POST-GRADUATE DIPLOMA IN COMPUTER APPLICATIONS

PAPER : PGDCA - 4
PROBLEM SOLVING USING C

SECTION-A

LESSON NO:

1.1 : Introduction to C Language and Program Development

1.2 : Identifiers, Keywords, Data Types and Type Conversion

1.3 : Performing Input Output Operations

1.4 : Operators and Expressions

1.5 : Library Functions

1.6 : Conditional Control Statements

1.7 : Iterative Control Statements

1.8 : Functions

1.9 : Storage Classes

INTRODUCTION TO C LANGUAGE AND PROGRAM DEVELOPMENT

1.1 Introduction
1.2 Objectives
1.3 Origin of C Language
1.4 Types of languages
1.5 Features of C Language
1.6 Structure of a C Program
1.7 Stages in Program Development
1.8 Summary
1.9 Keywords
1.10 Short Answer Type Questions
1.11 Long Answer Type Questions
1.12 Suggested Readings

1.1 Introduction

Computer can understand language of 0s and 1s only, therefore, to interact with computer we should know the binary language, which is extremely difficult to learn and implement because one wrong combination of 0s and 1s can mean entirely different thing. Human beings, on the other hand can converse in their own language which is not directly understandable to computers. Therefore, some inter-mediate or translator is required to facilitate communication between humans and computers. These translators should be able to convert human language to computer language and vice-versa. Computer language and human language are two extremes in the hierarchy of languages. What we speak at times, may mean differently for different persons. The same word or sentence may have different meaning. This is called ambiguity. Ambiguous language constructs can not be correctly understood by computers. Therefore some language is required which is unambiguous, close to human language, whose words or sentences may be translated and represent precisely one meaning. Such languages are called programming languages. C is one such language, used extensively by programmers around the globe for writing computer programs, which are translated to binary language for computer's understanding and functioning.

1.2 Objectives

In this lesson we shall learn about the origin and features of the C language. Let us begin with a quick introduction to C. Our aim is to show the essential elements of the language in real programs, but without getting bogged down by details, rules, and

exceptions. At this point, we are not trying to be complete or even precise (save that the examples are meant to be correct). One needs to concentrate on the basics: variables and constants, arithmetic, control flow, functions and the rudiments of input and output for starting to learn programming in any language. Some topics like pointers, structures, most of C's rich set of operators, several control-flow statements, and the standard library have not been touched upon in this lesson to keep the learning of the balanced in terms of complexity.

1.3 Origin of C Language

The C language was developed in 1970s at Bell Laboratories by a system programmer named Dennis Ritchie. It derives its name from the fact that it is based on a language B, developed by Ken Thompson, another system programmer at Bell Laboratories.

C is a general-purpose programming language. It has been closely associated with the UNIX operating system where it was developed, since both the system and most of the programs that run on it are written in C. The language, however, is not tied to any one operating system or machine; and although it has been called a "system programming language" because it is useful for writing compilers and operating systems, it has been used equally well to write major programs in many different domains.

Many of the important ideas of C stem from the language BCPL (Basic Combined Programming Language), developed by Martin Richards. The influence of BCPL on C proceeded indirectly through the language B, which was written by Ken Thompson in 1970 for the first UNIX system on the DEC PDP-7.

BCPL and B are "typeless" languages, means data type of variable need not be declared in advance. By contrast, C provides a variety of data types. The fundamental types are characters, integers and floating point numbers of several sizes. In addition, there is a hierarchy of derived data types created with pointers, arrays, structures and unions. Expressions are formed from operators and operands; any expression, including an assignment or a function call, can be a statement. Pointers provide for machine-independent address arithmetic.

1.4 Types of languages

In order to understand the features of C programming language we need to know the various types of programming languages and their features. The programming language can be divided into three categories:

i. **Low level languages**
ii. **Middle Level Languages**
iii. **High level languages**

i. **Low level languages: These are the languages which are closer to the machine languages.** These languages permit the efficient use of the machine. But these languages are hardware dependent, means programs written in one language may not run on other machines. Moreover, learning these languages is not an easy task. For learning the language programmers need to

possess thorough knowledge of the hardware. These languages include machine languages and assembly languages, though assembly language is also referred to as middle level languages in some language literature.

ii. **Middle Level Languages: These are the languages which are neither close to machine language nor near to human understandable languages. These are actually symbolic languages**. Symbols representing operations are the main building blocks. The symbols are mnemonics or acronyms of the operations to be performed. Programs written in middle level languages are very cryptic. Assembly language falls under this category. High level languages are sometimes converted to middle level languages before further translation to low level languages. In C language we can even write code in middle level language but in that case the middle level language code is translated by the respected language compiler, which needs to be installed in the machine and C environment must be configured to use that compiler.

iii. **High level languages: These are the languages which are closer to the human understandable languages and include FORTRAN, BASIC, PASCAL, COBOL, PL/1 etc.** These languages have been designed for better programming efficiency and have the following advantages:

- The syntax is like English language. This enables the programmer to easily learn the language. Additionally, the programs written in these languages are easily understandable.
- The programs written in these languages are portable means the programs are not hardware dependent.

The C language stands between these two types of languages. It has some features of the low level languages along with the features of high level languages. The C language has capability of directly interacting with the hardware.

A program written in high level language needs to be compiled for checking syntax or grammatical errors. If the program is free of error then it is translated to low level language which can be directly executed on the computer.

1.5 Features of C Language

C provides the fundamental control-flow constructions required for well-structured programs: statement grouping, decision making (if-else), selecting one of a set of possible values (switch), looping with the termination test at the top (while, for) or at the bottom (do), and early loop exit (break).

C language is case sensitive. **'A' and 'a' mean differently in the language.** Functions may return values of basic types, structures, unions or pointers. Any function may be called recursively. Local variables are typically "automatic", or created a new with each invocation. Function definitions may not be nested but variables may be declared in a block-structured fashion. The functions of a C program may exist in separate source files that are compiled separately. Variables may be internal to a function, external

but known only within a single source file or visible to the entire program.

A preprocessing step performs macro substitution on program text, inclusion of other source files and conditional compilation.

C is a relatively "low-level" language. This characterization is not pejorative; it simply means that C deals with the same sort of objects that most computers do, namely characters, numbers and addresses. These may be combined and moved about with the arithmetic and logical operators implemented by real machines.

C provides no operations to deal directly with composite objects such as character strings, sets, lists or arrays. There are no operations that manipulate an entire array or string, although structures may be copied as a unit. The language does not define any storage allocation facility other than static definition and the stack discipline provided by the local variables of functions; there is no heap or garbage collection. Finally, C itself provides no input/output facilities; there are no READ or WRITE statements and no built-in file access methods. All of these higher-level mechanisms must be provided by explicitly called functions. Most C implementations have included a reasonably standard collection of such functions.

Similarly, C offers only straightforward, single-thread control flow: tests, loops, grouping, and subprograms, but not multiprogramming, parallel operations, synchronization or co-routines.

Although the absence of some of these features may seem like a grave deficiency, ("You mean I have to call a function to compare two character strings?"), keeping the language down to modest size has real benefits. Since C is relatively small, it can be described in small space and learned quickly. A programmer can reasonably expect to know and understand and indeed regularly use the entire language.

For many years, the definition of C was the reference manual in the first edition of *The C Programming Language*. In 1983, the American National Standards Institute (ANSI) established a committee to provide a modern, comprehensive definition of C. The resulting definition, the ANSI standard, or "ANSI C", was completed in late 1988. Most of the features of the standard are already supported by modern compilers.

The standard is based on the original reference manual. The language is relatively little changed; one of the goals of the standard was to make sure that most existing programs would remain valid or failing that, that compilers could produce warnings of new behavior.

For most programmers, the most important change is the new syntax for declaring and defining functions. A function declaration can now include a description of the arguments of the function; the definition syntax changes to match. This extra information makes it much easier for compilers to detect errors caused by mismatched arguments; in our experience, it is a very useful addition to the language.

There are other small-scale language changes. Structure assignment and enumerations, which had been widely available are now officially part of the language. Floating-point

computations may now be done in single precision. The properties of arithmetic, especially for unsigned types are clarified. The preprocessor is more elaborate. Most of these changes will have only minor effects on most programmers.

A second significant contribution of the standard is the definition of a library to accompany C. It specifies functions for accessing the operating system (for instance, to read and write files), formatted input and output, memory allocation, string manipulation and the like. A collection of standard headers provides uniform access to declarations of functions in data types. Programs that use this library to interact with a host system are assured of compatible behavior. Most of the library is closely modeled on the "standard I/O library" of the UNIX system. This library was described in the first edition and has been widely used on other systems as well. Again, most programmers will not see much change.

Because the data types and control structures provided by C are supported directly by most computers, the run-time library required to implement self-contained programs is tiny. The standard library functions are only called explicitly, so they can be avoided if they are not needed. Most can be written in C and except for the operating system details they conceal are themselves portable.

Although C matches the capabilities of many computers, it is independent of any particular machine architecture. With a little care it is easy to write portable programs, that is, programs that can be run without change on a variety of hardware. The standard makes portability issues explicit and prescribes a set of constants that characterize the machine on which the program is run.

C is not a strongly-typed language, but as it has evolved, its type-checking has been strengthened. The original definition of C frowned on, but permitted, the interchange of pointers and integers; this has long since been eliminated and the standard now requires the proper declarations and explicit conversions that had already been enforced by good compilers. The new function declarations are another step in this direction. Compilers will warn of most type errors and there is no automatic conversion of incompatible data types. Nevertheless, C retains the basic philosophy that programmers know what they are doing; it only requires that they state their intentions explicitly.

C, like any other language, has its blemishes. Some of the operators have the wrong precedence; some parts of the syntax could be better. Nonetheless, C has proven to be an extremely effective and expressive language for a wide variety of programming applications.

Self Check Exercise

Q: What are the high level languages?

Q: Write the various features of C language?

1.6 Structure of a C Program

The only way to learn a new programming language is by writing programs in it. The first program to write is the same for all languages:

Print the words

hello, world

This is a big hurdle; to leap over it you have to be able to create the program text somewhere, compile it successfully, load it, run it and find out where your output went. With these mechanical details mastered, everything else is comparatively easy.

In C, the program to print "hello, world" is

```
#include <stdio.h>
main()
{
        printf("hello, world\n");
}
```

To run this program first it needs to be compiled and translated to machine level language, which we shall discuss in the next section.After executing the program it will print

hello, world

Now, for some explanations about the program itself. A C program, whatever its size, consists of *functions* and *variables*. A function contains *statements* that specify the computing operations to be done and variables store values used during the computation. C functions are like the subroutines and functions in Fortran or the procedures and functions of Pascal. Our example is a function named main. Normally you are at liberty to give functions whatever names you like, but "main" is special - your program begins executing at the beginning of main. This means that every program must have a main somewhere.

main will usually call other functions to help perform its job, some that you wrote and others from libraries that are provided for you. The first line of the program,

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

tells the compiler to include information about the standard input/output library; the line appears at the beginning of many C source files. The standard library is described in following lessons.

One method of communicating data between functions is for the calling function to provide a list of values called *arguments*, to the function it calls. The parentheses after the function name surround the argument list. In this example, main is defined to be a function that expects no arguments, which is indicated by the empty list ( ).

```
#include <stdio.h>     include information about standard library
main()                 define a function called main that received
                       no argument values, statements of main are
{                      enclosed in braces main calls library
        printf("hello, world\n"); function printf to print this sequence
                                  of characters, \n represents the
                                  newline character
}
```

The first C program

The statements of a function are enclosed in braces { }. The function main contains only one statement,

printf("hello, world\n");

A function is called by naming it, followed by a parenthesized list of arguments, so this calls the function printf with the argument "hello, world\n". printf is a library function that prints output, in this case the string of characters between the quotes.

A sequence of characters in double quotes, like "hello, world\n", is called a *character string* or *string constant*. For the moment our only use of character strings will be as arguments for printf and other functions.

The sequence \n in the string is C notation for the *newline character*, which when printed advances the output to the left margin on the next line. If you leave out the \n (a worthwhile experiment), you will find that there is no line advance after the output is printed. You must use \n to include a newline character in the printf argument; if you try something like

```
printf("hello, world
");
```

the C compiler will produce an error message.

printf never supplies a newline character automatically, so several calls may be used to build up an output line in stages. Our first program could just as well have been written

```
#include <stdio.h>
main()
{
        printf("hello, ");
        printf("world");
        printf("\n");
}
```

to produce identical output.

Notice that \n represents only a single character. An *escape sequence* like \n provides a general and extensible mechanism for representing hard-to-type or invisible characters. Among the others that C provides are \t for tab, \b for backspace, \" for the double quote, \a for producing a bell sound and \\ for the backslash itself.

1.7 Stages in Program Development

The following are the various stages involved in development of computer program ready to be executed on the computer:

i. **Developing the program:** The first and fore most task for develop the program for a particular problem is to understand the problem in hand to be solved. Analysis of the problem will reveal the input required and output produced by the problem solution. Some input is clearly visible from the problem statement itself and some

other input may be hidden which is revealed while developing the solution. Output to be produced by the program is clearly stated by the problem itself. Some auxiliary output may also be produced, however, which may or may not be of some use. The next step is to prepare a detailed list of steps required to be carried out for solving the problem, which is called the algorithm. Once input, output and algorithm have been clearly defined, the next step is to translate these steps in a computer program using any high level language. The program in high level language is called the source code and is stored in a disk file. This program contains the logic or steps for solving the problem.

ii. **Compile the program:** The next step is to compile the program for checking syntax errors and translation. This is done by the C compiler. The output of compilation is the object code, if no syntax errors are reported by the compiler. The following figure shows the compilation process.

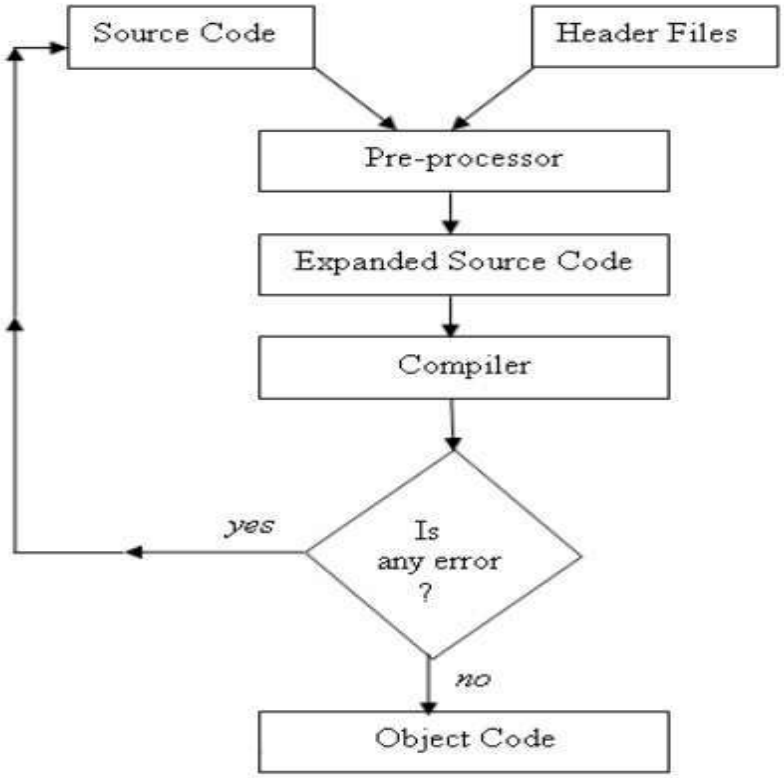


Figure 1: Stages in Compilation of a C program

C compiler has in-built pre-processor. The pre-processor processes the source code before it is passed to the compiler for compilation. Pre-processor

commands also known as directives, tell the pre-processor how to process the source code. Depending on the pre-processor directives, the pre-processor processes the source code and produces the expanded version of source code. The C complier takes expanded version of the source code as its input and if there are no errors in the source code, it produces a machine code (object code) version of the program which is saved on the disk file.

If there are some errors during compilation phase, known as syntax errors, then the compiler reports these errors in the form of diagnostic messages, which tell the cause and origin of errors and compilation process terminates. Having corrected the reported errors the source code is compiled again as shown in figure 1.

iii. Linking the program: After the compilation stage, the machine code version of the program is ready, but it can not be directly executed, as it may contain references to the library functions or user defined functions in other object modules which are compiled separately. In order to produce an executable code, these object codes are to be linked together and also with the system library. The process of linking is shown in figure 2 below:

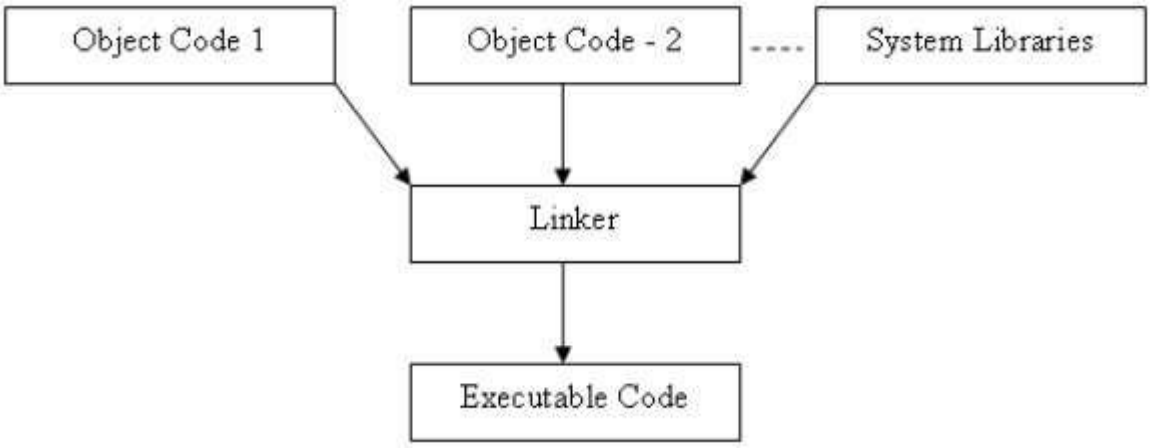


Figure 2: Linking of object code(s) with system libraries

Once the linking process is over, another disk with the same name as of the program file with extension exe is produced. This is the file that contains the code which can be directly executed on the computer.

iv. Debugging the program: The next stage in the program development is debugging of the program to make sure that the program is free of bugs and produces the desired outputs for all possible inputs. In this stage the program is executed with all possible values of input data for which result is known or can be computed manually. The program is declared correct if the output of the program matches the expected results. If the output does not match the expected results, then the program is said to be semantically or logically incorrect and needs to be

corrected by scanning the logic used in the source code. The logical or semantic errors are removed and the program is compiled and linked again.

v. Documenting the program: The final stage in the development of a program is documentation. The term documentation means recording the important information regarding the program. The documentation enables other users or programmers to understand the logic and purpose of the program. This facilitates maintenance and upgradation of the program.

Some compilers, like Turbo C, have integrated development environment (IDE), which facilitates program writing (editing), compilation, linking and execution of the program from one place. But in some other systems like in UNIX we need separate tools for editing, compilation, linking and execution of the program.

1.8 Summary

The C language was developed in Bell Laboratories by Dennis Ritchie and his associates. It is a refined version of BCPL with enhanced features. The C language is a high level language with capabilities of low level language as well. It has defined data types and program constructs like sequential, conditional and iterative flows. A program written in C language is first checked for syntax correctness and then converted to object code, thereafter it is linked with other libraries and finally an executable file is produced.

1.9 Keywords

High level languages: These are the languages which are closer to the human understandable languages and have been designed for better programming efficiency.

Debugging : It is the process of isolating and correcting the errors.

compiler : A program that translates a program written in a high level language into machine language so that it can be executed.

1.10 Short Answer Type Questions

1. What are the various types of programming languages?
2. What are the various types of errors?
3. Name any four high level programming languages?

1.11 Long Answer Type Questions

1. Discuss in detail the origin of C language.
2. What are the various features of C programming language?
3. What are various parts of a C program?
4. What are various stages of program development using C language?

1.12 Suggested Readings

1. Let us C — Yashwant Kanefkar
2. C Programming using Turbo C — Robert Lafore
3. Programming with ANSI and Turbo C — Ashok. N. Kamthane
4. C Programming — Balagurusamy

IDENTIFIERS, KEYWORDS, DATA TYPES AND TYPE CONVERSION

1.2.1 Introduction
1.2.2 Objectives
1.2.3 Character Set
1.2.4 Identifiers
1.2.5 Keywords
1.2.6 Data Types
1.2.7 Type Conversion
1.2.8 Variables and constants
1.2.9 Summary
1.2.10 Keywords
1.2.11 Short Answer Type Questions
1.2.12 Long Answer Type Questions
1.2.13 Suggested Readings

1.2.1 Introduction

Identifiers and data types are the basic building blocks of a programming language. A C program starts with the declaration of data types of the various identifiers to be used in the program. Then the behaviour of the identifiers also needs to be defined that whether these are variables or constants. Study of type conversion methodology is also important as it facilitates safe programming.

1.2.2 Objectives

In this lesson we will learn about the basic concepts used in the C programming language, which include, identifiers, variables and constants, keywords, data types and type conversion.

1.2.3 Character Set

The character set used to form words, numbers and expressions depend upon the computer on which the program runs. The characters in C are classified in the following categories:

i. Letters
ii. Digits
iii. White spaces
iv. Special characters

The C language character set is listed in the following table:

Letters	A to Z, a to z
Digits	0 to 9
White Spaces	Blank, Horizontal tab, Vertical tab, new line, form feed

Special characters All other characters available on standard keyboard.

1.2.4 Identifiers

Every program element must be named to distinguish it from other elements. The name assigned to the element should be meaningful, though it is not necessary, but it facilitates easy understanding of the program. Identifier is the name given to some program element. The program element is then identified by that name. The element may be some variable, constant, data structure, program block, function, pointer, file etc. The identifier naming rules are as follows:

a. Identifier name should begin with an alphabet or underscore (_) but never with a digit.

b. The following characters may be any combination of alphabets, digits and special symbol (underscore) but two consecutive underscores are not permitted.

c. In some C language compilers identifier length is restricted to some limit which varies from 32 to 48.

d. No other symbol or special character is permitted.

The following are valid C identifiers:

sum, factorial, number, first_name, permanent_address.

However, sum, SUM and Sum are different identifier because C language is case sensitive.

The following are invalid C identifiers:

5thdigit	(first letter should be alphabet or underscore)
first name	(Identifier can't contain spaces)
char	(It is a reserve word, discussed in next section)

It's wise to choose identifier names that are related to the purpose of the identifier, and that are unlikely to get mixed up typographically. We tend to use short names for local variables, especially loop indices and longer names for external variables.

1.2.5 Keywords

There is a set of words whose meaning is predefined in the C language and these words can not be used as identifier. These words are also called reserve words. The following is the set of words used as reserve words in the C language.

void	int	char	float	double	long	short	signed
for	do	while	If	else	break	continue	goto
static	auto	extern	case	switch	default	return	struct
sizeof	enum	typedef	const	volatile	register	unsigned	union

1.2.6 Data Types

Since, the C language is a strongly typed language therefore data type of all the variables need to be declared in advance. There are only a few basic data types in C:

char	a single byte, capable of holding one character in the local character set
int	an integer, typically reflecting the natural size of integers on the host machine
float	single-precision floating point
double	double-precision floating point

In addition, there are a number of qualifiers that can be applied to these basic types. short and long apply to integers:

```
short int sh;
long int counter;
```

The word int can be omitted in such declarations, and typically it is. The intent is that short and long should provide different lengths of integers where practical; int will normally be the natural size for a particular machine, short is often 16 bits long, and int either 16 or 32 bits. Each compiler is free to choose appropriate sizes for its own hardware, subject only to the the restriction that shorts and ints are at least 16 bits, longs are at least 32 bits, and short is no longer than int, which is no longer than long.

The qualifier signed or unsigned may be applied to char or any integer, unsigned numbers are always positive or zero and obey the laws of arithmetic modulo 2^n, where n is the number of bits in the type. So, for instance, if chars are 8 bits, unsigned char variables have values between 0 and 255, while signed chars have values between -128 and 127 (in a two's complement machine.) Whether plain chars are signed or unsigned is machine-dependent, but printable characters are always positive.

The type long double specifies extended-precision floating point. As with integers, the sizes of floating-point objects are implementation-defined; float, double and long double could represent one, two or three distinct sizes.

The standard headers <limits.h> and <float.h> contain symbolic constants for all of these sizes, along with other properties of the machine and compiler.

Data type	Size (in bytes)	Range	Format String
char	1	-128 to 127	%c
unsigned char	1	0 to 255	%c
short or int	2	-32768 to 32767	%i or %d
unsigned int	2	0 to 65535	%u
long	4	-2147483648 to 2147483647	%ld
unsigned long	4	0 to 4294967295	%lu
float	4	3.4 e-38 to 3.4 e+38	%f or %g
double	8	1.7 e-308 to 1.7 e+308	%lf
long double	10	3.4 e-4932 to 1.1 e+4932	%lf

1.2.7 Type Conversion

Some times data type of values needs to be modified. For example, if two integer values are divided then the result may be required in float but since the variable are of type integer, the result produced will also be of type integer.

If a = 5 and b = 2 and both a and b are integer then

result = a/b;

will store 2 in result irrespective of the data type of variable result. However to obtain float value we need to modify the data type of the argument variables of the expression, which can be done as follows:

result = (float)a/b; and value of result will be 2.5

this is called type casting and is done explicitly. Implicit data type conversion is also done while evaluating expression containing mixed types of variable. This is called **coercion**. In this case the data type of the lower sized or ranged variable is converted to the upper sized or ranged variable, for example, if b is float in the above example then the value of result will be 2.5.

When an operator has operands of different types, they are converted to a common type according to a small number of rules. In general, the only automatic conversions are those that convert a "narrower" operand into a "wider" one without losing information, such as converting an integer into floating point in an expression like f + i. Expressions that don't make sense, like using a float as a subscript, are disallowed. Expressions that might lose information, like assigning a longer integer type to a shorter, or a floating-point type to an integer, may draw a warning, but they are not illegal.

A char is just a small integer, so chars may be freely used in arithmetic expressions. This permits considerable flexibility in certain kinds of character transformations. One is exemplified by this naive implementation of the function atoi, which converts a string of digits into its numeric equivalent.

```
/* atoi:  convert s to integer */
int atoi(char s[])
{
    int i, n;

    n = 0;
    for (i = 0; s[i] >= '0' && s[i] <= '9'; ++i)
        n = 10 * n + (s[i] - '0');
    return n;
}
```

The expression

```
s[i] - '0'
```

gives the numeric value of the character stored in s[i], because the values of '0', '1', etc., form a contiguous increasing sequence.

Another example of char to int conversion is the function lower, which maps a single character to lower case *for the ASCII character set*. If the character is not an upper case letter, lower returns it unchanged.

```
/* lower:  convert c to lower case; ASCII only */
int lower(int c)
{
   if (c >= 'A' && c <= 'Z')
      return c + 'a' - 'A';
   else
      return c;
}
```

This works for ASCII because corresponding upper case and lower case letters are a fixed distance apart as numeric values and each alphabet is contiguous — there is nothing but letters between A and Z. This latter observation is not true of the EBCDIC character set, however, so this code would convert more than just letters in EBCDIC.

The standard header <ctype.h>, defines a family of functions that provide tests and conversions that are independent of character set. For example, the function tolower is a portable replacement for the function lower shown above. Similarly, the test

```
c >= '0' && c <= '9'
```

can be replaced by

```
isdigit(c)
```

We will use the <ctype.h> functions from now on.

There is one subtle point about the conversion of characters to integers. The language does not specify whether variables of type char are signed or unsigned quantities. When a char is converted to an int, can it ever produce a negative integer? The answer varies from machine to machine, reflecting differences in architecture. On some machines a char whose leftmost bit is 1 will be converted to a negative integer ("sign extension"). On others, a char is promoted to an int by adding zeros at the left end and thus is always positive.

The definition of C guarantees that any character in the machine's standard printing character set will never be negative, so these characters will always be positive quantities in expressions. But arbitrary bit patterns stored in character variables may appear to be negative on some machines, yet positive on others. For portability, specify signed or unsigned if non-character data is to be stored in char variables.

Relational expressions like i > j and logical expressions connected by && and || are defined to have value 1 if true, and 0 if false. Thus the assignment

```
d = c >= '0' && c <= '9'
```

sets d to 1 if c is a digit and 0 if not. However, functions like isdigit may return any non-zero value for true. In the test part of if, while, for, etc., "true" just means "non-zero", so this makes no difference.

Implicit arithmetic conversions work much as expected. In general, if an operator like + or * that takes two operands (a binary operator) has operands of different types, the "lower" type is *promoted* to the "higher" type before the operation proceeds. The result is of the integer type.

If there are no unsigned operands, however, the following informal set of rules will suffice:

- If either operand is long double, convert the other to long double.
- Otherwise, if either operand is double, convert the other to double.
- Otherwise, if either operand is float, convert the other to float.
- Otherwise, convert char and short to int.
- Then, if either operand is long, convert the other to long.

Notice that floats in an expression are not automatically converted to double; this is a change from the original definition. In general, mathematical functions like those in <math.h> will use double precision. The main reason for using float is to save storage in large arrays or less often, to save time on machines where double-precision arithmetic is particularly expensive.

Conversion rules are more complicated when unsigned operands are involved. The problem is that comparisons between signed and unsigned values are machine-dependent, because they depend on the sizes of the various integer types. For example, suppose that int is 16 bits and long is 32 bits. Then -1L < 1U, because 1U, which is an unsigned int, is promoted to a signed long. But -1L > 1UL because -1L is promoted to unsigned long and thus appears to be a large positive number.

Conversions take place across assignments; the value of the right side is converted to the type of the left, which is the type of the result.

A character is converted to an integer, either by sign extension or not, as described above.

Longer integers are converted to shorter ones or to chars by dropping the excess high-order bits. Thus in

```
int  i;
char c;
i = c;
c = i;
```

the value of c is unchanged. This is true whether or not sign extension is involved. Reversing the order of assignments might lose information, however.

If x is float and i is int, then x = i and i = x both cause conversions; float to int causes truncation of any fractional part. When a double is converted to float, whether the value is rounded or truncated is implementation dependent.

Since an argument of a function call is an expression, type conversion also takes place when arguments are passed to functions. In the absence of a function prototype, char and short become int and float becomes double. This is why we have

declared function arguments to be int and double even when the function is called with char and float.

Finally, explicit type conversions can be forced ("coerced") in any expression, with a unary operator called a cast. In the construction

(*type name*) *expression*

the *expression* is converted to the named type by the conversion rules above. The precise meaning of a cast is as if the *expression* were assigned to a variable of the specified type, which is then used in place of the whole construction. For example, the library routine sqrt expects a double argument, and will produce nonsense if inadvertently handled something else. (sqrt is declared in <math.h>.) So if n is an integer, we can use

```
sqrt((double) n)
```

to convert the value of n to double before passing it to sqrt. Note that the cast produces the *value* of n in the proper type; n itself is not altered. The cast operator has the same high precedence as other unary operators, as summarized in the table at the end of this lesson.

If arguments are declared by a function prototype, as they normally should be, the declaration causes automatic coercion of any arguments when the function is called. Thus, given a function prototype for sqrt:

```
double sqrt(double)
```

the call

```
root2 = sqrt(2)
```

coerces the integer 2 into the double value 2.0 without any need for a cast.

Self Check Exercise

Q: What do you mean by Identifier?

Q: What do you mean by Reserve Word?

1.2.8 Variables and constants

Variables and constants are the basic data objects manipulated in a program. Declarations list the variables to be used and state what type they have and perhaps what their initial values are. Operators specify what is to be done to them. Expressions combine variables and constants to produce new values. The type of an object determines the set of values it can have and what operations can be performed on it.

Declaring variables:

The declaration of all variables to be used in a function should be made in the variable declaration part of the function. All the variables must be declared before they can be used. A declaration specifies a type and contains a list of one or more variables of that type. It also provides a variable name to the compiler and tells the data type of the variable which helps in determining the memory requirements for the variable. The syntax for variable declaration is as follows:

```
data_type variable_name
```

Example

```
int rollno;
char c;
float amount;
double d;
```

Commas in the variable declaration separate the variables of the same type, as in

```
int  lower, upper, step;
char c, line[1000];
```

Variables can be distributed among declarations in any fashion; the lists above could well be written as

```
int  lower;
int  upper;
int  step;
char c;
char line[1000];
```

The latter form takes more space, but is convenient for adding a comment to each declaration for subsequent modifications.

A variable may also be initialized in its declaration. If the name is followed by an equals sign and an expression, the expression serves as an initializer, as in

```
char  esc = '\\';
int   i = 0;
int   limit = MAXLINE+1;
float eps = 1.0e-5;
```

If the variable in question is not automatic, the initialization is done once only, conceptionally before the program starts executing and the initializer must be a constant expression. An explicitly initialized automatic variable is initialized each time the function or block it is in is entered; the initializer may be any expression. External and static variables are initialized to zero by default. Automatic variables for which there is no explicit initializer have undefined (i.e., garbage) values.

Declaring constants:

The qualifier **const** can be applied to the declaration of any variable to specify that its value will not be changed. For an array, the const qualifier says that the elements will not be altered.

```
const double e = 2.71828182845905;
const char msg[] = "warning: ";
```

The const declaration can also be used with array arguments, to indicate that the function does not change that array:

```
int strlen(const char[]);
```

The result is implementation-defined if an attempt is made to change a const.

An integer constant like 1234 is an int. A long constant is written with a terminal l (ell) or L, as in 123456789L; an integer constant too big to fit into an int will also

be taken as a long. Unsigned constants are written with a terminal u or U and the suffix ul or UL indicates unsigned long.

Floating-point constants contain a decimal point (123.4) or an exponent (1e-2) or both; their type is double, unless suffixed. The suffixes f or F indicate a float constant; l or L indicate a long double.

The value of an integer can be specified in octal or hexadecimal instead of decimal. A leading 0 (zero) on an integer constant means octal; a leading 0x or 0X means hexadecimal. For example, decimal 31 can be written as 037 in octal and 0x1f or 0x1F in hex. Octal and hexadecimal constants may also be followed by L to make them long and U to make them unsigned: 0XFUL is an *unsigned long* constant with value 15 decimal.

A character constant is an integer, written as one character within single quotes, such as 'x'. The value of a character constant is the numeric value of the character in the machine's character set. For example, in the ASCII character set the character constant '0' has the value 48, which is unrelated to the numeric value 0. If we write '0' instead of a numeric value like 48 that depends on the character set, the program is independent of the particular value and easier to read. Character constants participate in numeric operations just as any other integers, although they are most often used in comparisons with other characters.

Certain characters can be represented in character and string constants by escape sequences like \n (newline); these sequences look like two characters, but represent only one. In addition, an arbitrary byte-sized bit pattern can be specified by

'*ooo*'

where *ooo* is one to three octal digits (0...7) or by

'\x*hh*'

where *hh* is one or more hexadecimal digits (0...9, a...f, A...F). So we might write

```
#define VTAB '\013'   /* ASCII vertical tab */
#define BELL '\007'   /* ASCII bell character */
```

or, in hexadecimal,

```
#define VTAB '\xb'   /* ASCII vertical tab */
#define BELL '\x7'   /* ASCII bell character */
```

The complete set of **escape sequences** is

\a	alert (bell) character	\\	backslash
\b	backspace	\?	question mark
\f	formfeed	\'	single quote
\n	newline	\"	double quote
\r	carriage return	*ooo*	octal number
\t	horizontal tab	\x*hh*	hexadecimal number
\v	vertical tab		

The character constant '\0' represents the character with value zero, the **null**

character. '\0' is often written instead of 0 to emphasize the character nature of some expression, but the numeric value is just 0.

A ***constant expression*** is an expression that involves only constants. Such expressions may be evaluated at during compilation rather than run-time and accordingly may be used in any place that a constant can occur, as in

```
#define MAXLINE 1000
char line[MAXLINE+1];
```

or

```
#define LEAP 1 /* in leap years */
int days[31+28+LEAP+31+30+31+30+31+31+30+31+30+31];
```

A ***string constant***, or *string literal*, is a sequence of zero or more characters surrounded by double quotes, as in

```
"I am a string"
```

or

```
"" /* the empty string */
```

The quotes are not part of the string, but serve only to delimit it. The same escape sequences used in character constants apply in strings; \" represents the double-quote character. String constants can be concatenated at compile time:

```
"hello, " "world"
```

is equivalent to

```
"hello, world"
```

This is useful for splitting up long strings across several source lines.

Technically, a string constant is an array of characters. The internal representation of a string has a null character '\0' at the end, so the physical storage required is one more than the number of characters written between the quotes. This representation means that there is no limit to how long a string can be but programs must scan a string completely to determine its length. The standard library function strlen(s) returns the length of its character string argument s, excluding the terminal '\0'. Here is our version:

```
/* strlen:  return length of s */
int strlen(char s[])
{
int i;

while (s[i] != '\0')
++i;
return i;
}
```

strlen and other string functions are declared in the standard header <string.h>.

Be careful to distinguish between a character constant and a string that con-

tains a single character: 'x' is not the same as "x". The former is an integer, used to produce the numeric value of the letter *x* in the machine's character set. The latter is an array of characters that contains one character (the letter *x*) and a '\0'.

Enumeration Constant

There is one other kind of constant, the *enumeration constant*. An enumeration is a list of constant integer values, as in

```
enum boolean { NO, YES };
```

The first name in an enum has value 0, the next 1, and so on, unless explicit values are specified. If not all values are specified, unspecified values continue the progression from the last specified value, as the second of these examples:

```
enum escapes { BELL = '\a', BACKSPACE = '\b', TAB = '\t',
        NEWLINE = '\n', VTAB = '\v', RETURN = '\r' };

enum months { JAN = 1, FEB, MAR, APR, MAY, JUN,
        JUL, AUG, SEP, OCT, NOV, DEC };
    /* FEB = 2, MAR = 3, etc. */
```

Names in different enumerations must be distinct. Values need not be distinct in the same enumeration.

Enumerations provide a convenient way to associate constant values with names, an alternative to #define with the advantage that the values can be generated for you. Although variables of enum types may be declared, compilers need not check that what you store in such a variable is a valid value for the enumeration. Nevertheless, enumeration variables offer the chance of checking and so are often better than #defines. In addition, a debugger may be able to print values of enumeration variables in their symbolic form.

1.2.9 Summary

Character set of a language specifies the valid set of characters using which words of the language are formed for identifier declaration.

Identifier is the name given to some program element. The element may be some variable, constant, data structure, program block, function, pointer, file etc. Identifier is the name given to some program element.

There is a set of words whose meaning is predefined in the C language and these words can not be used as identifier. These words are also called reserve words.

Since, the C language is a strongly typed language therefore data type of all the variables need to be declared in advance. The qualifier signed or unsigned may be applied to char or any integer, unsigned numbers are always positive or zero and obey the laws of arithmetic modulo 2^n, where n is the number of bits in the type.

Type conversion facilitates the conversion of data type of the result of expression. It may be explicit, called type casting or implicit, called conversion.

Variables and constants are the basic data objects manipulated in a pro-

gram. Declarations list the variables to be used, and state what type they have and perhaps what their initial values are. Operators specify what is to be done to them. Expressions combine variables and constants to produce new values. The type of an object determines the set of values it can have and what operations can be performed on it.

1.2.10 Keywords

Variable : A Variable is a data name that may be used to store a data value.

Constants : There are the fixed values that do not change during the execution of a program.

Keywords : There are the set of words whose meaning is predefined in the C language and these words can not be used as a identifier.

Identifiers : There are the names you supply for variables, types, functions and labels in your program.

1.2.11 Short Answer Type Questions

Q1. What are valid characters in the C character set?

Q2. What is the need of declaring the type of a variable?

Q3. What do you mean by type conversion?

Q4. What is the difference between variable and constant?

Q5. Explain Enumeration Constants.

1.2.12 Long Answer Type Questions

Q1. Discuss the various identifier naming rules.

Q2. Write any 24 reserve words of C language.

Q3. What are the basic data types available in C language? What is the size of each data type?

Q4. What are the various types of constant declarations? Explain giving examples.

1.2.13 Suggested Readings

The C Programming language	Brian W. Kernigham Dennis M. Ritchie
Programming with ANSI and Turbo C	Ashok N. Kamthane
C Programming	E. Balagurusamy
Let Us C	Yashavant Kanetkar

Web Resources :

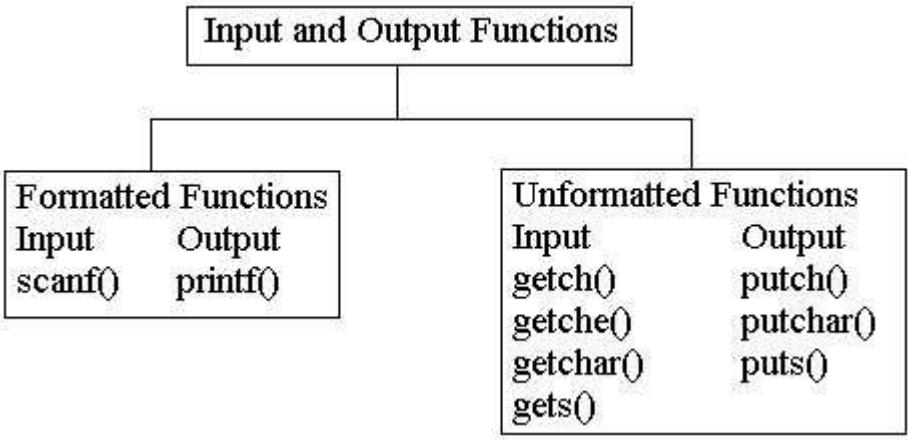
www.cprogramming.com/tutorial/c/lesson1.html
www.programiz.com/c-programming
www.w3schools.in/c-tutorial/
www.learn-c.org
www.tutorialspoint.com/cprogramming/

PERFORMING INPUT OUTPUT OPERATIONS

1.3.1 Introduction
1.3.2 Objectives
1.3.3 Unformatted Input Statements
1.3.4 Formatted Input - scanf
1.3.5 Unformatted Output Statements
1.3.6 Formatted Output – printf
1.3.7 Escape Sequences
1.3.8 Summary
1.3.9 Keywords
1.3.10 Short Answer Type Questions
1.3.11 Long Answer Type Questions
1.3.12 Suggested Readings

1.3.1 Introduction

Input output statements facilitate interaction between program and the users. Through input statements user provide input to the program and through the output statements prompts and results are displayed. The following are the input output functions which we shall discuss in this lesson.



1.3.2 Objectives

In this lesson we will discuss the role, types and usage of input and output statements.

1.3.3 Unformatted Input Statements

Input and output are not part of the C language itself, so we have not emphasized them in our presentation thus far. Nonetheless, programs interact with their environment in much more complicated ways than those we have shown before. In

this lesson we will describe the standard library, a set of functions that provide input and output, string handling, storage management, mathematical routines and a variety of other services for C programs. We will concentrate on input and output
The ANSI standard defines these library functions precisely, so that they can exist in compatible form on any system where C exists. Programs that confine their system interactions to facilities provided by the standard library can be moved from one system to another without change.
The properties of library functions are specified in more than a dozen headers; we have already seen several of these, including <stdio.h>, <string.h> and <ctype.h>. We will not present the entire library here, since we are more interested in writing C programs that use it.

Standard Input

As we said, the library implements a simple model of text input and output. A text stream consists of a sequence of lines; each line ends with a newline character. If the system doesn't operate that way, the library does whatever necessary to make it appear as if it does. For instance, the library might convert carriage return and linefeed to newline on input and back again on output.
The simplest input mechanism is to read one character at a time from the *standard input*, normally the keyboard, with getchar:

```
int getchar(void)
```

getchar returns the next input character each time it is called, or EOF when it encounters end of file. The symbolic constant EOF is defined in <stdio.h>. The value is typically -1, but tests should be written in terms of EOF so as to be independent of the specific value.
In many environments, a file may be substituted for the keyboard by using the < convention for input redirection: if a program prog uses getchar, then the command line

```
prog <infile
```

causes prog to read characters from infile instead. The switching of the input is done in such a way that prog itself is oblivious to the change; in particular, the string "<infile" is not included in the command-line arguments in argv. Input switching is also invisible if the input comes from another program via a pipe mechanism: on some systems, the command line

```
otherprog | prog
```

runs the two programs otherprog and prog and pipes the standard output of otherprog into the standard input for prog.

Following are the unformatted input functions

a. **getchar() fucntion:** This function returns a single character from the standard input device i.e. keyboard. The typed character is echoed on the screen. After typing the appropriate character, the user is required to press Enter Key. General syntax of using getchar function is : character variable = getchar

() e.g.

char ch;

ch = getchar ()

b. getche()function: This function also returns a character that has been recently typed. The character is also echoed on the screen, but it is not required to press Enter Key after typing the character. The syntax is same as getchar ().

c. getch() function: This function also returns a character that has been recently typed. But here, user is not required to press the Enter Key and the character being type is not echoed on the screen. The syntax is same as getchar () and getche ().

d. gets() : This function is used for accpeting any string through stdin (keyboard) until enter key is pressed. The header file stdio.h is needed for implementing this function.

1.3.4 Formatted Input – scanf

The function scanf is the input analog of printf, providing many of the same conversion facilities in the opposite direction.

```
int scanf(char *format, ...)
```

scanf reads characters from the standard input, interprets them according to the specification in format and stores the results through the remaining arguments. The format argument is described below; the other arguments, *each of which must be a pointer*, indicate where the corresponding converted input should be stored. As with printf, this section is a summary of the most useful features, not an exhaustive list.

scanf stops when it exhausts its format string, or when some input fails to match the control specification. It returns as its value the number of successfully matched and assigned input items. This can be used to decide how many items were found. On the end of file, EOF is returned; note that this is different from 0, which means that the next input character does not match the first specification in the format string. The next call to scanf resumes searching immediately after the last character already converted.

There is also a function sscanf that reads from a string instead of the standard input:

```
int sscanf(char *string, char *format, arg1, arg2, ...)
```

It scans the string according to the format and stores the resulting values through arg1, arg2, etc. These arguments must be pointers.

The format string usually contains conversion specifications, which are used to control conversion of input. The format string may contain:

- Blanks or tabs, which are not ignored.
- Ordinary characters (not %), which are expected to match the next non-white space character of the input stream.
- Conversion specifications, consisting of the character %, an optional

assignment suppression character *, an optional number specifying a maximum field width, an optional h, l or L indicating the width of the target and a conversion character.

A conversion specification directs the conversion of the next input field. Normally the result is placed in the variable pointed to by the corresponding argument. If assignment suppression is indicated by the * character, however, the input field is skipped; no assignment is made. An input field is defined as a string of non-white space characters; it extends either to the next white space character or until the field width, specified, is exhausted. This implies that scanf will read across boundaries to find its input, since newlines are white space. (White space characters are blank, tab, newline, carriage return, vertical tab, and formfeed.)

The conversion character indicates the interpretation of the input field. The corresponding argument must be a pointer, as required by the call-by-value semantics of C. Conversion characters are shown in following table.

Basic scanf Conversions

Character	Input Data; Argument type
D	decimal integer; int *
i	integer; int *. The integer may be in octal (leading 0) or hexadecimal (leading 0x or 0X).
o	octal integer (with or without leading zero); int *
u	unsigned decimal integer; unsigned int *
x	hexadecimal integer (with or without leading 0x or 0X); int *
c	characters; char *. The next input characters (default 1) are placed at the indicated spot. The normal skip-over white space is suppressed; to read the next non-white space character, use %1s
s	character string (not quoted); char *, pointing to an array of characters long enough for the string and a terminating '\0' that will be added.
e, f, g	floating-point number with optional sign, optional decimal point and optional exponent; float *
%	literal %; no assignment is made.

The conversion characters d, i, o, u, and x may be preceded by h to indicate that a pointer to short rather than int appears in the argument list, or by l (letter ell) to

indicate that a pointer to long appears in the argument list.

As a first example, the rudimentary calculator can be written with scanf to do the input conversion:

```
#include <stdio.h>
main()  /* rudimentary calculator */
{
        double sum, v;
        sum = 0;
        while (scanf("%lf", &v) == 1)
                printf("\t%.2f\n", sum += v);
        return 0;
}
```

Suppose we want to read input lines that contain dates of the form

```
25 Dec 1988
```

The scanf statement is

```
int day, year;
char monthname[20];
scanf("%d %s %d", &day, monthname, &year);
```

No & is used with monthname, since an array name is a pointer.

Literal characters can appear in the scanf format string; they must match the same characters in the input. So we could read dates of the form mm/dd/yy with the scanf statement:

```
int day, month, year;
scanf("%d/%d/%d", &month, &day, &year);
```

scanf ignores blanks and tabs in its format string. Furthermore, it skips over white space (blanks, tabs, newlines, etc.) as it looks for input values. To read input whose format is not fixed, it is often best to read a line at a time, then pick it apart with scanf. For example, suppose we want to read lines that might contain a date in either of the forms above. Then we could write

```
while (getline(line, sizeof(line)) > 0) {
        if (sscanf(line, "%d %s %d", &day, monthname, &year) == 3)
                printf("valid: %s\n", line); /* 25 Dec 1988 form */
        else if (sscanf(line, "%d/%d/%d", &month, &day, &year) == 3)
                printf("valid: %s\n", line); /* mm/dd/yy form */
        else
                printf("invalid: %s\n", line); /* invalid form */
}
```

Calls to scanf can be mixed with calls to other input functions. The next call to any input function will begin by reading the first character not read by scanf.

A final warning: the arguments to scanf and sscanf *must* be pointers. By far the most common error is writing

scanf("%d", n);

instead of

scanf("%d", &n);

This error is not generally detected at compile time.

Self Check Exercise

Q: Explain the purpose of following functions:

- getchar

- getch

- gets

Q: What are the formatted input output functions available in C?

1.3.5 Unformatted Output Statements

The function

int putchar(int)

is used for output: putchar(c) puts the character c on the standard output, which is by default the screen. putchar returns the character written, or EOF if an error occurs. Again, output can usually be directed to a file with >*filename*: if prog uses putchar,

prog >outfile

will write the standard output to outfile instead. If pipes are supported,

prog | anotherprog

puts the standard output of prog into the standard input of anotherprog. Output produced by printf also finds its way to the standard output. Calls to putchar and printf may be interleaved - output happens in the order in which the calls are made.

Each source file that refers to an input/output library function must contain the line

#include <stdio.h>

before the first reference. When the name is bracketed by < and > a search is made for the header in a standard set of places (for example, on UNIX systems, typically in the directory /usr/include).

Following are the unformatted output functions:

a. putchar(): This functions prints one character on the screen at a time.

b. putch(): This function prints any character taken by the standard input devices.

c. puts(): This function prints the string or character array.

Many programs read only one input stream and write only one output stream; for such programs, input and output with getchar, putchar and printf may be entirely adequate, and is certainly enough to get started. This is particularly true if redirection is used to connect the output of one program to the input of the next. For example, consider the program lower, which converts its input to lower case:

#include <stdio.h>

```
#include <ctype.h>
main() /* lower: convert input to lower case*/
{
        int c
        while ((c = getchar()) != EOF)
                putchar(tolower(c));
        return 0;
}
```

The function tolower is defined in <ctype.h>; it converts an upper case letter to lower case, and returns other characters untouched. As we mentioned earlier, "functions" like getchar and putchar in <stdio.h> and tolower in <ctype.h> are often macros, thus avoiding the overhead of a function call per character. Regardless of how the <ctype.h> functions are implemented on a given machine, programs that use them are shielded from knowledge of the character set.

1.3.6 Formatted Output - printf

The output function printf translates internal values to characters. We have used printf informally in previous lessons. The description here covers most typical uses but is not complete; for the full story, refer the books given at the end of this lesson.

```
int printf(char *format, arg1, arg2, ...);
```

printf converts, formats, and prints its arguments on the standard output under control of the format. It returns the number of characters printed.

The format string contains two types of objects: ordinary characters, which are copied to the output stream, and conversion specifications, each of which causes conversion and printing of the next successive argument to printf. Each conversion specification begins with a % and ends with a conversion character. Between the % and the conversion character there may be, in order:

- A minus sign, which specifies left adjustment of the converted argument.
- A number that specifies the minimum field width. The converted argument will be printed in a field at least this wide. If necessary it will be padded on the left (or right, if left adjustment is called for) to make up the field width.
- A period, which separates the field width from the precision.
- A number, the precision, that specifies the maximum number of characters to be printed from a string, or the number of digits after the decimal point of a floating-point value, or the minimum number of digits for an integer.
- An h if the integer is to be printed as a short, or l (letter ell) if as a long.

Conversion characters are shown in the following table. If the character after the % is not a conversion specification, the behavior is undefined.

A width or precision may be specified as *, in which case the value is computed by converting the next argument (which must be an int). For example, to print at most max characters from a string s,

```
printf("%.*s", max, s);
```

Basic printf Conversions

Character	Argument type; Printed As
d,i	int; decimal number
o	int; unsigned octal number (without a leading zero)
x,X	int; unsigned hexadecimal number (without a leading 0x or 0X), using abcdef or ABCDEF for 10, ...,15.
u	int; unsigned decimal number
c	int; single character
s	char *; print characters from the string until a '\0' or the number of characters given by the precision.
f	double; [-]*m.dddddd*, where the number of *d*'s is given by the precision (default 6).
e,E	double; [-]*m.dddddd*e+/-*xx* or [-]*m.dddddd*E+/-*xx*, where the number of *d*'s is given by the precision (default 6).
g,G	double; use %e or %E if the exponent is less than -4 or greater than or equal to the precision; otherwise use %f. Trailing zeros and a trailing decimal point are not printed.
p	void *; pointer (implementation-dependent representation).
%	no argument is converted; print a %

Along with the conversion characters, precision can also be defined for reserving or limiting the space for output. Most of the format conversions have been illustrated in earlier sections. One exception is the precision as it relates to strings. The field width of the strings is specified by using the format % w. p s where w is the field width and only p character of string w are to be displayed in right justified way.The following table shows the effect of a variety of specifications in printing "hello, world" (12 characters). We have put colons around each field so you can see its extent.

:%s:	:hello, world:
:%10s:	:hello, world:
:%.10s:	:hello, wor:
:%-10s:	:hello, world:
:%.15s:	:hello, world:

:%-15s:	:hello, world:
:%15.10s:	: hello, wor:
:%-15.10s:	:hello, wor:

A warning: printf uses its first argument to decide how many arguments follow and what their type is. It will get confused, and you will get wrong answers, if there are not enough arguments of if they are the wrong type. You should also be aware of the difference between these two calls:

```
printf(s);        /* FAILS if s contains % */
printf("%s", s);  /* SAFE */
```

The function sprintf does the same conversions as printf does, but stores the output in a string:

```
int sprintf(char *string, char *format, arg1, arg2, ...);
```

sprintf formats the arguments in arg1, arg2, etc., according to format as before, but places the result in string instead of the standard output; string must be big enough to receive the result.

1.3.7 Escape Sequences

The printf() and scanf() statements follow the combination of characters called as escape sequences. The following are the escape sequences with their use and ASCII value.

Escape Sequence	Use	ASCII Value
\n	New line	10
\b	Backspace	8
\f	Form Feed	12
\'	Single Quote	39
\\	Backslash	92
\0	Null	0
\t	Horizontal Tab	9
\r	Carriage Return	13
\a	Alert	7
\"	Double Quote	34
\v	Vertical Tab	11
\?	Question Mark	63

Escape sequences facilitate formatting of output, generating alerts and printing some characters which can not be directly printed using output functions.

1.3.8 Summary

Input and output are not part of the C language itself, so we have not emphasized them in our previous lessons. Nonetheless, programs interact with their environment in much more complicated ways than those we have shown before. In this lesson we have described the standard library, a set of functions that provide input and output functions.

1.3.9 Keywords

printf : printf function converts formats and print its arguments on the standard output under the control of the fomat.

scanf : scanf reads characters from the standard input interprets them according to the specification in format and stores the result through the remaining arguments.

1.3.10 Short Answer Type Questions

1. What are the basic input output functions available in C?
2. What do you mean by formatted I/O?
3. What is the difference between getch() and getche() functions?
4. Why ampersand (&) is used in scanf while reading numeric or character data types?

1.3.11 Long Answer Type Questions

1. Discuss in detail the various types of input and output functions used in C language. Also discuss the syntax of each of the functions giving examples.
2. Which characters are used for conversion in printf and scanf statements?
3. What are the various escape sequences? Write their use as well.

1.3.12 Suggested Readings

The C Programming language	Brian W. Kernigham Dennis M. Ritchie
Programming with ANSI and Turbo C	Ashok N. Kamthane
C Programming	E. Balagurusamy
Let Us C	Yashavant Kanetkar

Web Resources :

www.cprogramming.com/tutorial/c/lesson1.html
www.programiz.com/c-programming
www.w3schools.in/c-tutorial/
www.learn-c.org
www.tutorialspoint.com/cprogramming/

OPERATORS AND EXPRESSIONS

1.4.1 Introduction
1.4.2 Objectives
1.4.3 Arithmetic Operators
1.4.4 Relational Operators and Logical Operators
1.4.5 Bitwise Operators
1.4.6 Assignment Operator and Expression Evaluation
1.4.7 Conditional Expression
1.4.8 Comma Operator
1.4.9 Operator Precedence and Associativity
1.4.10 Summary
1.4.11 Keywords
1.4.12 Short Answer Type Questions
1.4.13 Long Answer Type Questions
1.4.14 Suggested Readings

1.4.1 Introduction

In order to perform different types of operations, C uses different kind of operators. An operator indicates an operation to be performed on data that yields a value. With the help of various operators available in C language, one can link the variables and constants. An operand is a data item on which operators perform the operations. C provides four classes of operators. They are 1) Arithmetic 2) Relational 3) Logical and 4) bitwise. Along with these operators there are other operators like unary, conditional, assignment and comma operator.

The following are the various types of operators available in C language:

Type of operator	**Symbolic representation**
Arithmetic	+, -, *, / and %
Relational	>, >, >=, <=, == and !=
Logical	&&, || and !
Increment and decrement	++ and —
Assignment	=
Bitwise	&, |, ^, >>, << and ~
Comma	,
Conditional	? :

1.4.2 Objectives

In this lesson we shall discuss the various types of operators available in the C language. We shall also discuss the method of using these operators, their precedence and their order of evaluation in expressions.

1.4.3 Arithmetic Operators

The binary arithmetic operators are +, -, *, /, and the modulus operator %. Integer division truncates any fractional part. The expression

```
x % y
```

produces the remainder when x is divided by y, and thus is zero when y divides x exactly. For example, a year is a leap year if it is divisible by 4 but not by 100, except that years divisible by 400 *are* leap years.

Therefore

```
if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0)
        printf("%d is a leap year\n", year);
else
        printf("%d is not a leap year\n", year);
```

The % operator cannot be applied to a float or double. The direction of truncation for / and the sign of the result for % are machine-dependent for negative operands, as is the action taken on overflow or underflow.

The binary + and - operators have the same precedence, which is lower than the precedence of *, / and %, which is in turn lower than unary + and -. Arithmetic operators associate left to right.

Unary Operators

C provides two unusual operators for incrementing and decrementing variables. The increment operator ++ adds 1 to its operand, while the decrement operator -- subtracts 1. We have frequently used ++ to increment variables, as in

```
if (c == '\n')
  ++nl;
```

The unusual aspect is that ++ and -- may be used either as prefix operators (before the variable, as in ++n), or postfix operators (after the variable: n++). In both cases, the effect is to increment n. But the expression ++n increments n *before* its value is used, while n++ increments n *after* its value has been used. This means that in a context where the value is being used, not just the effect, ++n and n++ are different. If n is 5, then

```
x = n++;
```

sets x to 5, but

```
x = ++n;
```

sets x to 6. In both cases, n becomes 6. The increment and decrement operators can only be applied to variables; an expression like (i+j)++ is illegal.

In a context where no value is wanted, just the incrementing effect, as in

```
if (c == '\n')
nl++;
```

prefix and postfix are the same. But there are situations where one or the other is specifically called for. For instance, consider the function squeeze(s,c), which removes all occurrences of the character c from the string s.

```
/* squeeze:  delete all c from s */
void squeeze(char s[], int c)
{
    int i, j;
    for (i = j = 0; s[i] != '\0'; i++)
        if (s[i] != c)
            s[j++] = s[i];
    s[j] = '\0';
}
```

Each time a non-c occurs, it is copied into the current j position and only then is j incremented to be ready for the next character. This is exactly equivalent to

```
if (s[i] != c) {
    s[j] = s[i];
    j++;
}
```

Another example of a similar construction comes from the getline function that we wrote in lesson 1, where we can replace

```
if (c == '\n') {
    s[i] = c;
    ++i;
}
```

by the more compact

```
if (c == '\n')
    s[i++] = c;
```

As a third example, consider the standard function strcat(s,t), which concatenates the string t to the end of string s. strcat assumes that there is enough space in s to hold the combination. As we have written it, strcat returns no value; the standard library version returns a pointer to the resulting string.

```
/* strcat:  concatenate t to end of s; s must be big enough */
void strcat(char s[], char t[])
{
    int i, j;
    i = j = 0;
    while (s[i] != '\0') /* find end of s */
```

```
        i++;
    while ((s[i++] = t[j++]) != '\0') /* copy t */
        ;
}
```

As each member is copied from t to s, the postfix ++ is applied to both i and j to make sure that they are in position for the next pass through the loop.

1.4.4 Relational Operators and Logical Operators

The relational operators are

```
>   >=   <   <=
```

They all have the same precedence. Just below them in precedence are the equality operators:

```
==   !=
```

Relational operators have lower precedence than arithmetic operators, so an expression like i < lim-1 is taken as i < (lim-1), as would be expected.

More interesting are the logical operators && and ||. Expressions connected by && or || are evaluated left to right, and evaluation stops as soon as the truth or falsehood of the result is known. Most C programs rely on these properties. For example, here is a loop from the input function getline:

```
for (i=0; i < lim-1 && (c=getchar()) != '\n' && c != EOF; ++i)
    s[i] = c;
```

Before reading a new character it is necessary to check that there is room to store it in the array s, so the test i < lim-1 *must* be made first. Moreover, if this test fails, we must not go on and read another character.

Similarly, it would be unfortunate if c were tested against EOF before getchar is called; therefore the call and assignment must occur before the character in c is tested.

The precedence of && is higher than that of ||, and both are lower than relational and equality operators, so expressions like

```
i < lim-1 && (c=getchar()) != '\n' && c != EOF
```

need no extra parentheses. But since the precedence of != is higher than assignment, parentheses are needed in

```
(c=getchar()) != '\n'
```

to achieve the desired result of assignment to c and then comparison with '\n'.

By definition, the numeric value of a relational or logical expression is 1 if the relation is true and 0 if the relation is false.

The unary negation operator ! converts a non-zero operand into 0 and a zero operand in 1. A common use of ! is in constructions like

```
if (!valid)
```

rather than

```
if (valid == 0)
```

It's hard to generalize about which form is better. Constructions like !valid

read nicely ("if not valid"), but more complicated ones can be hard to understand.

1.4.5 Bitwise Operators

C provides six operators for bit manipulation; these may only be applied to integral operands, that is, char, short, int and long, whether signed or unsigned.

&	bitwise AND
|	bitwise inclusive OR
^	bitwise exclusive OR
<<	left shift
>>	right shift
~	one's complement (unary)

The bitwise AND operator & is often used to mask off some set of bits, for example

```
n = n & 0177;
```

In the above example 0177 is an octal number 177 whose binary form is 0000000001111111. So when we AND nwith 01777, all the bits of n other than the last seven bits will be set to zero irrespecitve of their value.

sets to zero all but the low-order 7 bits of n.

The bitwise OR operator | is used to turn bits on:

```
x = x | SET_ON;
```

sets to one in x the bits that are set to one in SET_ON.

The bitwise exclusive OR operator ^ sets a one in each bit position where its operands have different bits and zero where they are the same.

One must distinguish the bitwise operators & and | from the logical operators && and ||, which imply left-to-right evaluation of a truth value. For example, if x is 1 and y is 2, then x & y is zero while x && y is one.

The shift operators << and >> perform left and right shifts of their left operand by the number of bit positions given by the right operand, which must be non-negative. Thus x << 2 shifts the value of x by two positions, filling vacated bits with zero; this is equivalent to multiplication by 4. Right shifting an unsigned quantity always fits the vacated bits with zero. Right shifting a signed quantity will fill with bit signs ("arithmetic shift") on some machines and with 0-bits ("logical shift") on others.

The unary operator ~ yields the one's complement of an integer; that is, it converts each 1-bit into a 0-bit and vice versa. For example

```
x = x & ~077
```

sets the last six bits of x to zero. Note that x & ~077 is independent of word length, and is thus preferable to, for example, x & 0177700, which assumes that x is a 16-bit quantity. The portable form involves no extra cost, since ~077 is a constant expression that can be evaluated at compile time.

As an illustration of some of the bit operators, consider the function getbits(x,p,n) that returns the (right adjusted) n-bit field of x that begins at position p. We assume that bit position 0 is at the right end and that n and p are sensible positive values. For

example, getbits(x,4,3) returns the three bits in positions 4, 3 and 2, right-adjusted.

```
/* getbits:  get n bits from position p */
unsigned getbits(unsigned x, int p, int n)
{
    return (x >> (p+1-n)) & ~(~0 << n);
}
```

The expression x >> (p+1-n) moves the desired field to the right end of the word. ~0 is all 1-bits; shifting it left n positions with ~0<<n places zeros in the rightmost n bits; complementing that with ~ makes a mask with ones in the rightmost n bits.

Self Check Exercise

Q: What is a Unary Operator?

Q: What are the various bitwise operators?

1.4.6 Assignment Operator and Expression Evaluation

An expression such as

```
i = i + 2
```

in which the variable on the left side is repeated immediately on the right, can be written in the compressed form

```
i += 2
```

The operator += is called an **assignment operator.**

Most binary operators (operators like + that have a left and right operand) have a corresponding assignment operator *op*=, where *op* is one of

```
+   -   *   /   %   <<   >>   &   ^   |
```

If $expr_1$ and $expr_2$ are expressions, then

$expr_1$ *op*= $expr_2$

is equivalent to

$expr_1 = (expr_1)$ *op* $(expr_2)$

except that $expr_1$ is computed only once. Notice the parentheses around $expr_2$:

```
x *= y + 1
```

means

```
x = x * (y + 1)
```

rather than

```
x = x * y + 1
```

As an example, the function bitcount counts the number of 1-bits in its integer argument.

```
/* bitcount:  count 1 bits in x */
int bitcount(unsigned x)
{
    int b;
    for (b = 0; x != 0; x >>= 1)
            if (x & 01)
```

```
        b++;
    return b;
}
```

Declaring the argument x to be an unsigned ensures that when it is right-shifted, vacated bits will be filled with zeros, not sign bits, regardless of the machine the program is run on.

Quite apart from conciseness, assignment operators have the advantage that they correspond better to the way people think. We say "add 2 to i" or "increment i by 2", not "take i, add 2, then put the result back in i". Thus the expression i += 2 is preferable to i = i+2. In addition, for a complicated expression like

```
yyval[yypv[p3+p4] + yypv[p1]] += 2
```

the assignment operator makes the code easier to understand, since the reader doesn't have to check painstakingly that two long expressions are indeed the same, or to wonder why they're not. And an assignment operator may even help a compiler to produce efficient code.

We have already seen that the assignment statement has a value and can occur in expressions; the most common example is

```
while ((c = getchar()) != EOF)
    ...
```

The other assignment operators (+=, -=, etc.) can also occur in expressions, although this is less frequent.

In all such expressions, the type of an assignment expression is the type of its left operand, and the value is the value after the assignment.

1.4.7 Conditional Expression

The statements

```
if (a > b)
    z = a;
else
    z = b;
```

compute in z the maximum of a and b. The **conditional expression,** written with the ternary operator "?:", provides an alternate way to write this and similar constructions. In the expression

$expr_1$? $expr_2$: $expr_3$

the expression $expr_1$ is evaluated first. If it is non-zero (true), then the expression $expr_2$ is evaluated, and that is the value of the conditional expression. Otherwise $expr_3$ is evaluated and that is the value. Only one of $expr_2$ and $expr_3$ is evaluated. Thus to set z to the maximum of a and b,

```
z = (a > b) ? a : b;    /* z = max(a, b) */
```

It should be noted that the conditional expression is indeed an expression and it can be used wherever any other expression can be. If $expr_2$ and $expr_3$ are of

different types, the type of the result is determined by the conversion rules discussed earlier in the previous lesson. For example, if f is a float and n an int, then the expression

```
(n > 0) ? f : n
```

is of type float regardless of whether n is positive.

Parentheses are not necessary around the first expression of a conditional expression, since the precedence of ?: is very low, just above assignment. They are advisable anyway, however, since they make the condition part of the expression easier to see.

The conditional expression often leads to succinct code. For example, this loop prints n elements of an array, 10 per line, with each column separated by one blank, and with each line (including the last) terminated by a newline.

```
for (i = 0; i < n; i++)
    printf("%6d%c", a[i], (i%10==9 || i==n-1) ? '\n' : ' ');
```

A newline is printed after every tenth element, and after the n-th. All other elements are followed by one blank. This might look tricky, but it's more compact than the equivalent if-else. Another good example is

```
printf("You have %d items %s.\n", n, n==1 ? "" : "s");
```

Here if n = 1, then the word item will be printed, otherwise items wil be printed.

1.4.8 Comma Operator

The comma operator is used to separate two or more expressions. The comma operator has the lowest priority among all the operators. It is not essential to enclose the expressions with comma operators within the parenthesis.

Example:

A=2,b=3,c=5; or (A=2,b=3,c=5;)

are valid statements.

1.4.9 Operator Precedence and Associativity

Table below summarizes the rules for precedence and associativity of all operators, including those that we have not yet discussed. Operators on the same line have the same precedence; rows are in order of decreasing precedence, so, for example, *, /, and % all have the same precedence, which is higher than that of binary + and -. The "operator" () refers to function call. The operators -> and . are used to access members of structures; they will be covered in lessons 11 and 12, along with sizeof (size of an object). Lesson 11 discusses * (indirection through a pointer) and & (address of an object).

Operators (in order of their precedence)	Operation	Associativity	Priority
() [] . -> .	Function call Array expression or square bracket Structure operator Structure operator	left to right	1
! ~ ++ -- + - * & (*type*) sizeof	Not operator Ones complement Increment Decrement Unary plus Unary minus Pointer operator Address operator Type cast Size of an object	right to left	2
* / %	Multiplication Division Modular division	left to right	3
+ -	Addition Subtraction	left to right	4
<< >>	Left shift Right shift	left to right	5
< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	left to right	6
== !=	Equality Inequality	left to right	7
&	Bitwise AND	left to right	8
^	Bitwise XOR	left to right	9
|	Bitwise OR	left to right	10
&&	Logical AND	left to right	11
||	Logical OR	left to right	12
?:	Conditional operator	right to left	13
= += -= *= /= %= &= ^= |= <<= >>=	Assignment operators	right to left	14
,	Comma operator	left to right	15

Unary & +, -, and * have higher precedence than the binary forms.
Note that the precedence of the bitwise operators &, ^, and | falls below ==

and !=. This implies that bit-testing expressions like

```
if ((x & MASK) == 0) ...
```

must be fully parenthesized to give proper results.

C, like most languages, does not specify the order in which the operands of an operator are evaluated. (The exceptions are &&, ||, ?:, and ‘,’.) For example, in a statement like

```
x = f() + g();
```

f may be evaluated before g or vice versa; thus if either f or g alters a variable on which the other depends, x can depend on the order of evaluation. Intermediate results can be stored in temporary variables to ensure a particular sequence.

Similarly, the order in which function arguments are evaluated is not specified, so the statement

```
printf("%d %d\n", ++n, power(2, n));   /* WRONG */
```

can produce different results with different compilers, depending on whether n is incremented before power is called. The solution of course is to write

```
++n;
printf("%d %d\n", n, power(2, n));
```

Function calls, nested assignment statements, and increment and decrement operators cause “side effects” - some variable is changed as a by-product of the evaluation of an expression. In any expression involving side effects, there can be subtle dependencies on the order in which variables taking part in the expression are updated. One unhappy situation is typified by the statement

```
a[i] = i++;
```

The question is whether the subscript is the old value of i or the new. Compilers can interpret this in different ways and generate different answers depending on their interpretation. The standard intentionally leaves most such matters unspecified. When side effects (assignment to variables) take place within an expression is left to the discretion of the compiler, since the best order depends strongly on machine architecture. (The standard does specify that all side effects on arguments take effect before a function is called, but that would not help in the call to printf above.)

The moral is that writing code that depends on order of evaluation is a bad programming practice in any language. Naturally, it is necessary to know what things to avoid, but if you don’t know *how* they are done on various machines, you won’t be tempted to take advantage of a particular implementation.

1.4.10 Summary

Operators are the building blocks of expressions. The C language uses different kind of operators. There are arithmetic, relational, logical, assignment, conditional, comma and bitwise operators available in C.

1.4.11 Keywords

Operator: An operator indicates an operation to be performed on data that yields a value.

Operand : An operand is a data item on which operators performs the opeations.

Precedence : It is the priority for grouping different types of operators with their operands.

Associativity : It is the left to right or right to left order for grouping operands to operators that have the same precedence.

1.4.12 Short Answer Type Questions

1. What is the precedence of different arithmetic operators?
2. What is a ternary operator?
3. What are the various relational operators?
4. What is the role of comma operator?

1.4.13 Long Answer Type Questions

1. What is the difference between precedence and associativity?
2. What are the rule governing the use of logical operators?
3. How bitwise operators are used?

1.4.14 Suggested Readings

The C Programming language	Brian W. Kernigham Dennis M. Ritchie
Programming with ANSI and Turbo C	Ashok N. Kamthane
Programming using C	E. Balagurusamy
Let Us C	Yashavant Kanetkar

Web Resources :

www.cprogramming.com/tutorial/c/lesson1.html
www.programiz.com/c-programming
www.w3schools.in/c-tutorial/
www.learn-c.org
www.tutorialspoint.com/cprogramming/

LESSON NO. 1.5 **AUTHOR : DHARAM VEER SHARMA**

Library Functions

1.5.1 Introduction
1.5.2 Objectives of the Lesson
1.5.3 Arithmetic Functions
1.5.4 Character Functions
1.5.5 String Functions
1.5.6 Other Functions
1.5.7 Summary
1.5.8 Short Answer Type Questions
1.5.9 Long Answer Type Questions
1.5.10 Suggested Books

1.5.1 Introduction

Library functions in C are inbuilt functions which are grouped together and placed in a common place called Library. All C library functions are declared in many header files which are saved as File_name.u. These functions, available in any language, provide ready to use codes and facilitate rapid application development. The C language has a rich set of library functions which can make programming task very easy. It is not a subject matter of discussing all the functions here, still some important and frequently used functions will be discussed in this lesson.

1.5.2 Objectives of the Lesson

In this lesson we shall discuss various common library functions used in C programming. These functions include arithmetic, character, string etc.

1.5.3 Arithmetic Functions

All the mathematical functions of C language are declared in **math.h** header file. Following are some of the mathematical functions:

abs() and its variants:

abs (a macro) gets the absolute value of an integer

cabs and cabsl (macros) calculate the absolute value of a complex number

fabs and fabsl calculate the absolute value of a floating-point number

labs calculates the absolute value of a long number

Declaration:

```
int abs(int x);
```

```
double abs(complex x);
double cabs(struct complex z);
long double cabsl(struct _complexl (z));
double fabs(double x);
long double fabsl(long double @E(x));
long int labs(long int x);
```

Remarks:

All of these routines return the absolute value of their argument. abs, abs and cabsl are macros; fabs and labs are functions.

abs x, an integer
cabs z, a complex number
cabsl z, a complex number
fabs x, a double
labs x, a long

ceil, ceill, floor, floor:

ceil and **ceill** round up
floor and **floorl** round down

Declaration:

```
double ceil(double x);
double floor(double x);
long double ceill(long double (x));
long double floorl(long double (x));
```

Remarks:

ceil finds the smallest integer not < x.
ceill finds the smallest (long double) integer not < x.
floor finds the largest integer not > x.
floorl finds the largest (long double) integer not > x.

Return Value:

Both ceil and floor return the integer found as a double; ceill and floorl return the integer found as a long double.

Example (for both ceil and floor):

```
#include <math.h>
#include <stdio.h>
int main(void)
{
        double number = 123.54;
        double down, up;
        down = floor(number);
        up = ceil(number);
```

```
        printf("original number     %1.5.2lf\n", number);
        printf("number rounded down %5.2lf\n", down);
        printf("number rounded up   %5.2lf\n", up);
        return 0;
    }
```

pow, powl:

Power function, x to the y (x**y)

Declaration:

```
double pow(double x, double y);
long double pow(long double (x), long double (y));
```

Remarks:

pow and powl calculate x**y.

Return Value:

On success, pow and powl return the value calculated, x**y.

If x and y are both 0, they return 1.

If x is real and < 0, and y is not a whole number, these functions set errno to EDOM (domain error).

Sometimes the arguments passed to these functions produce results that overflow or are incalculable. When the correct value would overflow, pow returns HUGE_VAL and powl returns _LHUGE_VAL.

Results of excessively large magnitude can cause pow or powl to set errno to ERANGE (result out of range).

Error handling for pow can be modified via matherr; for powl, via _matherrl.

Example:

```
#include <math.h>
#include <stdio.h>
int main(void)
{
        double x = 2.0, y = 3.0;
printf("%lf raised to %lf is %lf\n", x, y, pow(x, y));
        return 0;
}
```

cos, sin, tan:

Cosine, sine, and tangent functions

Declaration:

```
double cos(double x);
double sin(double x);
double tan(double x);
long double cosl(long double x);
```

long double sinl(long double x);
long double tanl(long double x);

Remarks:

Real versions

cos and **cosl** compute the cosine of the input value
sin and **sinl** compute the sine of the input value
tan and **tanl** calculate the tangent of the input value

Angles are specified in radians.

Return Value:

On success, cos and cosl return the cosine of the input value (in the range -1 to 1)

sin and sinl return the sine of the input value (in the range -1 to 1)

tan returns the tangent of x, sin(x)/cos(x).

Error handling for these routines can be modified via matherr and _matherrl.

1.5.4 Character Functions

All the character type checking and conversion functions of C language are declared in **ctype.h** header file. Following are some of these functions:

tolower()

Translate characters to lowercase

Declaration:

int tolower(int ch);

Remarks:

tolower is a function that converts an integer ch (in the range EOF to 255) to its lowercase value (a to z; if it was uppercase, A to Z). All others are left unchanged.

Character classification macros

Declarations:

int isalnum(int c);
int islower(int c);
int isalpha(int c);
int isprint(int c);
int isascii(int c);
int ispunct(int c);
int iscntrl(int c);
int isspace(int c);
int isdigit(int c);
int isupper(int c);
int isgraph(int c);
int isxdigit(int c);

Remarks:

The is... macros classify ASCII coded integer values by table lookup.

Each macro is a predicate that returns a non-zero value for true and 0 for false.

isascii is defined on all integer values. The other is... macros are defined only when isascii(c) is true or c is EOF.

Return Value:

The is... macros return a non-zero value on success. For each macro, success is defined as follows:

isalpha: c is a letter (A to Z or a to z)

isascii: the low order byte of c is in the range 0 to 127 (0x00--0x7F)

iscntrl: c is a delete character or ordinary control character (0x7F or 0x00 to 0x1F)

isdigit: c is a digit (0 to 9)

isgraph: c is a printing character, like isprint, except that a space character is excluded

islower: c is a lowercase letter (a to z)

isprint: c is a printing character (0x20 to 0x7E)

ispunct: c is a punctuation character (iscntrl or isspace)

isspace: c is a space, tab, carriage return, new line, vertical tab, or formfeed (0x09 to 0x0D, 0x20)

isupper: c is an uppercase letter (A to Z)

isxdigit: c is a hexadecimal digit (0 to 9, A to F, a to f)

toupper()

Translate characters to uppercase

Declaration:

```
int toupper(int ch);
```

Remarks

toupper is a function that converts an integer ch (in the range EOF to 255) to its uppercase value (A to Z; if it was lowercase, a to z). All others are left unchanged.

1.5.5 String Functions

All the string related functions of C language are declared in string.h header file. Following are some of the string functions:

strcpy()

Copies one string into another

Declaration:

```
char *stpcpy(char *dest, const char *src);
```

Remarks:

stpcpy copies the string src to dest, stopping after the terminating null character of src has been reached.
Return Value: dest + strlen(src)
Example:

```
#include <stdio.h>
#include <string.h>
int main(void)
{
        char string[10];
        char *str1 = "abcdefghi";
        stpcpy(string, str1);
        printf("%s\n", string);
        return 0;
}
```

strcat()
Appends one string to another
Declaration:

```
char *strcat(char *dest, const char *src);
```

Remarks:
strcat appends a copy of src to the end of dest. The length of the resulting string is strlen(dest) + strlen(src).
Return Value:
strcat returns a pointer to the concatenated strings.
Example:

```
#include <string.h>
#include <stdio.h>
int main(void)
{
        char destination[25];
        char *blank = " ", *c = "C++", *turbo = "Turbo";
        strcpy(destination, turbo);
        strcat(destination, blank);
        strcat(destination, c);
        printf("%s\n", destination);
        return 0;
}
```

strchr()
Scans a string for the first occurrence of a given character
Declaration:

```
char *strchr(const char *s, int c);
```

Remarks:

strchr scans a string in the forward direction, looking for a specific character. The function finds the first occurrence of the character c in the string s.

The null-terminator is considered to be part of the string; for example,

```
strchr(strs, 0)
```

returns a pointer to the terminating null character of the string strs.

Return Value:

On success, returns a pointer to the first occurrence of the character c in string s.

On error (if c does not occur in s), returns null.

Example:

```
#include <string.h>
#include <stdio.h>
int main(void)
{
        char string[15];
        char *ptr, c = 'r';

        strcpy(string, "This is a string");
        ptr = strchr(string, c);
        if (ptr)
                printf("The character %c is at position: %d\n", c,
                ptr-string);
        else
                printf("The character was not found\n");
        return 0;
}
```

strcmp, strcmpi, stricmp

strcmp compares two strings

strcmpi (a macro) compares two strings without case sensitivity

stricmp compares two strings without case sensitivity

Declaration:

```
int strcmp(const char *s1, const char*s2);
int strcmpi(const char *s1, const char *s2)
int stricmp(const char *s1, const char *s2);
```

Remarks:

strcmp performs an unsigned comparison of s1 to s2.

strcmpi (implemented as a macro that calls stricmp) performs an unsigned comparison of s1 to s2, without case sensitivity.

stricmp performs an unsigned comparison of s1 to s2, without case sensitivity.

The string comparison starts with the first character in each string and continues with subsequent characters until the corresponding characters differ or until the end of the strings is reached.

To use strcmpi, you must include STRING.H. This macro is provided for compatibility with other C compilers.

Return Value:

These routines return an int value that is

< 0 if s1 < s2

== 0 if s1 == s2

> 0 if s1 > s2

strlen()

Calculates length of a string

Declaration:

```
size_t strlen(const char *s);
size_t far _fstrlen(const char far *s);
```

Remarks:

strlen calculates the length of s.

Return Value:

Returns the number of characters in s, not counting the terminating null character.

Example:

```
#include <stdio.h>
#include <string.h>
int main(void)
{
        char *string = "Borland International";
        printf("%d\n", strlen(string));
        return 0;
}
```

strlwr, strupr

Converts case of s

Declaration:

```
char *strlwr(char *s);
char *strupr(char *s);
```

Remarks:

strlwr converts uppercase letters (A to Z) in string s to lowercase (a to z).

strupr converts lowercase letters (a to z) in string s to uppercase (A to Z).

No other characters are changed.

Return Value:

A pointer to the string s.

1.5.6 Other Functions

The following are some of the other important functions frequently used in C programming:

clrscr() declared in <conio.h>

Clears the screen.

Declaration:

```
Void clrscr();
```

Remarks

clrscr clears the console.

Since its return type is void this function does not return any value.

atoi() declared in <stdlib.h>

Macro that converts string to integer

Declaration:

```
int atoi(const char *s);
```

Remarks:

atoi converts a string pointed to by s to int.

atoi recognizes (in the following order)

an optional string of tabs and spaces

an optional sign

a string of digits

The characters must match this format:

[ws] [sn] [ddd]

In atoi, the first unrecognized character ends the conversion.

There are no provisions for overflow in atoi (results are undefined).

Return Value:

On success, returns the converted value of the input string.

If the string can't be converted, returns 0.

Example:

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
        int n;
```

```
        char *str = "12345.67";
        n = atoi(str);
        printf("string = %s integer = %d\n", str, n);
        return 0;
}
```

malloc() declared in <ALLOC.H, STDLIB.H>

Allocates memory

Declaration:

```
void *malloc(size_t size);
```

Remarks:

malloc allocates a block of size bytes from the memory heap. It allows a program to allocate memory explicitly as it's needed and in the exact amounts needed.

The heap is used for dynamic allocation of variable-sized blocks of memory. Many data structures, such as trees and lists, naturally employ heap memory allocation.

All the space between the end of the data segment and the top of the program stack is available for use in the small data models, except for a small margin immediately before the top of the stack.

This margin is intended to allow the application some room to make the stack larger, in addition to a small amount needed by DOS.

In the large data models, all the space beyond the program stack to the end of available memory is available for the heap.

Return Value:

On success, malloc returns a pointer to the newly allocated block of memory.

On error (if not enough space exists for the new block), malloc returns null. The contents of the block are left unchanged.

If the argument size == 0, malloc returns null.

Example:

```
#include <stdio.h>
#include <string.h>
#include <alloc.h>
#include <process.h>
int main(void)
{
        char *str;
        /* allocate memory for string */
        if ((str = (char *) malloc(10)) == NULL)
        {
```

```
                printf("Not enough memory to allocate buffer\n");
                exit(1);  /* terminate program if out of memory */
        }
        /* copy "Hello" into string */
        strcpy(str, "Hello");
        /* display string */
        printf("String is %s\n", str);
        /* free memory */
        free(str);
        return 0;
}
```

farfree() declared in <ALLOC.H>

free() declared in <STDLIB.H, ALLOC.H>

farfree frees a block from far heap

free frees allocated blocks

Declaration:

```
void farfree(void far *block);
void free(void *block);
```

Remarks:

farfree releases a block of memory previously allocated from the far heap. A tiny model program can't use farfree.

free deallocates a memory block allocated by a previous call to calloc, malloc or realloc.

In the small and medium memory models,

blocks allocated by farmalloc can't be freed with free.

blocks allocated with malloc can't be freed with farfree.

In these models, the two heaps are completely distinct.

Return Value: None

1.5.7 Summary

Library function is a subroutine that is part of a function library. Library functions are also called inbuilt functions. These are the functions which are provided with the built in language definition. The C language supports a huge set of library functions. The most commonly used functions are mathematical, string or character functions.

1.5.8 Short Answer Type Questions

1. What are the mathematical functions for truncation and rounding of real values?
2. What are functions for checking and converting case of characters?
3. Through which function can we know the length of the string?

1.5.9 Long Answer Type Questions

1. What are the various trigonometric functions available in C?
2. What are various character functions for checking type of the character?
3. Which string functions are used for copying, concatenation and comparison of strings?
4. What are the various memory management functions available in C?

1.5.10 Suggested Books

The C Programming language	Brian W. Kernigham Dennis M. Ritchie
Programming with ANSI and Turbo C	Ashok N. Kamthane
Programming using C	E. Balagurusamy
Application Programming in C	R. S. Salaria
Let us C	Yashavant Karetkar

Web Resources

www.cprogramming.com/tutorial/c/lesson1.html
www.programiz.com/c-programming
www.w3schools.in/c-tutorial/
www.learn-c.org
www.tutorialspoint.com/cprogramming/

LESSON NO. 1.6 | **AUTHOR : DHARAM VEER SHARMA**

Sequential and Conditional Control Statements

1.6.1 Introduction
1.6.2 Objectives of the Lesson
1.6.3 Statements and Blocks
1.6.4 if ... else Construct
1.6.5 else ... if
1.6.6 Using logical operators in if construct
1.6.7 switch ... case Construct
1.6.8 goto and labels
1.6.9 Summary
1.6.10 Short Answer Type Questions
1.6.11 Long Answer Type Questions
1.6.12 Suggested Books

1.6.1 Introduction

The control-flow of a language specifies the order in which computations are performed. In C language there are sequential, conditional and iterative control structures available for program design. In this lesson we shall discuss the various conditional control structures and iterative control structures will be discussed in the next lesson. Conditional constructs are required for making decision and choosing some execution path based on the satisfied condition.

1.6.2 Objectives of the Lesson

In this lesson we shall discuss the various conditional control structures available in the C language. These constructs provide branching with in the program based on some condition.

1.6.3 Statements and Blocks

An expression such as x = 0 or i++ or printf(...) becomes a *statement* when it is followed by a semicolon, as in

```
x = 0;
i++;
printf(...);
```

In C, the semicolon is a statement terminator rather than a separator as it is in languages like Pascal.

Braces { and } are used to group declarations and statements together into a

compound statement or *block*, so that they are syntactically equivalent to a single statement. The braces that surround the statements of a function are one obvious example; braces around multiple statements after an if, else, while, or for are another. (Variables can be declared inside *any* block) There is no semicolon after the right brace that ends a block. A block can be created anywhere with in the program.

Example

```
main()
{
        int a = 10;
        {
                int a = 20;
                printf("Value of a inside the block is -> %d",a);
        }
        printf("Value of a outside the block is -> %d",a);
        return 0;
}
```

In the above example, for the second declaration of a, a's scope is limited to the block only and output will be 20 for the first printf statement and for the second printf the output will be 10.

1.6.4 if ... else Construct

The if-else statement is used to express decisions. Formally the syntax is

if (*expression*)

statement$_1$

else

statement$_2$

where the else part is optional. The *expression* is evaluated; if it is true (that is, if *expression* has a non-zero value), *statement*$_1$ is executed. If it is false (*expression* is zero) and if there is an else part, *statement*$_2$ is executed instead.

Since an if tests the numeric value of an expression, certain coding shortcuts are possible. The most obvious is writing

if (*expression*)

instead of

if (*expression* != 0)

Sometimes this is natural and clear; at other times it can be cryptic.

Because the else part of an if-else is optional, there is an ambiguity when an else if omitted from a nested if sequence. This is resolved by associating the else with the closest previous else-less if. For example, in

```
if (n > 0)
        if (a > b)
```

```
                z = a;
        else
                z = b;
```

the else goes to the inner if, as we have shown by indentation. If that isn't what you want, braces must be used to force the proper association:

```
if (n > 0) {
        if (a > b)
                z = a;
}
else
        z = b;
```

The ambiguity is especially pernicious in situations like this:

```
if (n > 0)
        for (i = 0; i < n; i++)
                if (s[i] > 0) {
                        printf("...");
                        return i;
                }
else    /* WRONG */
        printf("error -- n is negative\n");
```

The indentation shows unequivocally what you want, but the compiler doesn't get the message and associates the else with the inner if. This kind of bug can be hard to find; it's a good idea to use braces when there are nested ifs.

By the way, notice that there is a semicolon after z = a in

```
if (a > b)
        z = a;
else
        z = b;
```

This is because grammatically, a *statement* follows the if and an expression statement like ``z = a;'' is always terminated by a semicolon.

1.6.5 else ... if

The construction

if (*expression*)
    statement
else if (*expression*)
    statement
else if (*expression*)
    statement
else if (*expression*)

statement

else

statement

occurs so often that it is worth a brief separate discussion. This sequence of if statements is the most general way of writing a multi-way decision. The *expressions* are evaluated in order; if an *expression* is true, the *statement* associated with it is executed and this terminates the whole chain. As always, the code for each *statement* is either a single statement, or a group of them in braces.

The last else part handles the ``none of the above'' or default case where none of the other conditions is satisfied. Sometimes there is no explicit action for the default; in that case the trailing

else

statement

can be omitted, or it may be used for error checking to catch an ``impossible'' condition.

To illustrate a three-way decision, here is a binary search function that decides if a particular value x occurs in the sorted array v. The elements of v must be in increasing order. The function returns the position (a number between 0 and n-1) if x occurs in v and -1 if not.

Binary search first compares the input value x to the middle element of the array v. If x is less than the middle value, searching focuses on the lower half of the table, otherwise on the upper half. In either case, the next step is to compare x to the middle element of the selected half. This process of dividing the range in two continues until the value is found or the range is empty.

```
/* binsearch:  find x in v[0] <= v[1] <= ... <= v[n-1] */
int binsearch(int x, int v[], int n)
{
        int low, high, mid;
        low = 0;
        high = n - 1;
        while (low <= high) {
                mid = (low+high)/2;
                if (x < v[mid])
                        high = mid + 1;
                else if (x  > v[mid])
                        low = mid + 1;
                else    /* found match */
                        return mid;
        }
        return -1;   /* no match */
```

```
}
```

The fundamental decision is whether x is less than, greater than or equal to the middle element v[mid] at each step; this is a natural for else-if.

Utmost care should be taken while using conditional constructs as is evident from the following example.

```
if (day == 1)
        printf("Monday");
if (day == 2)
        printf("Tuesday");
if (day == 3)
        printf("Wednesday");
if (day == 4)
        printf("Thursday");
if (day == 5)
        printf("Friday");
if (day == 6)
        printf("Saturday");
else
        printf("Sunday");
```

The above use of if is wrong as for any value of day between 1 and 5 it will print the Sunday as well because in the last if statement it will always printf Sunday if value of day is not 6. Therefore, in the above example if else if construct should be used as given below.

```
if (day == 1)
        printf("Monday");
else if (day == 2)
        printf("Tuesday");
else if (day == 3)
        printf("Wednesday");
else if (day == 4)
        printf("Thursday");
else if (day == 5)
        printf("Friday");
else if (day == 6)
        printf("Saturday");
else
        printf("Sunday");
```

In this case any condition will be checked only if its previous condition is false.

1.6.6 Using logical operators in if construct

Logical operators are indispensable part of if ... else construct, but these should be used with utmost caution. There is a need of understanding the way these are evaluated.

```
if (condition1 && condition2)
        statement
```

In this construct condition2 is evaluated only if condition1 is true, otherwise condition2 is never reached. Therefore if some calculation is involved in condition2 then that calculation will also not be performed. Therefore, care should be taken while using && operator.

```
if (condition1 || condition2)
        statement1
```

In this construct conditon2 is evaluated only if condition1 is false, otherwise condition2 is never reached. Therefore the problem is the same as in the case of && operator.

1.6.7 switch ... case Construct

The switch statement allows you to select from multiple choices based on a set of fixed values for a given expression or the switch statement is a multi-way decision that tests whether an expression matches one of a number of *constant* integer values and branches accordingly.

```
switch (expression) {
        case const-expr: statements
        case const-expr: statements
        default: statements
}
```

Important distinction between use of if and switch construct is that for each switch construct there is an equivalent if construct available. However, the reverse is not true. switch can replace only those if constructs where the value of only one variable is tested for different integer values.

Example

```
if (day == 1)
        printf("Monday");
else if (day == 2)
        printf("Tuesday");
else if (day == 3)
        printf("Wednesday");
else if (day == 4)
        printf("Thursday");
else if (day == 5)
        printf("Friday");
```

```
else if (day == 6)
        printf("Saturday");
else
        printf("Sunday");
```

For this situation where value of day is checked, switch construct is the most suitable.

```
switch (day)
{
        case 1: printf("Monday");
                break;
        case 2: printf("Tuesday");
                break;
        case 3: printf("Wednesday");
                break;
        case 4: printf("Thursday");
                break;
        case 5: printf("Friday");
                break;
        case 6: printf("Saturday");
                break;
        case 7: printf("Sunday");
                break;
}
```

Each case is labeled by one or more integer-valued constants or constant expressions. If a case matches the expression value, execution starts at that case. All case expressions must be different. The case labeled default is executed if none of the other cases are satisfied. A default is optional; if it isn't there and if none of the cases match, no action at all takes place. Cases and the default clause can occur in any order.

1.6.8 goto and labels

C provides the infinitely-abusable goto statement and labels to branch to. Formally, the goto statement is never necessary and in practice it is almost always easy to write code without it. We have not used goto in this book.

Nevertheless, there are a few situations where gotos may find a place. The most common is to abandon processing in some deeply nested structure, such as breaking out of two or more loops at once. The break statement cannot be used directly since it only exits from the innermost loop. Thus:

```
for ( ... )
        for ( ... ) {
```

```
                ...
                if (disaster)
                goto error;
        }
        ...
error:
/* clean up the mess */
```

This organization is handy if the error-handling code is non-trivial and if errors can occur in several places.

A label has the same form as a variable name and is followed by a colon. It can be attached to any statement in the same function as the goto. The scope of a label is the entire function.

As another example, consider the problem of determining whether two arrays a and b have an element in common. One possibility is

```
for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
                if (a[i] == b[j])
                        goto found;
                /* didn't find any common element */
...
found:
/* got one: a[i] == b[j] */
...
```

Code involving a goto can always be written without one, though perhaps at the price of some repeated tests or an extra variable. For example, the array search becomes

```
found = 0;
for (i = 0; i < n && !found; i++)
        for (j = 0; j < m && !found; j++)
                if (a[i] == b[j])
                        found = 1;
if (found)
        /* got one: a[i-1] == b[j-1] */
        ...
else
        /* didn't find any common element */
        ...
```

With a few exceptions like those cited here, code that relies on goto statements is generally harder to understand and to maintain than code without

gotos. Although we are not dogmatic about the matter, it does seem that goto statements should be used rarely, if at all.

1.6.9 Summary

Conditional flow of program provides decision making ability. In C language there are if …else and switch constructs for implementing conditional flow. if statements can be nested. In case of switch statement value of some variable is checked against integer constants. Logical operators are to be used cautiously for combining conditions.

1.6.10 Short Answer Type Questions

1. What are the various conditional constructs available in C?
2. What is the purpose of switch statement?
3. Why break is needed after cases?
4. What is a default case?

1.6.11 Long Answer Type Questions

1. What is the difference between if and switch constructs?
2. In what type of situation switch will be preferred over if statements?
3. What are the rules of using logical operators with conditions?
4. Is it possible to replace all kinds of if constructs with switch? If not then why?
5. WAP to check whether the person is eligible for voting or not.
6. WAP to check whether the entered number is +ve or –ve.
7. WAP to calculate the total marks, percentage and division of the student using nested If-Else statement.
8. WAP to print day of the week using switch statement.

1.6.12 Suggested Books

The C Programming language	Brian W. Kernigham Dennis M. Ritchie
Programming with ANSI and Turbo C	Ashok N. Kamthane
Programming using C	E. Balagurusamy
Application Programming in C	R. S. Salaria
Let us C	Yashwant Kanettkar

Web Resources

www.cprogramming.com/tutorial/c/lesson1.html
www.programiz.com/c-programming
www.w3schools.in/c-tutorial/
www.learn-c.org
www.tutorialspoint.com/cprogramming/

LESSON NO. 1.7 **AUTHOR : DHARAM VEER SHARMA**

Iterative Control Statements

1.7.1 Introduction
1.7.2 Objectives of the Lesson
1.7.3 while Loop
1.7.4 for Loop
1.7.5 do ... while Loop
1.7.6 Nested loops
1.7.7 Sequence Breaking Control Statements
1.7.8 Summary
1.7.9 Short Answer Type Questions
1.7.10 Long Answer Type Questions
1.7.11 Suggested Books

1.7.1 Introduction

Iterative control structures provide repetitive computations. In case where some set of statements are to be executed repeatedly, the iterative control structures can be used. The C language provides three iterative control structures namely for, while and do ... while loops. for and while loops are entry control loops where as do ... while is an exit controlled loop.

1.7.2 Objectives of the Lesson

In this lesson we shall discuss the various iterative control structures available in the C language. These constructs provide repetitive execution of some set of statements.

1.7.3 while Loop

As discussed earlier while is an entry controlled loop, means if the condition is true only then statements under the loop will be executed. In

while (*expression*)
statement

the *expression* is evaluated, if it is non-zero, *statement* is executed and *expression* is re-evaluated. This cycle continues until *expression* becomes zero, at which point execution resumes after *statement*. However if the expression evaluates to zero then the statement is not executed. while loop is used in case the number of iterations are not known in advance.

The following variant of while loop produces an infinite loop

while (1)

In this case since expression always evaluates to non-zero value, the loop is not terminated by just checking the expression. Instead, some internal control breaking mechanism is required to come out of the loop. The mechanism has been discussed later in this lesson.

1.7.4 for Loop

Like while loop, for is also an entry controlled loop. The for statement

for (*expr*$_1$; *expr*$_2$; *expr*$_3$)
statement

is equivalent to

expr$_1$;
while (*expr*$_2$) {
statement
expr$_3$;
}

except for the behaviour of break or continue.

A loop control variable is used for controlling the number of times for which the loop will be executed.

Grammatically, the three components of a for loop are expressions. Most commonly, *expr*$_1$ and *expr*$_3$ are assignments or function calls and *expr*$_2$ is a relational expression. Any of the three parts can be omitted, although the semicolons must remain. If *expr*$_1$ or *expr*$_3$ is omitted, it is simply dropped from the expansion. If the test, *expr*$_2$, is not present, it is taken as permanently true, so

for (;;) {
...
}

is an ``infinite'' loop, presumably to be broken by other means, such as a break or return.

The three expressions of the for loop are executed in the following way:

expr1: It is executed only once, primarily for initializing the loop control variable.

expr2: It is executed repeatedly for checking the truthfulness of condition, up to which the body of the loop is to be executed. When the condition evaluates to false the loop is terminated.

expr3: It is also repeatedly executed for incrementing or decrementing the value of the loop control variable.

In general, for loop is used when number of iterations is known in advance and while is used when iterations are not known. However these can be used

interchangeably. Whether to use while or for largely becomes a matter of personal preference. For example, in

```
while ((c = getchar()) == ' ' || c == '\n' || c = '\t')
        ;   /* skip white space characters */
```

there is no initialization or re-initialization, so the while is most natural.

The for is preferable when there is a simple initialization and increment since it keeps the loop control statements close together and visible at the top of the loop. This is most obvious in

```
for (i = 0; i < n; i++)
        ...
```

which is the C idiom for processing the first n elements of an array, the analog of the Fortran DO loop or the Pascal for. The analogy is not perfect, however, since the index variable i retains its value when the loop terminates for any reason. Because the components of the for are arbitrary expressions, for loops are not restricted to arithmetic progressions. Nonetheless, it is bad style to force unrelated computations into the initialization and increment of a for, which are better reserved for loop control operations.

As a larger example, here is another version of atoi for converting a string to its numeric equivalent. It copes with optional leading white space and an optional + or - sign.

The structure of the program reflects the form of the input:

skip white space, if any
get sign, if any
get integer part and convert it

Each step does its part, and leaves things in a clean state for the next. The whole process terminates on the first character that could not be part of a number.

```
#include <ctype.h>
/* atoi:  convert s to integer; version 2 */
int atoi(char s[])
{
        int i, n, sign;
        for (i = 0; isspace(s[i]); i++)  /* skip white space */
        ;
        sign = (s[i] == '-') ? -1 : 1;
        if (s[i] == '+' || s[i] == '-')  /* skip sign */
                i++;
        for (n = 0; isdigit(s[i]); i++)
                n = 10 * n + (s[i] - '0');
        return sign * n;
```

```
}
```

The standard library provides a more elaborate function strtol for conversion of strings to long integers.

The advantages of keeping loop control centralized are even more obvious when there are several nested loops. The following function is a Shell sort for sorting an array of integers.

The basic idea of this sorting algorithm, which was invented in 1959 by D. L. Shell, is that in early stages, far-apart elements are compared, rather than adjacent ones as in simpler interchange sorts. This tends to eliminate large amounts of disorder quickly, so later stages have less work to do. The interval between compared elements is gradually decreased to one, at which point the sort effectively becomes an adjacent interchange method.

```
/* shellsort:  sort v[0]...v[n-1] into increasing order */
void shellsort(int v[], int n)
{
        int gap, i, j, temp;
        for (gap = n/2; gap > 0; gap /= 2)
                for (i = gap; i < n; i++)
                        for (j=i-gap; j>=0 && v[j]>v[j+gap]; j-=gap) {
                                temp = v[j];
                                v[j] = v[j+gap];
                                v[j+gap] = temp;
                        }
}
```

There are three nested loops. The outermost controls the gap between compared elements, shrinking it from n/2 by a factor of two each pass until it becomes zero. The middle loop steps along the elements. The innermost loop compares each pair of elements that is separated by gap and reverses any that are out of order. Since gap is eventually reduced to one, all elements are eventually ordered correctly. Notice how the generality of the for makes the outer loop fit in the same form as the others, even though it is not an arithmetic progression.

One final C operator is the comma ``,'', which most often finds use in the for statement. A pair of expressions separated by a comma is evaluated left to right, and the type and value of the result are the type and value of the right operand. Thus in a for statement, it is possible to place multiple expressions in the various parts, for example to process two indices in parallel. This is illustrated in the function reverse(s), which reverses the string s in place.

```
#include <string.h>
/* reverse:  reverse string s in place */
```

```
void reverse(char s[])
{
int c, i, j;

        for (i = 0, j = strlen(s)-1; i < j; i++, j--) {
                c = s[i];
                s[i] = s[j];
                s[j] = c;
        }
}
```

The commas that separate function arguments, variables in declarations, etc., are *not* comma operators and do not guarantee left to right evaluation.

Comma operators should be used sparingly. The most suitable uses are for constructs strongly related to each other, as in the for loop in reverse and in macros where a multistep computation has to be a single expression. A comma expression might also be appropriate for the exchange of elements in reverse, where the exchange can be thought of a single operation:

```
for (i = 0, j = strlen(s)-1; i < j; i++, j--)
        c = s[i], s[i] = s[j], s[j] = c;
```

Forms of for loop

Syntax	Output	Remarks
for (;;)	Infinite loop	No arguments means condition is always true, therefore the loop executes for infinite number of times.
for (a=0;a<=20;)	Infinite loop	Value of 'a' is not modified, therefore, condition will always evaluate to true making the loop an infinite loop.
for (a=0;a<=10;a++) printf("%d ",a);	Displays values from 0 to 10	Since initial value of 'a' is 0 and it is incremented by one, the values from 0 to 10 will be printed. When 'a' will become 11 the condition will be false and loop will terminate.
for (a=10;a>=0;a--)	Displays values from 10 to 1	Since initial value of 'a' is 10 and it is decremented by one, the values from 10 to 0 will be printed. When 'a' will become -1 the condition will be false and loop will terminate.

1.7.5 do ... while Loop

As we discussed in lesson 1, the while and for loops test the termination condition at the top. By contrast, the third loop in C, the do-while, tests at the bottom *after* making each

pass through the loop body; the body is always executed at least once. Therefore, do … while loop is termed as exit controlled loop.

The syntax of the do is

do

statement

while (*expression*);

The *statement* is executed, then *expression* is evaluated. If it is true, *statement* is evaluated again and so on. When the expression becomes false, the loop terminates. Except for the sense of the test, do-while is equivalent to the Pascal repeat-until statement.

Experience shows that do-while is much less used than while and for. Nonetheless, from time to time it is valuable, as in the following function itoa, which converts a number to a character string (the inverse of atoi). The job is slightly more complicated than might be thought at first, because the easy methods of generating the digits generate them in the wrong order. We have chosen to generate the string backwards, then reverse it.

```
/* itoa:  convert n to characters in s */
void itoa(int n, char s[])
{
        int i, sign;
        if ((sign = n) < 0)  /* record sign */
                n = -n;          /* make n positive */
        i = 0;
        do {      /* generate digits in reverse order */
                s[i++] = n % 10 + '0';  /* get next digit */
        } while ((n /= 10) > 0);    /* delete it */
        if (sign < 0)
                s[i++] = '-';
        s[i] = '\0';
        reverse(s);
}
```

The do-while is necessary or at least convenient, since at least one character must be installed in the array s, even if n is zero. We also used braces around the single statement that makes up the body of the do-while, even though they are unnecessary, so the hasty reader will not mistake the while part for the *beginning* of a while loop.

do … while construct is best suited for situations where some program or block is to be repeatedly executed but that must be executed at least once. Following is the example demonstrating the use of do … while construct:

Example

```
/* program for finding sum any n numbers */
#include <stdio.h>
#include <conio.h>
void main()
{
        int i, x, n, sum;
        char choice;
        do {
                sum = 0;
                clrscr();
                printf("Enter the number of values -> ");
                scanf("%d",&n);
                for (i=0; i < n;i++)
                {
                        printf("\nEnter some number -> ");
                        scanf("%d",&x);
                        sum += x;
                }
                printf("\nSum of entered numbers is -> %d",sum);
                printf("Want to run the program again <Y/N>");
                choice = getch();
        } while (choice == 'Y' || choice == 'y');
        printf("\n!!! That's all folks !!!");
        getch();
}
```

In the above example the program runs for the first time and then asks the user if he wants to run the program again. Depending on the choice of the user the program either executes again or is exits.

1.7.6 Nested loops

Loops can be nested in any order within a program. If iterations with in iterations are to be performed then nested loops can be used. In such cases loop control variable, which controls the number of iterations to be performed, should be chosen separately for each of the inner loops.

This can be well demonstrated from the following example in which all possible outcomes of throwing three dice can be generated:

Example

```
#include <stdio.h>
#include <conio.h>
void main()
{
        int i,j,k;
        printf("Possible outcomes of throwing three dice are -> \n");
        for (i=1;i<=6;i++)
                for (j=1;j<=6;j++)
                        for (k=1;k<=6;k++)
                                printf("%d %d %d\t",i,j,k);
}
```

This program will produce all possible outcomes of throwing three dice simultaneously.

1.7.7 Sequence Breaking Control Statements

It is sometimes convenient to be able to exit from a loop other than by testing at the top or bottom. The break statement provides an early exit from for, while and do, just as from switch. A break causes the innermost enclosing loop or switch to be exited immediately.

The following function, trim, removes trailing blanks, tabs and newlines from the end of a string, using a break to exit from a loop when the rightmost non-blank, non-tab, non-newline is found.

```
/* trim:  remove trailing blanks, tabs, newlines */
int trim(char s[])
{
        int n;
        for (n = strlen(s)-1; n >= 0; n--)
                if (s[n] != ' ' && s[n] != '\t' && s[n] != '\n')
                        break;
        s[n+1] = '\0';
        return n;
}
```

strlen returns the length of the string. The for loop starts at the end and scans backwards looking for the first character that is not a blank or tab or newline. The loop is broken when one is found or when n becomes negative (that is, when the entire string has been scanned). You should verify that this is correct behavior even when the string is empty or contains only white space characters.

The continue statement is related to break, but less often used; it causes the next iteration of the enclosing for, while, or do loop to begin. In the while and do, this

means that the test part is executed immediately; in the for, control passes to the increment step. The continue statement applies only to loops, not to switch. A continue inside a switch inside a loop causes the next loop iteration.

As an example, this fragment processes only the non-negative elements in the array a; negative values are skipped.

```
for (i = 0; i < n; i++)
        if (a[i] < 0)   /* skip negative elements */
                continue;
... /* do positive elements */
```

The continue statement is often used when the part of the loop that follows is complicated, so that reversing a test and indenting another level would nest the program too deeply.

1.7.8 Summary

Iterative control structures are used when some statements are to be executed repetitively. In C language there are three iterative control structures available. for and while are entry controlled loops and can be used interchangeably. do …while is an exit controlled loop. for or while loops should not be used in place of do … while loop.

1.7.9 Short Answer Type Questions

1. What are the various iterative controlled structures available in C?
2. What is the difference between for and while loop?
3. Atleast how many times statements are executed in do … while loop?
4. What is the difference between break and continue?

1.7.10 Long Answer Type Questions

1. Discuss in detail the syntax and use of for and while loops.
2. Is it possible to use for or while loops in place of do … while loop? Explain your answer.
3. Explain the meaning of sequence breaking control statements.
4. WAP to print Even numbers less than 100.
5. WAP to print the following :

```
1
1  2
1  2  3
1  2  3  4
1  2  3  4  5
```

1.7.11 Suggested Books

The C Programming language	Brian W. Kernigham Dennis M. Ritchie
Programming with ANSI and Turbo C	Ashok N. Kamthane
Programming using C	E. Balagurusamy
Application Programming in C	R. S. Salaria
Let us C	Yashwant Kanettkar

Web Resources

www.cprogramming.com/tutorial/c/lesson1.html
www.programiz.com/c-programming
www.w3schools.in/c-tutorial/
www.learn-c.org
www.tutorialspoint.com/cprogramming/

LESSON NO. 1.8 **AUTHOR : DHARAM VEER SHARMA**

FUNCTIONS

1.8.1 Introduction
1.8.2 Objectives of the Lesson
1.8.3 Basics of Functions
1.8.4 Functions Returning Non-integers
1.8.5 External Variables
1.8.6 Formal and Actual Arguments
1.8.7 Scope Rules
1.8.8 Header Files
1.8.9 Recursion
1.8.10 Summary
1.8.11 Short Answer Type Questions
1.8.12 Long Answer Type Questions
1.8.13 Suggested Books

1.8.1 Introduction

A Function in C is a block of code that performs a specific task. The C language is basically a functional programming language, in which the program is divided in small manageable pieces of code called functions, which break large computing tasks into smaller ones and enable people to build on what others have done instead of starting over from scratch. Appropriate functions hide details of operation from parts of the program that don't need to know about them, thus clarifying the whole and easing the pain of making changes.

C has been designed to make functions efficient and easy to use; C programs generally consist of many small functions rather than a few big ones. A program may reside in one or more source files. Source files may be compiled separately and loaded together, along with previously compiled functions from libraries. We will not go into that process here, however, since the details vary from system to system.

1.8.2 Objectives of the Lesson

In this lesson, we shall discuss the program structuring tool, called function. We shall discuss function prototyping, function definition and function calling methods. The discussion will also include methods of parameter passing and recursive functions.

1.8.3 Basics of Functions

A function is a block of code that performs a calculation and returns a value. To begin with, let us design and write a program to print each line of its input that contains a particular ``pattern'' or string of characters. For example, searching for the pattern of letters ``ould'' in the set of lines

Ah Love! could you and I with Fate conspire
To grasp this sorry Scheme of Things entire,
Would not we shatter it to bits -- and then
Re-mould it nearer to the Heart's Desire!

will produce the output

Ah Love! could you and I with Fate conspire
Would not we shatter it to bits -- and then
Re-mould it nearer to the Heart's Desire!

The job falls neatly into three pieces:

while (*there's another line*)
if (*the line contains the pattern*)
print it

Although it's certainly possible to put the code for all of this in main, a better way is to use the structure to advantage by making each part a separate function. Three small pieces are better to deal with than one big one, because irrelevant details can be buried in the functions and the chance of unwanted interactions is minimized. And the pieces may even be useful in other programs.

``While there's another line'' is getline, a function that we wrote in lesson 1, and ``print it'' is printf, which someone has already provided for us. This means we need only write a routine to decide whether the line contains an occurrence of the pattern.

We can solve that problem by writing a function strindex(s,t) that returns the position or index in the string s where the string t begins or -1 if s does not contain t. Because C arrays begin at position zero, indexes will be zero or positive and so a negative value like -1 is convenient for signaling failure. When we later need more sophisticated pattern matching, we only have to replace strindex; the rest of the code can remain the same. (The standard library provides a function strstr that is similar to strindex, except that it returns a pointer instead of an index.)

Given this much design, filling in the details of the program is straightforward. Here is the whole thing, so you can see how the pieces fit together. For now, the pattern to be searched for is a literal string, which is not the most general of mechanisms. We will return shortly to a discussion of how to initialize character arrays, and will show how to make the pattern a parameter that is set when the program is run. There is also a slightly different version of getline; you might find it instructive to compare it to the one in lesson 2.

```
#include <stdio.h>
#define MAXLINE 1000 /* maximum input line length */
int getline(char line[], int max)
int strindex(char source[], char searchfor[]);
char pattern[] = "ould";   /* pattern to search for */
/* find all lines matching pattern */
main()
{
        char line[MAXLINE];
        int found = 0;
        while (getline(line, MAXLINE) > 0)
                if (strindex(line, pattern) >= 0) {
                        printf("%s", line);
                        found++;
                }
        return found;
}
/* getline:  get line into s, return length */
int getline(char s[], int lim)
{
        int c, i;
        i = 0;
        while (--lim > 0 && (c=getchar()) != EOF && c != '\n')
                s[i++] = c;
        if (c == '\n')
                s[i++] = c;
        s[i] = '\0';
        return i;
}

/* strindex:  return index of t in s, -1 if none */
int strindex(char s[], char t[])
{
        int i, j, k;
        for (i = 0; s[i] != '\0'; i++) {
                for (j=i, k=0; t[k]!='\0' && s[j]==t[k]; j++, k++)
                        ;
                if (k > 0 && t[k] == '\0')
                        return i;
```

```
        }
        return -1;
}
```

Each function definition has the form

return-type function-name(*argument declarations*)
{
    declarations and statements
}

Various parts may be absent; a minimal function is

```
dummy() {}
```

which does nothing and returns nothing. A do-nothing function like this is sometimes useful as a place holder during program development. If the return type is omitted, int is assumed.

A program is just a set of definitions of variables and functions. Communication between the functions is by arguments and values returned by the functions, and through external variables. The functions can occur in any order in the source file and the source program can be split into multiple files, so long as no function is split.

The return statement is the mechanism for returning a value from the called function to its caller. Any expression can follow return:

return *expression*;

The *expression* will be converted to the return type of the function if necessary. Paresthesis are often used around the *expression*, but they are optional.

The calling function is free to ignore the returned value. Furthermore, there need to be no expression after return; in that case, no value is returned to the caller. Control also returns to the caller with no value when execution ``falls off the end'' of the function by reaching the closing right brace. It is not illegal, but probably a sign of trouble, if a function returns a value from one place and no value from another. In any case, if a function fails to return a value, its ``value'' is certain to be garbage.

The pattern-searching program returns a status from main, the number of matches found. This value is available for use by the environment that called the program

The mechanics of how to compile and load a C program that resides on multiple source files vary from one system to the other. Suppose that the three functions are stored in three files called main.c, getline.c and strindex.c. Then the command

```
cc main.c getline.c strindex.c
```

compiles the three files, placing the resulting object code in files main.o, getline.o, and strindex.o, then loads them all into an executable file called a.out. If there is an error say in main.c, the file can be recompiled by itself and the result loaded with the previous object files, with the command

cc main.c getline.o strindex.o

The cc command uses the ``.c'' versus ``.o'' naming convention to distinguish source files from object files.

1.8.4 Functions Returning Non-integers

So far our examples of functions have returned either no value (void) or an int. What if a function must return some other type? many numerical functions like sqrt, sin, and cos return double; other specialized functions return other types. To illustrate how to deal with this, let us write and use the function atof(s), which converts the string s to its double-precision floating-point equivalent. atof if an extension of atoi, which we showed versions of in lessons 2 and 3. It handles an optional sign and decimal point, and the presence or absence of either part or fractional part. Our version is *not* a high-quality input conversion routine; that would take more space than we care to use. The standard library includes an atof; the header <stdlib.h> declares it.

First, atof itself must declare the type of value it returns, since it is not int. The type name precedes the function name:

```
#include <ctype.h>
/* atof:  convert string s to double */
double atof(char s[])
{
        double val, power;
        int i, sign;
        for (i = 0; isspace(s[i]); i++)  /* skip white space */
                ;
        sign = (s[i] == '-') ? -1 : 1;
        if (s[i] == '+' || s[i] == '-')
                i++;
        for (val = 0.0; isdigit(s[i]); i++)
                val = 10.0 * val + (s[i] - '0');
        if (s[i] == '.')
                i++;
        for (power = 1.0; isdigit(s[i]); i++) {
                val = 10.0 * val + (s[i] - '0');
                power *= 10;
        }
        return sign * val / power;
}
```

Second, and just as important, the calling routine must know that atof returns a non-int value. One way to ensure this is to declare atof explicitly in the

calling routine. The declaration is shown in this primitive calculator (barely adequate for check-book balancing), which reads one number per line, optionally preceded with a sign and adds them up, printing the running sum after each input:

```
#include <stdio.h>
#define MAXLINE 100
/* rudimentary calculator */
main()
{
        double sum, atof(char []);
        char line[MAXLINE];
        int getline(char line[], int max);
        sum = 0;
        while (getline(line, MAXLINE) > 0)
                printf("\t%g\n", sum += atof(line));
        return 0;
}
```

The declaration

```
double sum, atof(char []);
```

says that sum is a double variable and that atof is a function that takes one char[] argument and returns a double.

The function atof must be declared and defined consistently. If atof itself and the call to it in main have inconsistent types in the same source file, the error will be detected by the compiler. But if (as is more likely) atof were compiled separately, the mismatch would not be detected, atof would return a double that main would treat as an int and meaningless answers would result.

In the light of what we have said about how declarations must match definitions, this might seem surprising. The reason a mismatch can happen is that if there is no function prototype, a function is implicitly declared by its first appearance in an expression, such as

```
sum += atof(line)
```

If a name that has not been previously declared occurs in an expression and is followed by a left parentheses, it is declared by context to be a function name, the function is assumed to return an int and nothing is assumed about its arguments. Furthermore, if a function declaration does not include arguments, as in

```
double atof();
```

that too is taken to mean that nothing is to be assumed about the arguments of atof; all parameter checking is turned off. This special meaning of the empty argument list is intended to permit older C programs to compile with new compilers. But it's a bad

idea to use it with new C programs. If the function takes arguments, declare them; if it takes no arguments, use void.

Given atof, properly declared, we could write atoi (convert a string to int) in terms of it:

```
/* atoi:  convert string s to integer using atof */
int atoi(char s[])
{
        double atof(char s[]);
        return (int) atof(s);
}
```

Notice the structure of the declarations and the return statement. The value of the expression in

return *expression*;

is converted to the type of the function before the return is taken. Therefore, the value of atof, a double, is converted automatically to int when it appears in this return, since the function atoi returns an int. This operation does potentionally discard information, however, some compilers warn of it. The cast states explicitly that the operation is intended and suppresses any warning.

1.8.5 External Variables

A C program consists of a set of external objects, which are either variables or functions. The adjective ``external'' is used in contrast to ``internal'', which describes the arguments and variables defined inside functions. External variables are defined outside of any function and are thus potentionally available to many functions. Functions themselves are always external, because C does not allow functions to be defined inside other functions. By default, external variables and functions have the property that all references to them by the same name, even from functions compiled separately, are references to the same thing. (The standard calls this property *external linkage*.) In this sense, external variables are analogous to Fortran COMMON blocks or variables in the outermost block in Pascal. We will see later how to define external variables and functions that are visible only within a single source file. Because external variables are globally accessible, they provide an alternative to function arguments and return values for communicating data between functions. Any function may access an external variable by referring to it by name, if the name has been declared somehow.

If a large number of variables must be shared among functions, external variables are more convenient and efficient than long argument lists. As pointed out in lesson 1, however, this reasoning should be applied with some caution, for it can have a bad effect on program structure and lead to programs with too many data connections between functions.

External variables are also useful because of their greater scope and lifetime. Automatic variables are internal to a function; they come into existence when the function is entered and disappear when it is left. External variables, on the other hand, are permanent, so they can retain values from one function invocation to the next. Thus if two functions must share some data, yet neither calls the other, it is often most convenient if the shared data is kept in external variables rather than being passed in and out via arguments.

1.8.6 Formal and Actual Arguments

The values passed to the function at the time of call are called actual parameters. The values of the actual parameters are received by the variable declared at the time of the function declaration. These receiving variable are called formal parameters.

Example:

```
#include <stdio.h>
unsigned long GetSquare(unsigned long x)
{
        return x * x;
}
void main()
{
        int a = 10;
        printf("Square of value %d is %ld", GetSquare(a));
}
```

In the above example, while calling the function GetSquare(a), the parameter passed is an actual parameter, while when this values is received by the function then it is stored in formal parameter x.

1.8.7 Scope Rules

The functions and external variables that make up a C program need not all be compiled at the same time; the source text of the program may be kept in several files, and previously compiled routines may be loaded from libraries. Among the questions of interest are

- How are declarations written so that variables are properly declared during compilation?
- How are declarations arranged so that all the pieces will be properly connected when the program is loaded?
- How are declarations organized so there is only one copy?
- How are external variables initialized?

Let us discuss these topics by reorganizing the calculator program into several files. As a practical matter, the calculator is too small to be worth splitting, but it is a fine illustration of the issues that arise in larger programs.

The *scope* of a name is the part of the program within which the name can be used. For an automatic variable declared at the beginning of a function, the scope is the function in which the name is declared. Local variables of the same name in different functions are unrelated. The same is true of the parameters of the function, which are in effect local variables.

The scope of an external variable or a function lasts from the point at which it is declared to the end of the file being compiled. For example, if main, sp, val, push, and pop are defined in one file, in the order shown above, that is,

```
main() { ... }
int sp = 0;
double val[MAXVAL];
void push(double f) { ... }
double pop(void) { ... }
```

then the variables sp and val may be used in push and pop simply by naming them; no further declarations are needed. But these names are not visible in main, nor are push and pop themselves.

On the other hand, if an external variable is to be referred to before it is defined, or if it is defined in a different source file from the one where it is being used, then an extern declaration is mandatory.

It is important to distinguish between the *declaration* of an external variable and its *definition*. A declaration announces the properties of a variable (primarily its type); a definition also causes storage to be set aside. If the lines

```
int sp;
double val[MAXVAL];
```

appear outside of any function, they *define* the external variables sp and val, cause storage to be set aside and also serve as the declarations for the rest of that source file. On the other hand, the lines

```
extern int sp;
extern double val[];
```

declare for the rest of the source file that sp is an int and that val is a double array (whose size is determined elsewhere), but they do not create the variables or reserve storage for them.

There must be only one *definition* of an external variable among all the files that make up the source program; other files may contain extern declarations to access it. (There may also be extern declarations in the file containing the definition.) Array sizes must be specified with the definition, but are optional with an extern declaration. Initialization of an external variable goes only with the definition.

Although it is not a likely organization for this program, the functions push and pop could be defined in one file and the variables val and sp defined and initialized in

another. Then these definitions and declarations would be necessary to tie them together:

in file1:

```
extern int sp;
extern double val[];
void push(double f) { ... }
double pop(void) { ... }
```

in file2:

```
int sp = 0;
double val[MAXVAL];
```

Because the extern declarations in *file1* lie ahead of and outside the function definitions, they apply to all functions; one set of declarations suffices for all of *file1*. This same organization would also bee needed if the definition of sp and val followed their use in one file.

1.8.8 Header Files

Let us now consider dividing the calculator program into several source files, as it might be if each of the components were substantially bigger. The main function would go in one file, which we will call main.c; push, pop, and their variables go into a second file, stack.c; getop goes into a third, getop.c. Finally, getch and ungetch go into a fourth file, getch.c; we separate them from the others because they would come from a separately-compiled library in a realistic program.

There is one more thing to worry about - the definitions and declarations shared among files. As much as possible, we want to centralize this, so that there is only one copy to get and keep right as the program evolves. Accordingly, we will place this common material in a *header file*, calc.h, which will be included as necessary. (The #include line is described in) The resulting program then looks like this:

calc.h

```
#define NUMBER '0'
void push(double);
double pop(void);
int getop(char []);
int getch(void);
void ungetch(int);
```

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "calc.h"
#define MAXOP 100
main() {
    ...
}
```

getop.c

```
#include <stdio.h>
#include <ctype.h>
#include "calc.h"
getop() {
    ...
}
```

stack.c

```
#include <stdio.h>
#include "calc.h"
#define MAXVAL 100
int sp = 0;
double val[MAXVAL];
void push(double) {
    ...
}
double pop(void) {
    ...
}
```

getch.c

```
#include <stdio.h>
#define BUFSIZE 100
char buf[BUFSIZE];
int bufp = 0;
int getch(void) {
    ...
}
void ungetch(int) {
    ...
}
```

There is a tradeoff between the desire that each file have access only to the information it needs for its job and the practical reality that it is harder to maintain more header files. Up to some moderate program size, it is probably best to have one header file that contains everything that is to be shared between any two parts of the program; that is the decision we made here. For a much larger program, more organization and more headers would be needed.

1.8.9 Recursion

C functions may be used recursively; that is a function may call itself either directly or indirectly. Consider printing a number as a character string. As we mentioned before, the digits are generated in the wrong order: low-order digits are available before high-order digits, but they have to be printed the other way around.

There are two solutions to this problem. On is to store the digits in an array as they are generated, then print them in the reverse order, as we did with itoa in lesson 7. The alternative is a recursive solution, in which printd first calls itself to cope with any leading digits, then prints the trailing digit. Again, this version can fail on the largest negative number.

```
#include <stdio.h>
/* printd:  print n in decimal */
void printd(int n)
{
        if (n < 0) {
                putchar('-');
                n = -n;
        }
        if (n / 10)
                printd(n / 10);
        putchar(n % 10 + '0');
}
```

When a function calls itself recursively, each invocation gets a fresh set of all the automatic variables, independent of the previous set. Thus in printd(123) the first printd receives the argument n = 123. It passes 12 to a second printd, which in turn passes 1 to a third. The third-level printd prints 1, then returns to the second level. That printd prints 2, then returns to the first level. That one prints 3 and terminates.

Recursion may provide no saving in storage, since somewhere a stack of the values being processed must be maintained. Nor will it be faster. But recursive code is more compact, and often much easier to write and understand than the non-recursive equivalent. Recursion is the natural solution for some of the problems like factorial computation, Fibonacci series generation, quick sort, binary search etc.

While using recursion, care should be taken that recursive function always return some value or recursion has some terminating point.

1.8.10 Summary

Functions decompose a large program into manageable smaller units, which can be tested independently. Functions also facilitate breaking the program into logical units. While calling the functions, any parameter passed is the actual parameter and the variable receiving the value is called formal parameter. Recursive functions are those which call themselves directly or indirectly. Recursive functions should have a terminating point.

1.8.11 Short Answer Type Questions

1. What is the syntax of declaring a function?
2. What is the difference between function declaration and function definition?
3. What is the meaning of function prototyping?
4. Define recursion.

1.8.12 Long Answer Type Questions

1. What is the advantage of decomposing program into functions?
2. What type of values can be returned by functions?
3. What is the difference between formal and actual parameters?
4. WAP to calculate simple interest using function.
5. WAP to calculate factorial of a number using recursion.

1.8.13 Suggested Books

1.	Application Programming in C	R. S. Salaria
2.	C Programming using Turbo C	Robert Lafore
3.	Programming with ANSI and Turbo C	Ashok N. Kamthane
4.	Programming using C	E. Balagurusamy
5.	Let us C	Yashwant Kanettkar

Web Resources

www.cprogramming.com/tutorial/c/lesson1.html
www.programiz.com/c-programming
www.w3schools.in/c-tutorial/
www.learn-c.org
www.tutorialspoint.com/cprogramming/

Lesson No. 1.9 **Author : Dharam Veer Sharma**

Storage Classes

1.9.1 Introduction
1.9.2 Objectives of the Lesson
1.9.3 Scope
1.9.4 Linkage
1.9.5 Storage Duration (Life time)
1.9.6 automatic Storage Class
1.9.7 register Storage Class
1.9.8 static Storage Class
1.9.9 extern Storage Class
1.9.10 Summary
1.9.11 Short Answer Type Questions
1.9.12 Long Answer Type Questions
1.9.13 Suggested Books

1.9.1 Introduction

In C language, each variable has a storage class which decides scope, visibility and life time of that veriable. C provides five different models or storage classes for variables. There's also a sixth model, based on pointers, that we'll get to later in this lesson (in the section "Allocated Memory: malloc() and free()"). A storage class defines the scope and objective of variables or functions within a C program. You can describe a variable (or, more generally, a data object) in terms of its storage duration, which is how long it stays in memory and its scope and its linkage, which together indicate which parts of a program can use it by name. The different storage classes offer different combinations of scope, linkage and storage duration. You can have variables that can be shared over several files of source code, variables that can be used by any function in one particular file, variables that can be used only within a particular function, and even variables that can be used only within a subsection of a function. You can have variables that exist for the duration of a program and variables that exist only while the function containing them is executing. You also can store data in memory that is allocated and freed explicitly by means of function calls.

Before examining the five storage classes, we need to investigate the meaning of the terms scope, linkage and storage duration. After that, we'll return to the specific storage classes.

1.9.2 Objectives of the Lesson

In this lesson we shall discuss the various storage classes available in C language, purpose and behaviour of the storage classes. We shall also discuss the way these facilitate security, reliability and speed to the programs.

1.9.3 Scope

Scope describes the region or regions of a program that can access an identifier. A C variable has one of the following scopes: **block scope, function prototype scope or file scope**. The program examples to date have used block scope. A block, as you'll recall, is a region of code contained within an opening brace and the matching closing brace. For instance, the entire body of a function is a block. Any compound statement within a function also is a block. A variable defined inside a block has block scope and it is visible from the point it is defined until the end of the block containing the definition. Also, formal function parameters, even though they occur before the opening brace of a function, have block scope and belong to the block containing the function body. So the local variables we've used to date, including formal function parameters, have block scope. Therefore, the variables cleo and patrick in the following code both have block scope extending to the closing brace:

```
double blocky(double cleo)
{
        double patrick = 0.0;
        ...
        return patrick;
}
```

Variables declared in an inner block have scope restricted just to that block:

```
double blocky(double cleo)
{
        double patrick = 0.0;
        int i;
        for (i = 0; i < 10; i++)
        {
                double q = cleo * i; // start of scope for q
                ...
                patrick *= q;
        }                       // end of scope for q
        ...
        return patrick;
}
```

In this example, the scope of q is limited to the inner block and only code within that block can access q.

Function prototype scope applies to variable names used in function prototypes, as in the following:

```
int mighty(int mouse, double large);
```

Function prototype scope runs from the point the variable is defined to the end of the prototype declaration. What this means is that all the compiler cares about when handling a function prototype argument is the types; the names you use, if any, normally don't matter, and they needn't match the names you use in the function definition. One case in which the names matter a little is with variable-length array parameters:

```
void use_a_VLA(int n, int m, ar[n][m]);
```

If you use names in the brackets, they have to be names declared earlier in the prototype.

A variable with its definition placed outside of any function has file scope. A variable with file scope is visible from the point it is defined to the end of the file containing the definition. Take a look at this example:

```
#include <stdio.h>
int units = 0;          /* a variable with file scope */
void critic(void);
int main(void)
{
        ...
}

void critic(void)
{
        ...
}
```

Here, the variable units has file scope and it can be used in both main() and critic(). Because they can be used in more than one function, file scope variables are also called global variables.

There is one other type of scope, called function scope, but it applies only to labels used by goto statements. Function scope means a goto label in a particular function is visible to code anywhere in that function, regardless of the block in which it appears.

1.9.4 Linkage

Next, let's look at linkage. A C variable has one of the following linkages: **external linkage, internal linkage, or no linkage**. Variables with block scope or prototype scope have no linkage. That means they are private to the block or prototype in which they are defined. A variable with file scope can have either internal or external linkage. A variable with external linkage can be used anywhere in a multifile program. A variable with internal linkage can be used anywhere in a single file.

So how can you tell whether a file scope variable has internal or external linkage? You look to see if the storage class specifier static is used in the external definition:

```
int giants = 5;          // file scope, external linkage
static int dodgers = 3;  // file scope, internal linkage
int main()
{
        ...
}
        ...
```

The variable giants can be used by other files that are part of the same program. The dodgers variable is private to this particular file, but can be used by any function in the file.

1.9.5 Storage Duration (Life time)

A C variable has one of the following two storage durations: **static storage duration or automatic storage duration**. If a variable has static storage duration, it exists throughout program execution. Variables with file scope have static storage duration. Note that for file scope variables, the keyword static indicates the linkage type, not the storage duration. A file scope variable declared using static has internal linkage, but all file scope variables, using internal linkage or external linkage, have static storage duration.

Variables with block scope normally have automatic storage duration. These variables have memory allocated for them when the program enters the block in which they are defined and the memory is freed when the block is exited. The idea is that memory used for automatic variables is a workspace or scratch pad that can be reused. For example, after a function call terminates, the memory it used for its variables can be used to hold variables for the next function that is called.

The local variables we've used so far fall into the automatic category. For example, in the following code, the variables number and index come into being each time the bore() function is called and pass away each time the function completes:

```
void bore(int number)
{
        int index;
        for (index = 0; index < number; index++)
                puts("They don't make them the way they used to.\n");
        return 0;
}
```

C uses scope, linkage and storage duration to define five storage classes: **automatic, register, static with block scope, static with external linkage, and static with internal linkage**. The following lists the combinations. Now that we've covered scope, linkage, and storage duration, we can discuss these storage classes in more detail.

The Five Storage Classes

Storage Class	Duration	Scope	Linkage	How Declared
automatic	Automatic	Block	None	In a block
register	Automatic	Block	None	In a block with the keyword register
static with external linkage (global)	Static	File	External	Outside of all functions
static with internal linkage	Static	File	Internal	Outside of all functions with the keyword static
static with no linkage	Static	Block	None	In a block with the keyword static

1.9.6 automatic Storage Class

A variable belonging to the automatic storage class has automatic storage duration, block scope and no linkage. By default, any variable declared in a block or function header belongs to the automatic storage class. You can, however, make your intentions perfectly clear by explicitly using the keyword auto, as shown here:

```
int main(void)
{
        auto int plox;
```

You might do this, for example, to document that you are intentionally overriding an external function definition or that it is important not to change the variable to another storage class. The keyword auto is termed a storage class specifier.

Block scope and no linkage implies that only the block in which the variable is defined can access that variable by name. (Of course, arguments can be used to communicate the variable's value and address to another function, but that is indirect knowledge.) Another function can use a variable with the same name, but it will be an independent variable stored in a different memory location.

Recall that automatic storage duration means that the variable comes into existence when the program enters the block that contains the variable declaration. When the program exits the block, the automatic variable disappears. Its memory location can now be used for something else.

Let's look more closely at nested blocks. A variable is known only to the block in which it is declared and to any block inside that block:

```
int loop(int n)
{
```

```
        int m;          // m in scope
        scanf("%d", &m);
        {
                int i;    // both m and i in scope
                for (i = m; i < n; i++)
                        puts("i is local to a sub-block\n");
        }
        return m;     // m in scope, i gone
}
```

In this code, i is visible only within the inner braces. You'd get a compiler error if you tried to use it before or after the inner block. Normally, you wouldn't use this feature when designing a program. Sometimes, however, it is useful to define a variable in a sub-block if it is not used elsewhere. In that way, you can document the meaning of a variable close to where it is used. Also, the variable doesn't sit unused, occupying memory when it is no longer needed. The variables n and m, being defined in the function head and in the outer block, are in scope for the whole function and exist until the function terminates.

What if you declare a variable in an inner block that has the same name as one in the outer block? Then the name defined inside the block is the variable used inside the block. We say it hides the outer definition. However, when execution exits the inner block, the outer variable comes back into scope. The following listing illustrates these points and more.

The hiding.c Program

```
/* hiding.c -- variables in blocks */
#include <stdio.h>
int main()
{
        int x = 30;      /* original x          */
        printf("x in outer block: %d\n", x);
        {
                int x = 77;  /* new x, hides first x */
                printf("x in inner block: %d\n", x);
        }
        printf("x in outer block: %d\n", x);
        while (x++ < 33) /* original x          */
        {
                int x = 100; /* new x, hides first x */
                x++;
                printf("x in while loop: %d\n", x);
```

```
        }
        printf("x in outer block: %d\n", x);
        return 0;
}
```

Here's the output:

```
x in outer block: 30
x in inner block: 77
x in outer block: 30
x in while loop: 101
x in while loop: 101
x in while loop: 101
x in outer block: 34
```

First, the program creates a variable x with the value 30, as the first printf() statement shows. Then it defines a new x variable with the value 77, as the second printf() statement shows. That it is a new variable hiding the first x is shown by the third printf() statement. It is located after the first inner block and it displays the original x value, showing that the original x variable never went away and never got changed.

Perhaps the most intriguing part of the program is the while loop. The while loop test uses the original x:

```
while(x++ < 33)
```

Inside the loop, however, the program sees a third x variable, one defined just inside the while loop block. So when the code uses x++ in the body of the loop, it is the new x that is incremented to 101 and then displayed. When each loop cycle is completed, that new x disappears. Then the loop test condition uses and increments the original x, the loop block is entered again and the new x is created again. In this example, that x is created and destroyed three times. Note that, to terminate, this loop had to increment x in the test condition because incrementing x in the body increments a different x than the one used for the test.

The intent of this example is not to encourage you to write code like this. Rather, it is to illustrate what happens when you define variables inside a block.

Initialization of Automatic Variables

Automatic variables are not initialized unless you do so explicitly. Consider the following declarations:

```
int main(void)
{
        int repid;
        int tents = 5;
```

The tents variable is initialized to 5, but the repid variable ends up with whatever value happened to previously occupy the space assigned to repid. You cannot rely on this value being 0.

You can initialize an automatic variable with a nonconstant expression, provided any variables used have been defined previously:

```
int main(void)
{
        int ruth = 1;
        int rance = 5 * ruth;   // use previously defined variable
```

1.9.7 Register Storage Class

Variables are normally stored in computer memory. With luck, register variables are stored in the CPU registers or more generally, in the fastest memory available, where they can be accessed and manipulated more rapidly than regular variables. Because a register variable may be in a register rather than in memory, you can't take the address of a register variable. In most other respects, register variables are the same as automatic variables. That is, they have block scope, no linkage and automatic storage duration. A variable is declared by using the storage class specifier register:

```
int main(void)
{
        register int quick;
```

We say "with luck" because declaring a variable as a register class is more a request than a direct order. The compiler has to weigh your demands against the number of registers or amount of fast memory available, so you might not get your wish. In that case, the variable becomes an ordinary automatic variable; however, you still can't use the address operator with it.

You can request that formal parameters be register variables. Just use the keyword in the function heading:

```
        void macho(register int n)
```

The types that can be declared register may be restricted. For example, the registers in a processor might not be large enough to hold type double.

1.9.8 static Storage Class

The name static variable sounds like a contradiction, like a variable that can't vary. Actually, static means that the variable stays put. Variables with file scope automatically (and necessarily) have static storage duration. It's also possible to create local variables—that is, variables having block scope that have static storage. These variables have the same scope as automatic variables, but they don't vanish when the containing function ends its job. That is, such variables have block scope, no linkage, but static storage duration. The computer remembers their values from one function call to the next. Such variables are created by declaring them in a block (which provides the block scope and

lack of linkage) with the storage class specifier static (which provides the static storage duration). The example in the following listing illustrates this technique.

The loc_stat.c Program

```
/* loc_stat.c -- using a local static variable */
#include <stdio.h>
void trystat(void);
int main(void)
{
        int count;
        for (count = 1; count <= 3; count++)
        {
                printf("Here comes iteration %d:\n", count);
                trystat();
        }
        return 0;
}
        void trystat(void)
{
        int fade = 1;
        static int stay = 1;
        printf("fade = %d and stay = %d\n", fade++, stay++);
}
```

Note that trystat() increments each variable after printing its value. Running the program returns this output:

```
Here comes iteration 1:
fade = 1 and stay = 1
Here comes iteration 2:
fade = 1 and stay = 2
Here comes iteration 3:
fade = 1 and stay = 3
```

The static variable stay remembers that its value was increased by 1, but the fade variable starts anew each time. This points out a difference in initialization: fade is initialized each time trystat() is called, but stay is initialized just once, when trystat() is compiled. Static variables are initialized to zero if you don't explicitly initialize them to some other value.

The two declarations look similar:

```
int fade = 1;
static int stay = 1;
```

However, the first statement is really part of the trystat() function and is executed each time the function is called. It is a runtime action. The second statement isn't actually part of the trystat() function. If you use a debugger to execute the program step-by-step, you'll see that the program seems to skip that step. That's because static variables and external variables are already in place after a program is loaded into memory. Placing the statement in the trystat() function tells the compiler that only the trystat() function is allowed to see the variable; it's not a statement that's executed during runtime.

You can't use static for function parameters:

```
int wontwork(static int flu);   // not allowed
```

If you read some of the older C literature, you'll find this storage class referred to as the internal static storage class. However, the word internal was used to indicate internal to a function, not internal linkage.

1.9.9 extern Storage Class

A static variable with external linkage has file scope, external linkage, and static storage duration. This class is sometimes termed the external storage class, and variables of this type are called external variables. You create an external variable by placing a defining declaration outside of any function. As a matter of documentation, an external variable can additionally be declared inside a function that uses it by using the extern keyword. If the variable is defined in another file, declaring the variable with extern is mandatory. Declarations look like this:

```
int Errupt;                     /* externally defined variable    */
double Up[100];       /* externally defined array       */
extern char Coal;     /* mandatory declaration          */
                                /* Coal defined in another file   */
void next(void);
int main(void)
{
        extern int Errupt;  /* optional declaration          */
        extern double Up[]; /* optional declaration          */
        ...
}
void next(void)
{
        ...
}
```

The two declarations of Errupt are examples of linkage because they both refer to the same variable. External variables have external linkage, a point we'll return to later.

Note that you don't have to give the array size in the optional declaration of double Up. That's because the original declaration already supplied that information. The group of extern declarations inside main() can be omitted entirely because external variables have file scope, so they are known from the point of declaration to the end of the file. They do serve, however, to document your intention that main() use these variables.

If only extern is omitted from the declaration inside a function, a separate automatic variable is set up. That is, replacing

```
extern int Errupt;
```

with

```
int Errupt;
```

in main() causes the compiler to create an automatic variable named Errupt. It would be a separate, local variable, distinct from the original Errupt. The local variable would be in scope while the program executes main(), but the external Errupt would be in scope for other functions, such as next(), in the same file. In short, a variable in block scope "hides" a variable of the same name in file scope while the program executes statements in the block.

External variables have static storage duration. Therefore, the array Up maintains its existence and values regardless of whether the program is executing main(), next() or some other function.

The following three examples show four possible combinations of external and automatic variables. Example 1 contains one external variable: Hocus. It is known to both main() and magic().

```
/* Example 1 */
int Hocus;
int magic();
int main(void)
{
        extern int Hocus;  // Hocus declared external
        ...
}
int magic()
{
        extern int Hocus;  // same Hocus as above
        ...
}
```

Example 2 has one external variable, Hocus, known to both functions. This time, magic() knows it by default.

```
/* Example 2 */
int Hocus;
```

```
int magic();
int main(void)
{
        extern int Hocus;  // Hocus declared external
        ...
}
int magic()
{
        // Hocus not declared but is known
        ...
}
```

In Example 3, four separate variables are created. The Hocus variable in main() is automatic by default and is local to main. The Hocus variable in magic() is automatic explicitly and is known only to magic(). The external Hocus variable is not known to main() or magic() but would be known to any other function in the file that did not have its own local Hocus. Finally, Pocus is an external variable known to magic() but not to main() because Pocus follows main().

```
/* Example 3 */
int Hocus;
int magic();
int main(void)
{
        int Hocus;       // Hocus declared, is auto by default
        ...
}
int Pocus;
int magic()
{
        auto int Hocus;  // local Hocus declared automatic
        ...
}
```

These examples illustrate the scope of external variables: from the point of declaration to the end of the file. They also illustrate the lifetimes of variables. The external Hocus and Pocus variables persist as long as the program runs and because they aren't confined to any one function, they don't fade away when a particular function returns.

Initializing External Variables

Like automatic variables, external variables can be initialized explicitly. Unlike automatic variables, external variables are automatically initialized to zero if you

don't initialize them. This rule applies to elements of an externally defined array, too. Unlike the case for automatic variables, you can use only constant expressions to initialize file scope variables:

```
int x = 10;             // ok, 10 is constant
int y = 3 + 20;         // ok, a constant expression
size_t z = sizeof(int); // ok, a constant expression
int x2 = 2 * x;         // not ok, x is a variable
```

(As long as the type is not a variable array, a sizeof expression is considered a constant expression.)

Using an External Variable

Let's look at a simple example that involves an external variable. Specifically, suppose you want two functions, call them main() and critic(), to have access to the variable units. You can do this by declaring units outside of and above the two functions, as shown in the following listing.

The global.c Program

```
/* global.c  -- uses an external variable */
#include <stdio.h>
int units = 0;          /* an external variable      */
void critic(void);
int main(void)
{
        extern int units;  /* an optional redeclaration */
        printf("How many pounds to a firkin of butter?\n");
        scanf("%d", &units);
        while ( units != 56)
        critic();
        printf("You must have looked it up!\n");
        return 0;
}
void critic(void)
{
        /* optional redeclaration omitted */
        printf("No luck, chummy. Try again.\n");
        scanf("%d", &units);
}
```

Here is some sample output:

```
How many pounds to a firkin of butter?
14
No luck, chummy. Try again.
```

```
56
```

You must have looked it up!

Note how the second value for units was read by the critic() function, yet main() also knew the new value when it finished the while loop. So both the main() function and the critic() function use the identifier units to access the same variable. In C terminology, we say that units has file scope, external linkage and static storage duration.

We made units an external variable by defining it outside of (that is, external to) any function definition. That's all you need to do to make units available to all the subsequent functions in the file.

Let's look at some of the details. First, declaring units where it is declared makes it available to the functions below it without any further action taken. Therefore, critics() uses the units variable.

Similarly, nothing needed to be done to give main() access to units. However, main() does have the following declaration in it:

```
extern int units;
```

In the example, this declaration is mainly a matter of documentation. The storage class specifier extern tells the compiler that any mention of units in this particular function refers to a variable defined outside the function, perhaps even outside the file. Again, both main() and critic() use the externally defined units.

Definitions and Declarations

Let's take a longer look at the difference between defining a variable and declaring it. Consider the following example:

```
int tern = 1;          /* tern defined              */
main()
{
        external int tern;  /* use a tern defined elsewhere */
```

Here, tern is declared twice. The first declaration causes storage to be set aside for the variable. It constitutes a definition of the variable. The second declaration merely tells the compiler to use the tern variable that has been created previously, so it is not a definition. The first declaration is called a defining declaration and the second is called a referencing declaration. The keyword extern indicates that a declaration is not a definition because it instructs the compiler to look elsewhere.

Suppose you do this:

```
extern int tern;
int main(void)
{
```

The compiler will assume that the actual definition of tern is somewhere else in your program, perhaps in another file. This declaration does not cause space to be

allocated. Therefore, don't use the keyword extern to create an external definition; use it only to refer to an existing external definition.

An external variable can be initialized only once and that must occur when the variable is defined. A statement such as

```
extern char permis = 'Y';   /* error */
```

is in error because the presence of the keyword extern signifies a referencing declaration, not a defining declaration.

Static Variables with Internal Linkage

Variables of this storage class have static storage duration, file scope, and internal linkage. You create one by defining it outside of any function (just as with an external variable) with the storage class specifier static:

```
static int svil = 1;  // static variable, internal linkage
int main(void)
{
```

Such variables were once termed external static variables, but that's a bit confusing because they have internal linkage. Unfortunately, no new compact term has taken the place of external static, so we're left with static variable with internal linkage. The ordinary external variable can be used by functions in any file that's part of the program, but the static variable with internal linkage can be used only by functions in the same file. You can redeclare any file scope variable within a function by using the storage class specifier extern. Such a declaration doesn't change the linkage. Consider the following code:

```
int traveler = 1;       // external linkage
static int stayhome = 1; // internal linkage
int main()
{
        extern int traveler;  // use global traveler
        extern int stayhome;  // use global stayhome
        ...
```

Both traveler and stayhome are global for this particular file, but only traveler can be used by code in other files. The two declarations using extern document that main() is using the two global variables, but stayhome continues to have internal linkage.

Multiple Files

The difference between internal linkage and external linkage is important only when you have a program built from multiple files, so let's take a quick look at that topic.

Complex C programs often use several separate files of code. Sometimes these files might need to share an external variable. The ANSI C way to do this is to have a defining declaration in one file and referencing declarations in the other files. That is,

all but one declaration (the defining declaration) should use the extern keyword and only the defining declaration can be used to initialize the variable.

Note that an external variable defined in one file is not available to a second file unless it is also declared (by using extern) in the second file. An external declaration by itself only makes a variable potentially available to other files.

Historically, however, many compilers have followed different rules in this regard. Many Unix systems, for example, enable you to declare a variable in several files without using the extern keyword, provided that no more than one declaration includes an initialization. If there is a declaration with an initialization, it is taken to be the definition.

Storage-Class Specifiers

You may have noticed that the meaning of the keywords static and extern depends on the context. The C language has five keywords that are grouped together as storage-class specifiers. They are **auto, register, static, extern and typedef**. The typedef keyword doesn't say anything about memory storage, but it is thrown in for syntax reasons. In particular, you can use no more than one storage-class specifier in a declaration, so that means you can't use one of the other storage-class specifiers as part of a typedef.

The auto specifier indicates a variable with automatic storage duration. It can be used only in declaration of variables with block scope, which already have automatic storage duration, so its main use is documenting intent.

The register specifier also can be used only with variables of block scope. It puts a variable into the register storage class, which amounts to a request that the variable be stored in a register for faster access. It also prevents you from taking the address of the variable.

The static specifier, when used in a declaration for a variable with block scope, gives that variable static storage duration so it exists and retains its value as long as the program is running, even at times the containing block isn't in use. The variable retains block scope and no linkage. When used in a declaration with file scope, it indicates that the variable has internal linkage.

The extern specifier indicates that you are declaring a variable that has been defined elsewhere. If the declaration containing extern has file scope, the variable referred to must have external linkage. If the declaration containing extern has block scope, the referred-to variable can have either external linkage or internal linkage, depending on the defining declaration for that variable.

1.9.10 Summary

Automatic variables have block scope, no linking, and automatic storage duration. They are local and private to the block (typically a function) where they are defined. Register variables have the same properties as automatic variables, but the compiler may use faster memory or a register to store them. You can't take the address of a register variable.

Variables with static storage duration can have external linkage, internal linkage, or no linkage. When a variable is declared external to any function in a file, it's an external variable and has file scope external linkage and static storage duration. If you add the keyword static to such a declaration, you get a variable with static storage duration, file scope, and internal linkage. If you declare a variable inside a function and use the keyword static, the variable has static storage duration, block scope, and no linkage.

Memory for a variable with automatic storage duration is allocated when program execution enters the block containing the variable declaration and is freed when the block is exited. If uninitialized, such a variable has a garbage value. Memory for a variable with static storage duration is allocated at compile time and lasts as long as the program runs. If uninitialized, such a variable is set to 0.

A variable with block scope is local to the block containing the declaration. A variable with file scope is known to all functions in a file following its declaration. If a file scope variable has external linkage, it can be used by other files in the program. If a file scope variable has internal linkage, it can be used just within the file in which it is declared.

1.9.11 Short Answer Type Questions

1. How many storage classes are available in C?
2. Name the storage classes available in C?
3. What is the purpose of register storage class?
4. Which storage class variables are automatically initialize to zero?

1.9.12 Long Answer Type Questions

1. What is the purpose of static storage class?
2. What is the different between scope and lifetime of a variable?
3. What is the scope and lifetime of the various storage classes available in C?

1.9.13 Suggested Books

1.	Let Us C	Yashwant Kanetkar
2.	C Programming using Turbo C	Robert Lafore
3.	Programming with ANSI and Turbo C	Ashok N. Kamthane
4.	Programming using C	E. Balagurusamy
5.	The C Programming language	Brian W. Kernigham Dennis M. Ritchie

Web Resources

www.cprogramming.com/tutorial/c/lesson1.html
www.programiz.com/c-programming
www.w3schools.in/c-tutorial/
www.learn-c.org
www.tutorialspoint.com/cprogramming/