KARNATAKA STATE OPEN UNIVERSITY

MANASAGANGOTRI, MYSORE- 570 006

DEPERTAMENT OF STUDIES IN INFORMATION TECHNOLOGY

M.SC IN INFORMATION TECHNOLOGY

I SEMESTER

PROGRAMMING CONCEPTS AND C

(MSIT-102)

MSIT – 102

PROGRAMMING CONCEPTS AND C

Programming Concepts and C

Module	Unit No.	Page No.
1	Unit - 1	7 – 17
	Unit - 2	18 – 25
	Unit - 3	26 – 36
	Unit - 4	37 – 56
2	Unit - 5	57 – 70
	Unit - 6	71 – 86
	Unit - 7	87 – 97
	Unit - 8	98 – 109
3	Unit - 9	110 – 119
	Unit - 10	120 – 134
	Unit - 11	135 – 145
	Unit - 12	146 – 154
4	Unit - 13	155 – 174
	Unit - 14	175 – 195
	Unit - 15	196 – 207
	Unit - 16	208 - 217

Preface

The objective of ***Programming Concepts and C*** is to understand and effective use of the C programming language syntax to develop special programs, and provides I/O control for special applications. The students are expected use both the integrated environment development system and command line compilation. The ultimate goal of the Program Concept and C is make the reader clear, readable, and possibly even authorative in analyzing program concepts and improve their skills in programming with C language.

We introduce basic knowledge as well advanced programming concepts and C language in perceptive of information technology. Programming Concepts and C concerns with certain issues. Programming concepts concerns with visualizing real time problem in modularizing pattern and simulating them to work on the system. C language deals with programming models program concept to work on the machine with their defined syntax and semantics. Hence this course offers students in understanding their skills to maximum extent in both designing model and implanting them on PC.

This concise study material is an introduction to Programming Concept and C supports that makes foundation for problem solving model designing and foundation to master their programming skills in C language. Study material presents real time problem examples and rich set of C programming language tool and techniques.

Organization of the material: Whole material is break into four modules, a module consist four units each. The organization of book is of two fold. It means that when a problem is modeled as a programming concept, in subsequent section we talk about the tool and technique to solve them on machine through C language. So students will get good grip in better understand in concepts.

In the very first module, we begin with introduction to problem solving, introduction to C language, C standards and its characteristics. We give some useful information about current trend and details about advanced compilers. It has been neatly given for better understanding for students to set up compiler on their own. In next stage we have given all basic requirements which are very much necessary for further proceed.

In the second module, we move on to little bit deeper in program concept and C covering, Operators, expression and control flow statements.

In the third module, we discuss more and more advance and very important topics such as functions, arrays, string and pointer at this stage.

In the Fourth and final module, we have put maximum effort in creating real time example to deal with higher level concepts such as pointer and structure. A suitable example and codes are given in their respective sections.

All the best for the students.

Dr. H.K. Chethan & Mr. Vinay .K

Course Design and Editorial Committee

Prof. K. S. Rangappa Vice Chancellor & Chairperson Karnataka State Open University Manasagangotri, Mysore – 570 006	**Prof. Vikram Raj Urs** Dean (Academic) & Convener Karnataka State Open University Manasagangotri, Mysore – 570 006
Head of the Department – Incharge	**Course Co-Ordinator**
Prof. Kamalesh DIRECTOR IT&Technology Karnataka State Open University Manasagangotri, Mysore – 570 006	**Mr. Mahesha DM** Lecturer, DOS in Information Technology Karnataka State Open University Manasagangotri, Mysore – 570 006

Course Writers

Dr. H.K. Chethan Lecturer DOS in Computer Science University of Mysore Manasagangotri, Mysore – 570 006	**Modules 1 and 3**	**Units 1-4 and 9-12**
And		
Mr. Vinay .K Assistant Professor DOS in Computer Science University of Mysore Manasagangotri, Mysore – 570 006	**Modules 2 and 4**	**Units 5-8 and 13-16**

Publisher

Registrar
Karnataka State Open University
Manasagangotri, Mysore – 570 006

Developed by Academic Section, KSOU, Mysore
Karnataka State Open University, 2012

All rights reserved. No part of this work may be reproduced in any form, by mimeograph or any other means, without permission in writing from the Karnataka State Open University.

Further information on the Karnataka State Open University Programmes may be obtained from the University's Office at Manasagangotri, Mysore – 6.

Printed and Published on behalf of Karnataka State Open University, Mysore-6 by the **Registrar (Administration)**

Programming Concepts and C

Module - 1

Unit - 1

Structure

1.0 Objectives

1.1 Introduction

1.2 C language standards

1.3 C language features

1.4 Program Concepts and Characteristics

1.5 Structure of C Program

1.6 Summary

1.7 Key words

1.8 Questions

1.9 References

1.0 Objectives

At the end of this unit you will be able to

- Understand the evolution of C programming language
- List out the features and characteristics of C programming language
- Explain the structure of C program

1.1 Introduction to C

C was developed by Dennis Ritchie at Bell Laboratories in 1972. Most of its principles and ideas were taken from the earlier language B, BCPL and CPL. CPL was developed jointly between the Mathematical Laboratory at the University of Cambridge and the University of London Computer Unit in 1960s. CPL (Combined Programming Language) was developed with the purpose of creating a language that was capable of both machine independent programming and

would allow the programmer to control the behavior of individual bits of information. But the CPL was too large for use in many applications. In 1967, BCPL (Basic Combined Programming Language) was created as a scaled down version of CPL while still retaining its basic features. This process was continued by Ken Thompson. He made B Language during working at Bell Labs. B Language was a scaled down version of BCPL. B Language was written for the systems programming. In 1972, a co-worker of Ken Thompson, Dennis Ritchie developed C Language by taking some of the generality found in BCPL to the B language.

The original PDP-11 version of the Unix system was developed in assembly language. In 1973, C language had become powerful enough that most of the Unix kernel was rewritten in C. This was one of the first operating system kernels implemented in a language other than assembly.

During the rest of the 1970's, C spread throughout many colleges and universities because of its close ties to UNIX and the availability of C compilers. Soon, many different organizations began using their own versions of C Language. This was causing great compatibility problems. In 1983, the American National Standards Institute (ANSI) formed a committee to establish a standard definition of C Language. That is known as ANSI Standard C. Today C is the most widely used System Programming Language.

1.2 C language standards

With the introduction of new devices and extended character sets, new features may be added to this International Standard. Subclauses in the language and library clauses warn implementors and programmers of usages which, though valid in themselves, may conflict with future additions.

Certain features are obsolescent, which means that they may be considered for withdrawal in future revisions of this International Standard. They are retained because of their widespread use, but their use in new implementations (for implementation features) or new programs.

This International Standard is divided into four major subdivisions:

- preliminary elements
- the characteristics of environments that translate and execute C programs
- the language syntax, constraints, and semantics
- the library facilities

Examples are provided to illustrate possible forms of the constructions described. Footnotes are provided to emphasize consequences of the rules described in that subclause or elsewhere in this International Standard. References are used to refer to other related subclauses. Recommendations are provided to give advice or guidance to implementors. Annexes provide additional information and summarize the information contained in this International Standard. A bibliography lists documents that were referred to during the preparation of the standard.

The library clause is based on the 1984 /usr/group Standard.

1.3 C language Features

Characteristics of ‘c’ language/main features:

1. Modularity.
2. Portability.
3. Extendibility.
4. Speed.
5. Flexibility.

Modularity: Ability to breakdown a large module into manageable sub modules called as modularity, which is an important feature of structured programming languages.

Advantages:

- Projects can be completed in time.
- Debugging will be easier and faster.

Portability:

The ability to port i.e. to install the software in different platform is called portability.

Highest degree of portability: 'C' language offers highest degree of portability i.e., percentage of changes to be made to the sources code is at minimum when the software is to be loaded in another platform. Percentage of changes to the source code is minimum. The software that is 100% portable is also called as platform independent software or architecture neutral software. Eg: Java.

Extendibility: Ability to extend the existing software by adding new features is called as extendibility.

SPEED:

'C' is also called as middle level language because programs written in 'c' language run at the speeds matching to that of the same programs written in assembly language so 'c' language has both the merits of high level and middle level language and because if this feature it is mainly used in developing system software.

Flexibility: Key words or reverse words

ANSIC has 32 reverse words

'C' language has right number of reverse words which allows the programmers to have complete control on the language.

'C' is also called as programmer's language since it allows programmers to induce creativeness into the programmers.

1.4 Program Concepts and Characteristics

1. Program startup

The function called at program startup is named main. The implementation declares no prototype for this function. It shall be defined with a return type of int and with no parameters:

int main(void) { / ... */ }*

or with two parameters (referred to here as argc and argv, though any names may be used, as they are local to the function in which they are declared):

int main(int argc, char *argv[]) { /* ... */ }

or in some other implementation-defined manner.

If they are declared, the parameters to the main function shall obey the following constraints:

- The value of argc shall be nonnegative.
- argv[argc] shall be a null pointer.
- If the value of argc is greater than zero, the array members argv[0] through

argv[argc-1] inclusive shall contain pointers to strings, which are given implementation-defined values by the host environment prior to program startup. The intent is to supply to the program information determined prior to program startup from elsewhere in the hosted environment. If the host environment is not capable of supplying strings with letters in both uppercase and lowercase, the implementation shall ensure that the strings are received in lowercase.

If the value of **argc** is greater than zero, the string pointed to by **argv[0]** represents the *program name*; **argv[0][0]** shall be the null character if the program name is not available from the host environment. If the value of **argc** is greater than one, the strings pointed to by **argv[1]** through **argv[argc-1]** represent the *program parameters*.

2. Program execution

In a hosted environment, a program may use all the functions, macros, type definitions, and objects described in the library clause.

3. Program termination

If the return type of the **main** function is a type compatible with **int**, a return from the initial call to the **main** function is equivalent to calling the **exit** function with the value returned by the **main** function as its argument) reaching the **}** that terminates the **main** function returns a value of 0. If the return type is not compatible with **int**, the termination status returned to the host environment is unspecified.

4. Program execution

(i) The semantic descriptions in this International Standard describe the behavior of an abstract machine in which issues of optimization are irrelevant.

(ii) Accessing a volatile object, modifying an object, modifying a file, or calling a function that does any of those operations are all *side effects*,11) which are changes in the state of the execution environment. Evaluation of an expression may produce side effects. At certain specified points in the execution sequence called *sequence points*, all side effects of previous evaluations shall be complete and no side effects of subsequent evaluations shall have taken place.

(iii) In the abstract machine, all expressions are evaluated as specified by the semantics. An actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no needed side effects are produced (including any caused by calling a function or accessing a volatile object).

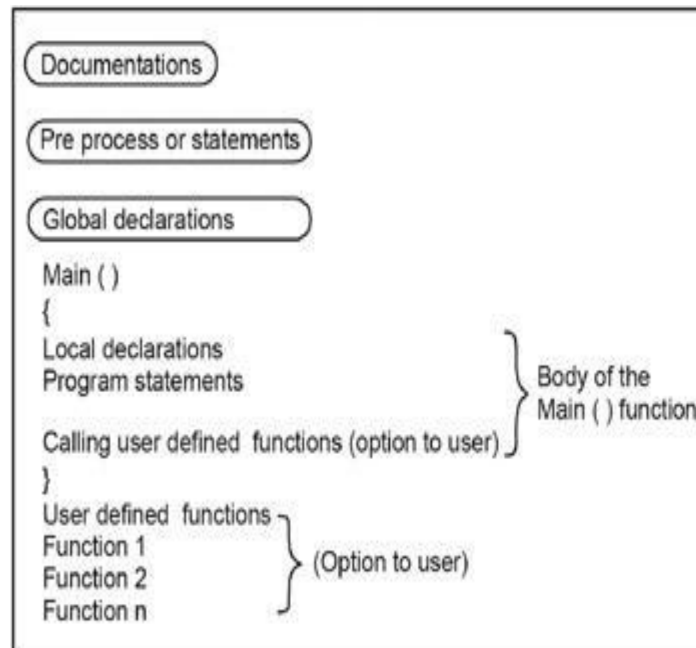
(iv) When the processing of the abstract machine is interrupted by receipt of a signal, only the values of objects as of the previous sequence point may be relied on. Objects that may be modified between the previous sequence point and the next sequence point need not have received their correct values yet.

(v) The least requirements on a conforming implementation are:

(vi) At sequence points, volatile objects are stable in the sense that previous accesses are complete and subsequent accesses have not yet occurred.

1.5 Structure of C Program

C language is very popular language among all the languages. The structure of a C program is a protocol (rules) to the programmer, while writing a C program. The general basic structure of C program is shown in the figure below. The whole program is controlled within main ( ) along with left brace denoted by "{" and right braces denoted by "}". If you need to declare local variables and executable program structures are enclosed within "{" and "}" is called the body of the main function. The main ( ) function can be preceded by documentation, preprocessor statements and global declarations.



Documentations

The documentation section consist of a set of comment lines giving the name of the program, the another name and other details, which the programmer would like to use later.

Preprocessor Statements

The preprocessor statement begin with # symbol and are also called the preprocessor directive. These statements instruct the compiler to include C preprocessors such as header files and

symbolic constants before compiling the C program. Some of the preprocessor statements are listed below.

```
# include <stdio.h>   )
# include <math.h>    } header files
# include <stdlib.h>  }
# include <CONIO.h>   )

# define P L 3.1412.  )
# define TRVE 1       } Symbolic constants
# define FALSE Ø      )
```

Global Declarations

The variables are declared before the main ( ) function as well as user defined functions are called global variables. These global variables can be accessed by all the user defined functions including main ( ) function.

The main ( ) function

Each and Every C program should contain only one main ( ). The C program execution starts with main ( ) function. No C program is executed without the main function. The main ( ) function should be written in small (lowercase) letters and it should not be terminated by semicolon. Main ( ) executes user defined program statements, library functions and user defined functions and all these statements should be enclosed within left and right braces.

Braces

Every C program should have a pair of curly braces ({, }). The left braces indicates the beginning of the main ( ) function and the right braces indicates the end of the main ( ) function. These braces can also be used to indicate the user-defined functions beginning and ending. These two braces can also be used in compound statements.

Local Declarations

The variable declaration is a part of C program and all the variables are used in main ( ) function should be declared in the local declaration section is called local variables. Not only variables, we can also declare arrays, functions, pointers etc. These variables can also be initialized with basic data types.

For Example

```
Main ( )
{
int sum = 0;
int x;
float y;
}
```

Here, the variable sum is declared as integer variable and it is initialized to zero. Other variables declared as int and float and these variables inside any function are called local variables.

Program statements

These statements are building blocks of a program. They represent instructions to the computer to perform a specific task (operations). An instruction may contain an input-output statements, arithmetic statements, control statements, simple assignment statements and any other statements and it also includes comments that are enclosed within /* and */ . The comment statements are not compiled and executed and each executable statement should be terminated with semicolon.

User defined functions

These are subprograms, generally, a subprogram is a function and these functions are written by the user are called user ; defined functions. These functions are performed by user specific tasks and this also contains set of program statements. They may be written before or after a main () function and called within main () function. This is an optional to the programmer.

First Program using C

Here is your first c program. Write carefully because C Language is a case sensitive language.

```
#include <stdio.h>
void main()
{
printf("Hello World\n Welcome to C Programming");
}
```

Press ALT+F9 to compile your program. If you have any error in your program, you will get the message, remove your errors and then execute your program you will got the output.

```
Hello World
Welcome to C Programming
```

Some other sample C programs

/*Display Current date and time*/	**/* addition of 2 numbers */**

#include <stdio.h>
#include <time.h>
 	
void main()
{
time_t t;
time(&t);
clrscr();
printf("Today's date and time : s",ctime(&t));
getch();
}
	#include<stdio.h>
	#include<conio.h>
	void main()
	{
	int a, b,c;
	c=0;
	a=3;
	b=4;
	c = a + b:
	printf("result of addition = %d",c);
	}

1.6 Summary

At the end this unit you are able to understand the history of C programming language and how it evolved as a powerful programming language. In this unit we have covered the topics such as C language standards, features and characteristics. It has been neatly covered with program concepts and different stages which involve in developing a program. At the end of this a simple C program has been illustrated to explain the structure of the simple C program.

1.7 Keywords

C programming language, program execution, program termination, structure of program

1.8 Questions

1. List out the characteristics of C program concepts?
2. With an example explain the structure of C program?

1.9 Reference

Programming In ANSI C by E Balagurusamy

The C Programming Language (Ansi C Version) by Brian W. Kernighan, Dennis M. Ritchie

Expert C Programming: Deep C Secrets by Peter Van, Der Linden

Unit - 2

Structure

2.0 Objectives

2.1 C Programming/Compiling

2.2 List of Compilers

2.3 IDE features of Turbo C compiler

2.4 Compiling C program

2.5 Summary

2.6 Key words

2.7 Questions

2.8 References

2.0 Objectives

At the end of this unit you will be able to

- Understand the C compilers
- Understand the creating and compiling C programs
- Explain the IDE features of C compilers
- Elucidate the common line options to compile C program

2.1 C Programming/Compiling

Like any programming language, C by itself is completely incomprehensible to a microprocessor. Its purpose is to provide an intuitive way for humans to provide instructions that can be easily converted into machine code that is comprehensible to a microprocessor. The **compiler** is what takes this code, and translates it into the machine code.

To those new to programming, this seems fairly simple. A naive compiler might read in every source file, translate everything into machine code, and write out an executable. This could work, but has two serious problems. First, for a large project, the computer may not have enough memory to read all of the source code at once. Second, if you make a change to a single source file, you would rather not have to recompile the entire application.

To deal with these problems, compilers break their job down into steps; for each source file (each .c file), the compiler reads the file, reads the files it references with #include, and translates it to machine code. The result of this is an "object file" (.o). Once every object file is made, a "linker" collects all of the object files and writes the actual program. This way, if you change one source file, only that file needs to be recompiled and then the application needs to be re-linked.

Without going into the painful details, it can be beneficial to have a superficial understanding of the compilation process.

Preprocessor

The preprocessor provides the ability for the inclusion of header files, macro expansions, conditional compilation, and line control. Many times you will need to give special instructions to your compiler. This is done by inserting preprocessor directives into your code. When you begin compiling your code, a special program called the preprocessor scans the source code and performs simple substitution of tokenized strings for others according to predefined rules. The preprocessor is not a part of the C language.

In C language, all preprocessor directives begin with the pound character (#). You can see one preprocessor directive in the Hello world program introduced in A taste of C:

Example:

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

This directive causes the header to be included into your program. Other directives such as #pragma control compiler settings and macros. The result of the preprocessing stage is a text string. You can think of the preprocessor as a non-interactive text editor that prepares your code

for the compilation step. The language of preprocessor directives is agnostic to the grammar of C, so the C preprocessor can also be used independently to process other kinds of text files.

Syntax Checking

This step ensures that the code is valid and will sequence into an executable program. Under most compilers, you may get messages or warnings indicating potential issues with your program (such as a statement always being true or false, etc.)

When an error is detected in the program, the compiler will normally report the file name and line that is preventing compilation.

Object Code

The compiler produces a machine code equivalent of the source code that can then be linked into the final program. The code itself can't be executed yet, as it has to complete the linking stage.

It's important to note after discussing the basics that compilation is a "one way street". That is, compiling a C source file into machine code is easy, but "decompiling" (turning machine code into the C source that creates it) is not. Decompilers for C do exist, but they rarely create useful code.

Linking

Linking combines the separate object codes into one complete program by integrating libraries and the code and producing either an executable program or a library. Linking is performed by a linker, which is often part of a compiler.

Common errors during this stage are either missing functions, or duplicate functions.

Automation

For large C projects, many programmers choose to automate compilation, both in order to reduce user interaction requirements and to speed up the process by only recompiling modified files.

Most integrated development environments have some kind of project management, which makes such automation very easy. On UNIX-like systems, make and Makefiles are often used to accomplish the same.

2.2 List of Free compilers available

- GNU C Compiler: The most famous and widely used at present. Primarly for Unix and Unix-like platforms.
- DJGPP: Port of the GNU development utilties to the Intel 32-bit platforms.
- Cygwin: This is an environment and a port of most GNU utilties including the GCC set to the Windows platform.
- MinGW: Minimalist GNU for Windows. Includes support for the GNU Compiler Collection (GCC).
- Turbo C: Remember Turbo C from Borland? Now it is available as a free download. Though it's a little out of date, it still is extremely usable.
- Borland C++ Builder: The compiler and command line tools are available as a free download.
- OpenWatcom Compilers: The original Watcom compilers are now available as open source software.
- Digital Mars C/C++ Compiler: Offshoot of the Zortech/Symantec C/C++ Compilers (probably it is the Zortech compiler re-incarnated). Touted as drop in replacements for the Symantec C/C++ compilers.
- lcc: Portable compiler for ANSI C.
- lcc-win32: Free for non-commerical use. Windows support with IDE, etc.

- Pelles C: Based on lcc. With IDE, etc for Windows platform.
- Ch: This is actually an interpreter. Standard edition is free.
- EiC: Extensible Interactive C. Another interpreter.

2.3 IDE features of Turbo C compiler

Turbo C is an Integrated Development Environment and compiler for the C programming language from Borland. First introduced in 1987, it was noted for its integrated development environment, small size, fast compile speed, comprehensive manuals and low price.

In May 1990, Borland replaced Turbo C with Turbo C++. In 2006, Borland reintroduced the Turbo moniker.

History

Turbo C had the same properties as Turbo Pascal: an integrated development environment (IDE), a fast compiler, a good editor and a competitive price. Turbo C was not as successful as the Pascal-sister product. First, C was a language for professional programming and systems development rather than a school language. Turbo C competed with other professional programming tools (Microsoft C, Lattice C, Watcom C, etc.).

Versions

Version 2.0, Released on May 13, 1987, offered the first integrated edit-compile-run development environment for C on IBM PCs. The software was, like many Borland products of the time, bought from another company and branded with the "Turbo" name, in this case **Wizard C** by Bob Jervis[1] (Borland's flagship product at that time, Turbo Pascal, which at this time did not have pull-down menus, would be given a facelift with version 4 released late in 1987 to make it look more like Turbo C.) It ran in 384 kB of memory. It allowed inline assembly with full access to C symbolic names and structures, supported all memory models, and offered optimizations for speed, size, constant folding, and jump elimination.[2]

Turbo C 2.5 startup screen.

Version 2.5, in January 1988 was an incremental improvement over version 2.0. It included more sample programs, improved manuals and bug fixes. It was shipped on five 360 KB diskettes of uncompressed files, and came with sample C programs, including a stripped down spreadsheet called mcalc. This version introduced the <conio.h> header file (which provided fast, PC-specific console I/O routines).

Turbo C 2.0 startup screen.

Version 2.0, in 1989 was released was in late 1988, and featured the first "blue screen" version, which would be typical of all future Borland releases for MS-DOS. The American release did not have Turbo Assembler or a separate debugger. (These were sold separately as Turbo Assembler.) Turbo C, Asm, and Debugger were sold together as a suite. This seems to describe another release: FeaturedTurbo Debugger, Turbo Assembler, and an extensive graphics library. This version of Turbo C was also released for the Atari ST, but distributed in Germany only.

2.4 Compiling C Program

There are two types of C compilers, each of which has advantages and disadvantages:

(i) Command-line C compilers and

(ii) IDE or Windows C compilers

To compile and run a C program using a **command-line** C compiler, you have to go through the following steps:

1. **Write the C program** (call it ``myfile.c'') in a text editor or word processor (for example, the simple ``Hello'' program below)
2. **Save it as a file** on your computer's hard disk,
3. **``Compile it'' to a computer-executable program** by entering a compile command

at a command prompt, for example for the following C compiler programs:

gcc -Wall -o myfile myfile.c (on gcc compiler)

(using the GNU C compiler, UNIX or Microsoft Windows)

cl myfile.c

(Microsoft Visual C++ command-line compiler)

bcc32 myfile.c

(Borland C/C++ compiler, Microsoft Windows)

followed by the ``Enter'' key, and finally

4. **Run the program** by entering

myfile

at a command prompt, again followed by ``Enter''. If you want to save the output of ``myfile'' as a text file ``myfile.txt'', enter instead

myfile > myfile.txt

Example code:

myfile.c

```
#include <stdio.h>


int main()
```

```
        {
         printf("Hello world\n");
         return 0;
}
```

2.5 Summary

In this unit we gave discussed about the various compiling options in executing C programs viz., IDE features and Common line options. It has been list out the various freely available C compilers along the features. At the end of this unit you will be able understand the compilation of C programs.

2.6 Keywords

Compilers, Turbo C, gcc, IDE

2.7 Questions

1. Explain the different compile options to execute C program?
2. List of the IDE feature available under various compilers for C programs?

2.8 Reference

Programming In ANSI C by E Balagurusamy

The C Programming Language (Ansi C Version) by Brian W. Kernighan, Dennis M. Ritchie

Expert C Programming: Deep C Secrets by Peter Van, Der Linden

Unit - 3

Structure

3.0 Objectives

3.1 Introduction

3.2 Identifiers

3.3 Keywords

3.4 Variable

3.5 Constants

3.6 Scope and Life of Variable

3.7 Data type and Size

3.8 Declaring, Initialization and Assigning

3.9 Symbolic Constants

3.10 Summary

3.11 Key words

3.12 Questions

3.13 References

3.0 Objectives

At the end of this unit you will be able to

- Understand the C compilers
- Understand the creating and compiling C programs
- Explain the IDE features of C compilers
- Elucidate the common line options to compile C program

3.1 Introduction

IDENTIFIERS AND KEYWORDS

Every word in C language is a keyword or an identifier/variable. Keywords in C language cannot be used as a variable name. They are specifically used by the compiler for its own purpose and they serve as building blocks of a c program.

3.2 Identifiers

- Identifiers are the names given to variables, classes, methods and interfaces.
- It must be a whole word and starts with either an alphabet or an underscore.
- They are case sensitive.
- No commas or blanks are allowed in it
- No special symbol other than an underscore can be used in it.

Ex.: marks gracesem1_rollno alpha

3.3 Keywords

- Keywords are words that have special meaning to the C compiler.
- An identifier cannot have the same spelling and case as a C keyword.
- C makes use of only 32 keywords or reserved words which combine withthe formal syntax to the form the C programming language.
- All keywords in C are written in lower case.
- A keyword may not be used as a variable name.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef

char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

3.4 Variables

A variable is value that can change any time. It is a memory location used to store a data value. A variable name should be carefully chosen by the programmer so that its use is reflected in a useful way in the entire program. Variable names are case sensitive.

Examples of variable names are

sum, number, salary, emp_name, average1

Types of variables

integer, char, string etc. Each type of variable has their own size and range. For ex. Range of integer is -32768 to 32767 and the range for it is 2, 4 bytes (1 byte = 8 bits).

The size of a char is 1 byte. And range is from 1 to 255.

Initialization of Variable

Initialize a variable in c to assign it a starting value. Without this we can't get whatever happened to memory at that moment.

C does not initialize variables automatically. So if you do not initialize them properly, you can get unexpected results. Fortunately, C makes it easy to initialize variables when you declare them.

For Example:

```
int x=45;
int month_lengths[] = {23,34,43,56,32,12,24};
struct role = { "Hamlet", 7, FALSE, "Prince of Denmark ", "Kenneth Branagh"};
```

Note: The initialization of variable is a good process in programming.

3.5 Constants

A constant value is the one which does not change during the execution of a program. C supports several types of **constants**.

1. Integer Constants
2. Real Constants
3. Single Character Constants
4. String Constants

Integer Constants

An integer constant is a sequence of digits. There are 3 types of integers namely decimal integer, octal integers and hexadecimal integer.

Decimal Integers consists of a set of digits 0 to 9 preceded by an optional + or - sign. Spaces, commas and non digit characters are not permitted between digits.

Example for valid decimal integer constants are

123, -3, 0, 562321, + 78

Some examples for invalid integer constants are

15 750, 20,000, Rs. 1000

Octal Integers constant consists of any combination of digits from 0 through 7 with a O at the beginning. Some examples of octal integers are

O26, O, O347, O676

Hexadecimal integer constant is preceded by OX or Ox, they may contain alphabets from A to F or a to f. The alphabets A to F refers to 10 to 15 in decimal digits.

Example of valid hexadecimal integers are

OX2, OX8C, OXbcd, Ox

1. Real Constants

Real Constants consists of a fractional part in their representation. Integer constants are inadequate to represent quantities that vary continuously. These quantities are represented by numbers containing fractional parts like 26.083.

Example of real constants are

0.0026, -0.97, 435.29, +487.0

Real Numbers can also be represented by exponential notation. The general form for exponential notation is mantissa exponent. The mantissa is either a real number expressed in decimal notation or an integer. The exponent is an integer number with an optional plus or minus sign.

Single Character Constants

A Single Character constant represent a single character which is enclosed in a pair of quotation symbols.

Example for character constants are

8

'5', 'x', ';', ' '

All character constants have an equivalent integer value which is called ASCII Values.

String Constants

String constant is a set of characters enclosed in double quotation marks. The characters in a string constant sequence may be an alphabet, number, special character and blank space. Example of string constants are

"VISHAL", "1234", "God Bless", "!.....?"

3.6 Scope and Life Span Of a Variable

The area of the program where that variable is valid, the amount of time that a variable is retained, as well as where it can be accessed from, depends on its specified location and type. The life span of the variable, i.e. the length of time that the variable remains in memory. The programmer has to be very much aware of the type of the variable he/she going to use. Each and every sort of scenario requires different type of scope. For example suppose the variable which should remain constant and also required by external functions then it is not good practice to declare same variable again and again instead of that it may be called as a global variable.

On the other hand, for the sake security reason some variable has be visible in only one function can be treated and declared as static variable.

This section of the unit will be discusses thoroughly in forth coming units when we introduce storage class and different types of variables.

3.7 Data types and Sizes

The basic data structure used in C programs is a number. C provides for two types of numbers - integers and floating point. Computer operations that involve text (characters, words or strings) still manipulate numbers. Each data item used by a program must have a data type. The basic data types provided by C are shown in Table 1. Each data type represents a different kind of number. You must choose an appropriate type for the kind of number you need in the variable declaration section of the program.

- In c, there are three types of data
- In C language the system has to know before-hand, the type of numbers or characters being used in the program. These are called data types. There are many data types in C language.
- A C programmer has to use appropriate data type as per his requirement.

C language data types can be broadly classified as

a) Primary data type
b) Derived data type
c) User-defined data type

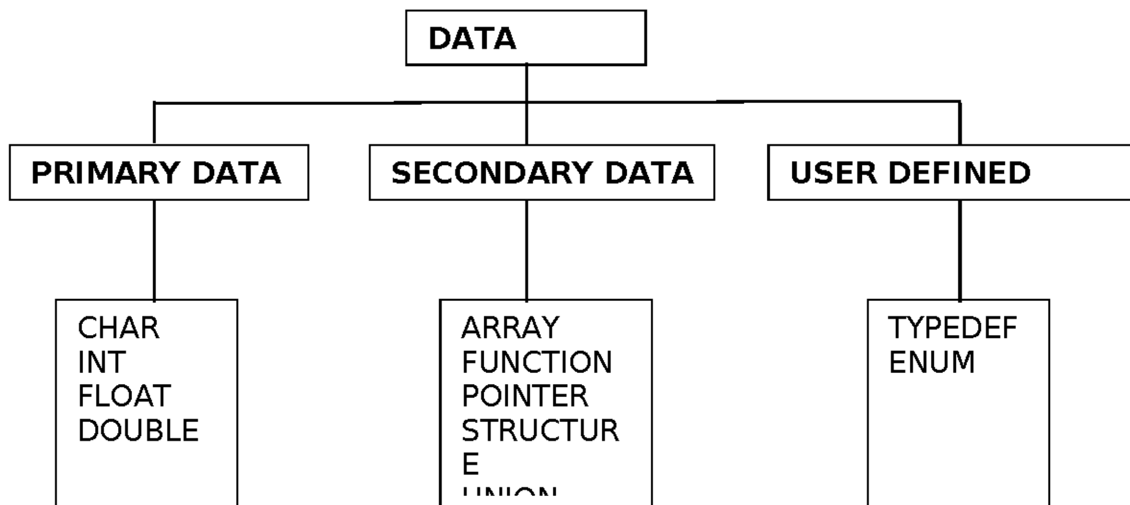


Fig: Data Types in C

Integer Data Type

- Integers are numbers without decimal point
- Integers are whole numbers with a range of values, range of values are machine dependent.
- Generally an integer occupies 2 bytes memory space and its value range limited to -32768 to +32767 (that is, -215 to +215-1).
- A signed integer use one bit for storing sign and rest 15 bits for number.
- To control the range of numbers and storage space, C has three classes of integer storage namely short int, int and long int.
- All three data types have signed and unsigned forms.
- A short int requires half the amount of storage than normal integer.
- Unlike signed integer, unsigned integers are always positive and use all the bits for the magnitude of the number.
- Therefore, the range of an unsigned integer will be from 0 to 65535.
- The long integers are used to declare a longer range of values and it occupies 4 bytes of storage space.

Syntax:

int <variable name>;

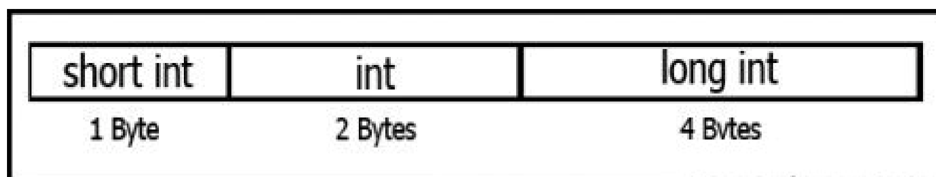
int num1;

short int num2;

long int num3;

Example: 5, 6, 100, 2500.

Integer Data Type Memory Allocation:



Floating Point Data Type

- The float data type is used to store fractional numbers (real numbers) with 6 digits of precision.
- Floating point numbers are denoted by the keyword float. When the accuracy of the floating point number is insufficient, we can use the double to define the number.
- The double is same as float but with longer precision and takes double space (8 bytes) than float.
- To extend the precision further we can use long double which occupies 10 bytes of memory space.

Syntax:

float <variable name>;

float num1;

double num2;

long double num3;

Example: 9.125, 3.1254.

Floating Point Data Type Memory Allocation:

float	double	long double
4 Bytes	8 Bytes	10 Bytes

Character Data Type

- Character type variable can hold a single character and are declared by using the keyword char.
- As there are singed and unsigned int (either short or long), in the same way there are signed and unsigned chars; both occupy 1 byte each, but having different ranges.
- Unsigned characters have values between 0 and 255, signed characters have values from –128 to 127.

  Syntax:

  char <variable name>;

char ch = ‘a’;

Example: a, b, g, S, j.

3.8 Declaring, Initializing, and Assigning Variables

Here is an example of declaring an integer, which we've called some_number. (Note the semicolon at the end of the line; that is how your compiler separates one program statement from another.)

int some_number;

This statement means we're declaring some space for a variable called some_number, which will be used to store integer data. Note that we must specify the type of data that a variable will store. There are specific keywords to do this – we'll look at them in the next section.

Multiple variables can be declared with one statement, like this:

int anumber, anothernumber, yetanothernumber;

We can also declare and assign some content to a variable at the same time.

int some_number=3;

This is called initialization.

After declaring variables, you can assign a value to a variable later on using a statement like this:

some_number=3;

You can also assign a variable the value of another variable, like so:

anumber = anothernumber;

Or assign multiple variables the same value with one statement:

anumber = anothernumber = yetanothernumber = 3;

This is because the assignment x = y returns the value of the assignment. x = y = z is really shorthand for x = (y = z).

3.9 Symbolic Constants

- C allows the definition of symbolic constants - names that will be replaced with their values when the program is compiled
- Symbolic constants are defined before main(), and the syntax is

#define *NAME value*

Example:

```
#define ANGLE_MIN 0
#define ANGLE_MAX 360
```

The above piece of code would define ANGLE_MIN and ANGLE_MAX to the values 0 and 360, respectively.

- C distinguishes between lowercase and uppercase letters in variable names.
- It is customary to use capital letters in defining global constants.

3.10 Summary

At the end of this unit we have discussed about the basic concepts of C programming viz., c tokens: Identifiers, Keywords, Variables and symbolic constants. We have also introduced the life and scope of variable and the same concepts will be discusses in detail in forth coming module.

3.11 Keywords

Identifier, Variable, Keyword

3.12 Questions

1. Explain the terms variable, keyword, constant and symbolic constant which are supported in C programming?
2. Briefly describe the life and scope of the variable in C program?
3. Discuss about the data type supported in C language?

3.13 Reference

Programming In ANSI C by E Balagurusamy

The C Programming Language (Ansi C Version) by Brian W. Kernighan, Dennis M. Ritchie

Expert C Programming: Deep C Secrets by Peter Van, Der Linden

Unit - 4

Structure

4.0 Objectives

4.1 Introduction

4.2 The Standard Input Output File

4.3 Character Input / Output

4.4 Formatted Input / Output

4.5 Whole Lines of Input and Output

4.6 Standard Library Function

4.7 Mathematical Function

4.8 Character Function

4.9 Summary

4.10 Key words

4.11 Questions

4.12 References

4.0 Objectives

At the end of this unit you will be able to

- Understand the C compilers
- Understand the creating and compiling C programs
- Explain the IDE features of C compilers
- Elucidate the common line options to compile C program

4.1 Introduction : Input and Output

Input and output are covered in some detail. C allows quite precise control of these. This section discusses input and output from keyboard and screen.

The same mechanisms can be used to read or write data from and to files. It is also possible to treat character strings in a similar way, constructing or analyzing them and storing results in variables. These variants of the basic input and output commands are discussed in the next section.

4.2 The Standard Input Output File

UNIX supplies a standard package for performing input and output to files or the terminal. This contains most of the functions which will be introduced in this section, along with definitions of the datatypes required to use them. To use these facilities, your program must include these definitions by adding the line. This is done by adding the line

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

near the start of the program file.

If you do not do this, the compiler may complain about undefined functions or datatypes.

4.3 Character Input / Output

This is the lowest level of input and output. It provides very precise control, but is usually too fiddly to be useful. Most computers perform buffering of input and output. This means that they'll not start reading any input until the return key is pressed, and they'll not print characters on the terminal until there is a whole line to be printed.

getchar

getchar returns the next character of keyboard input as an int. If there is an error then EOF (end of file) is returned instead. It is therefore usual to compare this value against EOF before using it.

If the return value is stored in a char, it will never be equal to EOF, so error conditions will not be handled correctly.

As an example, here is a program to count the number of characters read until an EOF is encountered. EOF can be generated by typing Control - d.

```
#include <stdio.h>

main()
{   int ch, i = 0;

    while((ch = getchar()) != EOF)
        i ++;

    printf("%d\n", i);
}
```

putchar

putchar puts its character argument on the standard output (usually the screen).

The following example program converts any typed input into capital letters. To do this it applies the function toupper from the character conversion library ctype.h to each character in turn.

```
#include <ctype.h>  /* For definition of toupper */
#include <stdio.h>  /* For definition of getchar, putchar, EOF */

main()
{   int ch;

    while((ch = getchar()) != EOF)
        putchar(toupper(ch));
}
```

4.4 Formatted Input / Output

We have met these functions earlier in the course. They are closest to the facilities offered by Pascal or Fortran, and usually the easiest to use for input and output. The versions offered under C are a little more detailed, offering precise control of layout.

printf

This offers more structured output than putchar. Its arguments are, in order; a control string, which controls what gets printed, followed by a list of values to be substituted for entries in the control string.

Control String Entry	What Gets Printed
%d	A Decimal Integer
%f	A Floating Point Value
%c	A Character
%s	A Character String

There are several more types available. You can refer the help section in usage compilers. It is also possible to insert numbers into the control string to control field widths for values to be displayed. For example %6d would print a decimal value in a field 6 spaces wide, %8.2f would print a real value in a field 8 spaces wide with room to show 2 decimal places. Display is left justified by default, but can be right justified by putting a - before the format information, for example %-6d, a decimal integer right justified in a 6 space field.

scanf

scanf allows formatted reading of data from the keyboard. Like printf it has a control string, followed by the list of items to be read. However scanf wants to know the address of the items to be read, since it is a function which will change that value. Therefore the names of variables are preceded by the & sign. Character strings are an exception to this. Since a string is already a character pointer, we give the names of string variables unmodified by a leading & Control string entries which match values to be read are preceded by the percentage sign in a similar way to their printf equivalents.

4.6 Whole Lines of Input and Output

Where we are not too interested in the format of our data, or perhaps we cannot predict its format in advance, we can read and write whole lines as character strings. This approach allows us to read in a line of input, and then use various string handling functions to analyze it at our leisure.

gets

gets reads a whole line of input into a string until a newline or EOF is encountered. It is critical to ensure that the string is large enough to hold any expected input lines. When all input is finished, NULL as defined in stdio.h is returned.

puts

puts writes a string to the output, and follows it with a newline character.

Example: Program which uses gets and puts to double space typed input.

```
#include <stdio.h>

main()
{       char line[256]; /* Define string sufficiently large to
                        store a line of input */

        while(gets(line) != NULL)   /* Read line        */
        {       puts(line);         /* Print line       */
                printf("\n");       /* Print blank line */
        }
}
```

Note that putchar, printf and puts can be freely used together. So can getchar, scanf and gets.

Apart from the above, all the standard input/output library function has been tabulated in next section.

4.7 Standard Library function

<stdio.h>

BUFSIZ

Size of buffer used by setbuf.

EOF

Value used to indicate end-of-stream or to report an error.

FILENAME_MAX

Maximum length required for array of characters to hold a filename.

FOPEN_MAX

Maximum number of files which may be open simultaneously.

L_tmpnam

Number of characters required for temporary filename generated by tmpnam.

NULL

Null pointer constant.

SEEK_CUR

Value for *origin* argument to fseek specifying current file position.

SEEK_END

Value for *origin* argument to fseek specifying end of file.

SEEK_SET

Value for *origin* argument to fseek specifying beginning of file.

TMP_MAX

Minimum number of unique filenames generated by calls to tmpnam.

_IOFBF

Value for *mode* argument to setvbuf specifying full buffering.

_IOLBF

Value for *mode* argument to setvbuf specifying line buffering.

_IONBF

Value for *mode* argument to setvbuf specifying no buffering.

stdin

File pointer for standard input stream. Automatically opened when program execution begins.

stdout

File pointer for standard output stream. Automatically opened when program execution begins.

stderr

File pointer for standard error stream. Automatically opened when program execution begins.

FILE

Type of object holding information necessary to control a stream.

fpos_t

Type for objects declared to store file position information.

size_t

Type for objects declared to store result of sizeof operator.

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

Opens file named *filename* and returns a stream, or NULL on failure. *mode* may be one of the following for text files:

"r"

text reading

"w"

text writing

"a"

text append

"r+"

text update (reading and writing)

"w+"

text update, discarding previous content (if any)

"a+"

text append, reading, and writing at end

or one of those strings with b included (after the first character), for binary files.

FILE* freopen(const char* *filename*, const char* *mode*, FILE* *stream*);

Closes file associated with *stream*, then opens file *filename* with specified mode and associates it with *stream*. Returns *stream* or NULL on error.

int fflush(FILE* *stream*);

Flushes stream *stream* and returns zero on success or EOF on error. Effect undefined for input stream. fflush(NULL) flushes all output streams.

int fclose(FILE* *stream*);

Closes stream *stream* (after flushing, if output stream). Returns EOF on error, zero otherwise.

int remove(const char* *filename*);

Removes specified file. Returns non-zero on failure.

int rename(const char* *oldname*, const char* *newname*);

Changes name of file *oldname* to *newname*. Returns non-zero on failure.

FILE* tmpfile();

Creates temporary file (mode "wb+") which will be removed when closed or on normal program termination. Returns stream or NULL on failure.

char* tmpnam(char *s*[L_tmpnam]);

Assigns to *s* (if *s* non-null) and returns unique name for a temporary file. Unique name is returned for each of the first TMP_MAX invocations.

int setvbuf(FILE* *stream*, char* *buf*, int *mode*, size_t *size*);

Controls buffering for stream *stream*. *mode* is _IOFBF for full buffering, _IOLBF for line buffering, _IONBF for no buffering. Non-null *buf* specifies buffer of size *size* to be used; otherwise, a buffer is allocated. Returns non-zero on error. Call must be before any other operation on stream.

void setbuf(FILE* *stream*, char* *buf*);

Controls buffering for stream *stream*. For null *buf*, turns off buffering, otherwise equivalent to (void)setvbuf(*stream*, *buf*, _IOFBF, BUFSIZ).

int fprintf(FILE* *stream*, const char* *format*, ...);

Converts (according to format *format*) and writes output to stream *stream*. Number of characters written, or negative value on error, is returned. Conversion specifications consist of:

- %
- (optional) flag:

-

left adjust

+

always sign

space

space if no sign

0

zero pad

#

Alternate form: for conversion character o, first digit will be zero, for [xX], prefix 0x or 0X to non-zero value, for [eEfgG], always decimal point, for [gG] trailing zeros not removed.

- (optional) minimum width: if specified as *, value taken from next argument (which must be int).
- (optional) . (separating width from precision):

- (optional) precision: for conversion character s, maximum characters to be printed from the string, for [eEf], digits after decimal point, for [gG], significant digits, for an integer, minimum number of digits to be printed. If specified as *, value taken from next argument (which must be int).
- (optional) length modifier:

h
short or unsigned short

l

long or unsigned long

L

long double

- conversion character:

d,i
int argument, printed in signed decimal notation

o

int argument, printed in unsigned octal notation

x,X

int argument, printed in unsigned hexadecimal notation

u

int argument, printed in unsigned decimal notation

c

int argument, printed as single character

s

char* argument

f

double argument, printed with format [-]*mmm.ddd*

e,E

double argument, printed with format [-]*m.dddddd*(e|E)(+|-)*xx*

g,G

double argument

p

void* argument, printed as pointer

n

int* argument : the number of characters written to this point is written *into* argument

%

no argument; prints %

int printf(const char* *format*, ...);

printf(f, ...) is equivalent to fprintf(stdout, f, ...)

int sprintf(char* *s*, const char* *format*, ...);

Like fprintf, but output written into string *s*, which **must be large enough to hold the output**, rather than to a stream. Output is NUL-terminated. Returns length (excluding the terminating NUL).

int vfprintf(FILE* *stream*, const char* *format*, va_list *arg*);

Equivalent to fprintf with variable argument list replaced by *arg*, which must have been initialised by the va_start macro (and may have been used in calls to va_arg).

int vprintf(const char* *format*, va_list *arg*);

Equivalent to printf with variable argument list replaced by *arg*, which must have been initialised by the va_start macro (and may have been used in calls to va_arg).

int vsprintf(char* *s*, const char* *format*, va_list *arg*);

Equivalent to sprintf with variable argument list replaced by *arg*, which must have been initialised by the va_start macro (and may have been used in calls to va_arg).

int fscanf(FILE* *stream*, const char* *format*, ...);

Performs formatted input conversion, reading from stream *stream* according to format *format*. The function returns when *format* is fully processed. Returns number of items converted and assigned, or EOF if end-of-file or error occurs before any conversion. Each of the arguments following *format* **must be a pointer**. Format string may contain:

- blanks and tabs, which are ignored
- ordinary characters, which are expected to match next non-white-space of input
- conversion specifications, consisting of:
  - %
  - (optional) assignment suppression character "*"
  - (optional) maximum field width
  - (optional) target width indicator:

h

argument is pointer to short rather than int

l

argument is pointer to long rather than int, or double rather than float

L

argument is pointer to long double rather than float

  - conversion character:

d

decimal integer; int* parameter required

i

integer; int* parameter required; decimal, octal or hex

o

octal integer; int* parameter required

u

unsigned decimal integer; unsigned int* parameter required

x

hexadecimal integer; int* parameter required

c

characters; char* parameter required; white-space is not skipped, and NUL-termination is not performed

s

string of non-white-space; char* parameter required; string is NUL-terminated

e,f,g

floating-point number; float* parameter required

p

pointer value; void* parameter required

n

chars read so far; int* parameter required

[...]

longest non-empty string from specified set; char* parameter required; string is NUL-terminated

[^...]

longest non-empty string not from specified set; char* parameter required; string is NUL-terminated

%

literal %; no assignment

int scanf(const char* *format*, ...);

scanf(f, ...) is equivalent to fscanf(stdin, f, ...)

int sscanf(char* *s*, const char* *format*, ...);

Like fscanf, but input read from string *s*.

int fgetc(FILE* *stream*);

Returns next character from (input) stream *stream*, or EOF on end-of-file or error.

char* fgets(char* *s*, int *n*, FILE* *stream*);

Copies characters from (input) stream *stream* to *s*, stopping when *n*-1 characters copied, newline copied, end-of-file reached or error occurs. If no error, *s* is NUL-terminated. Returns NULL on end-of-file or error, *s* otherwise.

int fputc(int *c*, FILE* *stream*);

Writes *c*, to stream *stream*. Returns *c*, or EOF on error.

char* fputs(const char* *s*, FILE* *stream*);

Writes *s*, to (output) stream stream. Returns non-negative on success or EOF on error.

int getc(FILE* *stream*);

Equivalent to fgetc except that it may be a macro.

int getchar(void);

Equivalent to getc(stdin).

char* gets(char* *s*);

Copies characters from stdin into *s* until newline encountered, end-of-file reached, or error occurs. Does not copy newline. NUL-terminates *s*. Returns *s*, or NULL on end-of-file or error. **Should not be used because of the potential for buffer overflow.**

int putc(int *c*, FILE* *stream*);

Equivalent to fputc except that it may be a macro.

int putchar(int *c*);

putchar(*c*) is equivalent to putc(*c*, stdout).

int puts(const char* *s*);

Writes *s* (excluding terminating NUL) and a newline to stdout. Returns non-negative on success, EOF on error.

int ungetc(int *c*, FILE* *stream*);

Pushes *c* (which must not be EOF), onto (input) stream *stream* such that it will be returned by the next read. Only one character of pushback is guaranteed (for each stream). Returns *c*, or EOF on error.

size_t fread(void* *ptr*, size_t *size*, size_t *nobj*, FILE* *stream*);

Reads (at most) *nobj* objects of size *size* from stream *stream* into *ptr* and returns number of objects read. (feof and ferror can be used to check status.)

size_t fwrite(const void* *ptr*, size_t *size*, size_t *nobj*, FILE* *stream*);

Writes to stream *stream*, *nobj* objects of size *size* from array *ptr*. Returns number of objects written.

int fseek(FILE* *stream*, long *offset*, int *origin*);

Sets file position for stream *stream* and clears end-of-file indicator. For a binary stream, file position is set to *offset* bytes from the position indicated by *origin*: beginning of file for SEEK_SET, current position for SEEK_CUR, or end of file for SEEK_END. Behaviour is similar for a text stream, but *offset* must be zero or, for SEEK_SET only, a value returned by ftell. Returns non-zero on error.

long ftell(FILE* *stream*);

Returns current file position for stream *stream*, or -1 on error.

void rewind(FILE* *stream*);

Equivalent to fseek(*stream*, 0L, SEEK_SET); clearerr(*stream*).

int fgetpos(FILE* *stream*, fpos_t* *ptr*);

Stores current file position for stream *stream* in **ptr*. Returns non-zero on error.

int fsetpos(FILE* *stream*, const fpos_t* *ptr*);

Sets current position of stream *stream* to **ptr*. Returns non-zero on error.

void clearerr(FILE* *stream*);

Clears end-of-file and error indicators for stream *stream*.

int feof(FILE* *stream*);

Returns non-zero if end-of-file indicator is set for stream *stream*.

int ferror(FILE* *stream*);

Returns non-zero if error indicator is set for stream *stream*.

void perror(const char* *s*);

Prints *s* (if non-null) and strerror(errno) to standard error as would:

fprintf(stderr, "%s: %s\n", (*s* != NULL ? *s* : ""), strerror(errno))

4.8 Mathematical Function Concept

<math.h>

On domain error, implementation-defined value returned and `errno` set to `EDOM`. On range error, `errno` set to `ERANGE` and return value is `HUGE_VAL` with correct sign for overflow, or zero for underflow. Angles are in radians.

HUGE_VAL

magnitude returned (with correct sign) on overflow error

double exp(double *x*);

exponential of *x*

double log(double *x*);

natural logarithm of *x*

double log10(double *x*);

base-10 logarithm of *x*

double pow(double *x*, double *y*);

x raised to power *y*

double sqrt(double *x*);

square root of *x*

double ceil(double *x*);

smallest integer not less than *x*

double floor(double *x*);

largest integer not greater than *x*

double fabs(double *x*);

absolute value of *x*

double ldexp(double *x*, int *n*);

x times 2 to the power *n*

double frexp(double *x*, int* *exp*);

if *x* non-zero, returns value, with absolute value in interval [1/2, 1), and assigns to **exp* integer such that product of return value and 2 raised to the power **exp* equals *x*; if *x* zero, both return value and**exp* are zero

double modf(double *x*, double* *ip*);

returns fractional part and assigns to **ip* integral part of *x*, both with same sign as *x*

double fmod(double *x*, double *y*);

if *y* non-zero, floating-point remainder of *x*/*y*, with same sign as *x*; if *y* zero, result is implementation-defined

double sin(double *x*);

sine of *x*

double cos(double *x*);

cosine of *x*

double tan(double *x*);

tangent of *x*

double asin(double *x*);

arc-sine of *x*

double acos(double *x*);

arc-cosine of *x*

double atan(double *x*);

arc-tangent of *x*

double atan2(double *y*, double *x*);

arc-tangent of *y*/*x*

double sinh(double *x*);

hyperbolic sine of *x*

double cosh(double *x*);

hyperbolic cosine of *x*

double tanh(double *x*);

hyperbolic tangent of *x*

4.8 Character Functions

<ctype.h>

int isalnum(int *c*);

isalpha(*c*) or isdigit(*c*)

int isalpha(int *c*);

isupper(*c*) or islower(*c*)

int iscntrl(int *c*);

is control character. In ASCII, control characters are 0x00 (NUL) to 0x1F (US), and 0x7F (DEL)

int isdigit(int *c*);

is decimal digit

int isgraph(int *c*);

is printing character other than space

int islower(int *c*);

is lower-case letter

int isprint(int *c*);

is printing character (including space). In ASCII, printing characters are 0x20 (' ') to 0x7E ('~')

int ispunct(int *c*);

is printing character other than space, letter, digit

int isspace(int *c*);

is space, formfeed, newline, carriage return, tab, vertical tab

int isupper(int *c*);

is upper-case letter

int isxdigit(int *c*);

is hexadecimal digit

int tolower(int *c*);

return lower-case equivalent

int toupper(int *c*);

return upper-case equivalent

4.9 Summary

At the end of this unit we have discussed about the basic input and output functions. We have list out various format of input and output functions such as Character IO function, formatted IO functions, Whole line IO functions. Apart from that it has been also covered about the various mathematical and character library functions available in C with their syntax and various flavors

4.10 Keywords

Printf, scanf, gets, puts, stdlib.h, math.h, getchar, putchar

4.11 Questions

1. List out the different types of formatting options provided by C language?
2. Briefly explain about, printf, scanf, gets, put, getchar, putchar functions with their usage?
3. Discuss about standard library functions of C language?
4. List out some standard functions provide under math.h and string.h in C language?

4.12 Reference

Programming In ANSI C by E Balagurusamy

The C Programming Language (Ansi C Version) by Brian W. Kernighan, Dennis M. Ritchie

Expert C Programming: Deep C Secrets by Peter Van, Der Linden

Programming Concepts and C

Module - 2

Unit - 5

Structure

5.0 Objectives

5.1 Expressions and Operators

5.2 Arithmetic Operators

5.3 Type Conversion

5.4 Comparison

5.5 Relational Operators

5.6 Logical Operators

5.7 Assignment Operators

5.8 Increment and Decrement Operators

5.9 Bitwise Operators

5.10 Summary

5.11 Key words

5.12 Questions

5.13 References

5.0 Objectives

At the end of this unit you will be able to

- Understand the concept of expressions and operators
- Understand the type conversion and comparison
- List out different operators supported in C language

5.1 Expressions and Operators

One reason for the power of C is its wide range of useful operators. An operator is a function which is applied to values to give a result. You should be familiar with operators such as +, -, /.

Arithmetic operators are the most common. Other operators are used for comparison of values, combination of logical states, and manipulation of individual binary digits. The binary operators are rather low level for so are not covered here.

Operators and values are combined to form expressions. The values produced by these expressions can be stored in variables, or used as a part of even larger expressions.

5.2 Arithmetic operators

Here are the most common arithmetic operators

```
+  Addition
-  Subtraction
*  Multiplication
/  Division
%  Modulo Reduction (Remainder from integer division)
```

*, / and % will be performed before + or - in any expression. Brackets can be used to force a different order of evaluation to this. Where division is performed between two integers, the result will be an integer, with remainder discarded. Modulo reduction is only meaningful between integers. If a program is ever required to divide a number by zero, this will cause an error, usually causing the program to crash.

Here are some arithmetic expressions used within assignment statements.

```
velocity = distance / time;
force = mass * acceleration;
count = count + 1;
```

C has some operators which allow abbreviation of certain types of arithmetic assignment statements.

Shorthand	Equivalent
`i++;` or `++i;`	`i = i + 1;`
`i--;` or `--i;`	`i = i - 1;`

These operations are usually very efficient. They can be combined with another expression.

`x = a * b++;` is equivalent to
```
x = a * b;
b = b + 1;
```

Versions where the operator occurs before the variable name change the value of the variable before evaluating the expression, so

`x = --i * (a + b);` is equivalent to
```
i = i - 1;
x = i * (a + b);
```

These can cause confusion if you try to do too many things on one command line. You are recommended to restrict your use of ++ and -- to ensure that your programs stay readable.
Another shorthand notation is listed below

Shorthand	Equivalent
`i += 10;`	`i = i + 10;`
`i -= 10;`	`i = i - 10;`
`i *= 10;`	`i = i * 10;`
`i /= 10;`	`i = i / 10;`

These are simple to read and use.

5.3 Type Conversion

You can mix the types of values in your arithmetic expressions. char types will be treated as int. Otherwise where types of different size are involved, the result will usually be of the larger size,

so a float and a double would produce a double result. Where integer and real types meet, the result will be a double.
There is usually no trouble in assigning a value to a variable of different type. The value will be preserved as expected except where;

- The variable is too small to hold the value. In this case it will be corrupted (this is bad).
- The variable is an integer type and is being assigned a real value. The value is rounded down. This is often done deliberately by the programmer.

Values passed as function arguments must be of the correct type. The function has no way of determining the type passed to it, so automatic conversion cannot take place. This can lead to corrupt results. The solution is to use a method called casting which temporarily disguises a value as a different type.
eg. The function sqrt finds the square root of a double.

```
int i = 256;
int root

root = sqrt( (double) i);
```

The cast is made by putting the bracketed name of the required type just before the value. (double) in this example. The result of sqrt((double) i); is also a double, but this is automatically converted to an int on assignment to root.

5.4 Comparison

C has no special type to represent logical or boolean values. It improvises by using any of the integral types char, int, short, long, unsigned, with a value of 0 representing false and any other value representing true. It is rare for logical values to be stored in variables. They are usually generated as required by comparing two numeric values. This is where the comparison operators are used, they compare two numeric values and produce a logical result.

C notation	Meaning
==	Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
!=	Not equal to

Note that == is used in comparisons and = is used in assignments. Comparison operators are used in expressions like the ones below.

x == y

i > 10

a + b != c

In the last example, all arithmetic is done before any comparison is made.

These comparisons are most frequently used to control an if statement or a for or a while loop. These will be introduced in later units.

5.5 Relational Operators

C provides you a list of relational operators to allow you to compare data. The relational operators enable you to check whether two variables or expressions are equal, not equal, which one is greater or less than other...etc. The relational operators are used in boolean conditions or expressions to return 0 or 1.The following table illustrates the relational operators in C:

Relational Operators	Description
==	Equal to
>	Greater than
<	Less than
>=	Greater than or equal to

Relational Operators	Description
<=	Less than or equal to
!=	Not equal to

The following program demonstrates the relational operators in C:

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

/*  Relational operators demo */
void demo(int x, int y) {
    printf("x = %d;y = %d\n", x, y);
    if (x == y) {
        printf("x is equal to y\n");
    } else if (x != y) {
        printf("x is not equal to y\n");
        if (x > y) {
            printf("x is greater than y\n");
        } else {
            printf("x is less than y\n");
        }
    }
}
/ *  main program */
int main(int argc, char** argv) {
    int x = 10;
    int y = 10;
    demo(x, y);
    x = 10;
    y = 20;
```

```
    demo(x, y);


    x = 20;
    y = 10;
    demo(x, y);


    return (0);
}
```

5.6 Logical Operators

These are the usual And, Or and Not operators.

Symbol	Meaning
&&	And
||	Or
!	Not

They are frequently used to combine relational operators, for example

```
x < 20 && x >= 10
```

In C these logical connectives employ a technique known as lazy evaluation. They evaluate their left hand operand, and then only evaluate the right hand one if this is required. Clearly false && anything is always false, true || anything is always true. In such cases the second test is not evaluated.

Not operates on a single logical value, its effect is to reverse its state. Here is an example of its use.

```
if ( ! acceptable )
        printf("Not Acceptable !!\n");
```

5.7 Assignment Operator

C assignment operators are used to assigned the value of a variable or expression to a variable. The syntax of assignment operators is:

```
1 var = expression;
2 var = var;
```

Beside = operator, C programming language supports other short hand format which acts the same assignment operator with additional operator such as +=, -=, *=, /=, %=.

```
1 var +=expression; //means
2 var = var + expression;
```

Each assignment operator has a priority and they are evaluated from right to left based on its priority. Here is assignment operator and its priority: =, +=, -=, *=, /=, %=.

A simple C program to demonstrate assignment operators:

```
#include <stdio.h>
/* a program demonstrates C assignment operator */
void main(){
   int x = 10;
      /* demonstrate = operator */
   int y = x;
   printf("y = %d\n",y);
    /* demonstrate += operator */
   y += 10;
   printf("y += 10;y = %d\n",y);
   /* demonstrate -= operator */
   y -=5;
   printf("y -=5;y = %d\n",y);
    /* demonstrate *= operator */
   y *=4;
   printf("y *=4;y = %d\n",y);
    /* demonstrate /= operator */
```

```
y /=2;
printf("y /=2;y = %d\n",y);
}
```

5.8 Increment and decrement operator

The traditional method of incrementing numbers is by coding something like:

```
a = a + 1;
```

Within C, this syntax is valid but you can also use the ++ operator to perform the same function.

```
a++;
```

will also add 1 to the value of **a**. By using a similar syntax you can also decrement a variable as shown below.

```
a--;
```

These operators can be placed as a prefix or post fix as below:

```
a++;        ++a;
```

When used on their own (as above) the prefix and postfix have the same effect BUT within an expression there is a subtle difference....

1. Prefix notation will increment the variable BEFORE the expression is evaluated.
2. Postfix notation will increment AFTER the expression evaluation.

Here is an example:

```
main()                        main()
{                             {
  int a=1;                      int a=1;
  printf(" a is %d", ++a);      printf(" a is %d", a++);
}                             }
```

In both examples, the final value of **a** will be 2. BUT the first example will print 2 and the second will print 1.

Example Code

```
#include<stdio.h>

main()
{
                /*
                 *      ++i - i incremented before i is used.
                 *      --i - i decremented before i is used.
                 *      j++ - j is incremented AFTER j has been used.
                 *      j-- - j is decremented AFTER j has been used.
                 */

  int i=1,j=1;

  puts("\tDemo 1");
  printf("\t%d %d\n",++i, j++);                      /* O/P  2 1     */
  printf("\t%d %d\n",i, j);                          /* O/P   2 2    */

  i=1;j=1;

  puts("\n\tDemo 2");
  printf("\t%d \n",i=j++);                           /* O/P  1       */
  printf("\t%d \n",i=++j);                           /* O/P  3       */


                                               /* Consider this code */

  i = 0; j = 0;

  puts("\n\tDemo 3");
  if ( (i++ == 1) && (j++ == 1)) puts("Some text");

        /* Will i and j get incremented? The answer is NO! Because
         * the expression in the left of '&&' resolves to false the
         * compiler does NOT execute the expression on the right and
         * so 'j' does not get executed!!!!! */


  printf("\t%d %d\n",i, j);                          /* O/P 1 0      */

}
```

5.9 Bitwise operators

C provides six bitwise operators for manipulating bit. The bitwise operator ca only be applied to integral operands such as char, short, int and long. The following table illustrate C bitwise operators:

C Bitwise Operators	Description
&	Bitwise AND
|	Bitwise inclusive OR
^	Bitwise exclusive OR
<<	Bitwise left shift
>>	Bitwise right shift
~	one's complement

The bitwise operators are preferred in some contexts because bitwise operations are faster than (+) and (-) operations and significantly faster than (*) and (/) operations.

Example of C Bitwise Operators

Here is a simple program that demonstrates C bitwise operators:

```
/* Purpose: Demonstrates C bitwise operators */
#include <stdio.h>
void main() {
   int d1 = 4, /* binary 101 */
       d2 = 6, /* binary 110 */
       d3;
   printf("\nd1=%d", d1);
   printf("\nd2=%d", d2);
```

```
d3 = d1 & d2; /* 0101 & 0110 = 0100 (=4) */
printf("\n Bitwise AND  d1 & d2 = %d", d3);

d3 = d1 | d2; /* 0101 | 0110 = 0110 (=6) */
printf("\n Bitwise OR d1 | d2 = %d", d3);

d3 = d1 ^ d2; /* 0101 & 0110 = 0010 (=2) */
printf("\n Bitwise XOR d1 ^ d2 = %d", d3);

d3 = ~d1; /* ones complement of 0000 0101 is 1111 1010 (-5) */
printf("\n Ones complement of d1 = %d", d3);

d3 = d1 << 2; /* 0000 0101 left shift by 2 bits is 0001 0000 */
printf("\n Left shift by 2 bits d1 << 2 = %d", d3);

d3 = d1 >> 2; /* 0000 0101 right shift by 2 bits is 0000 0001 */
printf("\n Right shift by 2 bits d1 >> 2 = %d", d3); }
```

5.10 Summary

Three types of expression have been introduced here;

- Arithmetic expressions are simple, but watch out for subtle type conversions. The shorthand notations may save you a lot of typing.
- Comparison takes two numbers and produces a logical result. Comparisons are usually found controlling if statements or loops.
- Logical connectors allow several comparisons to be combined into a single test. Lazy evaluation can improve the efficiency of the program by reducing the amount of calculation required.

5.11 Keywords

AND, OR, NOT, &, ! , type conversion, relational operator

5.12 Questions

1. List out the different operators available in C language?
2. Write a program which uses relational operators to display AND, OR and Not Truth table?
3. Explain the concept of type conversion? Why it is required?

5.13 Reference

Programming In ANSI C by E Balagurusamy

The C Programming Language (Ansi C Version) by Brian W. Kernighan, Dennis M. Ritchie

Expert C Programming: Deep C Secrets by Peter Van, Der Linden

Unit - 6

Structure

6.0 Objectives
6.1 Introduction
6.2 Arithmetic Expression
6.3 Type of Expressions
6.4 Operator Precedence
6.5 Arithmetic expression Evaluation
6.6 Some computational problems
6.7 Type conversion in expression
6.8 Operator precedence and Associativity
6.9 Mathematical functions
6.10 Summary
6.11 Key words
6.12 Questions
6.13 References

6.0 Objectives

At the end of this unit you will be able to

- Understand the concept of different types of expression
- Understand the computational complexity with the expression
- List out the mathematical functions available in C language

6.1 Introduction

Once we can declare data to be the type and size with the appropriate precision for our task, we would like to perform operations with the data. We have already discussed some of the basic C operators, and in this section we provide the complete precedence table for all C operators. We present a few new operators here, and others shown in the table will be discussed in detail in later chapters.

6.2 Arithmetic Expressions

- An expression is a string of symbols
- Arithmetic expressions are made up of variable names, binary operators and brackets. But in actual computer languages there are many other things such as powers(**),unary minus(-a),numbers(22/7*3.12a) and things like function(a=find(a,b)+c) and array references may be present.
- We are going to consider the expressions with variables(a-z), digits, binary operators(+,-,*,/) and brackets[(-left &)-right]
- Example of some arithmetic expression

  a+b-c

  a+b+c*d

  (a+b)*(c-d)

6.3 Types of Expression

An expression can be in 3 form

1. Infix Expression
2. Prefix Expression
3. Postfix Expression

- Infix, prefix and postfix notations are different ways of writing expression.
- In the 3 ways, the operands occur in the same order but the operators have to be moved.

- We are using infix type of expression in our daily life but the computer uses postfix or prefix type of expression

Infix Notation:

Operators are written in between their operands

- This is used in our common mathematical expressions.
- The operations(order of evaluation) are performed from left to right. and it obeys precedence rules ie multiplication and division are performed before addition and subtraction.
- Brackets can be used to change the order of evaluation

examples

A+B (b) X*(Y+Z)

Prefix Notation (Polish notation)

- Operators are written before their operands
- Order of evaluation from right to left.
- example

+AB (b)*X+YZ

Postfix Notation (Reverse Polish notation)

- Operators are written after their operands
- The order of evaluation of operators is always from left to right
- brackets cannot be used to change the order of evaluation.
- Example

AB+, (b) XYZ+*

6.4 Operator Precedence

- In the table the precedence is decreasing downwardly
- If there are two operators with same precedence then computer start to solve the expression from left to right

Operator
()
*,/,%
+,-

6.5 Arithmetic Expression Evaluation

An important application of stacks is parsing. ie a compiler must evaluate arithmetic expressions written using infix notation.

- The problem of parsing infix expression can be break in to 2 stages
    1. Infix to Postfix Conversion
    2. Evaluating a Postfix expression
- Converting an infix expression in to postfix expression and evaluating a Postfix expression is a easier problem than directly evaluating Infix expression

6.6 Some Computational Problems

Program: converting Infix expression to postfix expression

```
#include<stdio.h>
#define SIZE 40
char stack[SIZE];
int top=-1;
```

```c
void push(char data)
{
    if(top==SIZE-1)
    {
        printf("Stack is full\n");
        return;
    }
    else
    {
        top=top+1;
        stack[top]=data;
        printf("Pushed element is %c\n",data);
    }
}

char pop()
{
    char ch;
    if(top<0)
    {
        printf("stack is empty\n");
        return;
    }
    else
    {
        ch=stack[top];
        printf("poped element is%c\n",ch);
        top=top-1;
        return(ch);
    }
}

int check_pre(char a ,char b)
{
    //operators are arranged in the array based
    //on their priority. from low to high
    char op[]={'-','+','%','/','*','(',')'};
    int i,c1=0,c2=0;
    for(i=0;i<7;i++)
    {
        if(a==op[i])
```

```
            c1=i+1;
        else if(b==op[i])
            c2=i+1;
    }
    if(c1>c2)
        return(1);
    else if(c1<c2)
        return(-1);
    else
        return(0);
}


int main()
{
    char in_str[50],out_str[50];
    char ch,temp;
    int x=0,y=0,pre;
    printf("Enter the infix string\n");
    scanf("%s",in_str);
    ch=in_str[x];
    while(ch!='\0')
    {
        //for operand
        if((ch>='a' && ch<='z') ||
(ch<='A' && ch>='Z') ||
(ch>='0' && ch<='9'))
              out_str[y++]=ch;
        //for '(' paranthesis
        else if(ch=='(')
            push(ch);
        //for ')' parenthesis
        else if(ch==')')
        {
            temp=pop();
            while(temp!='(')
            {
                out_str[y++]=temp;
                temp=pop();
            }
          //  if(temp=='(')
```

```
                // pop();

        }
        //for operator
        else
        {
        //if the stack is empty or
        // the stack top element is '('
        //just push the operator in to the stack
            if (top==-1 || stack[top]=='(')
                push(ch);
            else
            {
                temp=stack[top];
                //check the precedence
                pre=check_pre(ch,temp);
                if(pre<0 )
                {
                    do{
                    out_str[y++]=pop();
                    temp=stack[top];
}while(top!=-1 && temp!='(' &&  (check_pre(ch,temp)<0));
                    push(ch);
                }
                else
                {
                    push(ch);
                }
            }
        }
        x++;
        ch=in_str[x];
    }
    while(top!=-1)
    {
        out_str[y++]=pop();
    }
    out_str[y]='\0';
    printf("Postfix string=%s\n",out_str);
    return;
```

```
}
```

Output:

```
Enter the infix string
((a+b)*c-(d-e))%(f+g)
Pushed element is (
Pushed element is (
Pushed element is +
poped element is+
poped element is(
Pushed element is *
poped element is*
Pushed element is -
Pushed element is (
Pushed element is -
poped element is-
poped element is(
poped element is-
poped element is(
Pushed element is %
Pushed element is (
Pushed element is +
poped element is+
poped element is(
poped element is%
Postfix string=ab+c*de--fg+%
```

6.7 Type Conversions in Expressions

Consider the following statements:

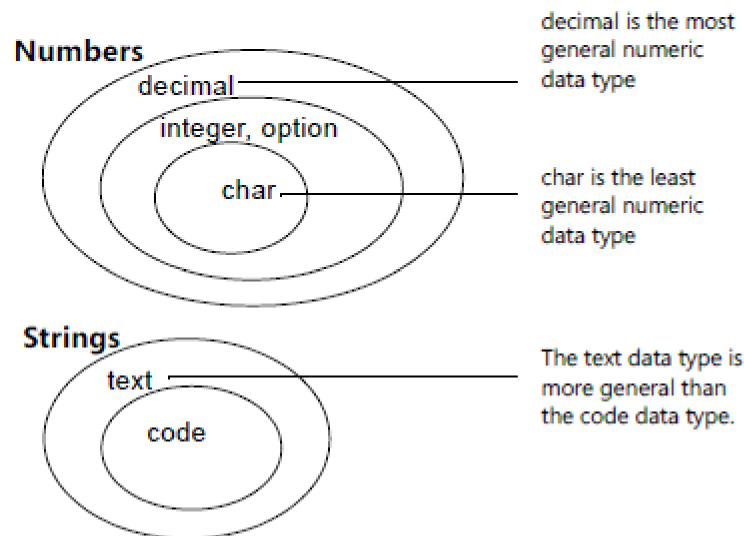
```
CharVar = 15; // A char variable
integerVar = 56000; // An integer variable
Sum:= CharVar + integerVar;
```

The last statement involves one or two type conversions. The rightmost statement involves the evaluation of the expression CharVar + integerVar (char + integer). In order to evaluate this expression, the first operand (CharVar) will have to be converted from char to integer. The addition operator will then return an integer result.

If the type of the leftmost variable has been declared as, for example, decimal, the result must be converted from integer to decimal before its value can be assigned to Sum.

The following examples describe the type conversions which can take place when expressions are evaluated. The following guidelines are used:

- When asked to evaluate an expression of mixed data types, the system will (if possible) always convert at least one of the operands to a more general data type.
- The data types in the two main groups, numbers and strings, can be ranked from "most general" to "least general."



- The most general data types include all the possible values from the less general data types: a decimal is more general than an integer, which is more general than a char.
- Type conversion can take place in some cases even though two operands have the same type.

These rules can be illustrated by the following examples.

Example 1

The following example shows the evaluation of a numeric expression.

```
integer + decimal
```

This expression contains two sub-expressions of different data types. Before it can add these two sub-expressions, the system must convert the leftmost sub-expression to decimal.

```
decimal + decimal
```

When the leftmost sub-expression has been converted, the expression can be evaluated, and the resulting data type will be decimal.

```
decimal + decimal = decimal
```

Example 2

The following example shows the evaluation of a string expression.

```
text + code
```

This expression contains two sub-expressions that must be concatenated. To do this, the system must convert the sub-expression of the least general data type (code) to the most general data type (text).

```
text + text
```

When the rightmost argument has been converted, the expression can be evaluated, and the resulting data type will be text.

```
text + text = text
```

6.8 Operator Precedence and Associativity

The data type and the value of an expression depend on the data types of the operands and the order of evaluation of operators which is determined by the precedence and associativity of operators. Let us first consider the order of evaluation. When expressions contain more than one operator, the order in which the operators are evaluated depends on their precedence levels. A higher precedence operator is evaluated before a lower precedence operator. If the precedence levels of operators are the same, then the order of evaluation depends on their associativity (or, grouping).

In the table, there are 15 precedence levels 0 through 14: higher level implies higher precedence. The precedence levels of operators are separated by solid lines with operators within solid lines having the same precedence level. For example, binary arithmetic operators, , and % have the same precedence level which is higher than binary +, and. Observe that the precedence of the assignment operator is lower than all but the ``comma'' operator (described below). This is in accordance with the rule that the expression on the right side of an assignment is evaluated first, and then its value is assigned to the left hand side object. On the other hand, ``function call'' has the highest precedence, since a function *value* is treated like a variable reference in an expression. In any expression, parentheses may be used to over ride the precedence of the operators - innermost parentheses are always evaluated first. The precedence of binary logical operators is lower than that of binary relational operators; that of binary relational operators is lower than that of binary arithmetic operators, and so forth. The unary NOT operator has precedence higher than that of all binary operators.

Table below shows the precedence levels and associativity of all C operators.

	Operator	Associativity	Precedence
() [] . ->	Function call Array subscript Dot (Member of structure) Arrow (Member of structure)	Left-to-Right	Highest 14
! ~ - ++ -- & * (type) sizeof	Logical NOT One's-complement Unary minus (Negation) Increment Decrement Address-of Indirection Cast Sizeof	Right-to-Left	13
* / %	Multiplication Division Modulus (Remainder)	Left-to-Right	12
+ -	Addition Subtraction	Left-to-Right	11
<< >>	Left-shift Right-shift	Left-to-Right	10
< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	Left-to-Right	8
== !=	Equal to Not equal to	Left-to-Right	8
&	Bitwise AND	Left-to-Right	7
^	Bitwise XOR	Left-to-Right	6
|	Bitwise OR	Left-to-Right	5
&&	Logical AND	Left-to-Right	4
||	Logical OR	Left-to-Right	3
? :	Conditional	Right-to-Left	2
=, += *=, etc.	Assignment operators	Right-to-Left	1
,	Comma	Left-to-Right	Lowest 0

When operators of the same precedence level appear in an expression, the order of evaluation is determined by the associativity. Except for the assignment operator, associativity of most binary operators is left to right; associativity of the assignment operator and most unary operators is right to left. Consider the following program fragment:

```
int x = 10, y = 7, z = 20;
```

When a logical operator is used in an expression, the entire expression is not evaluated if the result of the entire logical expression is clear. For example,

```
(x > 0) && (y > 0)

    (x > 0) || (y > 0)
```

In the first expression, if x > 0 is False, there is no need to evaluate the second part of the logical AND expression since the AND operation will be False. Similarly, in the second expression, the logical OR expression is True if the first part, x > 0, is True; there is no need to evaluate the second part. C evaluates only those parts of a logical expression that are required in order to arrive at the result of the expression.

When in doubt as to the order of evaluation within an expression, parentheses may be used to ensure evaluation is performed as intended.

6.9 Mathematical Function

<math.h>

Mathematics is relatively straightforward library to use again. You **must** #include <math.h> and must **remember** to link in the math library at compilation:

```
cc mathprog.c -o mathprog -lm
```

A common source of error is in forgetting to include the <math.h> file (and yes experienced prog rammers make this error also). Unfortunately the C compiler does not help much. Consider:

```
double x;

x = sqrt(63.9);
```

Having not seen the prototype for sqrt the compiler (by default) assumes that the function returns an int and converts the value to a double with meaningless results.

Math Functions available in C

Below we list some common math functions. Apart from the note above they should be easy to u se and we have already used some in previous examples. We give no further examples here:

double acos(double x) -- Compute arc cosine of x.
double asin(double x) -- Compute arc sine of x.
double atan(double x) -- Compute arc tangent of x.
double atan2(double y, double x) -- Compute arc tangent of y/x, using the signs of both argument s to determine the quadrant of the return value.
double ceil(double x) -- Get smallest integral value that exceeds x.
double cos(double x) -- Compute cosine of angle in radians.
double cosh(double x) -- Compute the hyperbolic cosine of x.
div_t div(int number, int denom) -- Divide one integer by another.
double exp(double x -- Compute exponential of x
double fabs (double x) -- Compute absolute value of x.
double floor(double x) -- Get largest integral value less than x.
double fmod(double x, double y) -- Divide x by y with integral quotient and return remainder.
double frexp(double x, int *expptr) -- Breaks down x into mantissa and exponent of no.
labs(long n) -- Find absolute value of long integer n.
double ldexp(double x, int exp) -- Reconstructs x out of mantissa and exponent of two.
ldiv_t ldiv(long number, long denom) -- Divide one long integer by another.
double log(double x) -- Compute log(x).
double log10 (double x) -- Compute log to the base 10 of x.
double modf(double x, double *intptr) -- Breaks x into fractional and integer parts.
double pow (double x, double y) -- Compute x raised to the power y.
double sin(double x) -- Compute sine of angle in radians.
double sinh(double x) - Compute the hyperbolic sine of x.
double sqrt(double x) -- Compute the square root of x.
void srand(unsigned seed) -- Set a new seed for the random number generator (rand).

double tan(double x) -- Compute tangent of angle in radians.

double tanh(double x) -- Compute the hyperbolic tangent of x.

6.10 Summary

At the end of this unit we have covered the arithmetic expressions and valuation of expressions. The same thing is demonstrated with an example of code which converts infix expression to post fix. Later we have focused on type converts in the expressions and we have mentioned the signifi cance of it. At the end the section we have list out some mathematic functions which are availabl e in C language.

6.11 Keywords

Expressions, INFIX, PREFIX, POSTFIX, type conversion, mathtype.h

6.12 Questions

1. Write a C program to evaluate following expressions?

a) Give postfix form of the following expression

1. A*(B+(C+D)*(E+F)/G)*H
2. A+((B+C)*(D+E)*F)/G
3. A*(B+D)/E-F-(G+H/K)
4. ((A-B)*(D/E))/(F*G*H)

b) Evaluate the following postfix expression using a stack and show the contents of stack after each step

1. 50 40 +18 14-4*+
2. 100 40 8+20 10 -+*
3. 5 6 9 +80 5*-/
4. 120 45+20 10-15+*
5. 20 45+20 10-15+*

2. Explain about the operator precedence and associativity?

6.13 Reference

Programming In ANSI C by E Balagurusamy

The C Programming Language (Ansi C Version) by Brian W. Kernighan, Dennis M. Ritchie

Expert C Programming: Deep C Secrets by Peter Van, Der Linden

Essential C

Solving computational problem using C

Unit - 7

Structure

7.0 Objectives

7.1 Control Statements

7.2 The if else Statement

7.3 The switch Statement

7.4 Loops

7.5 The while Loop

7.6 The do while Loop

7.7 The for Loop

7.8 The break Statement

7.9 The continue Statement

7.10 The goto Statement and Labels

7.11 Summary

7.12 Keywords

7.13 Questions

7.14 Reference

7.0 Objectives

At the end of this unit you will be able to

- Understand the concept of loops
- List out the working principle of do while, for loop statements
- Know about the break, switch statement
- Explain the role of goto and labels in c program

7.1 Control Statements

A program consists of a number of statements which are usually executed in sequence. Programs can be much more powerful if we can control the order in which statements are run.

Statements fall into three general types;

- Assignment, where values, usually the results of calculations, are stored in variables.
- Input / Output, data is read in or printed out.
- Control, the program makes a decision about what to do next.

This section will discuss the use of control statements in C. We will show how they can be used to write powerful programs by;

- Repeating important sections of the program.
- Selecting between optional sections of a program.

7.2 The if else Statement

This is used to decide whether to do something at a special point, or to decide between two courses of action.

The following test decides whether a student has passed an exam with a pass mark of 45

```
if (result >= 45)
        printf("Pass\n");
else
        printf("Fail\n");
```

It is possible to use the if part without the else.

```
if (temperature < 0)
        print("Frozen\n");
```

Each version consists of a test, (this is the bracketed statement following the if). If the test is true then the next statement is obeyed. If is is false then the statement following the else is obeyed if present. After this, the rest of the program continues as normal.

If we wish to have more than one statement following the if or the else, they should be grouped together between curly brackets. Such a grouping is called a compound statement or a block.

```
if (result >= 45)
{       printf("Passed\n");
        printf("Congratulations\n")
}
else
{       printf("Failed\n");
```

```
    printf("Good luck in the resits\n");
}
```

Sometimes we wish to make a multi-way decision based on several conditions. The most general way of doing this is by using the else if variant on the if statement. This works by cascading several comparisons. As soon as one of these gives a true result, the following statement or block is executed, and no further comparisons are performed. In the following example we are awarding grades depending on the exam result.

```
if (result >= 75)
    printf("Passed: Grade A\n");
else if (result >= 60)
    printf("Passed: Grade B\n");
else if (result >= 45)
    printf("Passed: Grade C\n");
else
    printf("Failed\n");
```

In this example, all comparisons test a single variable called result. In other cases, each test may involve a different variable or some combination of tests. The same pattern can be used with more or fewer else if's, and the final lone else may be left out. It is up to the programmer to devise the correct structure for each programming problem.

7.3 The switch Statement

This is another form of the multi way decision. It is well structured, but can only be used in certain cases where;

- Only one variable is tested, all branches must depend on the value of that variable. The variable must be an integral type. (int, long, short or char).
- Each possible value of the variable can control a single branch. A final, catch all, default branch may optionally be used to trap all unspecified cases.

Hopefully an example will clarify things. This is a function which converts an integer into a vague description. It is useful where we are only concerned in measuring a quantity when it is quite small.

```
estimate(number)

int number;

/* Estimate a number as none, one, two, several, many */

{       switch(number) {

        case 0 :

                printf("None\n");

                break;

        case 1 :

                printf("One\n");

                break;

        case 2 :

                printf("Two\n");

                break;

        case 3 :
```

```
    case 4 :
    case 5 :
        printf("Several\n");
        break;
    default :
        printf("Many\n");
        break;
    }
}
```

Each interesting case is listed with a corresponding action. The break statement prevents any further statements from being executed by leaving the switch. Since case 3 and case 4 have no following break, they continue on allowing the same action for several values of number.

Both if and switch constructs allow the programmer to make a selection from a number of possible actions.

The other main type of control statement is the loop. Loops allow a statement, or block of statements, to be repeated. Computers are very good at repeating simple tasks many times, the loop is C's way of achieving this.

7.4 Loops

C gives you a choice of three types of loop, while, do while and for.

- The while loop keeps repeating an action until an associated test returns false. This is useful where the programmer does not know in advance how many times the loop will be traversed.
- The do while loops is similar, but the test occurs after the loop body is executed. This ensures that the loop body is run at least once.

- The for loop is frequently used, usually where the loop will be traversed a fixed number of times. It is very flexible, and novice programmers should take care not to abuse the power it offers.

7.5 The while Loop

The while loop repeats a statement until the test at the top proves false.

As an example, here is a function to return the length of a string. Remember that the string is represented as an array of characters terminated by a null character '\0'.

```
int string_length(char string[])
{       int i = 0;

        while (string[i] != '\0')
                i++;

        return(i);
}
```

The string is passed to the function as an argument. The size of the array is not specified, the function will work for a string of any size.

7.6 The do while Loop

This is very similar to the while loop except that the test occurs at the end of the loop body. This guarantees that the loop is executed at least once before continuing. Such a setup is frequently used where data is to be read. The test then verifies the data, and loops back to read again if it was unacceptable.

```
do
{       printf("Enter 1 for yes, 0 for no :");
        scanf("%d", &input_value);
```

```
} while (input_value != 1 && input_value != 0)
```

7.7 The for Loop

The for loop works well where the number of iterations of the loop is known before the loop is entered. The head of the loop consists of three parts separated by semicolons.

- The first is run before the loop is entered. This is usually the initialisation of the loop variable.
- The second is a test, the loop is exited when this returns false.
- The third is a statement to be run every time the loop body is completed. This is usually an increment of the loop counter.

The example is a function which calculates the average of the numbers stored in an array. The function takes the array and the number of elements as arguments.

```
float average(float array[], int count)
{       float total = 0.0;
        int i;

        for(i = 0; i < count; i++)
                total += array[i];

        return(total / count);
}
```

The for loop ensures that the correct number of array elements are added up before calculating the average.

The three statements at the head of a for loop usually do just one thing each, however any of them can be left blank. A blank first or last statement will mean no initialisation or running increment. A blank comparison statement will always be treated as true. This will cause the loop to run indefinitely unless interrupted by some other means. This might be a return or a break statement.

It is also possible to squeeze several statements into the first or third position, separating them with commas. This allows a loop with more than one controlling variable. The example below

illustrates the definition of such a loop, with variables hi and lo starting at 100 and 0 respectively and converging.

```
for (hi = 100, lo = 0; hi >= lo; hi--, lo++)
```

The for loop is extremely flexible and allows many types of program behavior to be specified simply and quickly.

7.8 The break Statement

We have already met break in the discussion of the switch statement. It is used to exit from a loop or a switch, control passing to the first statement beyond the loop or a switch.
With loops, break can be used to force an early exit from the loop, or to implement a loop with a test to exit in the middle of the loop body. A break within a loop should always be protected within an if statement which provides the test to control the exit condition.

The while loop is used to look at the characters in the string one at a time until the null character is found. Then the loop is exited and the index of the null is returned. While the character isn't null, the index is incremented and the test is repeated.

7.9 The continue Statement

This is similar to break but is encountered less frequently. It only works within loops where its effect is to force an immediate jump to the loop control statement.

- In a while loop, jump to the test statement.
- In a do while loop, jump to the test statement.
- In a for loop, jump to the test, and perform the iteration.

Like a break, continue should be protected by an if statement. You are unlikely to use it very often.

7.10 The goto Statement and labels

C has a goto statement which permits unstructured jumps to be made. The goto statement is used to alter the normal sequence of program execution by transferring control to some other part of the program unconditionally. In its general form, the goto statement is written as

goto label;

Where the label is an identifier that is used to label the target statement to which the control is transferred. Control may be transferred to anywhere within the current function. The target statement must be labeled, and a colon must follow the label. Thus the target statement will appear as

label:statement;

Each labeled statement within the function must have a unique label, i.e., no two statements can have the same label.

7.11 Summary

In this unit we have introduced the concept of loop statements. We have covered all possible looping statements such as do-while, while, for, break, switch, goto and labels. Required examples are given in their sections. At the end of this unit reader are able to figure out the concept of loop constructs which are available in C.

7.12 Keyword

Loops, break, switch, goto, label, for, while, do-while

7.13 Questions

1. What is meant by looping? Why looping is required?
2. List out the difference between increment and decrement for loop
3. Differentiate conditional and unconditional looping with an example?

4. Write a c program to find the Fibonacci series using

(i) while loop

(ii) for loop

(iii) goto statement

5. Write a program to check the given character as vowel or consonant using switch statement?

7.14 Reference

Programming In ANSI C by E Balagurusamy

The C Programming Language (Ansi C Version) by Brian W. Kernighan, Dennis M. Ritchie

Expert C Programming: Deep C Secrets by Peter Van, Der Linden

Unit - 8

Structure

8.0 Objectives

8.1 Introduction

8.2 Automatic Variables

8.3 auto storage class

8.4 External variables

8.5 extern storage class

8.6 register storage class

8.7 Static Variables

8.8 static storage class

8.9 Summary

8.10 Keywords

8.11 Questions

8.12 Reference

8.0 Objectives

At the end of this unit you will be able to

- Understand the concept storage class
- Explain the scope and visibility of variable
- Elucidate optimization of C program

8.1 Introduction

Storage class refers to the permanence of a variable, and its scope which the program, i.e., the portion of the program over which the variable is recognized.

C has a concept of 'Storage classes' which are used to define the scope (visibility) and life time of variables and/or functions.

There are four different storage-class specifications in C:

- automatic,
- external,
- static
- register

They are identified by the keywords,

- auto
- extern,
- static,
- register, respectively.

We will discuss the automatic, external and static storage classes within this unit. The storage class associated with a variable can sometimes be established simply by the location of the variable declaration within the program. In other situations, however, the keyword that specifies a particular storage class must be placed at the beginning of the variable declaration.

8.2 AUTOMATIC VARIABLES

Automatic variables are always declared within a function and are local to the function in which they are declared; that is, their scope is confined to that function. Automatic variables defined in different functions will therefore be independent of one another, even though they may have the same name. Any variable declared within a function is interpreted as an automatic variable unless a different storage-class specification is shown within the declaration. This includes formal argument declarations. All of the variables in the programming examples encountered earlier in this book have been automatic variables. Since the location of the variable declarations within the program determines the automatic storage class, the keyword auto is not required at the beginning of each variable declaration. There is no harm in including an auto specification within a declaration, though this is normally not done.

8.3 auto - storage class

auto is the default storage class for local variables.

```
{
   int Count;
   auto int Month;
}
```

The example above defines two variables with the same storage class. auto can only be used within functions, i.e. local variables.

EXAMPLE: Calculating Factorials Consider once again the program for calculating factorials, originally shown in Example.. Within main, n is an automatic variable. Within factorial, i and prod, as well as the formal argument n, are automatic variables. - The storage-class designation auto could have been included explicitly in the variable declarations if we had wished. Thus, the program could have been written as follows.

```
/* calculate the factorial of an integer quantity */
#include <stdio. h>
long int factorial(int n);
main ()
{
auto int n;
1* read in the integer quantity *1
printf ("\nn =  " );
scanf("%d", &n);
/* calculate and display the factorial */
printf ("\nn! = %ld", factorial(n));
}
 long int factorial (auto int n) /* calculate the factorial */
{
  auto int i;
auto long int prod = 1;
if (n > 1)
for (i = 2; i <= n; ++j)
prod *= i;
return (prod);
}
```

Either method is acceptable. As a rule, however, the auto designation is not included in variable or formal argument declarations, since this is the default storage class. Thus, the program shown in Example 8.10 represents a more common programming style. Automatic variables can be assigned initial values by including appropriate expressions within the variable declarations, as in the above example, or by explicit assignment expressions elsewhere in the function. Such values will be reassigned each time the function is reentered. If an automatic variable is not initialized in some manner, however, its initial value will be unpredictable, and probably unintelligible.

An automatic variable does not retain its value once control is transferred out of its defining function. Therefore, any value assigned to an automatic variable within a function will be lost once the function is exited. If the program logic requires that an automatic variable be assigned a particular value each time the function is executed, that value will have to be reset whenever the function is reentered (i.e., whenever the function is accessed).

8.4 EXTERNAL (GLOBAL) VARIABLES

External variables, in contrast to automatic variables, are not confined to single functions. Their scope extends from the point of definition through the remainder of the program. They are often referred to as global variables.

- Since external variables are recognized globally, they can be accessed from any function that falls within their scope. They retain their assigned values within this scope. Therefore an external variable ‘can be assigned a value within one function, and this value can be used (by accessing the external variable) within another function.

- The use of external variables provides a convenient mechanism for transferring information back and forth between functions. In particular, we can transfer information into a function without using arguments. This is especially convenient when a function requires numerous input data items. Moreover, we now have a way to transfer multiple data items out of a function, since the return statement can return only one data item.

- When working with external variables, we must distinguish between external variable definitions and external variable declarations. An external variable definition is written in the same manner as an ordinary variable declaration. It must appear outside of, and usually before, the functions that access the external variables. An external variable definition will automatically allocate the required storage space for the external variables within the computer’s memory.

- The storage-class specifier extern is not required in an external variable definition, since the external variables will be identified by the location of their definition within the

program. In fact, many C compilers forbid the use of extern within an external variable definition. We will follow this convention within this book.

If a function requires an external variable that has been defined earlier in the program, then the function may access the external variable freely, without any special declaration within the function. (Remember however, that any alteration to the value an external variable within a function will be recognized within the entire scope of the external variable.)

On the other hand, if the function definition precedes the external variable definition, then the function must include a declaration for that external variable. The function definitions within a large program often include external variable declarations, whether they are needed or not, as a matter of good programming practice.

- An external variable declaration must begin with the storage-class specifier extern. The name of the external variable and its data type must agree with the corresponding external variable definition that appears outside of the function. Storage space for external variables will not be allocated as a result of an external variable declaration. Moreover, an external variable declaration cannot include the assignment of initial values. These distinctions between an external variable definition and an external variable declaration are very important.

Finally, it should be pointed out that there are inherent dangers in the use of external variables, since an alteration in the value of an external variable within a function will be carried over into other parts of the program. Sometimes this happens inadvertently, as a side effect of some other action. Thus, there is the possibility that the value of an external value will be changed unexpectedly, resulting in a subtle programming error. You should decide carefully which storage class is most appropriate for each particular programming situation.

8.5 extern - storage Class

extern defines a global variable that is visible to ALL object modules. When you use 'extern' the variable cannot be initialized as all it does is point the variable name at a storage location that has been previously defined.

Source 1	Source 2
--------	--------
extern int count;	int count=5;
write()	
{	main()
printf("count is %d\n",	{
count);	write();
}	}

Count in 'source 1' will have a value of 5. If source 1 changes the value of count - source 2 will see the new value.

8.6 register - Storage Class

register is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and cant have the unary '&' operator applied to it (as it does not have a memory location).

```
{
  register int  Miles;
}
```

Register should only be used for variables that require quick access - such as counters. It should also be noted that defining 'register' goes not mean that the variable will be stored in a register. It means that it might be stored in a register - depending on hardware and implementation restrictions.

8.7 STATIC VARIABLES

- Static variables are defined within individual functions and therefore have the same scope as automatic variables; i.e., they are local to the functions in which they are defined. Unlike automatic variables, however, static variables retain their values throughout the life of the program. Thus, if a function is exited and then re-entered at a later time, the static variables defined within that function will retain their former values. This feature allows functions to retain information permanently throughout the execution of a program.

- Static variables are defined within a function in the same manner as automatic variables, except that the variable declaration must begin with the static storage-class designation. Static variables can be utilized within the function in the same manner as other variables.

They cannot, however, be accessed outside of their defining function. It is not unusual to define automatic or static variables having the same names as external variables. In such situations the local variables will take precedence over the external variables, though the values of the external variables will be unaffected by any manipulation of the local variables. Thus the external variables maintain their independence from locally defined automatic and static variables. The same is true of local variables within one function that have the same names as local variables within another function.

EXAMPLE: Shown below is the skeletal structure of a C program that includes variables belonging to several different storage classes.

```
float a, b, c;
void dummy(void);
main ()
{
static float a;
…………..
}
```

```
void dummy(void)
{
static int a;
int b;
}
```

Within this program a, b and c are external, floating-point variables. However, a is redefined as a static floating-point variable within main. Therefore, b and c are the only external variables that will, be recognized within main. Note that the static local variable a will be independent of the external variable a. Similarly, a and b are redefined as integer variables within dummy. Note that a is a static variable, but b is an automatic variable. Thus, a will retain its former value whenever dummy is reentered, whereas b will lose its value whenever control is transferred out of dummy. Furthermore, c is the only external variable that will be recognized within dummy.

Since a and b are local to dummy, they will be independent of the external variables a, b and c, and the static variable a defined within main. The fact that a and b are declared as integer variables within dummy and floating-point variables elsewhere is therefore immaterial, Initial values can be included in the static variable declarations. The rules associated with the assignment of these values are essentially the same as the rules associated with the initialization of external variables, even though the static variables are defined locally within a function. In particular:

1. The initial values must be expressed as constants, not expressions.
2. The initial values are assigned to their respective variables at the beginning of program execution.

   The variables retain these values throughout the life of the program, unless different values are assigned during the course of the computation.
3. Zeros will be assigned to all static variables whose declarations do not include explicit initial values.

Hence, static variables will always have assigned values.

8.8 static - Storage Class

Static is the default storage class for global variables. The two variables below (count and road) both have a static storage class.

```
static int Count;
    int Road;

    main()
    {
      printf("%d\n", Count);
      printf("%d\n", Road);
}
```

'static' can also be defined within a function. If this is done, the variable is initialized at compilation time and retains its value between calls. Because it is initialized at compilation time, the initialization value must be a constant. This is serious stuff - treated with care.

```
void Func(void)
{
  static Count=1;
}
```

There is one very important use for 'static'. Consider this bit of code.

```
char *Func(void);

    main()
    {
      char *Text1;
      Text1 = Func();
    }

    char *Func(void)
    {
      char Text2[10]="martin";
      return(Text2);
}
```

‘Func' returns a pointer to the memory location where 'Text2' starts BUT Text2 has a storage class of auto and will disappear when we exit the function and could be overwritten by something else. The answer is to specify:

```
static char Text[10]="martin";
```

The storage assigned to 'Text2' will remain reserved for the duration if the program.

8.9 Summary

In this unit we have discussed in detail about the variables and storage classes. It has been covered in detail about automatic, external (global), register and static variable and storage class with an example.

8.10 Keywords

Auto, extern, register, static

8.11 Questions

1.Explain with an example about storage classes in C?

2.With an program show how efficiently you can utilize available storage class in C?

8.12 Reference

Programming In ANSI C by E Balagurusamy

The C Programming Language (Ansi C Version) by Brian W. Kernighan, Dennis M. Ritchie

Expert C Programming: Deep C Secrets by Peter Van, Der Linden

Programming Concepts and C

Module - 3

Unit - 9

Structure

9.0 Objectives

9.1 Introduction

9.2 Graphical Representation

9.3 The Need for a Function

9.4 Types of C Function

9.5 Library Function

9.6 User Defined Function

9.7 Prototype of Function

9.8 Calling a Function

9.9 Summary

9.10 Keywords

9.11 Questions

9.12 Reference

9.0 OBJECTIVES

After going through this lesson you will be able to

- explain of function
- describe access to function
- define parameters data types specification
- explain function prototype and recursion

9.1 Introduction to Functions

A number of statements grouped into a single logical unit are called a function. The use of function makes programming easier since repeated statements can be grouped into functions. Splitting the program into separate function make the program more readable and maintainable.

It is necessary to have a single function 'main' in every C program, along with other functions used/defined by the programmer.

A function definition has two principal components: the function header and body of the function. The function header is the data type of return value followed by function name and (optionally) a set of arguments separated by commas and enclosed in parenthesis. Associated type to which function accepts precedes each argument. In general terms function header statement can be written as

return_type function_name (type1 arg1,type2 arg2,..,typen argn)

where return_type represents the data type of the item that is returned by the function, function_name represents the name of the function, and type1,type2,…,typen represents the data type of the arguments arg1,arg2,..,argn.

Example: Following function returns the sum of two integers.

```
int add(int p, int q)
   {
   return p+q;         //Body of the function
   }
```

Here p and q are arguments. The arguments are called formal arguments or formal parameters, because they represent the name of the data item that is transferred into the function from the calling portion of the program. The corresponding arguments in the function call are called actual arguments or actual parameters, since they define the data items that are actually transferred.

A function can be invoked whenever it is needed. It can be accessed by specifying its name followed by a list of arguments enclosed in parenthesis and separated by commas. e.g., add(5,10);

The following condition must be satisfied for function call.

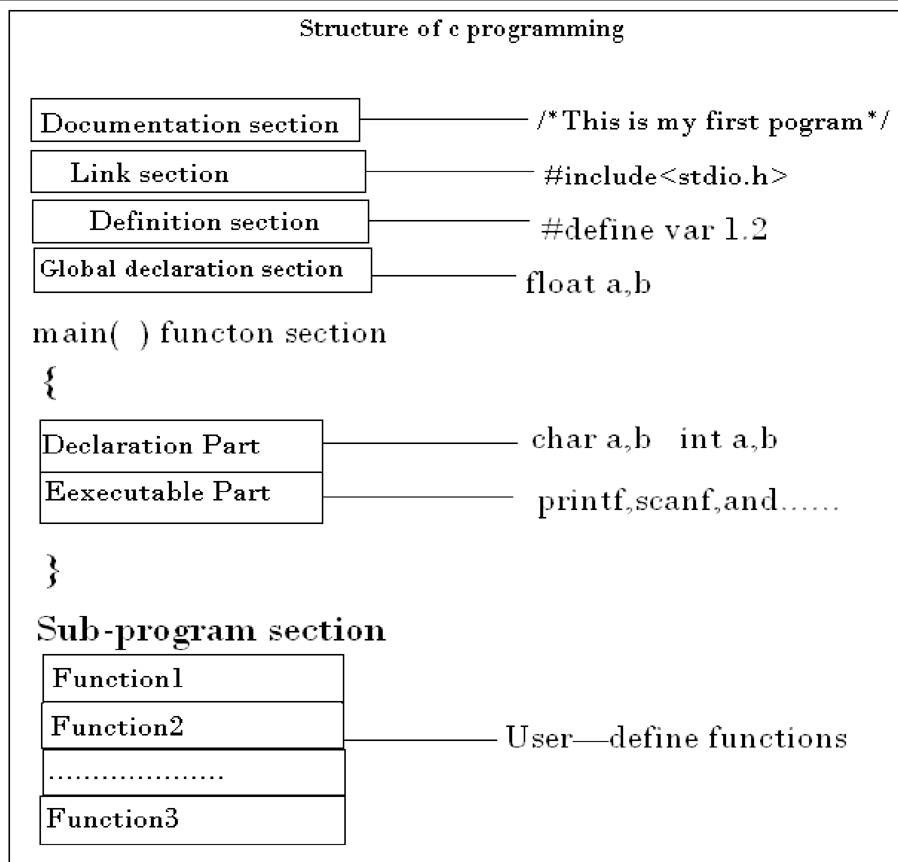
- The number of arguments in the function calls and function declaration must be same.
- The prototype of each of the argument in the function call should be same as the corresponding parameter in the function declaration statement.

For example the code shown bellow illustrate how function can be used in programming

```
//C Program for Addition of  Two Number's using User Define Function
#include<stdio.h>
#include<conio.h>
float add(float,float); // function declaration
void main()
{
float a,b,c;
clrscr();
printf("Enter the value for a & b\n\n");
scanf("%f
%f",&a,&b);
c=add(a,b);
printf("\nc=%f",c);
getch();
}
```

```
// Here is the function definitation
float add(float x,float y)
{
float z;
z=x+y;
return(z);
}
```

9.2 Graphical Representation of Working Principal of Functions



9.3 The Need for Function

Functions are very much required in hard core programming because of many reasons. Here we list out most common necessity for function requirement:

- Functions can be executed as many times as necessary from different points in a program.
- Without the ability to package a block of code into a function, programs would end up being much larger, since you would typically need to replicate the same code at various points in your program.
- But the real reason that you need functions is to modularize your program into easily manageable chunks.
- Functions allow a program to be segmented so that it can be written piecemeal, and each piece tested independently before bringing it together with the other pieces.
- It also allows the work to be divided among members of a programming team, with each team member taking responsibility for a tightly specified piece of the program, with a well defined functional interface to the rest of the code.
- Real world large scale programming projects are developed using this approach

9.4 Types of C functions

Basically, there are two types of functions in C on basis of whether it is defined by user or not.

- ❖ Library function
- ❖ User defined function

9.5 Library function

Library functions are the in-built function in C programming system.

For example:

main()

- The execution of every C program starts from this main.

printf()

- prinf() is used for displaying output in C.

scanf()

- scanf() is used for taking input in C.

for further information about the library functions readers can look into help section of C compiler and more library functions and their usage is discussed in forthcoming sections and modules.

9.6 User defined function

C provides programmer to define their own function according to their requirement known as user defined functions.

Example of how C function works

```
#include <stdio.h>
void function_name(){
................
................
}
int main(){
...........
...........
function_name();
...........
...........
}
```

As mentioned earlier, every C program begins from main() and program starts executing the codes inside main function. When the control of program reaches to function_name() inside main. The control of program jumps to "void function_name()" and executes the codes inside it. When, all the codes inside that user defined function is executed, control of the program jumps to statement just below it. Analyze the figure below for understanding the concept of function in C.

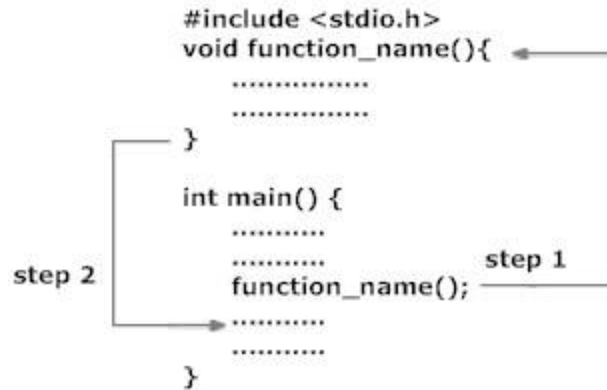


Fig: Working of Functions

Remember, the function name is an identifier and should be unique.

Advantages of user defined functions

- User defined functions helps to decompose the large program into small segments which makes programmer easy to understand, maintain and debug.
- If repeated code occurs in a program. Function can be used to include those codes and execute when needed by calling that function.
- Programmer working on large project can divide the workload by making different functions.

Example of user defined function

```
/*Program to demonstrate the working of user defined function*/
#include <stdio.h>
int add(int a, int b);              //function prototype(declaration)
int main(){
     int num1,num2,sum;
     printf("Enters two number to add\n");
     scanf("%d %d",&num1,&num2);
     sum=add(num1,num2);            //function call
     printf("sum=%d",sum);
     return 0;
}
int add(int a,int b)               //function declarator
{
```

```
/*line 13-19 is called function body
line 12-19 is called function definition.*/
     int add;
     add=a+b;
     return add;                  //return statement of function
}
```

9.7 Prototype of a Function

Every function in C programming should be declared before they are used. This type of declaration are also called function prototype. Function prototype gives compiler information about function name, type of arguments to be passed and return type.

Syntax of function prototype

return_type function_name(type(1) argument(1),....,type(n) argument(n));

Here, in the above example, function prototype is "int add(int a, int b);" which provides following information to the compiler:

1. name of the function is "add"
2. return type of the function is int.
3. two arguments of type int are passed to function.

Function prototype is not needed, if you write function definition above the main function. In this program if the code from line 12 to line 19 is written above main( ), there is no need of function prototype.

9.8 Calling a Function

Control of the program cannot be transferred to user-defined function (function definition) unless it is called (invoked).

Syntax of function call

function_name(argument(1),....argument(n));

In the above example, function call is made using statement "add(num1,num2);" in line 9. This makes control of program transferred to function declarator (line 12). In line 8, the value

returned by function is kept into "sum" which you will study in detail in function return type of next unit.

9.9 Summary

At the end of this unit you have learnt about functions, how to define and declare functions, what are the functions different data types. It has been also discussed about the library functions and user defined functions. You are now able to execute function from different places of a program.

9.10 Keywords

Function call, prototype, library function, used defined function

9.11 Questions

1. Distinguish between a user-defined and one supplied, functions in the C library.
2. In what sense does the user-defined function feature of 'C' extends its repertoire?
3. How many values can a function return?
4. How are argument data types specified in a function declaration?

9.12 Reference

Programming In ANSI C by E Balagurusamy

The C Programming Language (Ansi C Version) by Brian W. Kernighan, Dennis M. Ritchie

Expert C Programming: Deep C Secrets by Peter Van, Der Linden

Unit - 10

Structure

10.0 Objectives

10.1 Function Arguments

10.2 Passing Arguments to Function

10.3 The Return Statement

10.4 Nesting Functions

10.5 The main() Function

10.6 Recursion

10.7 Storage Class Specifier

10.8 Summary

10.9 Keywords

10.10 Questions

10.11 Reference

10.0 OBJECTIVES

After going through this lesson you will be able to

- explain of function arguments
- describe access to function and passing parameters to function
- understand the working of recursion
- explain about storage class specifier

10.1 Functions Arguments

Before the Standard, it was not possible to give any information about a function's arguments except in the definition of the function itself. The information was only used in the body of the function and was forgotten at the end. In those bad old days, it was quite possible to define a function that had three double arguments and only to pass it one int, when it was called. The program would compile normally, but simply not work properly. It was considered to be the programmer's job to check that the number and the type of arguments to a function matched correctly. As you would expect, this turned out to be a first-rate source of bugs and portability problems. Here is an example of the definition and use of a function with arguments, but omitting for the moment to declare the function fully.

```
#include <stdio.h>
#include <stdlib.h>
main(){
	void pmax();			/* declaration */
	int i,j;
	for(i = -10; i <= 10; i++){
		for(j = -10; j <= 10; j++){
			pmax(i,j);
		}
	}
	exit(EXIT_SUCCESS);
}
```

```
/*
* Function pmax.
* Returns:      void
* Prints larger of its two arguments.
*/
void
pmax(int a1, int a2){                       /* definition */
      int biggest;

      if(a1 > a2){
              biggest = a1;
      }else{
              biggest = a2;
      }

      printf("larger of %d and %d is %d\n",
              a1, a2, biggest);
}
```

To start with, notice the careful declaration that pmax returnsvoid. In the function definition, the matching void occurs on the line before the function name. The reason for writing it like that is purely one of style; it makes it easier to find function definitions if their names are always at the beginning of a line.

The function declaration (in main) gave no indication of any arguments to the function, yet the use of the function a couple of lines later involved two arguments. That is permitted by both the old and Standard versions of C, but **must nowadays be considered to be bad practice**. It is much better to include information about the arguments in the declaration too, as we will see. The old style is now an 'obsolescent feature' and may disappear in a later version of the Standard.

Now on to the function definition, where the body is supplied. The definition shows that the function takes two arguments, which will be known as a1 and a2 throughout the body of the function. The types of the arguments are specified too, as can be seen.

In the function definition you don't **have** to specify the type of each argument because they will default to int, but this is bad style. If you adopt the practice of always declaring arguments, even if they do happen to be int, it adds to a reader's confidence. It indicates that you meant to use that type, instead of getting it by accident: it wasn't simply forgotten. The definition of pmax **could** have been this:

```
  /* BAD STYLE OF FUNCTION DEFINITION */
  void
  pmax(a1, a2){
 /* and so on */
```

The proper way to declare and define functions is through the use of *prototypes*.

10.2 Passing Argument to a Function

Arguments can be passed to a function by two methods, they are called passing by value and passing by reference. When a single value is passed to a function via an actual argument, the value of the actual argument is copied into the function. Therefore, the value of the corresponding formal argument can be altered within the function, but the value of the actual argument within the calling routine will not change. This procedure for passing the value of an argument to a function is known as passing by value.

Let us consider an example

```
#include <stdio.h>
main()
{
int x=3;
printf("\n x=%d(from main, before calling the
function"),x);
change(x);
printf("\n\nx=%d(from main, after calling the
function)",x);
}
change(x)
int x;
```

```
{
x=x+3;
printf("\nx=%d(from the function, after being
modified)",x);
return;
}
```

The original value of x (i.e. x=3) is displayed when main begins execution. This value is then passed to the function change, where it is sum up by 3 and the new value displayed. This new value is the altered value of the formal argument that is displayed within the function. Finally, the value of x within main is again displayed, after control is transferred back to main from change.

x=3 (from main, before calling the function)

x=6 (from the function, after being modified)

x=3 (from main, after calling the function)

Passing an argument by value allows a single-valued actual argument to be written as an expression rather than being restricted to a single variable. But it prevents information from being transferred back to the calling portion of the program via arguments. Thus, passing by value is restricted to a one-way transfer of information. Arrays are passed differently than single-valued entities. If an array name is specified as an actual argument, the individual array elements are not copied to the function. Instead the location of the array is passed to the function. If an element of the array is accessed within the function, the access will refer to the location of that array element relative to the location of the first element. Thus, any alteration to an array element within the function will carry over to the calling routine.

10.3 The return statement

The return statement is very important. Every function except those returning void should have at least one, each return showing what value is supposed to be returned at that point. Although it is possible to return from a function by falling through the last }, unless the function returns void an unknown value will be returned, resulting in undefined behavior.

Here is another example function. It uses getchar to read characters from the program input and returns whatever it sees except for space, tab or newline, which it throws away.

```
#include <stdio.h>
int non_space(void) {
    int c;
    while ( (c=getchar ())=='\t' || c== '\n' || c==' ')
        ; /* empty statement */
    return (c);
}
```

Look at the way that all of the work is done by the test in the while statement, whose body was an empty statement. It is not an uncommon sight to see the semicolon of the empty statement sitting there alone and forlorn, with only a piece of comment for company and readability. Please, please, never write it like this:

```
while (something);
```

with the semicolon hidden away at the end like that. It's too easy to miss it when you read the code, and to assume that the following statement is under the control of the while.

The type of expression returned must match the type of the function, or be capable of being converted to it as if an assignment statement were in use. For example, a function declared to return double could contain

```
return (1);
```

and the integral value will be converted to double. It is also possible to have just return without any expression—but this is probably a programming error unless the function returns void. Following the return with an expression is **not** permitted if the function returns void.

10.4 Nesting Function

A nested function is a function defined inside the definition of another function. It can be defined wherever a variable declaration is permitted, which allows nested functions within nested functions. Within the containing function, the nested function can be declared prior to being defined by using the auto keyword. Otherwise, a nested function has internal linkage.

- A nested function can access all identifiers of the containing function that precede its definition.
- A nested function must not be called after the containing function exits.
- A nested function cannot use a goto statement to jump to a label in the containing function, or to a local label declared with the __label__ keyword inherited from the containing function.

10.5 The main() function

When a program begins running, the system calls the function main, which marks the entry point of the program. By default, main has the storage class extern. Every program must have one function named main, and the following constraints apply:

- No other function in the program can be called main.
- main cannot be defined as inline or static.
- main cannot be called from within a program.
- The address of main cannot be taken.

The function main can be defined with or without parameters, using any of the following forms:

```
int main (void)
int main ( )
int main(int argc, char *argv[])
int main (int argc, char ** argv)
```

Although any name can be given to these parameters, they are usually referred to as argc and argv. The first parameter, argc (argument count) is an integer that indicates how many arguments were entered on the command line when the program was started. The second parameter, argv(argument vector), is an array of pointers to arrays of character objects. The array

objects are null-terminated strings, representing the arguments that were entered on the command line when the program was started.

The first element of the array, argv[0], is a pointer to the character array that contains the program name or invocation name of the program that is being run from the command line. argv[1] indicates the first argument passed to the program, argv[2] the second argument, and so on.

10.6 Recursion

With argument passing safely out of the way we can look at recursion. Recursion is a topic that often provokes lengthy and unenlightening arguments from opposing camps. Some think it is wonderful, and use it at every opportunity; some others take exactly the opposite view. Let's just say that when you need it, you really **do** need it, and since it doesn't cost much to put into a language, as you would expect, C supports recursion.

Every function in C may be called from any other or itself. Each invocation of a function causes a new allocation of the variables declared inside it. In fact, the declarations that we have been using until now have had something missing: the keyword auto, meaning ‘automatically allocated’.

```
/* Example of auto */
main(){
    auto int var_name;
    .
    .
    .
}
```

The storage for auto variables is automatically allocated and freed on function entry and return. If two functions both declare large automatic arrays, the program will only have to find room for both arrays if both functions are active at the same time. Although auto is a keyword, it is never used in practice because it's the default for internal declarations and is invalid for external ones. If an explicit initial value isn't given for an automatic variable, then its value will be unknown when it is declared. In that state, any use of its value will cause undefined behavior.

The real problem with illustrating recursion is in the selection of examples. Too often, simple examples are used which don't really get much out of recursion. The problems where it really helps are almost always well out of the grasp of a beginner who is having enough trouble trying to sort out the difference between, say, definition and declaration without wanting the extra burden of having to wrap his or her mind around a new concept as well. The chapter on data structures will show examples of recursion where it is a genuinely useful technique.

The following example uses recursive functions to evaluate expressions involving single digit numbers, the operators *, %, /, +, - and parentheses in the same way that C does. The whole expression is evaluated and its value printed when a character not in the 'language' is read. For simplicity no error checking is performed. Extensive use is made of the ungetc library function, which allows the last character read by getchar to be 'unread' and become once again the next character to be read. Its second argument is one of the things declared in stdio.h.

The main places where recursion occurs are in the function unary_exp, which calls itself, and at the bottom level where primary calls the top level all over again to evaluate parenthesized expressions.

Try running following piece of code for better understanding. Trace its actions by hand on inputs such as

```
1
1+2
1+2 * 3+4
1+--4
1+(2*3)+4
```

```
/*Recursive code for simple C expressions*/

#include <stdio.h>
#include <stdlib.h>

int expr(void);
```

```
int mul_exp(void);
int unary_exp(void);
int primary(void);

main(){
      int val;

      for(;;){
              printf("expression: ");
              val = expr();
              if(getchar() != '\n'){
                      printf("error\n");
                      while(getchar() != '\n')
                              ; /* NULL */
              } else{
                      printf("result is %d\n", val);
              }
      }
      exit(EXIT_SUCCESS);
}

int
expr(void){
      int val, ch_in;

      val = mul_exp();
      for(;;){
              switch(ch_in = getchar()){
              default:
                      ungetc(ch_in,stdin);
                      return(val);
              case '+':
                      val = val + mul_exp();
                      break;
              case '-':
```

```
                        val = val - mul_exp();
                        break;
                }
        }
}

int
mul_exp(void){
        int val, ch_in;

        val = unary_exp();
        for(;;){
                switch(ch_in = getchar()){
                default:
                        ungetc(ch_in, stdin);
                        return(val);
                case '*':
                        val = val * unary_exp();
                        break;
                case '/':
                        val = val / unary_exp();
                        break;
                case '%':
                        val = val % unary_exp();
                        break;
                }
        }
}

int
unary_exp(void){
        int val, ch_in;

        switch(ch_in = getchar()){
        default:
```

```
            ungetc(ch_in, stdin);
            val = primary();
            break;
        case '+':
            val = unary_exp();
            break;
        case '-':
            val = -unary_exp();
            break;
        }
        return(val);
}

int
primary(void){
        int val, ch_in;

        ch_in = getchar();
        if(ch_in >= '0' && ch_in <= '9'){
            val = ch_in - '0';
            goto out;
        }
        if(ch_in == '('){
            val = expr();
            getchar();      /* skip closing ')' */
            goto out;
        }
        printf("error: primary read %d\n", ch_in);
        exit(EXIT_FAILURE);
out:
        return(val);
}
```

10.7 Storage Class Specifier

Every variable and function in C programming has two properties: type and storage class. Type refers to the data type of variable whether it is character or integer or floating-point value etc. There are 4 types of storage class:

- automatic
- external
- static
- register

Automatic storage class

Keyword for automatic variable

auto

Variables declared inside the function body are automatic by default. These variables are also known as local variables as they are local to the function and doesn't have meaning outside that function Since, variable inside a function is automatic by default, keyword auto are rarely used.

External storage class

External variable can be accessed by any function. They are also known as global variables. Variables declared outside every function are external variables.

In case of large program, containing more than one file, if the global variable is declared in file 1 and that variable is used in file 2 then, compiler will show error. To solve this problem, keyword extern is used in file 2 to indicate that, the variable specified is global variable and declared in another file.

Register Storage Class

Keyword to declare register variable: register

Example of register variable

```
register int a;
```

Register variables are similar to automatic variable and exists inside that particular function only.

If the compiler encounters register variable, it tries to store variable in microprocessor's register rather than memory. Value stored in register is much faster than that of memory.
In case of larger program, variables that are used in loops and function parameters are declared register variables. Since, there are limited number of register in processor and if it couldn't store the variable in register, it will automatically store it in memory.

Static Storage Class

The value of static variable persists until the end of the program. A variable can be declared static using keyword: static.
For example:

static int i;

Here, i is a static variable.

10.8 Summary

In this unit we have discussed about function in depth covering the topics such as how an arguments can be passes and about return statement. We have learnt the nesting of function and list out the special case of main() function. At the end of the section we have noted the role of storage class specifier such as auto, extern, register and static. With this knowledge a reader can be able to write bigger program in modular structure using functions.

10.9 Keywords

Function arguments, return, nesting of function, main(), recursion, storage class, auto, extern, static, register

10.10 Questions

1. Discuss in detail about passing arguments to the function?
2. What is recursion? With a suitable example explain the recursion?
3. Explain about main() function?
4. Write a program for

a) Program to reverse number using functions

b) Program to find power of number using recursion

c) Program to find highest common factor using recursion

10.11 Reference

Programming In ANSI C by E Balagurusamy

The C Programming Language (Ansi C Version) by Brian W. Kernighan, Dennis M. Ritchie

Expert C Programming: Deep C Secrets by Peter Van, Der Linden

Unit - 11

Structure

11.0 Objectives

11.1 Introduction: Basics of Array

11.2 One-dimensional Array

11.3 Multi dimensional Array

11.4 Initializing Two Dimension Array

11.5 Array as a Function Argument

11.6 Summary

11.7 Key words

11.8 Questions

11.9 References

11.0 Objectives

At the end of this unit you will be able to

- Understand basic of Array
- Deal with declaring one-dimensional and two-dimensional arrays
- Explain the usage of multidimensional array

11.1 Introduction: Basics of Array

Array is a collection of same type elements under the same variable identifier referenced by index number. Arrays are widely used within programming for different purposes such as sorting, searching and etc. Arrays allow you to store a group of data of a single type. Arrays are efficient and useful for performing operations. You can use them to store a set of high scores in a

video game, a two dimensional map layout, or store the coordinates of a multi-dimensional matrix for linear algebra calculations.

Arrays are of two types single dimension array and multi-dimension array. Each of these array types can be of either static array or dynamic array. Static arrays have their sizes declared from the start and the size cannot be changed after declaration. Dynamic arrays that allow you to dynamically change their size at runtime, but they require more advanced techniques such as pointers and memory allocation.

It helps to visualize an array as a spreadsheet. A single dimension array is represented be a single column, whereas a multiple dimensional array would span out n columns by n rows. In this unit, you will learn how to declare, initialize and access single and multi dimensional arrays.

11.2 One Dimension Arrays

Declaring Single Dimension Arrays

Arrays can be declared using any of the data types available in C. Array size must be declared using constant value before initialization. A single dimensional array will be useful for simple grouping of data that is relatively small in size. You can declare a single dimensional array as follows:

Sample Code

```
<data type> array_name[size_of_array];
```

Say we want to store a group of 3 possible data information that corresponds to a char value then we can declare it like this:

Sample Code

```
char arr_map[4];
```

Note: In C language the end of string is marked by the null character 'N'. Hence to store a group of 3 possible string data. We declare the array as char arr_map[4]; This applies for char type array.

One-dimensional string array containing 3 elements.

Array Element	arr_map[0]	arr_map[1]	arr_map[2]	arr_map[3]
Data	S	R	D	N

One-dimensional integer array containing 3 elements.

Sample Code

```
int emp_code[3];
```

Array Element	emp_code[0]	emp_code[1]	emp_code[2]
Data	7	5	8

Initializing Single Dimension Arrays

Array can be initialized in two ways, initializing on declaration or initialized by assignment.

Initializing on Declaration

If you know the values you want in the array at declaration time, you can initialize an array as follows:

Syntax:

Sample Code

```
<data type> array_name[size_of_array] = {element 1, element 2, ...};
```

Example:

Sample Code

```
char arr_map[3] = {'S', 'R', 'D'};
```

This line of code creates an array of 3 chars containing the values 'S', 'R ', and 'D'.

Initialized by Assignment

Sample Code

```
char arr_map[3];

arr_map[0] = 'S';

arr_map[1] = 'R';

arr_map[2] = 'D';
```

Accessing Single Dimension Array Elements

Arrays are 0-indexed, so the first array element is at index = 0, and the highest is size_of_array –

To access an array element at a given index you would use the following syntax:

Sample Code

```
array_name[index_of_element];
```

To set an element of an array equal to a value you would write:

Sample Code

```
array_name[index_of_element] = value;
```

To access one of the arr_map element:

Sample Code

```
arr_map[0]; //value of 'S'

arr_map[1]; //value of ' R'

arr_map[2]; //value of 'D'
```

Trying to access a value outside the bounds of index 1 through size_of_array – 1, results in runtime errors. Your compiler will not complain, but your program will crash when it executes.

11.3 Multidimensional Arrays

Arrays can have more than one dimension. Two dimensional arrays are widely used for tables, matrices and etc. You can have as many dimensions as you would like but you have to consider the complexity of the code as you add new dimension. Multidimensional arrays allow you to store data in a spreadsheet or matrix like format.

Declaring Multidimensional Arrays

To declare a multidimensional array:

```
<data type> array_name[size_of_first_dimension][size_of_second_dimension] ...
```

A very good example is the creation of 2D maps for arrs.

```
char arr_map[10][10];
```

This would declare a 2 dimensional array of chars that is 10 columns by 10 rows in size.

You could even do a 3D map for a arr using a 3 dimensional array.

```
char arr_map[10][10][5];
```

You can consider above array as being 10 tiles by 10 tiles on the horizontal plane, and 5 tiles high on the vertical plane.

11.4 Initializing Multidimensional Arrays

Just like single dimension array, you can initialize the multidimensional array also upon declaration as well as initialize by assignment.

Sample Code

```
<datatype>
array_name[size_of_dimension1][size_of_second_dimension2] ... = {
{element 1, element 2, element 3, …},
.
```

```
.
. }
};
```

Sample Code

```
char arr_map[2][4];

array [Y][X]
arr_map[2][4];X direction ►
Y direction▼  arr_map[0][0] arr_map[0][1] arr_map[0][2] arr_map[0][3]
arr_map[1][0] arr_map[2][1] arr_map[3][2] arr_map[4][3]
Sample Code
char arr_map[2][4] = { {'x', 'b', 'f', '&#65533;'} , {'b', 't', '&#65533;' } };
array [Y][X]
arr_map[2][4];X direction ►
Y direction▼  x    b    f    �
b    t    �    empty
```

Here is what the above 2D arr_map array will look like if you decided to initialize it during declaration.

Sample Code

```
char arr_map[10][10] = {
{'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'},
{'X', 'H', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'X'},
{'X', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'X'},
{'X', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'X'},
{'X', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'X'},
{'X', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'X'},
{'X', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'X'},
{'X', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'X'},
{'X', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'X'},
{'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'}
};
```

The first index accesses the rows in the array, the second one accesses the columns. Dimensions more than 2 gets difficult to work with, as far as initializing a 3 or more dimensional array goes, you usually only see them assigned values using a series of nested for loops. Here is a simple example.

Sample Code

```
//loop through first dimension
for (i = 0; i < 10; i++) {
  //loop through second dimension
  for (j = 0; j < 10; j++) {
    //loop through third dimension
    for (k = 0; k < 10; k++)
        int_array[i][j][k]) = i * j * k;
    }
}
```

Accessing Elements in Multidimensional Arrays

To access elements in a multidimensional array, all you have to do is provide a combination of index values for each dimension to reach the desired element, just like a bunch of cells in a spreadsheet or coordinates in a n dimensional matrix or graph.

```
array_name[dim1_index][dim2_index] ... [dimn-1_index];
```

Setting values in a multidimensional array is just as simple as in a basic array, you need to add another index per dimension.

```
array_name[index1][index2] ... [index s-1] = value;
```

Let's return to our arr_map 2d array for an example of setting a value in a multidimensional array. We could position our hero character in the arr_map array using this snippet:

```
arr_map[1][1] = 't';
```

That would insert the character 'H' at the intersection of the cell found at coordinate pair (1, 1) in the array.

Here we will look at retrieving a value from a multidimensional array using the above syntax. If we wanted to print out our arr_level map we declared earlier so that it gets displayed in the console we would use the following group of nested for loops to go through both dimensions and print every value we encounter.

Sample Code

```
for (i = 0; i < 10; i++) {
    for (j = 0; j < 10; j++) {
        printf("%c", arr_map[i][j]);
    }
}
```

The first for loop goes through the index values from 0 to 9 of the first dimension of the 2d array. The second for loop does the same for the second dimension. We use the stdio built-in printf function to print the char ("%c") found at the coordinate pair (i, j). because of the nested loops, this code will print every value in the 2d array arr_map out to the console from indexes (0, 0) through (9, 9).

11.5 Array as Function Argument

We have now seen two examples of the use of arrays - to hold numeric data such as test scores, and to hold character strings. We have also seen two methods for determining how many cells of an array hold useful information - storing a count in a separate variable, and marking the end of the data with a special character. In both cases, the details of array processing can easily obscure the actual logic of a program - processing a set of scores or a character string. It is often best to treat an array as an abstract data type with a set of allowed operations on the array which are performed by functional modules. Let us return to our exam score example to read and store

scores in an array and then print them, except that we now wish to use functions to read and print the array.

LIST1: Read an array and print a list of scores using functional modules.

The algorithm is very similar to our previous task, except that the details of reading and printing the array is hidden by functions. The function, read_intaray(), reads scores and stores them, returning the number of scores read. The function, print_intaray(), prints the contents of the array. The refined algorithm for main() can be written as:

print title, etc.

    n = read_intaray(exam_scores, MAX);

    print_intaray(exam_scores, n);

Notice we have passed an array, exam_scores, and a constant, MAX (specifying the maximum size of the proposed list), to read_intarray() and expect it to return the number of scores placed in the array. Similarly, when we print the array using print_intarray, we give it the array to be printed and a count of elements it contains. We saw in Chapter that in order for a called function to access objects in the calling function (such as to store elements in an array) we must use indirect access, i.e. pointers. So, read_intaray() must indirectly access the array, exam_scores, in main(). One unique feature of C is that array access is always indirect; thus making it particularly easy for a called function to indirectly access elements of an array and store or retrieve values. As we will see in later sections, array access by index value is interpreted as an indirect access, so we may simply use array indexing as indirect access.

We are now ready to implement the algorithm for main() using functions to read data into the array and to print the array

```
/*  File: scores2.c
    This program uses functions to read scores into an array and to print
    the scores.
*/
#include <stdio.h>
#define MAX 10

int read_intaray(int scores[], int lim);
void print_intaray(int scores[], int lim);
main()
{   int n, exam_scores[MAX];

    printf("***List of Exam Scores***\n\n");
    n = read_intaray(exam_scores, MAX);
    print_intaray(exam_scores, n);
}

/* Function reads scores in an array. */
int read_intaray(int scores[], int lim)
{   int n, count = 0;

    printf("Type scores, EOF to quit\n");

    while ((count < lim) && (scanf("%d", &n) != EOF)) {
         scores[count] = n;
         count++;
    }
    return count;
}

/* Function prints lim elements in the array scores. */
void print_intaray(int scores[], int lim)
{   int i;

    printf("\n***Exam Scores***\n\n");
    for (i = 0; i < lim; i++)
         printf("%d\n", scores[i]);
}
```

The function calls in main() pass the name of the array, exam_scores, as an argument because the name of an array in an expression evaluates to a pointer to the array. In other words, the expression, exam_scores, is a pointer to (the first element of) the array, exam_scores[]. Its type is, therefore, int *, and a called function uses this pointer (passed as an argument) to indirectly access the elements of the array. As seen in the Figure, for both functions, the headers and the prototypes show the first formal parameter as an integer array without specifying the size. In C, this syntax is interpreted as a pointer variable; so scores is declared aa an int * variable.

11.6 Summary

At the end of this unit we have learnt basic of array such as definition of array, types of array. In later section we have discussed about one dimensional and two dimension array in detail covering about declaration and access. A suitable piece of code has been given as an demonstration example. At the end of the unit we have focused on how arrays say two dimension or higher dimension can be treated as a function argument. Readers are instructed to implement the different programs to better understanding of array concepts.

11.7 Keywords

Array, single dimension array, multi dimension array, array as argument

11.8 Questions

1. What is array? Differentiate between single and multi dimension array?
2. How an array can be passed as an function argument?
3. List out the use of array structure in programming languages?
4. Write program to find the product of two matrices using arrays? Use array as an function argument and find the product in function by receive argument?

11.9 Reference

Programming In ANSI C by E Balagurusamy

The C Programming Language (Ansi C Version) by Brian W. Kernighan, Dennis M. Ritchie

Expert C Programming: Deep C Secrets by Peter Van, Der Linden

Unit - 12

Structure

12.0 Objectives

12.1 Introduction

12.2 Declaration and Initialization of String

12.3 Reading String from Terminal

12.4 Writing String to screen

12.5 Arithmetic Operations on String

12.6 String Handling Function

12.7 Summary

12.8 Key words

12.9 Questions

12.10 References

12.0 Objectives

At the end of this unit you will be able to

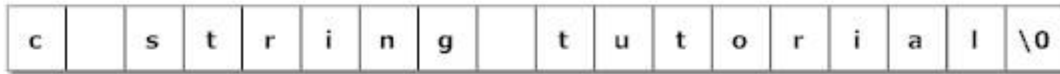
- Understand the string concept
- Describe the string handling
- Explain string handling function

12.1 String

In C, array of character are called strings. A string is terminated by null character /0. For example:

"c string unit"

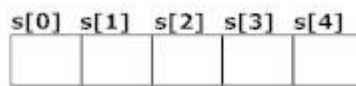
Here, "c string unit" is a string. When, compiler encounters strings, it appends null character at the end of string.



12.2 Declaration and Initialization of Strings

Strings is declared in C in similar manner as arrays. Only difference is that, strings are of char type.

char s[5];



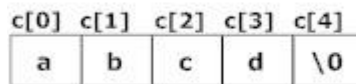
Strings can also be declared using pointer.

char *p

Initialization of strings

In C, string can be initialized in different number of ways.

```
char c[]="abcd";
     OR,
char c[5]="abcd";
     OR,
char c[]={'a','b','c','d','\0'};
     OR;
char c[5]={'a','b','c','d','\0'};
```



String can also be initialized using pointers

char *c="abcd";

12.3 Reading Strings from Terminal

Reading words from user

char name[20];

scanf("%s",name);

String variable, name can only take a word. It is because when white space is encountered, the scanf() function terminates.

C program to illustrate how to read string from terminal.

```
#include <stdio.h>
int main(){
    char name[20];
    printf("Enter name: ");
    scanf("%s",name);
    printf("Your name is %s.",name);
    return 0;
}
```

Output

Enter name: Dennis Ritchie

Your name is Dennis.

Here, program will ignore Ritchie because, when the scanf() function takes only string before the white space.

Reading a line of text

C program to read line of text manually.

```
#include <stdio.h>
int main(){
    char name[30],ch;
    int i=0;
    printf("Enter name: ");
    while(ch!='\n')    //terminates if user hit enter
    {
```

```
            ch=getchar();
            name[i]=ch;
            i++;
        }
        name[i]='\0';        //inserting null character at end
        printf("Name: %s",name);
        return 0;
    }
```

This process to take string is tedious. There are predefined functions gets() and puts in C language to read and display string respectively.

```
#include<stdio.h>
int main(){
    char name[30];
    printf("Enter name: ");
    gets(name);     //Function to read string from user.
    printf("Name: ");
    puts(name);    //Function to display string.
    return 0;
}
```

Both, the above program have same output below:

Output

Enter name: Anil Ross

Name: Anil Ross

Passing Strings to Functions

String can be passed in similar manner as arrays as, string is also an array (of characters).

```
#include <stdio.h>
void Display(char ch[]);
int main(){
    char c[50];
    printf("Enter string: ");
    gets(c);
```

```
        Display(c);          //Passing string c to function.
        return 0;
    }
    void Display(char ch[]){
        printf("String Output: ");
        puts(ch);
    }
```

Here, string c is passed from main() function to user-defined function Display(). In function declaration in line 10, ch[] is the formal argument. It is not necessary to give the size of array in function declaration.

12.4 Writing Strings to the Screen

To write strings to the terminal, we use a file stream known as stdout. The most common function to use for writing to stdout in C is the printf function, defined as follows:

Sample Code

```
int printf(const char *format, ...);
```

To print out a prompt for the user you can:

Sample Code

```
printf("Please type a name: \n");
```

The above statement prints the prompt in the quotes and moves the cursor to the next line.

If you wanted to print a string from a variable, such as our fname string above you can do this:

Sample Code

```
printf("First Name: %s", fname);
```

You can insert more than one variable, hence the "..." in the prototype for printf but this is sufficient. Use %s to insert a string and then list the variables that go to each %s in your string you are printing. It goes in order of first to last. Let's use a first and last name printing example to show this:

Sample Code

```
printf("Full Name: %s %s", fname, lname);
```

The first name would be displayed first and the last name would be after the space between the %s's.

12.5 Arithmetic Operations on Strings

Characters in C can be used just like integers when used with arithmetic operators. This is nice, for example, in low memory applications because unsigned chars take up less memory than do regular integers as long as your value does not exceed the rather limited range of an unsigned char.

For example: consider the following piece of code

charmath.c:

```
#include <stdio.h>
void main() {
 unsigned char val1 = 20;
 unsigned char val2 = 30;
 int answer;

 printf("%d\n", val1);
 printf("%d\n", val2);

 answer = val1 + val2;
 printf("%d + %d = %d\n", val1, val2, answer);

 val1 = 'a';
 answer = val1 + val2;

 printf("%d + %d = %d\n", val1, val2, answer);
}
```

First we make two unsigned character variables and give them number values. We then add them together and put the answer into an integer variable. We can do this without a cast because

characters are an alphanumeric data type. Next we set var1 to an expected character value, the letter lowercase a. Now this next addition adds 97 to 30.

This is because the ASCII value of lowercase a is 97. So it adds 97 to 30, the current value in var2. Notice it did not require casting the characters to integers or having the compiler complain. This is because the compiler knows when to automatically change between characters and integers or other numeric types.

12.6 String handling functions

One can perform different type of string operations manually like: finding length of string, concatenating (joining) two strings etc. But, this process is tedious. To perform these operations, there are different library functions.

String Manipulations Using Library Functions

Strings are often needed to be manipulated according to the problem. All string manipulation can be done manually by the programmer but, this makes programming complex and large. To solve this, the C library supports a large number of string handling functions.

There are numerous functions defined in "string.h" header file. Few commonly used string handling functions are discussed below:

Function	Work Of Function
strlen()	Calculates the length of string
strcpy()	Copies a string to another string
strcat()	Concatenates(joins) two strings
strcmp()	Compares two string
strlwr()	Converts string to lowercase
strupr()	Converts string to uppercase

Strings handling functions are defined under "string.h" header file, i.e, you have to include the code below to run string handling functions.

<string.h>

#include <string.h>

Example to show how to "string.h" header file

```
#include <stdio.h>
#include <string.h>
int main(){
   char s[]="Programiz";
   int length;
   length=strlen(s);    /* Compiler will show error if you use
this statement without using code #include <string.h>
in line number 2*/
   printf("%d",length);
   return 0;
}
```

gets() and puts()

Functions gets() and puts() are two string functions to take string input from user and display string respectively as mentioned in previous chapter.

```
#include<stdio.h>
int main(){
    char name[30];
    printf("Enter name: ");
    gets(name);     //Function to read string from user.
    printf("Name: ");
    puts(name);    //Function to display string.
    return 0;
}
```

Though, gets() and puts() function handle string, both these functions are defined in "stdio.h" header file.

12.7 Summary

In this unit we have introduced string operations which are supported in C programming. We have seen how strings can be declared and initialized. Later we have seen an example how to read and write the string from terminal and to screen respectively. At the end the section we have list out some standard library function in support of string operations.

12.8 Keywords

String, char, strlen, strrev, gets(), puts(), string.h

12.10 Questions

1. Explain about string data type which is supported in C?
2. How can an arithmetic operations can be performed on string?
3. Write a C program to accept two strings and concatenate them?
4. Write a C program to reverse a string?

12.11 Reference

Programming In ANSI C by E Balagurusamy

The C Programming Language (Ansi C Version) by Brian W. Kernighan, Dennis M. Ritchie

Expert C Programming: Deep C Secrets by Peter Van, Der Linden

Programming Concepts and C

Module - 4

Unit - 13

Structure

13.0 Objectives

13.1 Introduction: Basics of Array

13.2 Pointer Declaration

13.3 The & and * Operators

13.4 Passing Pointers to a Function

13.5 Operations on Pointers

13.6 Pointer Arithmetic

13.7 Pointers & Array

13.7 Summary

13.8 Key words

13.9 Questions

13.10 References

13.0 Objectives

At the end of this unit you will be able to

- Understand the concepts of pointer
- Describe the functionality of pointer
- List out various operation on pointers
- Explain the arithmetic operation performed on pointers

13.1 Introduction

In c a pointer is a variable that points to or references a memory location in which data is stored. Each memory cell in the computer has an address that can be used to access that location so a pointer variable points to a memory location we can access and change the contents of this memory location via the pointer.

13.2 Pointer declaration

A pointer is a variable that contains the memory location of another variable. The syntax is as shown below. We start by specifying the type of data stored in the location identified by the pointer. The asterisk tells the compiler that we are creating a pointer variable. Finally we give the name of the variable.

type * variable name;

Example:

int *ptr;

float *string;

13.3 The & and * Operators

Once we declare a pointer variable we must point it to something we can do this by assigning to the pointer the address of the variable we want to point as in the following example:

ptr=#

This places the address where num is stores into the variable ptr. If num is stored in memory 21260 address then the variable ptr has the value 21260.

```
/* A program to illustrate pointer declaration*/
main()
{
int *ptr;
```

```
int sum;
sum=45;
ptr=sum;
printf ("\n Sum is %d\n", sum);
printf ("\n The sum pointer is %d", ptr);
}
```

We will get the same result by assigning the address of num to a regular (non-pointer) variable. The benefit is that we can also refer to the pointer variable as *ptr the asterisk tells to the computer that we are not interested in the value 21260 but in the value stored in that memory location. While the value of pointer is 21260 the value of sum is 45 however we can assign a value to the pointer * ptr as in *ptr=45.

This means place the value 45 in the memory address pointer by the variable ptr. Since the pointer contains the address 21260 the value 45 is placed in that memory location. And since this is the location of the variable num the value also becomes 45. this shows how we can change the value of pointer directly using a pointer and the indirection pointer.

13.4 Passing Pointers to a Function

The pointers are very much used in a function declaration. Sometimes only with a pointer a complex function can be easily represented and success. The usage of the pointers in a function definition may be classified into two groups.

1. Call by reference
2. Call by value

Call by value

We have seen that a function is invoked there will be a link established between the formal and actual parameters.

A temporary storage is created where the value of actual parameters is stored. The formal parameters picks up its value from storage area the mechanism of data transfer between actual and formal parameters allows the actual parameters mechanism of data transfer is referred as call by value.

The corresponding formal parameter represents a local variable in the called function .

The current value of corresponding actual parameter becomes the initial value of formal parameter.

The value of formal parameter may be changed in the body of the actual parameter.

The value of formal parameter may be changed in the body of the subprogram by assignment or input statements. This will not change the value of actual parameters.

```
/* Include< stdio.h >
void main()
{
int x,y;
x=20;
y=30;
printf("\n Value of a and b before function call =%d %d",a,b);
fncn(x,y);
printf("\n Value of a and b after function call =%d %d",a,b);
}
fncn(p,q)
int p,q;
{
p=p+p;
q=q+q;
}
```

Call by Reference

When we pass address to a function the parameters receiving the address should be pointers. The process of calling a function by using pointers to pass the address of the variable is known as call by reference. The function which is called by reference can change the values of the variable used in the call.

```
/* example of call by reference*/
/* Include< stdio.h >
void main()
{
int x,y;
```

```
x=20;
y=30;
printf("\n Value of a and b before function call =%d %d",a,b);
fncn(&x,&y);
printf("\n Value of a and b after function call =%d %d",a,b);
}
fncn(p,q)
int p,q;
{
*p=*p+*p;
*q=*q+*q;
}
```

13.5 Operations on Pointers

Pointer variables are not directly usable by many of the operators, functions, or procedures provided by IDL. You cannot, for example, do arithmetic on them or plot them. You can, of course, do these things with the heap variables referenced by such pointers, assuming that they contain appropriate data for the task at hand. Pointers exist to allow the construction of dynamic data structures that have lifetimes that are independent of the program scope they are created in.

There are 4 IDL operators that work with pointer variables: assignment, dereference, EQ, and NE. The remaining operators (addition, subtraction, etc.) do not make any sense for pointer types and are not defined.

Many non-computational functions and procedures in IDL do work with pointer variables. Examples are SIZE, N_ELEMENTS, HELP, and PRINT. It is worth noting that the only I/O allowed directly on pointer variables is default formatted output, where they are printed as a symbolic description of the heap variable they point at. This is merely a debugging aid for the IDL programmer—input/output of pointers does not make sense in general and is not allowed. Please note that this does *not* imply that I/O on the contents of non-pointer data held in heap variables is not allowed. Passing the contents of a heap variable that contains non-pointer data to the PRINT command is a simple example of this type of I/O.

Assignment

Assignment works in the expected manner—assigning a pointer to a variable gives you another variable with the same pointer. Hence, after executing the statements:

```
A = PTR_NEW(FINDGEN(10))
B = A
HELP, A, B
```

A and B both point at the same heap variable and we see the output:

```
A POINTER= <PtrHeapVar1>
B POINTER= <PtrHeapVar1>
```

Dereference

In order to get at the contents of a heap variable referenced by a pointer variable, you must use the *dereference operator*, which is * (the asterisk). The dereference operator precedes the variable dereferenced. For example, if you have entered the above assignments of the variables A and B:

PRINT, *B

IDL prints:

0.00000 1.00000 2.00000 3.00000 4.00000 5.00000
6.00000 7.00000 8.00000 9.00000

That is, IDL prints the contents of the heap variable pointed at by the pointer variable B.

Dereferencing Pointer Arrays

Note that the dereference operator requires a *scalar* pointer operand. This means that if you are dealing with a pointer array, you must specify which element to dereference. For example, create a three-element pointer array, allocating a new heap variable for each element:

```
ptarr = PTRARR(3, /ALLOCATE_HEAP)
```

To initialize this array such that the heap variable pointed at by the first pointer contains the integer zero, the second the integer one, and the third the integer two, you would use the following statement:

```
for i = 0,2 do *ptarr[i] = i
```

Note: The dereference operator is dereferencing only element I of the array for each iteration. Similarly, if you wanted to print the values of the heap variables pointed at by the pointers in ptarr, you might be tempted to try the following:

```
PRINT, *ptarr
```

IDL prints:

```
% Expression must be a scalar in this context: PTARR.
% Execution halted at: $MAIN$
```

To print the contents of the heap variables, use the statement:

```
FOR I = 0, N_ELEMENTS(ptarr)-1 DO PRINT, *ptarr[I]
```

Dereferencing Pointers to Pointers

The dereference operator can be applied as many times as necessary to access data pointed at indirectly via multiple pointers. For example, the statement:

```
A = PTR_NEW(PTR_NEW(47))
```

- Assigns to A a pointer to a pointer to a heap variable containing the value 47.

To print this value, use the following statement:

```
PRINT, **A
```

Dereferencing Pointers within Structures

If you have a structure field that contains a pointer, dereference the pointer by prepending the dereference operator to the front of the structure name. For example, if you define the following structure:

```
struct = {data:’10.0’, pointer:ptr_new(20.0)}
```

you would use the following command to print the value of the heap variable pointed at by the pointer in the pointer field:

PRINT, *struct.pointer

Defining pointers to structures is another common practice. For example, if you define the following pointer:

ptstruct = PTR_NEW(struct)

you would use the following command to print the value of the heap variable pointed at by the pointer field of the struct structure, which is pointed at by ptstruct:

PRINT, *(*pstruct).pointer

Note that you must dereference both the pointer to the structure and the pointer within the structure.

Dereferencing the Null Pointer

It is an error to dereference the NULL pointer, an invalid pointer, or a non-pointer. These cases all generate errors that stop IDL execution. For example:

PRINT, *45

IDL prints:

% Pointer type required in this context: <INT(45)>.

% Execution halted at: $MAIN$

For example:

```
A = PTR_NEW() & PRINT, *A
IDL prints:
% Unable to dereference NULL pointer: A.
% Execution halted at:  $MAIN$
For example:
A = PTR_NEW(23) & PTR_FREE, A & PRINT, *A
IDL prints:
% Invalid pointer: A.
% Execution halted at: $MAIN$
```

Equality and Inequality

The EQ and NE operators allow you to compare pointers to see if they point at the same heap variable. For example:

Make A a pointer to a heap variable containing 23.

```
A = PTR_NEW(23);
//B points at the same heap variable as A.
B = A;
//C contains the null pointer.
C = PTR_NEW();

PRINT, 'A EQ B: ', A EQ B & $
PRINT, 'A NE B: ', A NE B & $
PRINT, 'A EQ C: ', A EQ C & $
PRINT, 'C EQ NULL: ', C EQ PTR_NEW() & $
PRINT, 'C NE NULL:', C NE PTR_NEW()
IDL prints:
A EQ B:    1
```

```
A NE B:     0
A EQ C:     0
C EQ NULL: 1
C NE NULL: 0
```

13.6 Pointer Arithmetic

C has the speed and efficiency of assembly language combined with readability of assembly language. In other words, it's just a glorified assembly language.

It's perhaps too harsh a judgment of C, but certainly one of the reasons the language was invented was to write operating systems. Writing such code requires the ability to access addresses in memory in an efficient manner. This is why pointers are such an important part of the C language. They're also a big reason programmers have bugs.

If you're going to master C, you need to understand pointer arithmetic, and in particular, the relationship between arrays and pointers.

Arbitrary Pointer Casting

Because most ISAs use the same number of bits as integers, it's not so uncommon to cast integers as pointers.

Here's an example.

```
// Casting 32 bit int, 0x0000ffff, to a pointer
char * ptr  = reinterpret_cast<char *>( 0x0000ffff ) ;
char * ptr2 = reinterpret_cast<char *>( 0x0000ffff ) ;
```

In general, this is one of the pitfalls of C. Arbitrary pointer casting allows you to point anywhere in memory.

Unfortunately, this is not good for safe programs. In a safe programming language (say, Java), the goal is to access objects from pointers only when there's an object there. Furthermore, you want to call the correct operations based on the object's type.

Arbitrary pointer casting allows you to access any memory location and do anything you want at that location, regardless of whether you can access that memory location or whether the data is valid at that memory location.

13.7 Pointers and Arrays

In C, arrays have a strong relationship to pointers.

Consider the following declaration.

int arr[10] ;

You might think from the above statement that arr is of type int. However, arr, by itself, without any index subscripting, can be assigned to an integer pointer.

Do you know, however, that arr[i] is defined using pointer arithmetic?

In particular:

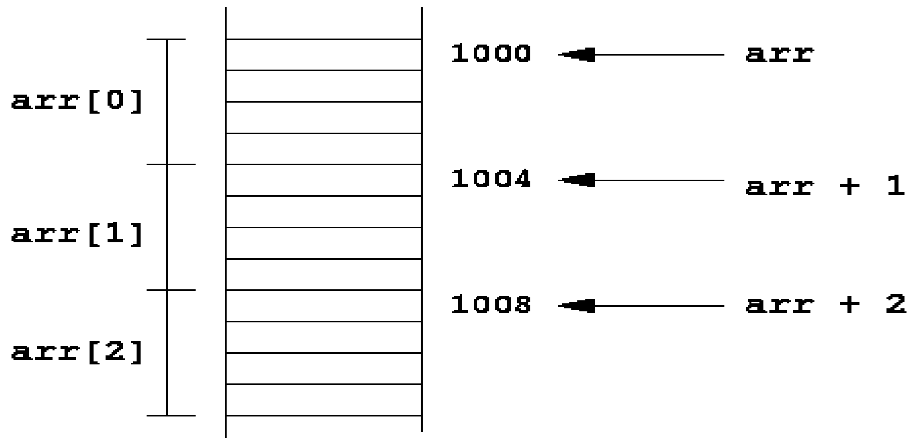
arr[i] == * (arr + i)

Let's take a closer look at arr + i. What does that mean? arr is a pointer to arr[0]. In fact, it is defined to be & arr[0].

A pointer is an address in memory. arr contains an address in memory---the address where arr[0] is located.

What is arr + i? If arr is a pointer to arr[0] then arr + i is a pointer to arr[i].

Perhaps this is easier shown in a diagram.



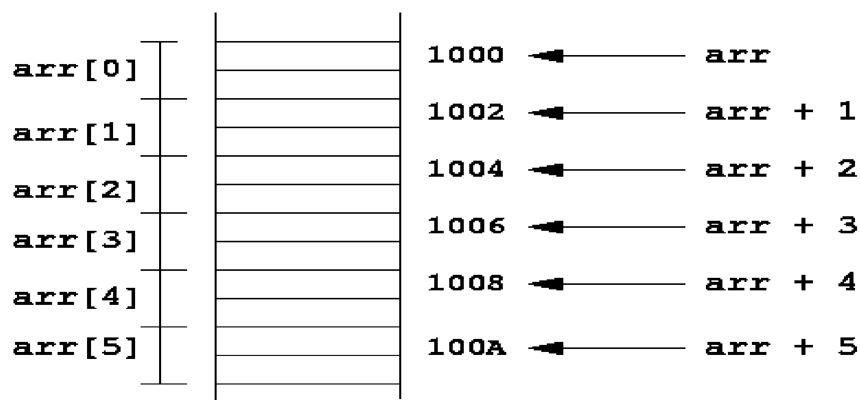
We assume that each int takes up 4 bytes of memory. If arr is at address 1000, then arr + 1 is address 1004, arr + 2 is address 1008, and in general, arr + i is address 1000 + (i * 4).

Now that you see the diagram, something may strange. Why is arr + 1 at address 1004 and not 1001?

That's pointer arithmetic in action. arr + 1 points to one element past arr. arr + 3 is points to 3 elements past arr. Thus, arr + i points to i elements past arr.

The idea is to have arr + i point to i elements after arr regardless of what type of element the array holds.

Suppose the array had contained short where a short is only 2 bytes. If arr is at address 1000, then, arr + 1 is address 1002 (instead of 1004). arr + 2 is at address 1004 (instead of 1008), and in general arr + i is at address 1000 + (2 * i).



Notice that arr + 1 is now 2 bytes after arr and arr + 2 is 4 bytes. When we had an int array, arr + 1 was 4 bytes after arr and arr + 2 was 8 bytes afterwards.

Why is there a difference? What's the difference between arr before and now?

The difference is in the type. Before, arr had (roughly) type int * and now arr has type short *.

In C, a pointer is not only an address, it tells you what (in principle) the data type at that address is. Thus, int * is a pointer to an int. Not only does it tell you the data type, but you can also determine the data type's size.

You can find out the data type's size with the sizeof() operator.

sizeof(int) ==> 4
sizeof(short) ==> 4
That's important, because that's how pointer arithmetic computes address. It uses the size of the data type, which it knows from the pointer's type.

Here's the formula for computing the address of ptr + i where ptr has type T *. then the formula for the address is:

addr(ptr + i) = addr(ptr) + [sizeof(T) * i]

T can be a pointer

How does pointer arithmetic work if you have an array of pointers, for example:

int * arr[10] ;
int ** ptr = arr ;
In this case, arr has type int **. Thus, T has type int *. All pointers have the same size, thus the address of ptr + i is:

addr(ptr + i) = addr(ptr) + [sizeof(int *) * i]
= addr(ptr) + [4 * i]

Static arrays are constant

When you declare

```
int arr[ 10 ] ;
```

arr is a constant. It is defined to be the address, & arr[0].

You can't do the following:

```
int arr[ 10 ] ;

arr = arr + 1 ; // NO! Can't reassign to arr--it's constant
```

However, you can declare a pointer *variable*.

```
int arr[ 10 ] ;
int * ptr ;

ptr = arr + 1 ; // This is OK. ptr is a variable.
```

Parameter Passing an Array

If you pass an array to a function, it's type changes.

```
void foo( int arr[ 10 ], int size ) {
// code here
}
```

The compiler translates arrays in a parameter list to:

```
void foo( int * arr, int size ) {
// code here
}
```

Thus, it becomes arr becomes a pointer variable.

Why doesn't this cause problem? After all, won't we pass arr as an argument? Isn't arr a constant?

Yes, it is. However, we pass a copy of the address to arr the parameter. Thus, the copy can be manipulated while the original pointer address that was passed during the function call is unchanged.

Subtraction

We can also compute ptr - i. For example, suppose we have an int array called arr.

```
int arr[ 10 ] ;
int * p1, * p2 ;

p1 = arr + 3 ; // p1 == & arr[ 3 ]
p2 = p1 - 2 ; // p1 == & arr[ 1 ]
```

We can have a pointer point to the middle of the array (like p1). We can then create a new pointer that is two elements back of p1.

As it turns out, we can even point way past the end of the array.

```
int arr[ 10 ];
int * p1, * p2;

p1 = arr + 100; // p1 == & arr[ 100 ]
p2 = arr - 100; // p1 == & arr[ -100 ]
```

The compiler still computes an address, and does not core dump. For example, if arr is address 1000, then p1 is address 1400 and p2 is address 600. The compiler still uses the formula for pointer arithmetic to compute the address.

Passing the Array Size

This is why it's usually important to keep track of the array size and pass it in as a parameter. When you're in a function, you can't tell the size of the array. All you have is a pointer, and that's it. No indication of how big that array is.

If you try to compute the array's size using sizeof, you just get 4.

```
// Compiler translates arr's type to int *
void foo ( int arr[], int size ) {
// Prints 4
cout << sizeof( arr ) ;

int arr2[ 10 ] ;
```

```
// Prints 40
cout << sizeof( arr2 ) ;
}
```

If you declare a local array (not using dynamic memory allocation), you can get the size of the array. However, once you pass that array, all that's passed is the address. There's no information about the array size anymore.

Pointers on two dimensional Arrays

Suppose you declare:

int arr[10][12] ;

What type is arr? You may have been told that it's int **, but that's incorrect. Two dimensional arrays (as declared above) are contiguous in memory. If you create an array of pointers to dynamically allocated arrays, such as:

int * arr[10] ;

then, arr has type int ** (or at least has a type compatible with int **).

The type is rather complicated, and is due to the fact that arr[0] = & arr[0][0], arr[1] = & arr[1][0], and in general, arr[i] = & arr[i][0].

Pointer arithmetic says that arr + i gives you & arr[i], yet this skips an entire row of 12 elements, i.e., skips 48 bytes times i. Thus, if arr is address 1000 then arr + 2 is address 1096.

If the array's type were truly int **, pointer arithmetic would say the address is 1008_{ten}, so that doesn't work.

Note: A two-dimensional array is not the same as an array of pointers to 1D array

The actual type for a two-dimensional array, is declared as:

int (*ptr)[10] ;

Which is a pointer to an array of 10 elements? Thus, when you do pointer arithmetic, it can compute the size of the array and handle it correctly. The parentheses are NOT optional above. Without the parentheses, ptr becomes an array of 10 pointers, not a pointer to an array of 10 int.

If you have a conventional two dimensional array, and you want to compute the address for arr[row][col] and you have ROWS rows (where ROWS is some constant) and COLS columns, then the formula for the address in memory is:

```
addr( & arr[ row ][ col ] ) = addr( arr ) + [ sizeof( int ) * COLS *
row ]+ [ sizeof( int ) * col ]
```

Two dimensional arrays are stored in row major order, that is, row by row. Each row contains COLS elements, which is why you see COLS in the formula. In fact, you don't see ROWS.

When you have a 2D array as a parameter to a function, there's no need to specify the number of rows. You just need to specify the number of columns. The reason is the formula above. The compiler can compute the address of an element in a 2D array just knowing the number of columns.

Thus, the following is valid in C, i.e. it compiles:

```
void sumArr( int arr[][ COLS ], int numRows, int numCols ) {
}
```

The following is also valid in C.

```
void sumArr( int arr[ ROWS ][ COLS ], int numRows, int numCols ) {
}
```

The compiler ignores ROWS. Thus, any 2D array with the COLS columns and any number of rows can be passed to this function.

The following, however, is NOT valid in C:

```
void sumArr( int arr[][], int numRows, int numCols ) {
}
```

It's not syntactically valid to declare int arr[][] in C.

However, it's OK to write:

```
void sumArr( int **arr, int numRows, int numCols ) {
}
```

Note that int **arr is an array of pointers (possibly to 1D arrays), while int arr[][COLS] is a 2D array. They are not the same type, and are not interchangeable.

Pointer Subtraction

It turns out you can subtract two pointers of the same type. The result is the distance (in array elements) between the two elements.

For example:

```
int arr[ 10 ] ;
int * p1 = arr + 2 ;
int * p2 = arr + 5 ;

cout << ( p2 - p1 ) ; // Prints 3
cout << ( p1 - p3 ) ; // Prints -3
```

The formula used is rather simple. Assume that p1 and p2 are both pointers of type T *. Then, the value computed is:

```
( p2 - p1 ) == ( addr( p2 ) - addr( p1 ) ) / sizeof( T )
```

This can result in negative values if p2 has a smaller address than p1.

p2 and p1 need not point to valid elements in an array. The formula above still works even when p2 and p1 contain invalid addresses (because they contain *some* address).

Pointer subtraction isn't used very much, but can be handy to determine the distances between two array elements (i.e., the difference in the array indexes). You may not know exactly which element you're pointing to using pointer subtraction, but you can tell relative distances.

13.8 Summary

At the end of this unit we have covered the topics such as basic terminology of pointer. It has been clearly given about how the pointer variables can be declared and how it can visualized with a suitable examples. In subsequent stages we have discussed about the role of & and * operators. In the later stages we had list out various arithmetic operations which can be performed using pointer. In the last section of the unit we have seen how exactly arrays can be handled using pointer.

13.9 Keywords

Pointer, & and * operator, Sizeof(), Null

13.10 Questions

1. Briefly explain the role of pointers in C programming?
2. Discuss about the advantages and disadvantages using pointer in programming?
3. What is meaning of following declaration?

   int(*ptr[5])();
4. Explain about various arithmetic operations on pointers?
5. Write a C program using pointer to find the memory address of all declared variables in a program?

13.11 Reference

"Common Pointer Pitfalls" by Dave Marshall

C Programming: A Modern Approach by K.N. King

C Programming in 12 Easy Lessons by Greg Perry

C for Dummies Vol. II by Dan Gookin

Unit - 14

Structure

14.0 Objectives

14.1 Pointer to Functions

14.2 Function returning Pointer

14.3 Static and Dynamic Memory Allocation

14.4 DMA Functions

14.5 Bitwise Operators

14.6 Preprocessor Directives

14.7 Summary

14.8 Key words

14.9 Questions

14.10 References

14.0 Objectives

At the end of this unit you will be able to

- Understand advance concept on pointers
- Sketch out the working principle of pointer to function as argument
- Explain various memory related function
- List out the various preprocess directives and bitwise operators

14.1 Pointer to Functions

A useful technique is the ability to have pointers to functions. Their declaration is easy: write the declaration as it would be for the function, say

int func(int a, float b);

and simply put brackets around the name and a * in front of it: that declares the pointer. Because of precedence, if we don't parenthesize the name, we declare a function returning a pointer:

/* function returning pointer to int */

int *func(int a, float b);

/* pointer to function returning int */

int (*func)(int a, float b);

Once we've got the pointer, we can assign the address of the right sort of function just by using its name: like an array, a function name is turned into an address when it's used in an expression. We can call the function using one of two forms:

```
(*func)(1,2);
/* or */
func(1,2);
```

The second form has been newly blessed by the Standard. Here's a simple example.

```
#include <stdio.h>
#include <stdlib.h>
void func(int);
main()
{
void (*fp)(int);
fp = func;
(*fp)(1);
fp(2);
exit(EXIT_SUCCESS);
}
void
func(int arg)
{
```

```
printf("%d\n", arg);
}
```

14.2 Function Returning Pointers

It's possible to take the address of a function, too. And, similarly to arrays, functions decay to pointers when their names are used. So if you wanted the address of, say, strcpy, you could say either strcpy or &strcpy. (&strcpy[0] won't work for obvious reasons.)

When you call a function, you use an operator called the function call operator. The function call operator takes a function pointer on its left side.

In this example, we pass dst and src as the arguments on the interior, and strcpy as the function (that is, the function pointer) to be called:

```
enum { str_length = 18U }; Remember the NUL terminator!
char src[str_length] = "This is a string.", dst[str_length];
strcpy(dst, src);
```

The function call operator in action (notice the function pointer on the left side).

There's a special syntax for declaring variables whose type is a function pointer.

```
char *strcpy(char *dst, const char *src); An ordinary function declaration, for reference
char *(*strcpy_ptr)(char *dst, const char *src);
//Pointer to strcpy-like function
strcpy_ptr =  strcpy;
strcpy_ptr = &strcpy; //This works too
strcpy_ptr = &strcpy[0]; //But not this
```

Note the parentheses around *strcpy_ptr in the above declaration. These separate the asterisk indicating return type (char *) from the asterisk indicating the pointer level of the variable (*strcpy_ptr — one level, pointer to function).

Also, just like in a regular function declaration, the parameter names are optional:

```
char *(*strcpy_ptr_noparams)(char *, const char *) = strcpy_ptr;
//Parameter names removed — still the same type
```

The type of the pointer to strcpy is char *(*)(char *, const char *); you may notice that this is the declaration from above, minus the variable name. You would use this in a cast. For example:

```
strcpy_ptr = (char *(*)(char *dst, const char *src))my_strcpy;
```

As you might expect, a pointer to a pointer to a function has two asterisks inside of the parentheses:

```
char *(**strcpy_ptr_ptr)(char *, const char *) = &strcpy_ptr;
```

We can have an array of function-pointers:

```
char *(*strcpies[3])(char *, const char *) = { strcpy, strcpy, strcpy };
char *(*strcpies[])(char *, const char *) = { strcpy, strcpy, strcpy };
//Array size is optional, same as ever

strcpies[0](dst, src);
```

Here's a pathological declaration, taken from the C99 standard. “[This declaration] declares a function f with no parameters returning an int, a function fip with no parameter specification returning a pointer to an int, and a pointerpfi to a function with no parameter specification returning an int.”

```
int f(void), *fip(), (*pfi)();
```

In other words, the above is equivalent to the following three declarations:

```
int f(void);
int *fip(); //Function returning int pointer
int (*pfi)(); //Pointer to function returning int
```

In order to explain this, We are going to summarize all the declaration syntax you've learned so far. First, declaring a pointer variable:

```
char *ptr;
```

This declaration tells us the pointer type (char), pointer level (*), and variable name (ptr). And the latter two can go into parentheses:

```
char (*ptr);
```

What happens if we replace the variable name in the first declaration with a name followed by a set of parameters?

```
char *strcpy(char *dst, const char *src);
```

Huh. A function declaration.

But we also removed the * indicating pointer level — remember that the * in this function declaration is part of the return type of the function. So if we add the pointer-level asterisk back (using the parentheses):

```
char *(*strcpy_ptr)(char *dst, const char *src);
```

A function is pointer variable. If this is a variable, and the first declaration was also a variable, can we not replace the variable name in THIS declaration with a name and a set of parameters? Yes we can and the result is the declaration of a function that returns a function pointer:

```
char *(*get_strcpy_ptr(void))(char *dst, const char *src);
```

Remember that the type of a pointer to a function taking no arguments and returning int is int (*)(void). So the type returned by this function is char *(*)(char *, const char *) (with, again, the inner * indicating a pointer, and the outer * being part of the return type of the pointed-to function). You may remember that that is also the type of strcpy_ptr.

So this function, which is called with no parameters, returns a pointer to a strcpy-like function:

```
strcpy_ptr = get_strcpy_ptr();
```

Because function pointer syntax is so mind-bending, most developers use typedefs to abstract them:

```
typedef char *(*strcpy_funcptr)(char *, const char *);

strcpy_funcptr strcpy_ptr = strcpy;
strcpy_funcptr get_strcpy_ptr(void);
```

14.3 Static and Dynamic Memory Allocation

Static Allocation means, that the memory for your variables is automatically allocated, either on the *Stack*or in other sections of your program. You do not have to reserve extra memory using them, but on the other hand, have also no control over the lifetime of this memory. E.g: a variable in a function, is only there until the function finishes.

```
void func() {
    int i; /* `i` only exists during `func` */
}
```

Dynamic memory allocation is a bit different. You now control the exact size and the lifetime of these memory locations. If you don't free it, you'll run into memory leaks, which may cause your application to crash, since it, at some point cannot allocate more memory.

```
int* func() {
    int* mem = malloc(1024);
    return mem;
}

int* mem = func(); /* still accessible */
```

In the upper example, the allocated memory is still valid and accessible, even though the function terminated. When you are done with the memory, you have to free it:

```
free(mem);
```

Difference between Static and Dynamic Memory Allocation

STATIC MEMORY ALLOCATION	DYNAMIC MEMORY ALLOCATION
Memory is allocated before the execution of the program begins. (During Compilation)	Memory is allocated during the execution of the program.
No memory allocation or deallocation actions are performed during Execution.	Memory Bindings are established and destroyed during the Execution.
Variables remain permanently allocated.	Allocated only when program unit is active.
Implemented using stacks and heaps.	Implemented using data segments.
Pointer is needed to accessing variables.	No need of Dynamically allocated pointers.
Faster execution than Dynamic.	Slower execution than static.
More memory Space required.	Less Memory space required.

14.4 DMA functions

Malloc Function

The malloc function takes one argument i.e. the number of bytes to be allocated. The syntax of the function is

void * malloc (size_t size) ;

It returns a void pointer to the starting of the chunk of the memory allocated from the heap in case of the availability of that memory. If the memory is not available or is fragmented (not in a sequence), malloc will return a NULL pointer. While using malloc, we normally make use sizeof operator and a call to malloc function is written in the following way.

malloc (1000 * sizeof(int)) ;

Here * is multiplication operator and not a dereference operator of a pointer.

In the above call, we request for 1000 spaces in the memory each of the size, which can accommodate an integer. The `sizeof(int)' means the number of bytes, occupied by an integer in the memory. Thus the above statement will allocate memory in bytes for 1000 integers. If on our machine, an integer occupies 4 bytes. A 1000 * 4 (4000) bytes of memory will be allocated. Similarly if we want memory for 1000 characters or 1000 floats, the malloc function will be written as

malloc (1000 * sizeof(char)) ;

and malloc (1000 * sizeof(float)) ;

respectively for characters and floats.

So in general, the syntax of malloc will be.

malloc (n * sizeof (datatype)) ;

where `n' represents the numbers of required data type. The malloc function differs from calloc in the way that the space allocated by malloc is not initialized and contains any values initially.

Let's say we have a problem that states `Calculate the average age of the students in your class.' The program prompts the user to enter the number of students in the class and also allows the user to enter the ages of the students. Afterwards, it calculates the average age.

Now in the program, we will use dynamic memory. At first, we will ask the user `How many students are in the class? The user enters the number of students. Let's suppose, the number is 35. This number is stored in a variable say `numStuds'. We will get the age of students in whole numbers so the data type to store age will be int. Now we require a memory space where we can store a number of integers equal to the value stored in numStuds. We will use a pointer to a memory area instead of an array. So we declare a pointer to an integer. Suppose we call it iptr. Now we make a call to calloc or malloc function. Both of them are valid. So we write the following statement

iptr = (int *) malloc (numStuds * sizeof (int)) ;

Now we immediately check iptr whether it has NULL value. If the value of iptr is not NULL, it will mean that we have allocated the memory successfully. Now we write a loop to get the ages of the students and store these to the memory, got through malloc function. We write these values of ages to the memory by using the pointer iptr with pointer arithmetic. A second pointer

say sptr can be used for pointer arithmetic so that the original pointer iptr should remain pointing to the starting position of the memory. Now simply by incrementing the pointer sptr, we get the ages of students and store them in the memory. Later, we perform other calculations and display the average age on the screen. The advantage of this (using malloc) is that there is no memory wastage as there is no need of declaring an array of 50 or 100 students first and keep the ages of 30 or 35 students in that array. By using dynamic memory, we accurately use the memory that is required.

Calloc Function

The syntax of the calloc function is as follows.

void *calloc (size_t n, size_t el_size)

This function takes two arguments. The first argument is the required space in terms of numbers while the second one is the size of the space. So we can say that we require n elements of type int. We have read a function sizeof. This is useful in the cases where we want to write a code that is independent of the particular machines that we are runningon. So if we write like

void calloc(1000, sizeof(int))

It will return a memory chunk from the heap of 1000 integers. By using sizeof (int) we are not concerned with the size of the integer on our machine whether it is of 4 bytes or 8 bytes. We will get automatically a chunk that can hold 1000 integers. The said memory will be returned if a chunk of similar size is available on the heap. Secondly, this memory should be available on heap in continuous space. It should not be in split blocks. The function returns a pointer to the starting point of the allocated memory. It means that if starting point of the chunk is gotten, then the remaining memory is available in a sequence from end to end. There cannot be gaps and holes between them. It should be a single block. Now we have to see what happens when either we ask for too much memory at a time of non-availability of enough memory on the heap or we ask for memory that is available on the heap , but not available as a single chunk?. In this case,
the call to calloc will fail. When a call to memory allocation functions fails, it returns a NULL pointer. It is important to understand that whenever we call a memory allocation function, it is necessary to check whether the value of the pointer returned by the function is NULL or not. If it is not NULL, we have the said memory. If it is NULL, it will mean that either we have asked for too much memory or a single chunk of that size is not available on the heap.

Suppose, we want to use the memory got through calloc function as an integer block We have to cast it before using. It will be written as the following statement.

(int *) calloc (1000, sizeof (int)) ;

Another advantage of calloc is that whenever we allocate memory by using it. The memory is automatically initialized to zeros. In other words it is set to zeros. For casting we normally declare a pointer of type which we are going to use. For example, if we are going to use the memory for integers. We declare an integer pointer like int *iptr; Then when we allocate memory through calloc, we write it as

iptr = (int *) calloc (1000, sizeof(int)) ;

(int *) means cast the pointer returned by calloc to an integer pointer and we hold it in the declared integer pointer iptr. Now iptr is a pointer to an integer that can be used to manipulate all the integers in that memory space. You should keep in mind that after the above statement, a NULL check of memory allocation is necessary. An `if statement' can be used to check the success of the memory allocation. It can be written as under if (iptr == NULL)

any error message or code to handle error .

If a NULL is returned by the calloc, it should be treated according to the logic so that the program can exit safely and it should not be crashed.

The next function used for allocating memory is malloc.

free ():

Whenever we get a benefit, there is always a cost. The dynamic memory allocation has also a cost. Here the cost is incurred in terms of memory management. The programmer itself has to manage the memory. It is the programmer's responsibility that when the memory allocated is no longer in use, it should be freed to make it a part of heap again. This will help make it available for the other programs. As long as the memory is allocated for a program, it is not available to other programs for use. So it is programmer's responsibility to free the memory when the program has done with it. To ensure it, we use a function free. This function returns the allocated memory, got through calloc or malloc, back to the heap. The argument that is passed to this function is the pointer through which we have allocated the memory earlier. In our program, we write

free (iptr) ;

By this function, we call the memory allocated by malloc and pointed by the pointer iptr is freed. It goes back to the heap and becomes available for use by other programs. It is very important to note that whenever we allocate memory from the heap by using calloc or malloc, it is our responsibility to free the memory when we have done with it.

Following is the code of the program discussed above.

```
//This program calculates the average age of a class of students
//using dynamic memory allocation
#include <iostream.h>
#include <stdlib.h>
#include <string.h>
int main( )
{
int numStuds, i, totalAge, *iptr, *sptr;
cout <<"How many students are in the class ?  " ;
cin >> numStuds;
// get the starting address of the allocated memory in pointer iptr
iptr = (int *) malloc(numStuds * sizeof(int));
//check for the success of memory allocation
if (iptr == NULL)
{
cout << "Unable to allocat space for " << numStuds << " students\n";
return 1;
// A nonzero return is usually used to indicate an error
}
sptr = iptr ; //sptr will be used for pointer arithmetic/manipulation
i=1;
totalAge = 0 ;
//use a loop to get the ages of students
for (i = 1 ; i <= numStuds ; i ++)
{
cout << "Enter the age of student " << i << " = " ;
cin >> *sptr ;
totalAge = totalAge + *sptr ;
sptr ++ ;
}
cout << "The average age of the class is " << totalAge / numStuds <<
endl;
//now free the allocated memory, that was pointed by iptr
free (iptr) ;
sptr = NULL ;
```

```
}
Following is a sample output of the program.
How many students are in the class ? 3
Enter the age of student 1 = 12
Enter the age of student 2 = 13
Enter the age of student 3 = 14
The average age of the class is 13
```

Realloc Function:

Sometimes, we have allocated a memory space for our use by malloc function. But we see later that some additional memory is required. For example, in the previous example, where (for example) after allocating a memory for 35 students, we wanted to add one more student. So we need same type of memory to store the new entry. Now the question arises `Is there a way to increase the size of already allocated memory chunk? Can the same chunk be increased or not? The answer is yes. In such situations, we can reallocate the same memory with a new size according to our requirement. The function that reallocates the memory is realloc. The syntax of realloc is given below.

void realloc (void * ptr, size_t size) ;

This function enlarges the space allocated to ptr (in some previous call of calloc or malloc) to a (new) size in bytes. This function receives two arguments. First is the pointer that is pointing to the original memory allocated already by using calloc or malloc. The second is the size of the memory which is a new size other than the previous size. Suppose we have allocated a memory for 20 integers by the following call of malloc and a pointer iptr points to the allocated memory.

(iptr *) malloc (20 * sizeof(int)) ;

Now we want to reallocate the memory so that we can store 25 integers. We can reallocate the same memory by the following call of realloc.

realloc (iptr, 25 * sizeof(int)) ;

There are two scenarios to ascertain the success of `realloc'. The first is that it extends the current location if possible. It is possible only if there is a memory space available contiguous to the previously allocated memory. In this way the value of the pointer iptr is the same that means it is pointing to the same starting position, but now the memory is more than the previous one. The second way is that if such contiguous memory is not available in the current location, realloc goes back to the heap and looks for a contiguous block of memory for the requested size. Thus it

will allocate a new memory and copy the contents of the previous memory in this new allocated memory. Moreover it will set the value of the pointer iptr to the starting position of this memory. Thus iptr is now pointing to a new memory location. The original memory is returned to the heap. In a way, we are handling dynamic arrays. The size of the array can be increased during the execution. There is another side of the picture. It may happen that we have stored the original value of iptr in some other pointer say sptr. Afterwards, we are manipulating the data through both the pointers. Then ,we use realloc for the pointer iptr. The realloc does not find contiguous memory with the original and allocates a new block of memory and points it by the pointer iptr. The original memory no longer exists now. The pointer iptr is valid now as it is pointing to the starting position of the new memory. But the other pointer sptr is no longer valid. It is pointing to an invalid memory that has been freed and may be is being used some other program. If we manipulate this pointer, very strange things can happen. The program may crash or the computer may halt. We don't know what can happen. Now it becomes the programmer's responsibility again to make it sure that after realloc, the pointer(s) that have the value of the original pointer have been updated. It is also important to check the pointer returned by realloc for NULL value. If realloc fails, that means that it cannot allocate the memory. In this case, it returns a NULL value. After checking NULL value, (if realloc is successful), we should update the pointer that was referencing the same area of the memory.
We have noticed while getting powers of dynamic memory allocation, we face some dangerous things along with it. These are real problems. Now we will talk about the common errors that can happen with the memory allocation.

Sizeof( )

The sizeof operator gives the amount of storage, in bytes, required to store an object of the type of the operand. This operator allows you to avoid specifying machine-dependent data sizes in your programs.

sizeof unary-expression
sizeof (type-name
If an unsized array is the last element of a structure, the sizeof operator returns the size of the structure without the array.

```
buffer = calloc(100, sizeof (int) );
```

This example uses the sizeof operator to pass the size of an int, which varies among machines, as an argument to a run-time function named calloc. The value returned by the function is stored in buffer.

```
static char *strings[] ={
"this is string one",
"this is string two",
"this is string three",
};
const int string_no = ( sizeof strings ) / ( sizeof strings[0] );
```

In this example, strings is an array of pointers to char. The number of pointers is the number of elements in the array, but is not specified. It is easy to determine the number of pointers by using the sizeof operator to calculate the number of elements in the array. The const integer value string_no is initialized to this number. Because it is a const value, string_no cannot be modified.

14.5 Bitwise Operators

When you learn to program in a high-level language like C (although C is fairly low-level, as high-level languages go), the idea is to avoid worrying too much about the hardware. You want the ability to represent mathematical abstractions, such as sets, etc. and have high level language features like threads, higher-order functions, exceptions.

High-level languages, for the most part, try to make you as unaware of the hardware as possible. Clearly, this isn't entirely true, because efficiency is still a major consideration for some programming languages.

C, in particular, was created to make it easier to write operating systems. Rather than write UNIX in assembly, which is slow process (because assembly code is tedious to write), and not very portable (because assembly is specific to an ISA), the goal was to have a language that provided good control-flow, some abstractions (structures, function calls), and could be efficiently compiled and run quickly.

Writing operating systems requires the manipulation of data at addresses, and this requires manipulating individual bits or groups of bits.

That's where two sets of operators are useful: *bitwise* operators and *bitshift* operators.

You can find these operators in C, C++, and Java (and presumably C#, since it's basically Java). Bitwise operators allow you to read and manipulate bits in variables of certain types.

Even though such features are available in C, they aren't often taught in an introductory level programming course. That's because intro level courses prefer to emphasize abstraction. With many departments using Java, there's a trend to increase what's abstract, and not get into the representation.

For example, some languages have support for stacks, queues, hashtables, and so forth. These "canned" data structures are meant to provide you, the programmer, with objects that perform certain tasks, while relieving you of the tedium and detail of understanding how the data is represented.

Nevertheless, if you intend to do some work in systems programming, or other forms of low-level coding (operating systems, device drivers, socket programming, network programming), knowing how to access and manipulate bits is important.

Generic Bitwise Operations

Bitwise operators only work on a limited number of types: int and char. This seems restrictive--and it is restrictive, but it turns out we can gain some flexibility by doing some C "tricks".

It turns out there's more than one kind of int. In particular, there's unsigned int, there's short int, there's long int, and then unsigned versions of those ints.

The "C" language does not specify the difference between a short int, an int and a long int, except to state that:

sizeof(short int) <= sizeof(int) <= sizeof(long)

You will find that these sizes vary from ISA to ISA, and possibly even compiler to compiler. The sizes do not have to be distinct. That means all three sizes could be the same, or two of three could be the same, provided that the above restrictions are held.

Bitwise operators fall into two categories: binary bitwise operators and unary bitwise operators. Binary operators take two arguments, while unary operators only take one.

We're going to discuss binary operators first, and we'll do in the context of a fake binary operator called @. Thus, we write x @ y to perform bitwise @ on x and y.

Bitwise operators, like arithmetic operators, do not change the value of the arguments. Instead, a temporary value is created. This can then be assigned to a variable. For example, when you write x + y, neither x nor y have their value changed.

Let's assume that we are doing bitwise @ on two unsigned int variables. Let's assume that unsigned ints use 32 bits of memory. We define two variables X and Y where

$X = x_{31}x_{30}....x_0$

$Y = y_{31}y_{30}....y_0$

We want to be able to refer to each bit of X and Y, which we do so by writing out the bits by writing the variable name in lowercase and adding the appropriate subscript for thie bit number.

The subscripts follow the convention we've used so far. The least significant bit has a subscript of 0, and is the rightmost bit. The most significant bit has a subscript of N-1 (for N bits), and is the leftmost bit. In this case N = 32 so the MSb has a subscript of 31.

Let's define @ to be the @ operation performed on two individual bits. @ is performed on all 32 bits simultaneously.

Furthermore, let's call the temporary value created, T. Normally, this temporary value does not have a name. It is generated while the program is running, and used in other computations. Thus, when you write c = a + b, a temporary value is created for the sum a + b, and this temporary value then gets copied into c.

Of course, for efficiency, the temporary value might not be generated, and the value written directly to c. For more complex statements, such as c = (a + b) - d, temporary values are often created to store intermediate results. These temporary values are created by the runtime system, and are not given variable names.

We define Z = X @ Y as:

$z_i = x_i @^1 y_i$, for 0 <= i <= N - 1

In words, to compute the ith bit of Z, you should perform the operation on the ith bit of X and i^{th} bit of Y.

Bitwise AND:

This makes more sense if we apply this to a specific operator. In C/C++/Java, the & operator is bitwise AND. The following is a chart that defines $\&^1$, defining AND on individual bits.

x_i	y_i	$x_i \&^1 y_i$
0	0	0
0	1	0
1	0	0
1	1	1

We can do an example of bitwise &. It's easiest to do this on 4 bit numbers, however.

Variable	b_3	b_2	b_1	b_0
x	1	1	0	0
y	1	0	1	0
z = x & y	1	0	0	0

Bitwise OR

The | operator is bitwise OR (it's a single vertical bar). The following is a chart that defines $|^1$, defining OR on individual bits.

x_i	y_i	$x_i \|^1 y_i$
0	0	0
0	1	1
1	0	1
1	1	1

We can do an example of bitwise |. It's easiest to do this on 4 bit numbers, however.

Variable	b_3	b_2	b_1	b_0
x	1	1	0	0
y	1	0	1	0
z = x | y	1	1	1	0

Bitwise XOR

The ^ operator is bitwise XOR. The usual bitwise OR operator is *inclusive* OR. XOR is true only if exactly one of the two bits is true. The XOR operation is quite interesting, but we defer talking about the interesting things you can do with XOR until the next set of notes.

The following is a chart that defines $\hat{}^1$, defining XOR on individual bits.

x_i	y_i	x_i ^y_i
0	0	0
0	1	1

1	0	1
1	1	0

We can do an example of bitwise ^. It's easiest to do this on 4 bit numbers, however.

Variable	b_3	b_2	b_1	b_0
x	1	1	0	0
y	1	0	1	0
z = x ^ y	0	1	1	0

Bitwise NOT

There's only one unary bitwise operator, and that's bitwise NOT. Bitwise NOT flips all of the bits.

There's not that much to say about it, other than it's not the same operation as unary minus.

The following is a chart that defines $\sim^1$, defining NOT on an individual bit.

x_i	$\sim^1 x_i$
0	1
1	0

We can do an example of bitwise ~. It's easiest to do this on 4 bit numbers (although only 2 bits are necessary to show the concept).

Variable	b_3	b_2	b_1	b_0
x	1	1	0	0
z = ~x	0	0	1	1

14.6 Preprocessor Directives

Preprocessor directives, such as #define and #ifdef, are typically used to make source programs easy to change and easy to compile in different execution environments. Directives in the source file tell the preprocessor to perform specific actions. For example, the preprocessor can replace tokens in the text, insert the contents of other files into the source file, or suppress compilation of part of the file by removing sections of text. Preprocessor lines are recognized and carried out before macro expansion. Therefore, if a macro expands into something that looks like a preprocessor command, that command is not recognized by the preprocessor.

Preprocessor statements use the same character set as source file statements, with the exception that escape sequences are not supported. The character set used in preprocessor statements is the same as the execution character set. The preprocessor also recognizes negative character values.

The preprocessor recognizes the following directives:

#define	#error	#import	#undef
#elif	#if	#include	#using
#else	#ifdef	#line	
#endif	#ifndef	#pragma	

The number sign (#) must be the first nonwhite-space character on the line containing the directive; white-space characters can appear between the number sign and the first letter of the directive. Some directives include arguments or values. Any text that follows a directive (except an argument or value that is part of the directive) must be preceded by the single-line comment delimiter (//) or enclosed in comment delimiters (/* */). Lines containing preprocessor directives can be continued by immediately preceding the end-of-line marker with a backslash (\).

Preprocessor directives can appear anywhere in a source file, but they apply only to the remainder of the source file.

14.7 Summary

At the end of this unit we have learnt pointer concept in depth, we dealt with advance concept on pointers such as passing pointer to function and returning pointer from function. In the later stage we focused on DMA functions covering malloc, calloc, sizeof()_ free functions with an syntax to use them and we have seen many examples. In the last section of the unit we have list out various preprocess directive which are available in C language and also we have seen the bitwise operators and their operations.

14.8 Keywords

Pointer to function, DMA function, malloc, calloc, free. Sizeof(0, bitwise

14.9 Questions

1. How can pointers be used as arguments for function? Demonstrate the stages with suitable code as an example?
2. What are DMA functions? Explain all available DMA function in C language?
3. Why we require memory related functions in programming? Give reason?
4. What are preprocess directives? Why we require them?
5. Explain bitwise operators available in C?

14.10 Reference

"Common Pointer Pitfalls" by Dave Marshall

C Programming: A Modern Approach by K.N. King

C Programming in 12 Easy Lessons by Greg Perry

C for Dummies Vol. II by Dan Gookin

Structure

15.0 Objectives

15.1 Introduction

15.2 Structure Variable Declaration

15.3 Accessing Members of Structure

15.4 Nested Structure

15.5 Structure and Functions

15.6 Union-Defining Structure

15.7 Summary

15.8 Key words

15.9 Questions

15.10 References

15.0 Objectives

At the end of this unit you will be able to

- Understand advance concept on pointers
- Sketch out the working principle of pointer to function as argument
- Explain various memory related function
- List out the various preprocess directives and bitwise operators

15.1 Structure

Structure is the collection of variables of different types under a single name for better handling. For example: You want to store the information about person about his/her name, citizenship number and salary. You can create this information separately but, better approach will be collection of this information under single name because all these information are related to person.

Structure Definition in C

Keyword struct is used for creating a structure.

Syntax of structure

```
struct structure_name
{
    data_type member1;
    data_type member2;
    .
    .
    data_type memebern;
};
```

We can create the structure as mentioned in above example as:

```
struct person
{
    char name[50];
    int cit_no;
    float salary;
};
```

This declaration above creates the derived data type struct person, i.e, a user-defined type.

15.2 Structure Variable Declaration

When a structure is defined, it creates a user-defined type but, no storage is allocated. You can use structure variable using tag name in any part of program. For example:

```
struct person
{
    char name[50];
    int cit_no;
    float salary;
};
struct person p1, p2, p[20];
Another way of creating sturcture variable is:
struct person
{
    char name[50];
    int cit_no;
    float salary;
}p1 ,p2 ,p[20];
```

In the above cases, 2 variables p1, p2 and array p having 20 elements of type **struct person** are created.

15.3 Accessing members of a Structure

There are two types of operators used for accessing members of a structure.

- Member operator(.)
- Structure pointer operator(->) (will be discussed in respective chapter of structure and pointers)

Any member of a structure can be accessed as: structure_variable_name.member_name

Suppose, if we want to access salary for variable p2. Then, it can be accessed as: p2.salary

Consider following Example of structure

```
#include <stdio.h>
struct Distance{
    int feet;
    float inch;
}d1,d2,sum;
int main(){
    printf("1st distance\n");
    printf("Enter feet: ");
    scanf("%d",&d1.feet);  //input of feet structure for d1
    printf("Enter inch: ");
    scanf("%f",&d1.inch);  //input of inch structure for d1
    printf("2nd distance\n");
    printf("Enter feet: ");
    scanf("%d",&d2.feet);  //input of feet for structure variable d2
    printf("Enter inch: ");
    scanf("%f",&d2.inch);  //input of inch for structure variable d2
    sum.feet=d1.feet+d2.feet;
    sum.inch=d1.inch+d2.inch;
    if (sum.inch>12){  //If inch is greater than 12, changing it to
feet.
        ++sum.feet;
        sum.inch=sum.inch-12;
    }
    printf("Sum of distances=%d\'-%.1f\"",sum.feet,sum.inch);
//printing sum of distance d1 and d2
    return 0;
}
```

Structure is the collection of variables of different types under a single name for better handling. For example: You want to store the information about person about his/her name, citizenship number and salary. You can create this information separately but, better approach will be collection of these information under single name because all these information are related to person.

Keyword typedef while using structure

Programmers are generally use typedef while using structure in C language. For example:

```
typedef struct complex{
  int imag;
  float real;
```

```
}comp;

comp c1,c2;
```

Here, typedef keyword is used in creating a type comp(which is same type as struct complex). Then, two structure variables c1 and c2 are created by this comp type.

15.4 Nested Structure

Structures within structures

Structures can be nested within other structures in C programming.

```
struct complex
{
 int imag_value;
 float real_value;
};
struct number{
   struct complex c1;
   int real;
}n1,n2;
```

Suppose you want to access imag_value for n2 structure variable then, structure member n1.c1.imag_value is used.

*typedef struct complex **complex**; complex_no c1,c2,c3;*

Here, typdef is used to create a type complex. This type complex is then used in declaring variables c1, c2 and c3.

15.5 Structure and Function

In C, structure can be passed to functions by two methods:

- Passing by value (passing actual value as argument)
- Passing by reference (passing address of an argument)

Passing structure by value

A structure variable can be passed to the function as an argument as normal variable. If structure is passed by value, change made in structure variable in function definition does not reflect in original structure variable in calling function.

For example consider the C program to create a structure student, containing name and roll. Ask user the name and roll of a student in main function. Pass this structure to a function and display the information in that function.

```
#include <stdio.h>
struct student{
    char name[50];
    int roll;
};
void Display(struct student stu);
/*function prototype should be below to the structure declaration
otherwise compiler shows error */
int main(){
    struct student s1;
1       printf("Enter student's name: ");
    scanf("%s",&s1.name);
    printf("Enter roll number:");
    scanf("%d",&s1.roll);
    Display(s1);   //passing structure variable s1 as argument
    return 0;
}
void Display(struct student stu){
  printf("Output\nName: %s",stu.name);
  printf("\nRoll: %d",stu.roll);
}
```

Output

Enter student's name: Kevin Amla

Enter roll number: 149

Output

Name: Kevin Amla

Roll: 149

Passing structure by reference

The address location of structure variable is passed to function while passing it by reference. If structure is passed by reference, change made in structure variable in function definition reflects in original structure variable in the calling function.

Write a C program to add two distances (feet-inch system) entered by user. To solve this program, make a structure. Pass two structure variable (containing distance in feet and inch) to add function by reference and display the result in main function without returning it.

```
#include <stdio.h>
struct distance{
    int feet;
    float inch;
};
void Add(struct distance d1,struct distance d2, struct distance *d3);

//function prototype
int main()
{
    struct distance dist1, dist2, dist3;
    printf("First distance\n");
    printf("Enter feet: ");
    scanf("%d",&dist1.feet);
    printf("Enter inch: ");
    scanf("%f",&dist1.inch);
    printf("Second distance\n");
    printf("Enter feet: ");
    scanf("%d",&dist2.feet);
    printf("Enter inch: ");
    scanf("%f",&dist2.inch);
    Add(dist1, dist2, &dist3);

/*passing structure variables dist1 and dist2 by value whereas
 passing structure variable dist3 by reference */

    printf("\nSum of distances = %d\'-%.1f\"",dist3.feet, dist3.inch);
    return 0;
}
void Add(struct distance d1,struct distance d2, struct distance *d3)
{
     //Adding distances d1 and d2 and storing it in d3
     d3->feet=d1.feet+d2.feet;
     d3->inch=d1.inch+d2.inch;
     if (d3->inch>=12) {
```

```
    //if inch is greater or equal to 12, converting it to feet.
            d3->inch-=12;
            ++d3->feet;
        }
    }
```

Output

First distance

Enter feet: 12

Enter inch: 6.8

Second distance

Enter feet: 5

Enter inch: 7.5

Sum of distances = 18'-2.3"

In this program, structure variables dist1 and dist2 are passed by value (because value of dist1 and dist2 does not need to be displayed in main function) and dist3 is passed by reference, i.e, address of dist3 (&dist3) is passed as an argument. Thus, the structure pointer variable d3 points to the address of dist3. If any change is made in d3 variable, effect of it is seed in dist3 variable in main function.

15.6 Union-Defining Structure

Unions are quite similar to the structures in C. Union is also a derived type as structure. Union can be defined in same manner as structures just the keyword used in defining union in **union** where keyword used in defining structure was **struct.**

```
union car{
  char name[50];
  int price;
};
```

Union variables can be created in similar manner as structure variable.

```
union car{
  char name[50];
  int price;
}c1, c2, *c3;
```

```
OR;

union car{
  char name[50];
  int price;
};
union car c1, c2, *c3;
```

In both cases union variables c1, c2 and union pointer variable c3 of type **union car** is created.

Accessing members of an union

Again, the member of unions can be accessed in similar manner as that structure. Suppose, we you want to access price for union variable c1 in above example, it can be accessed as c1.price. If you want to access price for union pointer variable c3, it can be accessed as (*c3).price or as c3->price.

Difference between union and structure

Though unions are similar to structure in so many ways, the difference between them is crucial to understand. This can be demonstrated by an example.

```
#include <stdio.h>
union job {         //defining a union
   char name[32];
   float salary;
   int worker_no;
}u;
struct job1 {
   char name[32];
   float salary;
   int worker_no;
}s;
int main(){
   printf("size of union = %d",sizeof(u));
   printf("\nsize of structure = %d", sizeof(s));
```

```
    return 0;
}
```

Output

size of union = 32

size of structure = 40

There is difference of memory allocation between union and structure as suggested in above example. The amount of memory required to store a structure variables is the sum of memory size of all members.

But, the memory required to store a union variable is the memory of largest element of union.

Difference between structure and union

As you know, all members of structure can be accessed at any time. But, only one member of union can be accessed at a time in case of union and other members will contain garbage value.

```
#include <stdio.h>
union job {
   char name[32];
   float salary;
   int worker_no;
}u;
int main(){
   printf("Enter name:\n");
   scanf("%s",&u.name);
   printf("Enter salary: \n");
   scanf("%f",&u.salary);
   printf("Displaying\nName :%s\n",u.name);
   printf("Salary: %.1f",u.salary);
   return 0;
}
```

Output

```
Enter name
Hillary
Enter salary
1234.23
Displaying
Name: f%Bary
Salary: 1234.2
```

Note: You may get different garbage value of name.

When code in line no. 9 is executed, Hillary will be stored in u.name and other members of union will contain garbage value. When code all will executed, 1234.23 will be stored in u.salary and other members will contain garbage value. Thus in output, salary is printed accurately but, name displays some random string.

15.7 Summary

At the end this unit we have learnt the concepts about structure and unions. We have introduced structure through basic thing such as declaring a structure, accessing structure members. A suitable piece of code as an example taken and demonstrated how exactly the structure concepts works. In later sections we have touch upon advanced concepts on structure such as passing structure as a function argument and returning the value. In the last section of this unit we have introduced union defining function.

15.8 Keywords

struct, typedef, passing by reference

15.9 Questions

1. What is structure? How this is useful in programming?
2. How can structure be treated as a function argument?
3. Explain the concept of union defined functions?

4. Do survey and prepare report on how exactly the structure is different from class in object orient programming?

15.10 Reference

C Programming: A Modern Approach by K.N. King

C Programming in 12 Easy Lessons by Greg Perry

C for Dummies Vol. II by Dan Gookin

Unit 16

Structure

16.0 Objectives

16.1 Array of Structure

16.2 Structure Assignment

16.3 Structure as Function Argument

16.4 Pointers to Structure

16.5 Typedef

16.6 Unions

16.7 Bit fields

16.8 Summary

16.9 Key words

16.10 Questions

16.11 References

16.0 Objectives

At the end of this unit you will be able to

- Explain the use of structure
- Understand the pointers to structure
- Understand typedef, bitfields and unions in detial

16.1 Array of structure

It is possible to define a array of structures for example if we are maintaining information of all the students in the college and if 100 students are studying in the college. We need to use an array than single variables. We can define an array of structures as shown in the following example:

```
structure information
{
int id_no;
char name[20];
char address[20];
char combination[3];
int age;
}
student[100];
```

An array of structures can be assigned initial values just as any other array can.
Each element is a structure that must be assigned corresponding initial values as illustrated below.

```
#include< stdio.h >
{
struct info
{
int id_no;
char name[20];
char address[20];
char combination[3];
int age;
}
struct info std[100];
int I,n;
printf("Enter the number of students");
scanf("%d",&n);
printf(" Enter Id_no,name address combination age\m");
```

```
for(I=0;I < n;I++)
scanf(%d%s%s%s
%d”,&std[I].id_no,std[I].name,std[I].address,std[I].combinatio
n,&std[I].age);
printf(“\n Student information”);
for (I=0;I< n;I++)
printf(“%d%s%s%s%d\n”,”,std[I].id_no,std[I].name,std[I].address,std[I]
.combination,std[I].age);
}
```

16.2 Structure Assignment

The following assignment of a struct to another struct does what one might expect. It is not necessary to use memcpy() to make a duplicate of a struct type. The memory is already given and zeroed by just declaring a variable of that type regardless of member initialization. This should not be confused with the requirement of memory management when dealing with a pointer to a struct.

```
#include <stdio.h>

/* Define a type point to be a struct with integer members x, y */
typedef struct {
int    x;
int    y;
} point;

int main(int argc, char * argv[]) {

/* Define a variable p of type point, and initialize all its members inline!
*/
point p = {1,2};

/* Define a variable q of type point. Members are initialized with the
defaults for their derivative types such as 0. */
point q;

/* Assign the value of p to q, copies the member values from p into q. */
q = p;

/* Change the member x of q to have the value of 2 */
q.x = 2;
```

```
/* Demonstrate we have a copy and that they are now different. */
if (p.x != q.x) printf("The members are not equal! %d != %d", p.x, q.x);
}
```

16.3 Structures as Function Arguments

A structure can be passed as a function argument just like any other variable. This raises a few practical issues. Where we wish to modify the value of members of the structure, we must pass a pointer to that structure. This is just like passing a pointer to an int type argument whose value we wish to change.

If we are only interested in one member of a structure, it is probably simpler to just pass that member. This will make for a simpler function, which is easier to re-use. Of course if we wish to change the value of that member, we should pass a pointer to it.

When a structure is passed as an argument, each member of the structure is copied. This can prove expensive where structures are large or functions are called frequently. Passing and working with pointers to large structures may be more efficient in such cases.

```
#include <stdio.h>
void input (dob_st *);
int main ()
{
    typedef struct
    {
        int year;
        int month;
        int day;
    }
    dob_st;

    dob_st date;
    dob_st *p;
    p=&date;
```

```
    input (*p);
    printf("%02i.",p->day);
    printf("%02i.",p->month);
    printf("%i.",p->year);

    return 0;
}
void upis (dob_st *p)
{
    printf ("Date of birth:\nDay?\n");
    scanf ("%i",&(p->day));
    printf ("Month?\n");
    scanf ("%i",&(p->month));
    printf ("Year?\n");
    scanf ("%i",&(p->year));
}
```

16.4 Pointers to Structures

Pointers can be used to refer to a struct by its address. This is particularly useful for passing structs to a function by reference. The pointer can be dereferenced just like any other pointer in C — using the `*`operator. There is also a `->` operator in C which dereferences the pointer to struct (left operand) and then accesses the value of a member of the struct (right operand).

```
struct point {
int x;
int y;
} my_point;

struct point *p = &my_point;   /* To declare p as a pointer of type
struct point */
```

```
(*p).x = 8;                          /* To access the first member of the struct */
p->x = 8;                            /* Another way to access the first member of the struct */
```

16.5 Typedefs

Typedef is a keyword. The more theoretical information about the typedef is given in last unit. Readers are directed to refer theoretical aspects from last unit. In this section we will see programmatic aspects using typedef.

```
typedef struct {
int    account_number;
char   *first_name;
char   *last_name;
float  balance;
} account;
```

Different users have differing preferences; proponents usually claim:

- shorter to write
- can simplify more complex type definitions

As an example, consider a type that defines a pointer to a function that accepts pointers to struct types and returns a pointer to struct:

Without typedef:

```
struct point {
int    x;
int    y;
};
typedef struct point *(*point_compare_t) (struct point *a, struct point *b);
```

With typedef:

```
struct point {
int    x;
int    y;
};
```

```
typedef struct point  point_t;
typedef point_t *(*point_compare_t) (point_t *a, point_t *b);
```

If neither typedef were used in defining a function that takes a pointer to a type of the above function pointer, the following code would have to be used. Although valid, it becomes increasingly hard to read quickly.

```
/* Using the struct point type from before */

/* Define a function that returns a pointer to the biggest point,
using a function to do the comparison. */
struct point * biggest_point (size_t size, struct point *points,
struct point *(*point_compare) (struct point *a, struct point *b))
{
int i;
struct point *biggest = NULL;

for (i=0; i < size; i++) {
biggest = point_compare(biggest, points + i);
}
return biggest;
}
```

Now with all of the typedefs being used you should see that the complexity of the function signature is drastically reduced.

```
/* Using the struct point type from before and all of the typedefs */

/* Define a function that returns a pointer to the biggest point,
using a function to do the comparison. */
point_t * biggest_point(size_t size, point_t * points, point_compare_t
point_compare )
{
int i;
```

```
point_t * biggest = NULL;

for (i=0; i < size; i++) {
biggest = point_compare(biggest, points + i);
}
return biggest;
}
```

However, there are a handful of disadvantages in using them:

- They pollute the main namespace (see below), however this is easily overcome with prefixing a library name to the type name.
- Harder to figure out the aliased type (having to scan/grep through code), though most IDEs provide this lookup automatically.
- Typedefs do not really "hide" anything in a struct or union — members are still accessible (account.balance) (To really hide struct members, one needs to use 'incompletely-declared' structs.)

16.6 Unions

In computer science, a union is a value that may have any of several representations or formats; or a data structure that consists of a variable which may hold such a value. Some programming languages support special data types, called union types, to describe such values and variables. In other words, a union type definition will specify which of a number of permitted primitive types may be stored in its instances, e.g. "float or long integer". Contrast with a record, which could be defined to contain a float *and* an integer; whereas, in a union, there is only one value at a time.

In type theory, a union has a sum type.

Depending on the language and type, a union value may be used in some operations, such as assignment and comparison for equality, without knowing its specific type. Other operations may require that knowledge, either by some external information, or by the use of a tagged union.

Because of the limitations of their use, untagged unions are generally only provided in untyped languages or in an unsafe way (as in C). They have the advantage over simple tagged unions of not requiring space to store the tag.

The name "union" stems from the type's formal definition. If one sees a type as the set of all values that that type can take on, a union type is simply the mathematical union of its constituting types, since it can take on any value any of its fields can. Also, because a mathematical union discards duplicates, if more than one field of the union can take on a single common value, it is impossible to tell from the value alone which field was last written.

However, one useful programming function of unions is to map smaller data elements to larger ones for easier manipulation. A data structure, consisting for example of 4 bytes and a 32-bit integer, can form a union (in this case with an unsigned 64-bit integer) and thus be more readily accessed for purposes of comparison etc.

Like a structure, all of the members of a union are by default public. The keywords private, public, and protected may be used inside a struct or a union in exactly the same way they are used inside a class for defining private, public, and protected members.

16.7 Bit fields

A bit field is a common idiom used in computer programming to compactly store multiple logical values as a short series of bits where each of the single bits can be addressed separately. A bit field is most commonly used to represent integral types of known, fixed bit-width. A well-known usage of bit-fields is to represent single bit flags with each flag stored in a separate bit.
A bit field is distinguished from a bit array in that the latter is used to store a large set of bits indexed by integers and is often wider than any integral type supported by the language. Bit fields, on the other hand, typically fit within a machine word, and the denotation of bits is independent of their numerical index.

16.8 Summary

In this unit we have learnt some advance concepts on structure and unions such as passing to function, pointers to structure. We have given programmatic approach to demonstrate use of typedef. In later section we covered topic on unions and bitfields.

16.9 Keywords

Typedef, bitfield, unions, pointer to structure, structure as function argument

16.10 Questions

1. Explain the concept of structure as function arguments and pointer to structure with suitable program?
2. Explain the terms unions, bitfield, typedef

16.11 Reference

C Programming: A Modern Approach by K.N. King

C Programming in 12 Easy Lessons by Greg Perry

C for Dummies Vol. II by Dan Gookin