PUNJABI UNIVERSITY PATIALA

M.Sc (IT) SEM-I **Paper: MITM1102T**

Computer Programming using C

UNIT No.1

Center for Distance and Online Education, PunjabiUniversity,Patiala

Lesson No:

1. : **Introduction to C Language and Program Development**
2. : **Identifiers, Keywords, Data Types and Type Conversion**
3. : **Performing Input Output Operations**
4. : **Operators and Expressions**
5. : **Library Functions**
6. : **Sequential and Conditional Control Statements**
7. : **Iterative Control Statements**
8. : **Arrays**

(Syllabus)

MITM1102T: Computer Programming using C

Maximum Marks: 70 **Maximum Time: 3 Hrs.**

Minimum Pass Marks: 35%

Course Objective: This course is designed to explore computing and to show students the art of computer programming. Students will be able to learn Understand programming using C concepts for writing good programs. On completion of this course, the students will be able to

- Write, compile and debug programs in C language.
- Use different data types, operators and console I/O function in a computer program.
- Design programs involving decision control statements, loop control statements and case control structures.
- Understand the implementation of arrays, pointers and functions and apply the dynamics of memory by the use of pointers.

Course Content

SECTION A

Problem Solving with Computers, c character set, identifier, constants, variables, rules for defining variables, Data types, operators: arithmetic, relational, logical, comma, conditional, assignment, arithmetic expressions, input and output statements, assignment statements.

Decision statement: if, if else, nested if, switch statement, break statement, continue statement, go to statement.

Loops and control statements: While loop, for loop and do-while loop, nested loops

Arrays: one dimensional Array, multi-dimensional arrays, array initialization.

SECTION B

Pointers: Pointer data type, pointers and arrays, pointers and functions.

Functions: definition, declaration, function prototype, types of functions, call by value, call by reference, recursion, processing character strings.

Structures: Using structures, arrays of structures and arrays in structures, union

Files in C: Sequential files, random access files, Unformatted files, Text files, binary files.

Pedagogy:

The Instructor is expected to use leading pedagogical approaches in the class room situation, research-based methodology, innovative instructional methods, extensive use of technology in the class room, online modules of MOOCS, and comprehensive assessment practices to strengthen teaching efforts and improve student learning outcomes.

The Instructor of class will engage in a combination of academic reading, analyzing case studies, preparing the weekly assigned readings and exercises, encouraging in class discussions, and live project based learning.

Text and Readings: Students should focus on material presented in lectures. The text should be used to provide further explanation and examples of concepts and techniques discussed in the course:

- E. Balagurusamy, "Programming in C", Tata McGraw Hill.
- Kamthane, "Programming with ANSI and Turbo C", Pearson Education
- Rajaraman, V, "Fundamentals of Computers", PHI
- Kanetkar, "Let Us C", BPB Publications.

Scheme of Examination

- English will be the medium of instruction and examination.
- Written Examinations will be conducted at the end of each Semester as per the Academic Calendar notified in advance
- Each course will carry 100 marks of which 30 marks shall be reserved for internal assessment and the remaining 70 marks for written examination to be held at the end of each semester.
- The duration of written examination for each paper shall be three hours.
- The minimum marks for passing the examination for each semester shall be 35% in aggregate as well as a minimum of 35% marks in the semester-end examination in each paper.
- A minimum of 75% of classroom attendance is required in each subject.

Instructions to the External Paper Setter

The question paper will consist of three Sections: A, B and C. Sections A and B will have four questions each from the respective section of the syllabus and will carry 10.5 marks for each question. Section C will consist of 7-15 short answer type questions covering the entire syllabus uniformly and will carry a total of 28 marks.

Instructions for candidates

- Candidates are required to attempt five questions in all, selecting two questions each from section A and B and compulsory question of section C.
- Use of non-programmable scientific calculator is allowed.

Lesson No. 1 **Author: Dr. DharamVeer Sharma**
Converted into SLM by: Dr. Vishal Singh

Introduction to C Language and Program Development

1. Objectives

In this lesson we shall learn about the origination 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 in 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.

2. 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 being, 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 languages 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 computers' understanding and functioning.

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, and 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.

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 two 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.

5. Feature 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 anew 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 ben an extremely effective and expressive language for a wide variety of programming applications.

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.

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 it self. 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.

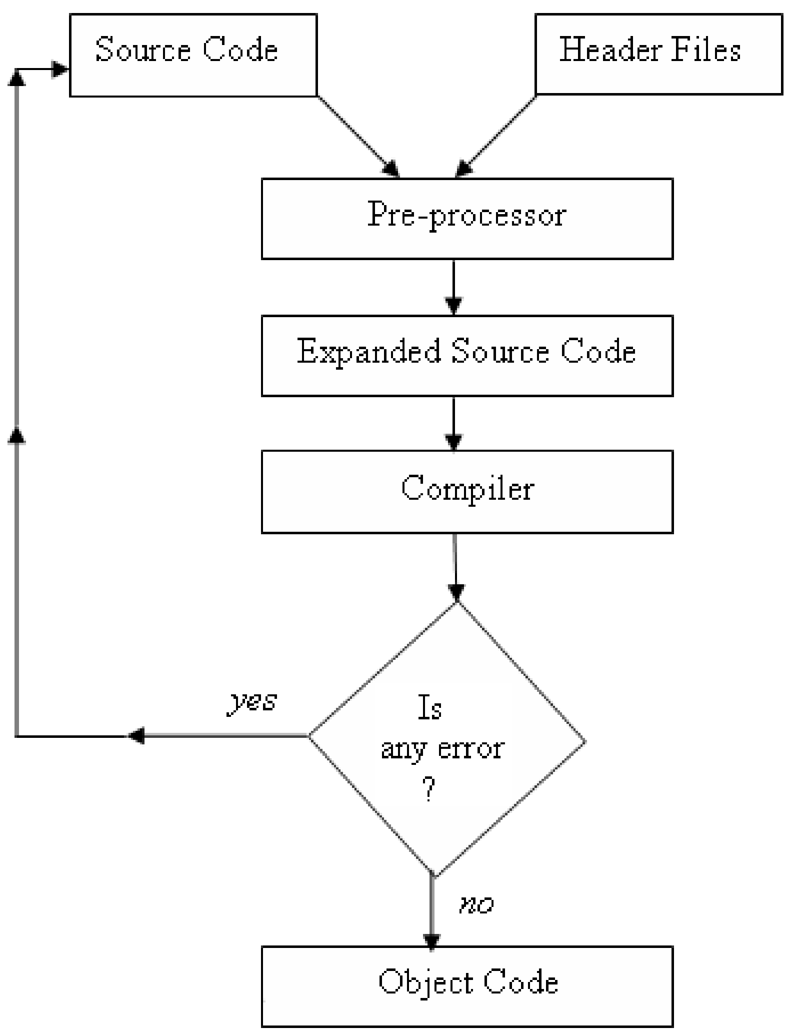


Figure 1: Stages in Compilation of a C program

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 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:

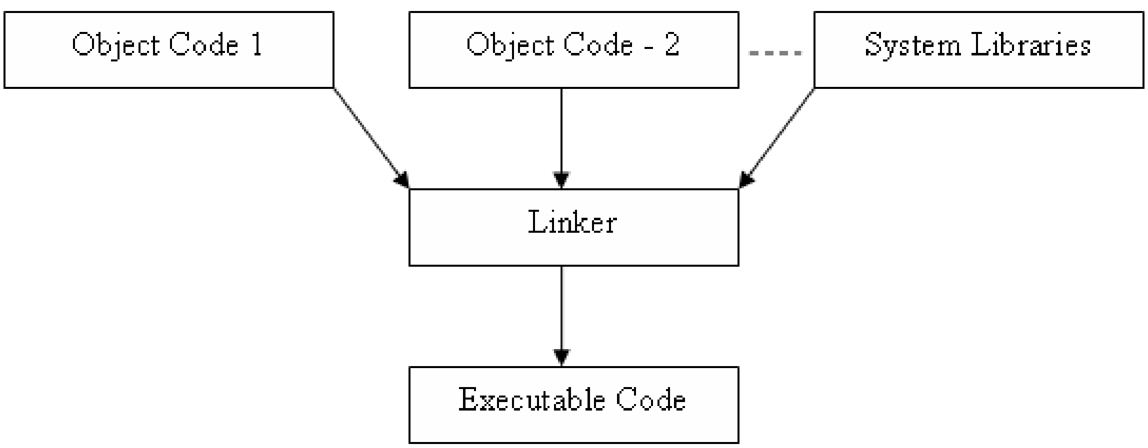


Figure 2: Linking of object code(s) with system libraries

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.

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 check for syntax correctness and then converted to object code, thereafter it is linked with other libraries and finally an executable file is produced.

9. 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?

10. 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?

11. Suggested Books

1.	Application Programming in C	R. S. Salaria
2.	C Programming using Turbo C	Robert Lafore

Lesson No. 2 **Author: Dr. DharamVeer Sharma**
Converted into SLM by: Dr. Vishal Singh

Identifiers, Keywords, Data Types and Type Conversion

1. 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.

2. 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.

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.

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.

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

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

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 chapter.

If arguments are declared by a function prototype, as the 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.

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 declared 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 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 contains 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'.

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.

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 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.

10. Short Answer Type Questions

1. What are valid characters in the C character set?
2. What do you mean by an identifier?
3. What do you mean by a reserve word?
4. What is the need of declaring the type of a variable?
5. What do you mean by type conversion?
6. What is the difference between variable and constant?

11. Long Answer Type Questions

1. Discuss the various identifier naming rules.
2. Write any 24 reserve words of C language.

3. What are the basic data types available in C language? What is the size of each data type?
4. What are the various types of constant declarations? Explain giving examples.

12. Suggested Books

Programming with ANSI and Turbo C	Ashok N. Kamthane
C Programming	E. Balagurusamy
Application Programming in C	R. S. Salaria

Lesson No. 3 **Author: Dr. DharamVeer Sharma**
Converted into SLM by: Dr. Vishal Singh

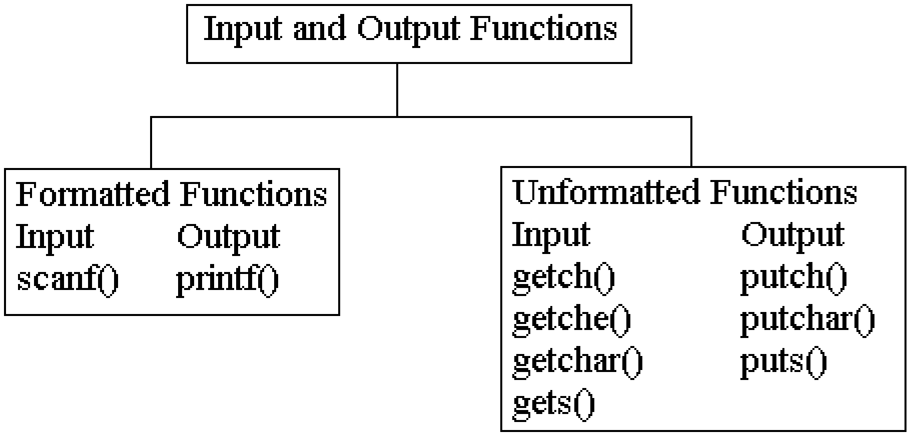
Performing Input Output Operations

1. Objectives

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

2. 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.



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 in, 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, bus 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():** This function reads character type data from the standard input. It reads one character at a time till the user presses the enter key.
b. **getch() and getche():** These functions read any character from the standard input device. The character entered is not displayed or echoed by getch() function. These functions are included in conio.h header file.
c. **gets():** This function is used for accepting any string through stdin (keyboard) until enter key is pressed. The header file stdio.h is needed for implementing this function.

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 in 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 places 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, is 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.

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 is 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 which is read by the standard input.

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.

6. Formatted Output - printf

The output function printf translates internal values to characters. We have used printf informally in previous chapters. 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 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 it 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.

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.

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.

9. 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?

10. 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.

11. Suggested Books

Programming with ANSI and Turbo C	Ashok N. Kamthane
C Programming	E. Balagurusamy
Application Programming in C	R. S. Salaria

Lesson No. 4 **Author: Dr. DharamVeer Sharma**
Converted into SLM by: Dr. Vishal Singh

Operators and Expressions

1. 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.

2. 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	? :

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.

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.

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.

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

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

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

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.

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.

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");
```

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.

9. Operator Precedence and Associativity

Table 2.1 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

<table>
<tr><td><<
>></td><td>Left shift
Right shift</td><td>left to right</td><td>5</td></tr>
<tr><td><
<=
>
>=</td><td>Less than
Less than or equal to
Greater than
Greater than or equal to</td><td>left to right</td><td>6</td></tr>
<tr><td>==
!=</td><td>Equality
Inequality</td><td>left to right</td><td>7</td></tr>
<tr><td>&</td><td>Bitwise AND</td><td>left to right</td><td>8</td></tr>
<tr><td>^</td><td>Bitwise XOR</td><td>left to right</td><td>9</td></tr>
<tr><td>|</td><td>Bitwise OR</td><td>left to right</td><td>10</td></tr>
<tr><td>&&</td><td>Logical AND</td><td>left to right</td><td>11</td></tr>
<tr><td>||</td><td>Logical OR</td><td>left to right</td><td>12</td></tr>
<tr><td>?:</td><td>Conditional operator</td><td>right to left</td><td>13</td></tr>
<tr><td>= += -= *= /= %=
&= ^= |= <<= >>=</td><td>Assignment operators</td><td>right to left</td><td>14</td></tr>
<tr><td>,</td><td>Comma operator</td><td>left to right</td><td>15</td></tr>
</table>

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.

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.

11. 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?

12. 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?

13. Suggested Books

Programming with ANSI and Turbo C	Ashok N. Kamthane
Programming using C	E. Balagurusamy
Application Programming in C	R. S. Salaria

Lesson No. 5 **Author: Dr. DharamVeer Sharma**
Converted into SLM by: Dr. Vishal Singh

Library Functions

1. Objectives

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

2. Introduction

Library 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.

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     %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.

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.

5. String Functions

All the strings 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 functions find 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.

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 clear 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

7. Summary

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

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?

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?

10. Suggested Books

Programming with ANSI and Turbo C	Ashok N. Kamthane
Programming using C	E. Balagurusamy
Application Programming in C	R. S. Salaria

Lesson No. 6 **Author: Dr. DharamVeer Sharma**
Converted into SLM by: Dr. Vishal Singh

Sequential and Conditional Control Statements

1. Objectives

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.

2. 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 are available for program design. In this lesson we shall discuss the various conditional control structures and iterative control structure will be discussed in the next lesson. Conditional constructs are required for making decision and choosing some execution path based on the satisfied condition.

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.

4. if ... else Construct

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

```
if (expression)
    statement1
else
    statement2
```

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.

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 use conditional construct, 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.

6. Using logical operators in if construct

Logical operators are indispensable part of if … else construct, but these should be use 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.

7. switch … case Construct

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.

Following is a program to count the occurrences of each digit, white space, and all other characters, using a switch:

```
#include <stdio.h>
main()  /* count digits, white space, others */
{
```

```
        int c, i, nwhite, nother, ndigit[10];
        nwhite = nother = 0;
        for (i = 0; i < 10; i++)
            ndigit[i] = 0;
        while ((c = getchar()) != EOF) {
            switch (c) {
                case '0': case '1': case '2': case '3': case '4':
                case '5': case '6': case '7': case '8': case '9':
                    ndigit[c-'0']++;
                    break;
                case ' ': case '\n': case '\t': nwhite++;
                    break;
                default:
                    nother++;
                    break;
            }
        }
        printf("digits =");
        for (i = 0; i < 10; i++)
            printf(" %d", ndigit[i]);
        printf(", white space = %d, other = %d\n", nwhite, nother);
        return 0;
    }
```

Note that multiple cases can be combined as is in the case with the digits in the above program. This is similar to the multiple conditions combined using logical or (||) with in one if statement. Therefore, it can be said that if multiple conditions involving single variable but combined using logical or (||) are under one if then those can be safely converted to switch construct. But if the logical and has been used to combine multiple conditions or if the multiple conditions involve more than one variable then switch construct can not be used.

The break statement causes an immediate exit from the switch. Because cases serve just as labels, after the code for one case is done, execution *falls through* to the next unless you take explicit action to escape. break and return are the most common ways to leave a switch. A break statement can also be used to force an immediate exit from while, for, and do loops, as will be discussed later in this chapter.

Falling through cases is a mixed blessing. On the positive side, it allows several cases to be attached to a single action, as with the digits in this example. But it also implies that normally each case must end with a break to prevent falling through to the next. Falling through from one case to another is not robust, being prone to disintegration when the program is modified. With the exception of multiple labels for a single computation, fall-throughs should be used sparingly, and commented.

Default case is used when none of the conditions specified under cases are encountered and thus may be ignored or an appropriate action may be taken. It mean all those cases which have not been handled using case labels.

As a matter of good form, put a break after the last case (the default here) even though it's logically unnecessary. Some day when another case gets added at the end, this bit of defensive programming will save you.

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.

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.

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?

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?

12. Suggested Books

Programming with ANSI and Turbo C	Ashok N. Kamthane
Programming using C	E. Balagurusamy
Application Programming in C	R. S. Salaria

Lesson No. 7 **Author: Dr. DharamVeer Sharma**
Converted into SLM by: Dr. Vishal Singh

Iterative Control Statements

1. Objectives

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.

2. 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.

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.

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.

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 them 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.

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 use. 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.

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.

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.

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?

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.

11. Suggested Books

Programming with ANSI and Turbo C	Ashok N. Kamthane
Programming using C	E. Balagurusamy
Application Programming in C	R. S. Salaria

M.Sc (IT) SEM-I | **Paper: MITM1102T**
Computer Programming using C

Lesson No. 8

Author: Dr. DharamVeer Sharma
Converted into SLM by: Dr. Vishal Singh

Arrays

1. Objectives

We shall discuss the declaration, initialization, printing and manipulation of array elements in this chapter. We shall see how arrays are passed to functions. We shall also discuss multi-dimensional array and using pointers with arrays.

2. Introduction

When multiple elements of the same data type are to be use then we need some such identifier which can store these multiple elements. In the C language array is one such data structure that can store groups of similar data type elements. These elements are stored in contiguous memory area. The array elements are accessed by providing the name of the storage area or the array and a subscript representing the position of the element within array.

3. Array basics

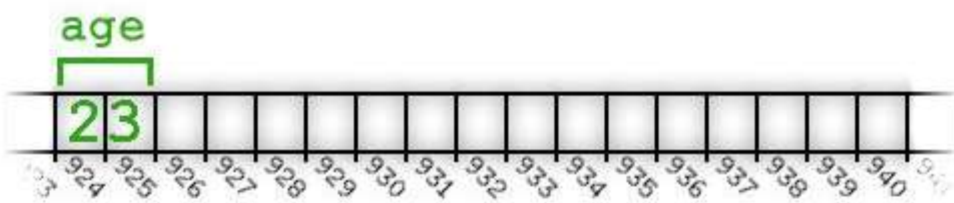
Let's start by looking at a single variable used to store a person's age.

```
#include <stdio.h>

int main()
{
  short age;
  age=23;
```

```
7:   printf("%d\n", age);
8:   return 0;
9: }
```

Not much to it. The variable age is created at line (5) as a short. A value is assigned to age. Finally, age is printed to the screen.

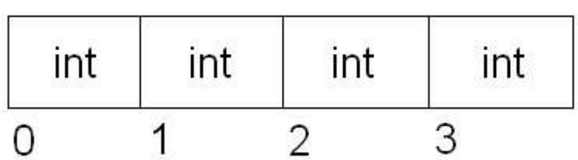


Now let's keep track of 4 ages instead of just one. We could create 4 separate variables, but 4 separate variables have limited appeal. (If using 4 separate variables **is** appealing to you, then consider keeping track of 1000 ages instead of just 4). Rather than using 4 separate variables, we'll use an array. Which can store a group of similar data type elements as single entity, whose each element is accessed by providing its offset with in the array.

An array is a simple sequence of objects. All of the objects in the sequence are of the same type. The following example presents an array of four integers.

Array of four integers

Each cell of the array is accessed through its index number. Arrays are zero-based. Thus the first cell is defined by index 0, the second by index 1, and so on.



Arrays are declared in the following manner:

```
type variable_name[array_size];
```

The following examples show different ways to declare various arrays.

```
int my_int_array[4]; // An array of 4 integers
double my_double_array[10]; // An array of 10 doubles
```

Accessing an element within an array

Elements within an array can be accessed (for reading or writing) with the subscript operator **[ ]**.

Example -

```
int my_array[10]; // create an array of 10 integers

my_array[3] = 15; // store the number 15 in the 4th cell

cout << my_array[3]; // display the number 15 we just put in
```

NOTE: In C, arrays do not have boundary checking. This means that the programmer is responsible for knowing the number of cells in the array, and thus, the last valid index which can be referenced. Accessing an element past the end of the array bounds will not cause a compiler error, but will crash your program at an unpredictable (but usually the worst possible) time. This is a very common bug which is found even in some of the most popular commercial software packages. Be careful – this is one of the most difficult bugs to track down.

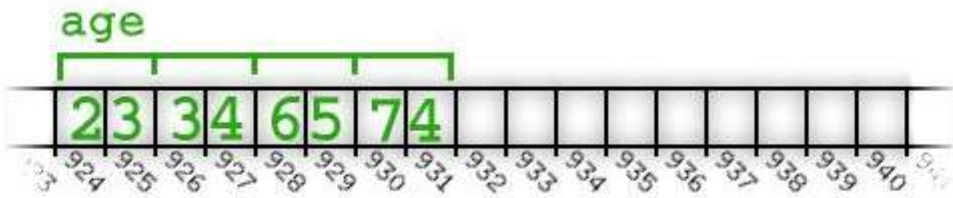
Example -

```
int the_array[2];          //array of 2 integers
the_array[0] = 42;         //valid access
the_array[1] = 1776;       //valid access
the_array[3] = 1492;       //oops! Error but nor reported by compiler.
```

Here's how to create an array and one way to initialize an array:

```
 1: #include <stdio.h>
 2:
 3: int main()
 4: {
 5:   short age[4];
 6:   age[0]=23;
 7:   age[1]=34;
 8:   age[2]=65;
 9:   age[3]=74;
10:   return 0;
11: }
```

On line (5), an array of 4 short's is created. Values are assigned to each variable in the array on line (6) through line (9).



Accessing any single short variable, or element, in the array is straightforward. Simply provide a number in square braces next to the name of the array. The number identifies which of the 4 elements in the array you want to access.

The program above shows that the first element of an array is accessed with the number 0 rather than 1. Later in the tutorial, We'll discuss why 0 is used to indicate the first element in the array.

4. Array Initialization

When an array of any type is created, the cells within the array are not empty. Depending on your setup, they will either have an already initialized value depending on the type of array you created, or the cells will contain garbage values left over from previous use of that piece of memory. In either case, it is usually a good idea to initialize the elements of the array prior to use. Two common ways of initializing an array is through an initializer list or through a loop. Array elements can be initialized at the time of declaration of the array as following.

Example (initializer list) -

```
int int_arr[] = { 34, 68, 7, 9, 20 };

double double_arr[] = { 22.78, 9.7, 3.1415, 2.71 };
char name[10] = "John";

char name[10]= { 'J', 'o','h','n','\0'};
```

In all the above declaration the array has been initialised there itself. If an array is partially initialized then the remaining elements are automatically initialized to 0 in case of numeric arrays.

```
int a[10] = {0};
```

The above example is a handy way of initializing all elements of an array to 0.

NOTE: When initializer lists are used, the size of the array inside the brackets is not needed. You are free to explicitly place the number there, but if you do not, the compiler will know the size based on the number of elements you initialized the array with.

Example (loop) -

```
const int ARRAY_SIZE = 10;
```

```
int my_array[ARRAY_SIZE];

for (int i = 0; i < ARRAY_SIZE; i++)

{
        my_array[i] = 0;

}
```

This method is useful when arrays are to be initialized with in the program.

NOTE: A 'for' loop is used by convention to initialize the array. When the value to be used to initialize the array is the same for all the cells, a loop is usually the way to go.

Arrays and Loops

Most of the useful work done with an array requires some sort of searching through the array. While searching through the array, you are accessing every cell in that array. This is done with the same kind of 'for' loop. The following example 'traverses' (runs through) the entire array in order to check if the number 2 appears anywhere within the array.

Example:

```
const int ARR_SIZE = 7;

int my_array[ARR_SIZE] = { 60, 3, 2, 8, 19, 2, 9 };

for (int i = 0; i < ARR_SIZE; i++)

{
        if (my_array[i] == 2)

        {

                cout << "Number 2 appears in index " << i;

        }
}
```

5. Arrays and Functions

Arrays, by default, are not passed to functions in the same way as regular variables are. In C, regular variables are passed by value, meaning that any changes you make to those variables in that function will not persist after you leave the function. Arrays are passed by reference (this isn't technically true, but the effect is the same), meaning that any changes you make to array cells in the function are still in effect when the function exits.

NOTE: (For the more inquisitive of you): The name of the array is actually a pointer to the first cell of the array. When you pass an array to a function, what is passed is actually the address of the first cell. Because of this, access to other cells of the array is freely available from inside the function through the use of the subscript operator or through pointer arithmetic.

What follows is a complete program which declares an array, initializes its cells and increments the value in each cell in a separate function.

Example -

```
void increment_all(int an_array[], int size)

{

    for (int i = 0; i < size; i++)

        an_array[i] += 1;

}

int main()

{
    const int ARR_SIZE = 10;

    int my_arr[ARR_SIZE];

    for (int i = 0; i < ARR_SIZE; i++)

        my_arr[i] = i;

    increment_all(my_arr, ARR_SIZE);

    return 0;

}
```

Like other languages, C uses arrays as a way of describing a collection of variables with identical properties. The group has a single name for all of the members, with the individual members being selected by an index.

Here's an array being declared:

```
double ar[100];
```

The name of the array is ar and its members are accessed as ar[0] through to ar[99] inclusive, as the following figure shows.

ar[0]	ar[1]	. . .	ar[99]

100 element array

Each of the hundred members is a separate variable whose type is double. Without exception, all arrays in C are numbered from 0 up to one less than the bound given in the declaration. This is a prime cause of surprise to beginners—watch out for it. For simple examples of the use of arrays, look back at earlier chapters where several problems are solved with their help.

One important point about array declarations is that they don't permit the use of varying subscripts. The numbers given must be constant expressions which can be evaluated at compile time, not run time. For example, this function incorrectly tries to use its argument in the size of an array declaration:

```
f(int x){
        char var_sized_array[x];        /* FORBIDDEN */
}
```

It's forbidden because the value of x is unknown when the program is compiled; it's a run-time, not a compile-time, value.

To tell the truth, it would be easy to support arrays whose first dimension is variable, but neither Old C nor the Standard permits it, although we do know of one Very Old C compiler that used to do it.

6. Multi-dimensional Arrays

Multidimensional arrays can be declared like this:

```
int three_dee[5][4][2];
int t_d[2][3]
```

The use of the brackets gives a clue to what is going on. If you refer to the precedence table you'll see that [] associates left to right and that, as a result, the first declaration gives us a five-element array called three_dee. The members of that array are each a four element array whose members are an array of two ints. We have declared arrays of arrays, as the following figure shows for two dimensions.

Two-dimensional array, showing layout

In the diagram, you will notice that t_d[0] is one element, immediately followed by t_d[1] (there is no break). It so happens that both of those elements are themselves arrays of three integers. Because of C's storage layout rules, t_d[1][0] is immediately after t_d[0][2]. It would be possible (but very poor practice) to access t_d[1][0] by making use of the lack of array-bound checking in C, and to use the expression t_d[0][3]. That is not recommended—apart from anything else, if the declaration of t_d ever changes, then the results will be likely to surprise you.

That's all very well, but does it really matter in practice? Not much it's true; but it is interesting to note that in terms of actual machine storage layout the rightmost subscript 'varies fastest'. This has an impact when arrays are accessed via pointers. Otherwise, they can be used just as would be expected; expressions like these are quite in order:

```
three_dee[1][3][1] = 0;
three_dee[4][3][1] += 2;
```

The second of those is interesting for two reasons. First, it accesses the very last member of the entire array—although the subscripts were declared to be [5][4][2], the highest usable subscript is always one less than the one used in the declaration. Second, it shows where the combined assignment operators are a real blessing. For the experienced C programmer it is much easier to tell that only one array member is being accessed, and that it is being incremented by two. Other languages would have to express it like this:

```
three_dee[4][3][1] = three_dee[4][3][1] + 2;
```

It takes a conscious effort to check that the same array member is being referenced on both sides of the assignment. It makes thing easier for the compiler too: there is only one array indexing calculation to do, and this is likely to result in shorter, faster code. (Of course a clever compiler would notice that the left- and right-hand sides look alike and would be able to generate equally efficient code—but not all compilers are clever and there are lots of special cases where even clever compilers are unable to make use of the information.)

It may be of interest to know that although C offers support for multidimensional arrays, they aren't particularly common to see in practice. One-dimensional arrays are present in most programs, if for no other reason than that's what strings are. Two dimensional arrays are seen occasionally, and arrays of higher order than that are most uncommon. One of the reasons is that the array is a rather inflexible data structure, and the ease of building and manipulating other types of data structures in C means that they tend to replace arrays in the more advanced programs. We will see more of this when we look at pointers.

7. Summary

Arrays are used for storing multiple elements of the same data type. Elements of array can be accessed by providing a subscript enclosed in square brackets. Arrays can be initialized at the time of declaration or using a for loop. If array is partially initialized then the remaining elements are automatically initialized to 0, in case of numeric arrays. Arrays can be passed as arguments to functions. Multi-dimensional arrays are used for storing matrix and other higher dimensional data.

8. Short Answer Type Questions

1. Define arrays.
2. How array can be initialized at the time of declaration?
3. Define multi-dimensional array.

9. Long Answer Type Questions

1. What are the methods of arrays initialization? Explain giving examples.
2. How arrays are passed as arguments to functions? Explain.
3. What is the purpose of multi-dimensional arrays? Give an example of these.

10. 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

Last Updated on June 2023

Mandatory Student Feedback Form

https://forms.gle/KS5CLhvpwrpgjwN98

Note: Students, kindly click this google form link, and fill this feedback form once.