/
        temp->num=val; /*put value to it*/
        temp->next=NULL; /*make it last element*/
        tail->next=temp; /*make it next to tail*/
        tail=temp; /*make new node as tail*/
    }
}
/*get function to get from head and return a value*/
int get()
{
    node *temp;
    int t;
    if(q_empty) /*if queue is empty*/
    {
        return (-1); /*return error=-1*/
    }
    temp=head; /*if not empty*/
    if(temp->next==NULL) /*if we are at the end of the queue*/
    {
        q_empty=1; /*make q_empty=1 to tell queue is empty*/
    }
    else /*if this is not last element*/
    {
        head=head->next; /*shift head to next element*/
    }
    t=temp->num; /*extract data from node*/
    free(temp); /*free the accessed node*/
    return(t); /*return data*/
}
main()
{
    put(1); /*put data to queue*/
    put(2); /*put data to queue*/
    put(3); /*put data to queue*/
    printf("%d\n",get()); /*get data from queue*/
    printf("%d\n",get()); /*get data from queue*/
    printf("%d\n",get()); /*get data from queue*/
```

```
    printf("%d\n",get()); /*queue is empty so -1 will be printed*/
    put(100); /*put data to queue*/
    put(101); /*put data to queue*/
    put(103); /*put data to queue*/
    printf("%d\n",get()); /*get data from queue*/
    printf("%d\n",get()); /*get data from queue*/
    put(104); /*put data to queue*/
    printf("%d\n",get()); /*get data from queue*/
    printf("%d\n",get()); /*get data from queue*/
    printf("%d\n",get()); /*queue is empty so -1 will be printed*/
}
```

The above example is self-explanatory; still we should discuss the method we used. As we said earlier, a queue has 2 ends, a head and a tail. We get and put at these ends respectively. So we declared them. As head and tail are addresses of nodes we declare them as pointer to node. Then we declare an integer variable q_empty and assigned it 1 (TRUE). As there is no element in the queue at beginning. Then we wrote two functions put() which puts an element into the queue and a get() which retrieves elements from the queue. The put() function creates an empty node, put the value and add it to the queue if queue is not empty; else it creates a node and make head and tail too and assign 0 (FALSE) to q_empty. As when there is only one element, the head and tail are the same. The get() checks the empty condition and retrieves the data from the queue and deletes the node if the queue is not empty. If the element is last element then it retrieves the value and set q_empty to 1 (TRUE). Else it returns an error-code. The code inside main() tests the proper working of the queue algorithm.

12.5 Summing up

This chapter gives some idea to the data structure. We cannot discuss all the aspect in a single chapter and also in a small tutorial like this. Several books are written on this subject. You can easily obtain one from your local store or library. Choose the one that is suitable for you.

This chapter described memory allocation, de-allocation and how to use them; Singly linked list, stack and queue. Note for convenience integer type is taken in the examples but that can be changed to any valid data type.

In the next chapter we will see file input output operations. We have seen the normal console input and output functions; we will compare them here.

Chapter 13

File Operations

We have heard lots about files. Data file, executable file, source file, Binary file, Text file and so on. But what is a file. To know what a file is, we have to go back to data. Datum (singular) is a basic unit. Collection of related data is known as record, collection of such similar records is known as a file. Next comes where the file exists and how it is managed. Nearly all operating system has a file system, except those ***embedded operating systems*** sit inside your TV, VCR/VCP, Audio systems and Mobile phones. Even some intelligent wristwatch can have an operating system. In those operating systems a file system is not necessary. This file system handles most of the job dealing with file. Of course we will never know when and how it does its job. We will just see how the file can be created and written.

Before doing anything with file the file has to be opened. After opening we can write into or read from the file. So this is the sequence maintained in any file operation. After our work is over the file should be closed. If the file is not closed before program exits, then the program forces the file to be closed and the data in the file is lost if it was opened for writing.

13.1 Opening and closing a file

Any operation on file must precede file opening. File is opened with library functions available with compiler. Standard file opening functions are **fopen ()** and **open ()**. Both the functions opens a file for input or output depending on the parameters passed.

*FILE *fopen (const char *filename, const char *mode);*

Filename is the name of the disk file we want to open, mode is the string, known as the attribute string or mode string, depending on which the further operations are done. FILE is a structure, fopen() returns a pointer to this structure. This file structure contains some vital information regarding file like which type, the collective mode with which the file is opened, and position of current operation and so on. We may not need to modify the structure. We only use it the internal fields are not so important for us

now. If the fopen() function succeeds then it returns a pointer to this FILE structure else it returns a NULL pointer. This means if the file exists then a valid FILE pointer will be returned else a NULL pointer will be returned. The mode string can have:

r read-only, open an existing file
w create a new file or overwrite a file if it exists
a append, open the file at end of file if exists or create a new one.
r+ open an existing file for update read and write
w+ create a new file for update; if file exists it will be overwritten
a+ open for append, update at the end of file or create a new one if the file does not exist

These above characters can be associated with **t** or **b** to signify the file type text or binary respectively. Hence **rt, r+t, wt+, r+b** etcetera are possible.

Any opened file should be closed. This closing is done by fclose () or close () library function. It has no such critical syntax it only takes the file handle as argument.

*int fclose (FILE *fp);*
On success it returns 0 or if error occurs while closing it returns EOF.

Now let us see how to open a file and close it.

```
/*opening a file to test its presence*/
#include<stdio.h>
main()
{
    char fn[15];
    FILE *fp;
    printf("enter a file name :"); /*get the file name from user*/
    scanf("%s",fn);
    fp=fopen(fn,"r"); /*try to open the file in readonly mode*/
    if(!fp) /*if fp==NULL*/
    {
        printf("file does not exist\n");
        exit(1);
    }
    printf("file exists\n");
    fclose(fp); /*close the file*/
}
```

The second library function is **open ()** let us discuss about it now. This function opens a file in a given mode. The syntax for this function is as follows.

*int open(const char *path, int access, unsigned mode);*

Here the **mode** is optional. **Path** is the path of file to be opened and the access is a combination of pre-defined constants. On success it returns a non-negative integer and put the file pointer at the beginning of file. On error it returns **-1** and set **errno** to one of the following:

ENOENT - file not found
EMFILE - too many open files
EACCES - permission denied
EINVACC - invalid access

The access field can be any one or a combination of these fields

O_RDONLY - open for reading
O_WRONLY - open for writing
O_RDWR - open for reading and writing
O_APPEND - open for append
if this is used the file pointer is set at the end of file
O_CREAT - creates and opens a file
If the file already exists then no thing happen else it is created
O_EXCL - exclusive open, used with O_CREAT, if file exists returns
error
O_TRUNC - open with truncation, if the file exists then the file ts truncated
to 0
O_BINARY - open the file in binary mode
O_TEXT - open in text mode

Several other access modes are available some of them are also operating system dependant check them with your compiler documentation. The mode is a combination of the follows:

S_IREAD - permission to read
S_IWRITE - permission to write

Now let us write a program to illustrate the **open ()** function.

```
/*file example using open()*/
#include<fcntl.h>
#include<sys\stat.h>
#include<io.h>
main()
{
int fh; /*file handle*/
char fn[15];
printf("Enter a file name :"); /*ask to input a file name*/
scanf("%s",fn);
fh=open(fn,O_RDONLY);
if(fh==-1)
{
printf("File does not exist\n");
```

```
exit(1);
}
printf("File present\n");
close(fh);
}
```

In the above examples we saw how to test the presence of a disk file. The key is open the file for read if it does not exist then it will give an error. Just by modifying the mode string in first example or access string in second example we can open it to write into the file. We will see them next.

13.2 Writing into a file

Till now we know **printf** (), **putc** (), **putch** () and **puts** () are output functions. These functions have their cousins for files. They are **fprintf** (), **fputc** (), **fputs** () etcetera. Check their syntaxes to know more about them. We will see some of them in our example. As we know **printf** () and its sister functions are capable of formatted output, we will use **fprintf** () more. We will use **fwrite** () function to write a buffer to a file.

*int fprintf (FILE *fp, const char *format-string, arguments);*

See the similarity between **printf** () and **fprintf** (). The only difference is **FILE*** field. Incase of **printf** () the **FILE*** field is not present because it is used for stdout. Here **fp** is a valid **FILE*** obtained from **fopen** (). See the example bellow.

```
/*writing to a file*/
#include<stdio.h>
main()
{
    char fn[15];
    char *temp;
    int p;
    float q;
    FILE *fp;
    printf("enter a file name :");/*ask the user to enter a filename*/
    scanf("%s",fn);
    fp=fopen(fn,"r"); /*open the file for reading*/
    if(!fp) /*if unsuccess*/
    {
        printf("file does not exist, creating it\n");
        fclose(fp);
        fp=fopen(fn,"wt+"); /*create it*/
        if(!fp) /*if unsuccess again*/
        {
            printf("unable to create file\n");
```

```
            fclose(fp);
            exit(1); /*exit */
        }
    }
    else /*if exists*/
    {
        printf("file exists, oppening for append\n");
        fclose(fp);
        fp=fopen(fn,"a+t"); /*open it for append*/
        if(!fp) /*unsuccess??*/
        {
            printf("unable to open file\n");
            exit(2); /*exit*/
        }
    }
    fprintf(fp,"This is my test file\n"); /*write a string to file*/
    fflush(stdin); /*flush the stdin buffer*/
    printf("enter a string : "); /*ask the user to enter a string*/
    gets(temp);
    fprintf(fp,"%s\n",temp); /*write the user entered stringto file*/
    printf("enter an integer and a float : "); /*ask for numbers*/
    scanf("%d%f",&p,&q);
    /*write a formatted string*/
    fprintf(fp,"you entered int=%d, float=%f\n",p,q);
    printf("closing file\n");
    fclose(fp); /*close the file*/
}
```

The above example is self-explanatory. The only unknown function is fflush(), this function is used to flush the stream or a buffer. Here we used to flush the stdin buffer. Try to run the program without fflush () and with fflush () to see what it does and analyze why it happens.

13.3 Reading from a file

Library functions to read from a file are cousins to **scanf ()**. Hence the first one we can guess is **fscanf ()** the opposite to **fprintf ()**. The **fscanf ()** function is used to read formatted values from a file. See the example bellow.

```
/*example showing formated reading from file*/
#include<stdio.h>
#include<stdlib.h>
main()
{
    int i,j,k,l;
```

```
    float f,g;
    char temp[50],temp2[50];
    FILE *fp;
    fp=fopen("test.dat","r");
    if(!fp)
    {
        printf("file not found, createing it\n");
    }
    printf("file found, overwriting it\n");
    fclose(fp);
    fp=fopen("test.dat","wt+");
    if(!fp)
    {
        printf("unable to create file\n");
        fclose(fp);
        exit(1);
    }
    printf("enter 2 integers a floating number and your name : ");
    scanf("%d%d%f%s",&i,&j,&f,temp);
    printf("writing these numbers to file\n");
    fprintf(fp,"%d %d %f %s\n",i,j,f,temp);
    fclose(fp);
    fflush(stdin);
    printf("reopening file for reading the numbers\n");
    fp=fopen("test.dat","r");
    fscanf(fp,"%d%d%f%s",&k,&l,&g,temp2);
    fclose(fp);
    printf("value read from file are %d %d %f %s\n",k,l,g,temp2);
}
```

This is just we are looking into the similarity of functions normally used and functions used with files. There are several other functions available for these purposes. Suppose we have to write a single character at a time and read in same way we should not use fprintf () or fscanf () functions. There we have to use getc (), putc (), fgetc () and fputc (). Let us discuss them now.

The getc () and putc () are macros defined to read or write a single character from or to a stream or file. After reading or writing they increment the file pointer to point to the next character. The fgetc () and fputc () are function version of the getc () and putc (). If an error occurs or if at end of file they return EOF.

```
/*character input output example*/
#include<stdio.h>
#include<stdlib.h>
main()
{
    char ch;
```

```
    FILE *fp;
    fp=fopen("test.dat","w+");
    if(!fp)
    {
        printf("cannot create file\n");
        fclose(fp);
        exit(1);
    }
    printf("type something on keyboard and end it with \\ ");
    do{
        fflush(stdin); /*flush stdin buffer*/
        ch=getc(stdin); /*read from stdin*/
        putc(ch,fp); /*write to file*/
    }while(ch!='\\'); /*until back slash*/
    fclose(fp);
    fflush(stdin);
    fp=fopen("test.dat","r+");
    if(!fp)
    {
        printf("error opening file\n");
        exit(2);
    }
    do{
        ch=getc(fp); /*read from file*/
        putc(ch,stdout); /*write to stdout */
    }while(ch!=EOF); /*until end of file*/
}
```

Rewrite the above program by changing the **getc ()** tp **fgetc()** and **putc ()** to **fputc ()** the program will work similarly. Till now we are dealing with predefined data-type, we will see next how to write a structure to a file.

13.4 Writing and reading a buffer

Every time we cannot write a string or a character to a file and if it is also possible it is time consuming. Hence we must follow some way to write a bulk of data to a file or read a bulk of data from a file. One may say using printf () we can write a large string, yes it is true but we have to prepare a string then we canwrite. Consider a situation in which we are writing students record to a file. In this case we have to prepare a string which should contain student name, marks in different subjects address and grade. For this purpose we have to convert all the required parameters to alpha type then copy them to a string. Then only we can write this string to a file. While reading we have to read the string entirely and break it up in to several strings. This is a cumbersome method. We have to find an alternative way. If we can write a structure to a file then our purpose is solved. This is achieved by using fwrite () and fread (). These two library

functions deal with buffer writing and buffer reading to or from file.

size_t fwrite(const void *buf, size_t size, size_t number_of_buffers, FILE *fp);
size_t fwrite(const void *buf, size_t size, size_t number_of_buffers, FILE *fp);
buf is the buffer to be written
size is size of buffer in bytes
number_of_buffers is number of items to be written
fp is the stream to which the data should be written.

Now let us consider an example. Suppose we need to write a structure which contains name, age, sex and phone number of our friend, and we need to store several such structure to a file called addbook.dat. See the following example to get an idea of how can it be done.

```
/*file demo for buffer writing using fopen, fread, fwrite and fclose*/
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
/*our structure to hold the details of our friend*/
struct friend_add
{
    char name[80]; /*name is 80 chars max*/
    int age;
    char sex[7];
    char phone[20]; /*phone number is stored as string*/
};
typedef struct friend_add address;
main()
{
    FILE *fp;
    char temp[80];
    int temp_age;
    address this_adrs;
    /*we will open our file to put the address*/
    fp=fopen("addbook.dat","r+");
    if(!fp)
    {
        printf("file not present\n");
    }
    fclose(fp);
    printf("opening for append...\n");
    fp=fopen("addbook.dat","a+");
    if(!fp)
    {
        fclose(fp);
        printf("error. couldnot open file\n");
        exit(1);
    }
    printf("Enter name : ");
```

```
    scanf("%[^\n]s",temp);
    strcpy(this_adrs.name,temp);
    fflush(stdin);
    printf("enter age :  ");
    scanf("%d",&temp_age);
    this_adrs.age=temp_age;
    fflush(stdin);
    printf("enter sex :  ");
    scanf("%[^\n]s",temp);
    strcpy(this_adrs.sex,temp);
    fflush(stdin);
    printf("enter phone number :  ");
    scanf("%[^\n]s",temp);
    strcpy(this_adrs.phone,temp);
    /*we complete reading a structure*/
    /*write structure this_adrs of size sizeof(address) 1 item to fp*/
    fwrite(&this_adrs,sizeof(address),1,fp);
    fclose(fp);
    /*now open for reading*/
    fp=fopen("addbook.dat","r");
    if(!fp)
    {
        fclose(fp);
        printf("error. reading file\n");
        exit(2);
    }
    memset(&this_adrs,'\0',sizeof(address)); /*clear the structure*/
    fread(&this_adrs,sizeof(address),1,fp);
    fclose(fp);
   printf("%s\n%d\n%s\n%s\n",this_adrs.name,this_adrs.age,this_adrs.sex,this_adrs.phone);
}
```

In the above program we first defined a structure, then we filled it up and write the whole structure to the file. Using memset (), we fill null characters in to the memory, where structure exists. We then reopen the file for reading and we read the whole structure into memory and print them. Check the printed value with input value; are they same or not. You may be wondering why the memory region filled up with null. The answer is to make sure there is no residual values exist. Follow the second program bellow. Which does the same thing but uses open, read, write and close.

```
/*buffer writing using open,read,write and close*/
#include<stdio.h>
#include<io.h>
#include<fcntl.h>
#include<sys\stat.h>
#include<stdlib.h>
#include<string.h>
/*our structure to hold the details of our friend*/
struct friend_add
```

```
{
    char name[80]; /*name is 80 chars max*/
    int age;
    char sex[7];
    char phone[20]; /*phone number is stored as string*/
};
typedef struct friend_add address;
main()
{
    int handle;
    int res;
    char temp[80];
    int temp_age;
    address this_adrs;
    /*we will open our file to put the address*/
    handle=open("addbook1.dat",O_RDONLY);
    if(handle==-1)
    {
        printf("file not present\n");
    }
    close(handle);
    printf("opening for append...\n");
    handle=open("addbook1.dat",O_RDWR|O_CREAT|O_APPEND);
    if(handle==-1)
    {
        close(handle);
        printf("error. couldnot open file\n");
        exit(1);
    }
    printf("Enter name : ");
    scanf("%[^\n]s",temp);
    strcpy(this_adrs.name,temp);
    fflush(stdin);
    printf("enter age : ");
    scanf("%d",&temp_age);
    this_adrs.age=temp_age;
    fflush(stdin);
    printf("enter sex : ");
    scanf("%[^\n]s",temp);
    strcpy(this_adrs.sex,temp);
    fflush(stdin);
    printf("enter phone number : ");
    scanf("%[^\n]s",temp);
    strcpy(this_adrs.phone,temp);
    /*we complete reading a structure*/
    /*write structure this_adrs of size sizeof(address) 1 item to fp*/
    res=write(handle,&this_adrs,sizeof(address));
    if(res!=sizeof(address))
    {
        printf("error. Writing to file\n");
```

```
        exit(2);
    }
    close(handle);
    /*now open for reading*/
    handle=open("addbook1.dat",O_RDONLY);
    if(handle==-1)
    {
        close(handle);
        printf("error. opening file for read\n");
        exit(3);
    }
    memset(&this_adrs,'\0',sizeof(address)); /*clear the structure*/
    res=read(handle,&this_adrs,sizeof(address));
    if(res!=sizeof(address))
    {
        printf("error. Reading from file\n");
        exit(4);
    }
    close(handle);
   printf("%s\n%d\n%s\n%s\n",this_adrs.name,this_adrs.age,this_adrs.sex,this_adrs.phone);
}
```

13.5 Random access files

Whatever examples or the methods of reading and writing we discussed are all sequential. We will discuss random access methods now. A random access is accessing data from any location within that file but skipping preceding records or data. Suppose I want to access 10th record or 100th character, what we generally do is browse through the file from beginning and stop at where our goal is. Incase of random access there is no need of browsing through all the records but we can directly jump to the desired location and start reading or writing. This random access is achieved by **seek** functions. Let us see the syntax of **seek**.

*int fseek(FILE *fp, long offset, int whence);*
fp file handle
offset number of bytes to skip
whence from where

The return value from **fseek()** is **0 for success and negative for error**. Here fp is a file handle returned by fopen(). **Offset is the distance in bytes** expressed in long. **Whence** is usually described by three pre-defined constants **SEEK_SET, SEEK_CUR and SEEK_END**; these 3 constants represent **beginning, current and end of file position** respectively. Let us see an example.

```
/*file example for random access*/
```

```
#include<stdio.h>
#include<io.h>
#include<string.h>
#include<stdlib.h>
main()
{
    FILE *fp;
    char str[]="a quick brown fox jumped over a lazydog";
    char str2[50];
    char ch;
    int l=strlen(str);
    fp=fopen("test.txd","a+"); /*open file for create or append*/
    if(!fp)
    {
        printf("error. file open\n");
        fclose(fp);
        exit(1);
    }
    fputs(str,fp); /*write the string to file*/
    fclose(fp); /*close the file*/
    fp=fopen("test.txd","r");
    if(!fp)
    {
        printf("error. reading file\n");
        fclose(fp);
        exit(2);
    }
    fgets(str2,l+1,fp);
    printf("original string in file is\n%s\n",str2);
    fseek(fp,0L,SEEK_SET); /*go to begining of file*/
    ch=fgetc(fp); /*get a character*/
    printf("%c\n",ch); /*print it*/
    fseek(fp,-7L,SEEK_END);/*goto 7 character before EOF*/
    fgets(str2,8,fp);
    printf("%s\n",str2);
    fseek(fp,-21L,SEEK_CUR); /*goto 21st character before
    current position*/
    fgets(str2,7,fp);
    printf("%s\n",str2);
    fseek(fp,1L,SEEK_CUR); /*skip a single character from current*/
    fgets(str2,5,fp);
    printf("%s\n",str2);
    fseek(fp,8L,SEEK_SET); /*skip 8 character from begining of file*/
    fgets(str2,10,fp);
    printf("%s\n",str2);
    fclose(fp);
```

```
}
```

See in above example when we refer to SEEK_END we use negative offset. The negative offset is due to the fact we cannot seek beyond the end of file and we have to go backward in file so negative sign is needed. Also see we have used negative offset with one SEEK_CUR this is because we want to go back from that point.

here is another library function lseek (), this function does the same what fseek() does but used with file handle returned from open() function. The only difference is this function returns a long instead of an integer as in the case of fseek(). The working of lseek is left for you as an assignment, try to write in the light of above example.

13.5.1 rewind()

This library function is responsible to take the file pointer to the beginning of file. The syntax is: void rewind(FILE *fp). This is same as fseek(fp, 0L, SEEK_SET). Let us see an example.

```
#include<stdio.h>
main()
{
    FILE *fp;
    char str[]="abcdefghijklmnopqrstuvwxyz";
    char ch;
    int i;
    /*create the file for writing*/
    fp=fopen("test4.dat","w");
    if (!fp)
    {
        printf("error opening file for write\n");
        exit(1);
    }
    /*write the string to file*/
    fprintf(fp,"%s",str);
    fclose(fp); /*close the file*/
    /*open the file again for reading*/
    fp=fopen("test4.dat","r");
    if(!fp)
    {
        printf("error opening file for read\n");
        exit(2);
    }
    /*read the file until k is reached*/
    while(ch!='k')
    {
        ch=getc(fp);
        printf("%c ",ch);
```

```
    }
    printf("\n");
    /*rewind the file*/
    rewind(fp);
    /*get 3 characters from file*/
    for(i=0;i<3;i++)
    {
        ch=getc(fp);
        printf("%c\n",ch);
    }
    fclose(fp);
}
```

The above example will print **a** to **k** then **abc**. When it gets a character from file using **getc()**, the file pointer advances after **k** is read the file pointer actually points to **l** but we rewind the file and forced the file pointer to be at the beginning of the file. So further reads are done at the beginning.

13.6 Summing up

This chapter dealt with file handling. In this chapter we discussed how to open files, different modes for open a file; then writing and reading from files. We also saw hoe to jump to different location this means random access. This chapter is as important as that of data structure because file handling is needed often in real world programming.

Chapter 14

Mixed Mode Programming

In this chapter we will see how we can interface other languages with C and how to interface C with other languages. That is we have some codes (collection of functions) in lets say in assembly language and we want to call those code from our C program and we have some C codes (functions) we want to use them in other languages like C++.

Before we dive into this sort of adventure it is better to know some interesting features of compilers. In the previous paragraph we stated our intention to use functions. Well you might think already there is a chapter regarding function, so what is more great about functions? Well yes, There are some thing regarding functions which we have not delet in the functions chapter. That is calling conventions. This is more or less specific to compiler implementation and language implementation.

14.1 Calling conventions

Different language (e.g Pascal, C, FORTRAN...) have different protocol to call functions. Basically there are two conventions.

Function entry and exit, call stack management

- Pascal calling convention
- C calling convention (cdcl)
- register calling convention (fastcall)
- stdcall

Calling convention influence two things, they are

- How the parameters to a function is passed
- How the housekipping of stack handling is done

14.1.1 Pascal calling convention

Pascal calling convention passes parameters on the stack and pushes them from left to right in the parameter list. And The parameters are cleared off the stack by the called function.

If a stack map is taken then it will be some thing like
ebp + 20 value of i, 4 bytes
ebp + 16 value of b, 4 bytes, only lowbyte significant
ebp + 08 value of d, 8 bytes
ebp + 04 return address, 4 bytes
ebp + 00 old ebp value
The above code is a part of Pascal code *FUNCTION test1(i:integer; b:boolean;d:double):integer;pasca*l;

14.1.2 cdcl

Cdecl calling convention passes parameters on the stack and pushes them from right to left in the parameter list. The parameters are cleared off the stack by the calling function.

the equivalant stack frame will be
ebp + 16 value of d, 8 bytes
ebp + 12 value of b, 4 bytes, only lowbyte significant
ebp + 08 value of i, 4 bytes
ebp + 04 return address, 4 bytes
ebp + 00 old ebp value
see the order has changed from abobe section.

14.1.3 Stdcall

Sdtcall calling convention passes parameters on the stack and pushes them from right to left in the parameter list. The parameters are cleared off the stack by the called function.

ebp + 16 value of d, 8 bytes
ebp + 12 value of b, 4 bytes, only lowbyte significant
ebp + 08 value of i, 4 bytes
ebp + 04 return address, 4 bytes
ebp + 00 old ebp value
same as above but the stack is cleared by the called function.

14.1.4 fastcall or register call

Register calling convention passes parameters in registers eax, edx, ecx and on the stack and processes them from left to right in the parameter list. There are rules to decide what goes into registers and what goes on the stack is defined by the language implementation document or by the compiler implementors choice.

ebp + 08 value of d, 8 bytes
ebp + 04 return address, 4 bytes

ebp + 00 old ebp value
The rest of the parameters are passed through registers
i in *eax* **b** in *edx* register.

Note: The above codes were taken from Delphi compiler. MS-VC supports cdcl pascal stdcall

14.2 Libraries

Libraries, A collection of books? Right, But in our context it is a collection of functions or procedures. The procedures or functions written by us or by a third party. Well the advantage of libraries you can guess from the fact that C has only 28 reserved key words. But how come we are using so many functions in our programs? Well that is the beauty of library. The C language is extensible using libraries.

There are libraries available for different purposes. If you are writing an application which deals with connecting to a database server and fetching data then you can write your data retrival functions but it will be reinventing the wheel. Again think about situation that you need similar functionality in several projects you do. Is it wise to rewrite the code in every project? No. The better option is to put those functions which are needed time to time and make library and include the library in your projects.

14.2.1 How to make a library

very small example will be define write all the functions in to a file compile into an object file and link it to your projects which demands functions from that library. But it is a very primitive way. The compiler system you are using already have a library system. for Borland group it is **tlib** and for gcc it is **ar**. Besides library a header file is needed too. This header file is the interface to the library. This header file contains the function prototypes which are there in the library.

Consider the following example

```
/*mylib.c this library is a demonstration library      */
/*the concerned interface header file is mylib .h     */
/*the linker expects mylib.lib or mylib.a during linking*/
/*depending on the compiler system                */
void foo(int bar)
{
        printf("%d bars in foo\n",bar);
}
int foo_bar(int car, int truck)
{
        return(car-truck);
}
```

lets assume you have several files with functions defined in them. Just compile them to object files and using the library manger for your compiler system createa a library named mylib.lib or mylib.a. for the header file which will be the interface to

mylib library just copy the prototypes into a file and name it as mylib .h. Now the library mylib is ready to work.

```
/*mylib.h this is the interface to mylib library      */
/*the linker expects mylib.lib or mylib.a during linking*/
/*depending on the compiler system                    */
#ifndef __myheader_h_
#define __myheader_h_
void foo(int bar);
int foo_bar(int car, int truck);
#endif
```

Now when ever you need any function from the mylib library, include mylib.h in your projects; And link mylib library to the project.

14.3 Using assembly language routine with C

We now knew some bit of calling convention and how to create a library. It is time to write some simple code in assembly and call that code from our C program. If you are so much enthuciastic about it, then please write a C function and call that function from main() , compile it and check the assembly code how a function is translated into assembly from C.

Consider this as an adventure. Hint: use compiler to generate assembly.

Here for assembly language we will discuss only x86 assembly in 32/16 bit and our assembler will be NASM. If you want to have a copy of nasm then you can get it from http://nasm.sourceforge.net/. For C compiler we will stick to gcc and TC. You may try it out with other compiler and assembler too.

14.3.1 An example assembly routine

Here we will write an assembly routine to accept 2 numbers from a c function and return the sum of both the number. Let us assume the name is ***addnum***. The calling convention will be cdcl. Let us write the function in assembly language here with NASM syntax.

```
[BITS 32]
global addnum
segment .data
segment .bss
segment .text
addnum
    push    ebp             ;save the ebp
    mov     ebp,esp         ;move esp to ebp
    mov     eax,[ebp+12]    ; access the first parameter and store it in eax
    mov     ebx,[ebp+8]     ; access the second parameter and store it in ebx
    add     eax,ebx         ; add the 2 numbers and put the sum in eax
    pop     ebp             ; pop ebp as we are going to exit from function
```

```
    ret                     ; return to calling function
```

14.3.2 A C program to call assembly language routine

Now letus write a C program to call the function addnum with 2 parameters.

```
#include<stdio.h>
extern int addnum(int a, int b);
int main()
{
    int res=0;
    res=addnum(5,6);
printf("result is %d\n",res);
return 0;
}
```

Now We will see how to make bothe the code and link them to get an executable. For this first of all the assembly code need to be assembled. To assemble the above code you must enter it in a text file. Let us call it *myfun.asm*, from command prompt fire *nasm -f elf myfun.asm*. This will generate a file *myfun.o*. Then enter the C code above in a file called *mytest.c*, and fire *gcc -c mytest.c*. This will generate *mytest.o*. If you find no error in above 2 steps then perform *gcc myfun.o mytest.o -o mytest*. This will create *mytest* executable; execute it. If it prints **result is 11** then you are done.

For BORLAND C 32bit you have to put an underscore(_) before *addnum* in assembly. during assembly fire *nasm -f obj myfun.asm*, replace gcc with bcc and rest are same.

> The reverse, that is calling C functions from asesembly language is also possible. http://drpaulcarter.com/pcasm/

14.4 A small project

This projeect is all about detecting CPU. The code is split in to two parts, assembly and C. Follow the procedure described above to make this project.

gcpun.asm

```
;*************************************************************
;********* GCPUN.ASM NASM FILE *******
;*********COPYRIGHT(C)2003-2004(GPL) ASHOK SHANKAR DAS*******
;********* ashok_s_das@yahoo.com *******
;*************************************************************
[BITS 32]
global _is_486
global _is_386DX
global _is_fpu
global _is_cyrix
```

```
global _is_cpuid_supported
global _get_cpuid_info
global _cyrix_read_reg
global _cyrix_write_reg
segment .data
segment .bss
;_reg_eax resd 1h
;_reg_ebx resd 1h
;_reg_ecx resd 1h
;_reg_edx resd 1h
segment .text
;
;_is_486 if a 486 is present
;
_is_486:
pushf ; /* save EFLAGS */
pop eax ; /* get EFLAGS */
mov ecx,eax; /* temp storage EFLAGS */
xor eax,0x40000;" /* change AC bit in EFLAGS */
push eax ; /* put new EFLAGS value on stack */
popf ; /* replace current EFLAGS value */
pushf ; /* get EFLAGS */
pop eax ; /* save new EFLAGS in EAX */
cmp eax,ecx ; /* compare temp and new EFLAGS */
jz a ;
mov eax,1 ; /* 80486 present */
jmp b ;
a:
mov eax,0 ;" /* 80486 not present */
b:
push ecx ; /* get original EFLAGS */
popf ; /* restore EFLAGS */
ret;
;
;_is_386DX if a 386DX is present
;
_is_386DX:
mov edx,cr0 ;" /* Get CR0 */
push edx ;" /* save CR0 */
and dl,0xef;" /* clear bit4 */
mov cr0,edx ;" /* set CR0 */
mov edx,cr0;" /* and read CR0 */
and dl,0x10;" /* bit4 forced high? */
pop edx ;" /* restore reg w/ CR0 */
mov cr0,edx ;" /* restore CR0 */
mov eax,1;" /* TRUE, 386DX */
```

```
jz c ;
mov eax,0;" /* FALSE, 386SX */
c:
ret;
;
;_is_fpu checks if floating point unit(math co processor)is present
;
_is_fpu:
mov sp,bp;
fninit ;
mov ax,0x5a5a;"
push ax;
fnstsw [bp-2] ;"
pop ax
cmp ax,0;"
jne d ;" no fpu
mov eax,1;"
jmp e ;"
d:
mov eax,0;"
e:
ret
;
;cyrix_write_reg(char index, char val) writes to cyrix cpu
;
_cyrix_write_reg:
push ebp;
mov ebp,esp;
push eax;
push ebx;
mov eax,dword[ebp+8]
mov ebx,dword[ebp+12]
pushf; /* save flags */
cli; /* clear interrupt in flags */
out 0x22, eax; /* tell CPU which config. register */
mov eax,ebx;
out 0x23, eax; /* write to CPU config. register */
popf; /* restore flags */
pop ebx;
pop eax;
pop ebp;
ret
;
;_cyrix_read_reg(char index) reads from register
;
_cyrix_read_reg:
```

```
push ebp;
mov ebp,esp;
mov eax,dword[ebp+8];
pushf; /* save flags */
cli; /* clear interrupt in flags */
out 0x22,eax; /* tell CPU which config. register */
in eax,0x23; /* read CPU config, register */
popf; /* restore flags */
pop ebp;
ret
;
;is_cyrix checks if it is a cyrix cpu
;
_is_cyrix:
xor ax,ax ;" /* clear eax */
sahf ; /* clear flags, bit 1 is always 1 in flags */
mov ax,5;"
mov bx,2;"
div bl ;" /* do an operation that does not change flags */
lahf ; /* get flags */
cmp ah,2;" /* check for change in flags */
jne f;" /* flags changed not Cyrix */
mov eax,1;" /* TRUE Cyrix CPU */
jmp g;"
f:
mov eax,0;" /* FALSE NON-Cyrix CPU */
g:
ret
;
;_is_cpuid_supported returns 1 if we can execute cpuid
;
_is_cpuid_supported:
pushf ; /* get extended flags */
pop eax ;
mov ebx,eax ;" /* save current flags */
xor eax,0x200000;" /* toggle bit 21 */
push eax ;" /* put new flags on stack */
popf ;" /* flags updated now in flags */
pushf ;" /* get extended flags */
pop eax ;"
xor eax,ebx ;" /* if bit 21 r/w then supports cpuid */
jz e0;"
mov eax,1;"
jmp e1;"
e0:
mov eax,0;"
```

```
e1:
ret
;get_cpu_info(int cpuid_level, &ret_struct)
;the retstruct will contain information of the registers. it is defined in
; C source file
_get_cpuid_info:
push ebp ; save ebp
mov ebp,esp ; stack pointer to ebp
push esi ;save esi
mov eax,dword[ebp+8] ; get first param in eax FIRST PARAM is at esp+8
mov esi,dword[ebp+0ch] ; get the address of structure in esi which is
; at esp+12d
cpuid ; ; execute cpuid instruction
mov dword[esi+0h],eax ;store eax at esi which is retstruct.reg_eax
mov dword[esi+04h],ebx ;store ebx at esi+4 retstruct.reg_ebx
mov dword[esi+08h],ecx ;store ecx at retstruct.reg_ecx
mov dword[esi+0ch],edx ;store edx at retstruct.reg_edx
pop esi ;restore esi
pop ebp ;restore ebp
ret
```

The underscores infront of function names may create problem In that case please remove leading underscores.

gcpu.c

```
/*CPU IDENTIFICATION PROGRAM modified by Ashok Shankar Das
** 1- Moved assembly codes to separate asm file "gcpun.asm"
** 2- This file now is a pure C file can be compiled
** with any 32-bit c-compiler tested with gcc(djgpp-3.2.3)
**CPU ID routines for 386+ CPU's
**Written by Phil Frisbie, Jr. (pfrisbie@geocities.com)
**Parts adapted from the cpuid algorithm by Robert Collins(rcollins@x86.org)
**and from Cyrix sample code.
**See cpu.txt for more details on Intel and Cyrix codes.
*/
#include <string.h>
#ifndef TRUE
#define TRUE 1
#define FALSE 0
#endif
typedef struct ret_regs
                {
                    unsigned int reg_eax;
                    unsigned int reg_ebx;
                    unsigned int reg_ecx;
                    unsigned int reg_edx;
```

```
}cpu_inf;
char *unknown_vendor = "NoVendorName";
char *cyrix = "CyrixInstead";
char cpu_vendor[16]; /* Vendor String, or Unknown */
char dType[20]="";
int cpu_family=0; /* 3=386, 4=486, 5=Pentium, 6=PPro, 7=Pentium II?, etc */
int cpu_model=0; /* other details such as SX, DX, overdrive, etc. */
int cpu_stepping=0; /* stepping*/
int cpu_ext_family=0; /* extended family*/
int cpu_ext_model=0; /*extended model info*/
int cpu_fpu = FALSE; /* TRUE or FALSE */
int cpu_mmx = FALSE; /* TRUE or FALSE */
int cpu_cpuid = FALSE; /* Whether the cpu supported the cpuid instruction */
/* if TRUE, you can trust the information returned */
/* if FALSE, be careful... ;) */
int brand=0;
extern int _is_486(void); /* return TRUE for 486+, and FALSE for 386 */
extern int _is_386DX(void); /* return TRUE for 386DX, and FALSE for 386SX */
extern int _is_fpu(void); /*true for yes FPU */
extern int _is_cyrix(void); /*true for Yes*/
extern int _cyrix_read_reg(int);
extern void _cyrix_write_reg(int,int);
#define UNKNOWN 0xff
#define Cx486_pr 0xfd /* ID Register not supported, software created */
#define Cx486S_a 0xfe /* ID Register not supported, software created */
#define CR2_MASK 0x4 /* LockNW */
#define CR3_MASK 0x80 /* Resereved bit 7 */
void cyrix_type(void)
{
    char temp, orgc2, newc2, orgc3, newc3;
    int cr2_rw=FALSE, cr3_rw=FALSE, type;
    type = UNKNOWN;
    /* Test Cyrix c2 register read/writable */
    orgc2 = cyrix_read_reg(0xc2); /* get current c2 value */
    newc2 = orgc2 ^ CR2_MASK; /* toggle test bit */
    cyrix_write_reg(0xc2, newc2); /* write test value to c2 */
    cyrix_read_reg(0xc0); /* dummy read to change bus */
    if (cyrix_read_reg(0xc2) != orgc2) /* did test bit toggle */
    cr2_rw = TRUE; /* yes bit changed */
    cyrix_write_reg(0xc2, orgc2); /* return c2 to original value */
    /* end c2 read writeable test */
    /* Test Cyrix c3 register read/writable */
    orgc3 = cyrix_read_reg(0xc3); /* get current c3 value */
    newc3 = orgc3 ^ CR3_MASK; /* toggle test bit */
    cyrix_write_reg(0xc3, newc3); /* write test value to c3 */
    cyrix_read_reg(0xc0); /* dummy read to change bus */
```

```
        if (cyrix_read_reg (0xc3) != orgc3) /* did test bit change */
        cr3_rw = TRUE; /* yes it did */
        cyrix_write_reg (0xc3, orgc3); /* return c3 to original value */
        /* end c3 read writeable test */
        if ((cr2_rw && cr3_rw) || (!cr2_rw && cr3_rw)) /*DEV ID register ok */
        {
            /* <<<<<<< READ DEVICE ID Reg >>>>>>>> */
            type = cyrix_read_reg (0xfe); /* lower byte gets IDIR0 */
        }
        else if (cr2_rw && !cr3_rw) /* Cx486S A step */
        {
            type = Cx486S_a; /* lower byte */
        }
        else if (!cr2_rw && !cr3_rw) /* Pre ID Regs. Cx486SLC or DLC */
        {
            type = Cx486_pr; /* lower byte */
        }
        /* This could be broken down more, but is it needed? */
        if (type < 0x30 || type > 0xfc)
        {
            cpu_family = 4; /* 486 class-including 5x86 */
            cpu_model = 15; /* Unknown */
        }
        else if (type < 0x50)
            {
                cpu_family = 5; /* Pentium class-6x86 and Media GX */
                cpu_model = 15; /* Unknown */
            }
            else
              {
                cpu_family = 6; /* Pentium || class- 6x86MX */
                cpu_model = 15; /* Unknown */
                cpu_mmx = TRUE;
              }
        }
    extern int is_cpuid_supported(void); /* true if Supported*/
    extern void get_cpuid_info(int cpuid_levels,cpu_inf *t); /* This is so simple! */
    void check_cpu(void) /* This is the function to call to set the globals */
    {
        long cpuid_levels;
        long vendor_temp[3];
        cpu_inf t;
        memset(cpu_vendor, 0, 16);
        if (is_cpuid_supported ())
        {
            cpu_cpuid = TRUE;
```

```
        t.reg_eax = t.reg_ebx = t.reg_ecx = t.reg_edx = 0;
        get_cpuid_info(0,&t);
        cpuid_levels = t.reg_eax;
        vendor_temp[0] = t.reg_ebx;
        vendor_temp[1] = t.reg_edx;
        vendor_temp[2] = t.reg_ecx;
        memcpy(cpu_vendor, vendor_temp, 12);
        if (cpuid_levels > 0)
        {
            t.reg_eax = t.reg_ebx = t.reg_ecx = t.reg_edx = 0;
            get_cpuid_info (1,&t);
            cpu_family = ((t.reg_eax>>8) & 0xf);
            cpu_model = ((t.reg_eax>>4) & 0xf);
            cpu_stepping=(t.reg_eax & 0xf);
            cpu_ext_family=((t.reg_eax>>20) & 0xff);
            cpu_ext_model=((t.reg_eax>>16) & 0xf);
            switch(((t.reg_eax>>12)&0x7))
            {
                case 0:
                    strcpy(dType, "Original");
                    break;
                case 1:
                    strcpy(dType, "OverDrive");
                    break;
                case 2:
                    strcpy(dType, "Dual");
                    break;
            }
            brand=t.reg_ebx&0xff;
            cpu_fpu = (t.reg_edx & 1 ? TRUE : FALSE);
            cpu_mmx = (t.reg_edx & 0x800000 ? TRUE: FALSE);
        }
    }
    else
    {
        memcpy(cpu_vendor, unknown_vendor, 12);
        cpu_fpu = is_fpu();
        if (!is_486())
        {
            if (is_386DX()) /* It is a 386DX */
            {
                cpu_family = 3; /* 386 */
                cpu_model = 0; /* DX */
            }
            else /* It is a 386SX */
            {
```

```
                    cpu_family = 3; /* 386 */
                    cpu_model = 1; /* SX */
                }
            }
            else /* It is a 486+ */
            {
                if(is_cyrix())
                {
                    memcpy(cpu_vendor, cyrix, 12);
                    cyrix_type();
                }
                else
                {
                    cpu_family = 4; /* 486 */
                    cpu_model = 15; /* unknown */
                }
            }
        }
}
int main(void) /* Sample program */
{
    check_cpu();
    printf("CPU has cpuid instruction? %s\n", cpu_cpuid ? "yes": "no");
    printf("CPU vender is %s\n", cpu_vendor);
    printf("CPU has fpu? %s\n", cpu_fpu ? "yes": "no");
    printf("CPU has mmx? %s\n", cpu_mmx ? "yes": "no");
    printf("%s\n", dType);
   printf("Family %i, Model %i, Stepping %i\n", cpu_family, cpu_model, cpu_stepping);
    printf("Extended Family %i, Extended Model %i\n", cpu_ext_family, cpu_ext_model);
    printf("CPU brand %d\n",brand);
    return 1;
}
```

14.5 Summing Up

In this chapter we saw

- calling conventions and how they are implemented in compilers.
- Next we saw how to create a library for using in C projects.
- how to interface assembly language with C
- A small project using assembly language and C.

The project described here is created after doing research on several example CPU detection codes. Those codes are available on Internet in plenty.