

These notes encompass the theoretical aspect of **CSE 1121** in accordance with the International Islamic University Chittagong's syllabus.

In the realm of applied disciplines like Computer Science and Engineering (CSE), it's essential to grasp theory. However, the path to excellence also hinges on the adage '**practice makes perfect**'. Engaging practically with theoretical concepts not only enhances memory but also simplifies understanding.

Programming Language I

B. Sc. Engineering in CSE
Midterms, Autumn 2023

Assembled by Shawon – C233449

This outline provides a broad overview of the theoretical topics for revision. For a more in-depth understanding, it's recommended to refer to the original textbooks or study materials provided by the teacher. While online resources can be valuable, active participation and attentiveness in class are essential for comprehensive learning.

Find Your Path Here

<u>BASIC ORGANIZATION OF COMPUTER</u>	<u>V</u>
<u>DEFINITION OF SOFTWARE AND ITS CLASSIFICATION</u>	<u>VI</u>
<u>PROBLEM SOLVING STEPS</u>	<u>VII</u>
ALGORITHMS	VII
FLOW CHARTS	VIII
PSEUDO CODE	IX
<u>INTRODUCTION OF C</u>	<u>X</u>
HISTORY AND CHARACTERISTICS OF C	X
IDENTIFIERS AND KEYWORDS	X
DATA TYPES	XI
CONSTANTS	XII
VARIABLES	XII
STATEMENTS	XIII
SYMBOLIC CONSTANT	XIII
<u>OPERATORS</u>	<u>XIV</u>
ARITHMETIC	XIV
UNARY	XIV
RELATIONAL	XIV
LOGICAL	XIV
ASSIGNMENT	XV
CONDITIONAL OPERATORS	XV
<u>PRECEDENCE AND ASSOCIATIVITY OF OPERATORS</u>	<u>XVI</u>
<u>EXPRESSION</u>	<u>XVIII</u>
<u>TYPE CONVERSIONS</u>	<u>XVIII</u>
<u>LIBRARY FUNCTIONS</u>	<u>XXII</u>

INPUT AND OUTPUT	XXVI
-------------------------	-------------

MANAGING DATA INPUT	XXVI
----------------------------	-------------

SCANF	XXVI
-------	------

GETCHAR	XXVI
---------	------

GETS	XXVI
------	------

MANAGING DATA OUTPUT	XXVI
-----------------------------	-------------

PRINTF	XXVI
--------	------

PUTCHAR	XXVI
---------	------

PUTS	XXVI
------	------

FORMATTED INPUT AND OUTPUT	XXVII
-----------------------------------	--------------

CONTROL STATEMENTS	XXX
---------------------------	------------

BRANCHING	XXX
------------------	------------

IF	XXX
----	-----

ELSE	XXX
------	-----

ELSE...IF	XXX
-----------	-----

NESTED IF	XXXI
-----------	------

SWITCH	XXXI
--------	------

LOOPING	XXXII
----------------	--------------

FOR	XXXII
-----	-------

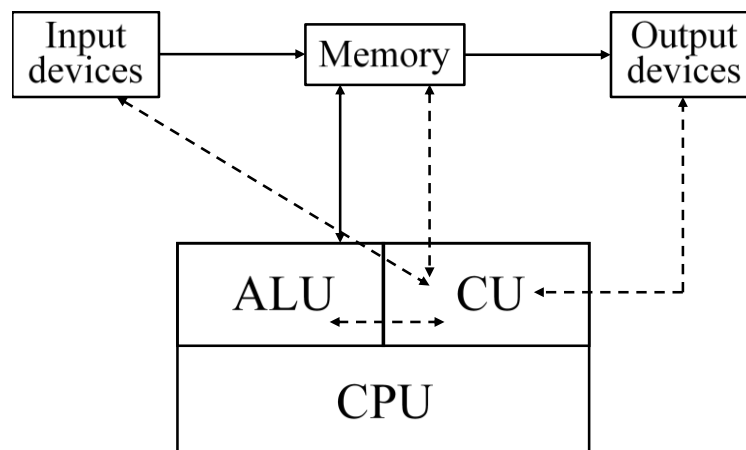
WHILE	XXXII
-------	-------

DO...WHILE	XXXII
------------	-------

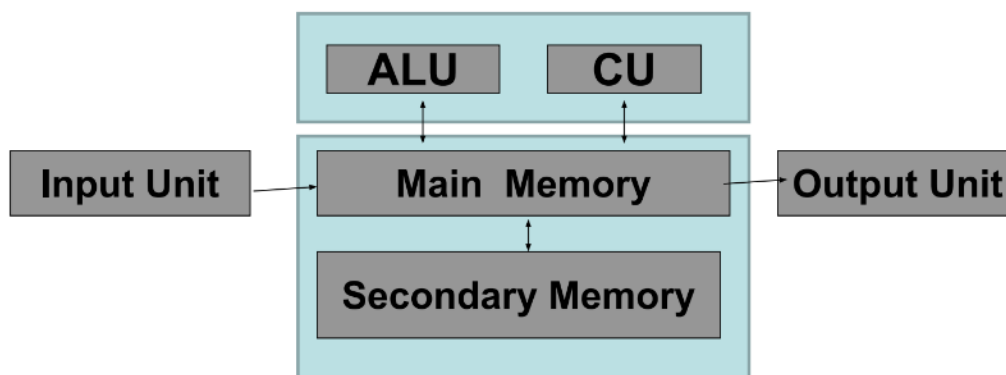
Basic Organization of Computer

A computer system comprises a processing unit, memory unit, and input-output devices. The processing unit, often the CPU, governs system functions. Memory includes an arithmetic and logic unit. Input devices (e.g., keyboard, mouse, microphone, scanner, joystick, stylus, trackball, etc.) enable data input, while output devices (monitor, printer, loudspeaker, projector, plotter, headphones, etc.) yield results. Input gets processed in memory, directed by the CPU, composed of Arithmetic-Logic Unit (ALU) and Control Unit (CU). ALU manages arithmetic operations and logical decisions, while CU handles control signals.

This can be represented through a Block Diagram of Computer.



Programming With C (Third Edition) by Byron Gottfried - Schaum's Outline



Slides by NN - Assistant Lecturer (CSE)

Definition of Software and Its Classification

A computer requires a set of instructions to carry out the specific task. This set of instructions is known as program and one or more programs along with their documentation are called software.

Software used on computer can be classified as:

- i) System software — refers to the essential programs that manage and control computer hardware and provide a foundation for other software to operate effectively.
e.g.: operating systems, compilers, editors, etc.
- ii) Application software — comprises specialized programs designed to fulfil specific tasks or purposes on a computer, serving as interfaces for users to interact with the system.
e.g.: weather forecasting software, Microsoft Word(MS Word), Code::Blocks, etc.

Problem Solving Steps

1. Problem Analysis
identify inputs, outputs, and processing requirements.
2. Development
identify various processing steps needed to solve the problem and represent them in a particular way (algorithm, flow charts, pseudo code, etc.).
3. Program Coding
refine step2; detail all the processing steps properly.
4. Program Compilation & Execution
add the syntax of the programming language to the above representation and it becomes the program. Then type the program and compile it after removing all the syntax related errors.
5. Program Debugging & Testing
run the program and check it with different types of input to verify its accuracy and precision. In case the output is incorrect for any combination of inputs; Debugging:
 - Hand/ desk Checking
 - Syntax (compile time) error
 - Execution (run time error (divide by zero))
 - Logical error
6. Program Documentation
 - Comments: /* */ or //
 - User manual

ALGORITHMS




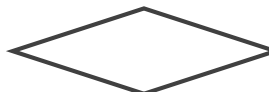

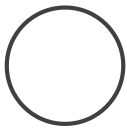
An algorithm is a set of well-defined instructions that is designed to solve a specific task.

Rules for constructing an algorithm:

- Input: There should be zero or more values which are to be supplied.
- Output: At least one output is to be produced.
- Definiteness: Each step must be clear and unambiguous.
- Finiteness: If we trace the steps of an algorithm, then for all cases, the algorithm must terminate after a finite number of steps.
- Effectiveness: Each step must be sufficiently basic that a person using only paper and pencil can in principle carry it out. In addition, not only each step is definite, it must also be feasible.
- Comment Session: Comment session is not mandatory for writing an algorithm. Comment is additional info of program for easily modification. In algorithm comment would be appear between two square brackets '[]'.

FLOW CHARTS

It is the pictorial representation of the process. It describes the sequence & flow of control & information in a process. It uses different symbols for depicting different activities.

Start, Stop	
Read, Print	
Processing Statements	
Condition Check	
Direction of flow	
Connectors (for longer flow charts)	

PSEUDO CODE

'Pseudo' means imitates and 'Code' means instruction. It is a formal design tool. It is also called Program Design Language.

Keywords:

- READ, GET
- PRINT, DISPLAY
- COMPUTE, CALCULATE

Guideline for writing Pseudocode:

- Steps should be understandable
- Capitalize the keyword
- Indent to show hierarchy
- End multiple line structure etc.

Introduction of C

HISTORY AND CHARACTERISTICS OF C

C was originally developed in the 1970s by Dennis Ritchie at Bell Telephone laboratories, Inc. It was outgrowth of two earlier languages, called BCPL and B.

(more details on this can be found at page 1.18 pf Programming With C (Third Edition) by Byron Gottfried - Schaum's Outline)

IDENTIFIERS AND KEYWORDS

Identifiers are names given to various program elements such as variables, functions and arrays.

- The underscore character '_' is often used in the middle of an identifier. An identifier may also begin with an underscore, though this is rarely done in practice.
- Identifiers consist of letters(a-z, A-Z), digits(0-9), and underscore character(_) in any order, except that the first character must not be a number.
- Both upper and lowercase letters are permitted, though common usage favours the use of lowercase letters for most types of identifiers.
- Upper and lowercase letters are not interchangeable (i.e., an uppercase letter is not equivalent to the corresponding lowercase letter).

rules for identifiers:

1. First character must be an alphabet (or underscore).
2. Must consist of only letters, digits, or underscore.
3. Only first 31 characters are significant.
4. Cannot use keyword.
5. Must not contain white space.

Keywords are certain reserved words that have standard, predefined meanings in C. Keywords serve as basic building blocks for program statements. All keywords must be written in lower case.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	floatn	short	unsigned
continue	for	signed	void
default	go to	size of	volatile
do	if	static	while

DATA TYPES

The basic kinds of data:

- *Numeric*
Integers
Real (floating point) numbers
- *Character* (are enclosed by single quotes in C)
All letters, numbers, and special symbols
Ex. 'A', '+', '5'
- *String* (are enclosed by double quotes in C)
Are combinations of more than one character
Ex. "programming", "ME 30"
- *Logical* (also called 'Boolean' named after George Boole an English mathematician from the early 1800's)
True or False (1 or 0)

The 4 basic data types are:

int	Represents integer numbers (whole numbers) without decimal points. Range: -2,147,483,648 to 2,147,483,647 (32 bits). Size: 4 bytes.	%d
char	Represents a single character (letter, digit, symbol) from the character set. Range: -128 to 127 (8 bits). Size: 1 byte.	%c
float	Represents floating-point numbers with decimal points, offering a compromise between range and precision. Range: Approximately 3.4×10^{-38} to 3.4×10^{38} or 1.2×10^{-38} to 3.4×10^{38} (32 bits). Size: 4 bytes.	%f
double	Represents double-precision floating-point numbers with higher precision compared to float. Range: Approximately 1.7×10^{-308} to 1.7×10^{308} or 2.2×10^{-308} to 1.8×10^{308} (64 bits). Size: 8 bytes.	%lf

Here, take s'more of these, data types:

long long int	%lld
short int	%hd
long int	%ld
unsigned long int	%lu
unsigned char	%c
long double	%Lf

CONSTANTS

There are *integer constants*, *floating-point constants*, *character constants*, *string constants*, *enumeration constants*.

Rules for numeric-type (integer, floating-point) constants:

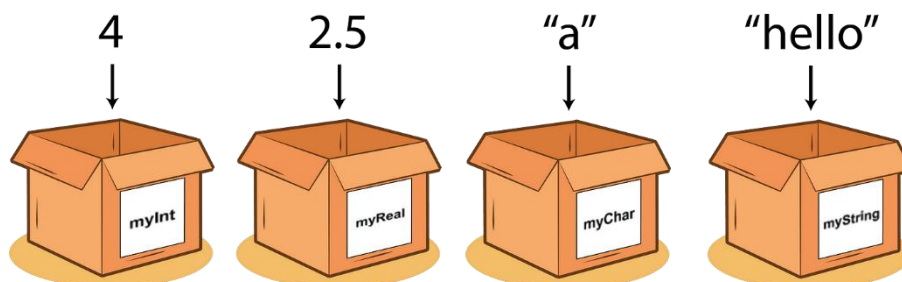
- Commas and blank space cannot be included within the constants.
- Constant is allowed to be preceded by a minus '-' sign.
- The value of constant cannot exceed the min and max bounds.

VARIABLES

Variable names correspond to locations in the computer's memory.

- Every variable has a name, a type, a size and a value.
- Whenever a new value is placed into a variable (e.g., through scanf or assigning), it replaces (and destroys) previous value
- Reading variables from memory does not change them

e.g., `printf("type/format specifier", what/variable);`



STATEMENTS

The statement causes the computer to carry out some actions. There are expression statements, compound statements, and control statements.

Expression statement consists of an expression followed by a semicolon. The execution of an expression statement causes the expression to be evaluated.

Compound statement consists of several individual statements enclosed within a pair of braces { }. Unlike an expression statement, a compound statement does not end with a semicolon.

Control statements are used to create special program features, such as logical tests, loops and branches. Many control statements require that other statements be embedded within them.

SYMBOLIC CONSTANT

A symbolic constant is a name that stands in for a sequence of characters, which can be a numeric, character, or string constant. During compilation, instances of a symbolic constant are substituted with their associated character sequences.

```
#define NAME text
```

Operators

ARITHMETIC

Arithmetic operators in programming perform basic math on numeric operands.

+ - × ÷ %

UNARY

Unary operators act on a single operand to produce a new value. They are usually placed before the operand, but in some cases, they can follow it.

- ++ -- ! sizeof (*type casting*)

These operators require a single variable as their operand.

Note: "post" operators (post-increment and post-decrement) perform the increment or decrement after the current value is used in the expression, while the "pre" operators (pre-increment and pre-decrement) do so before the value is used.

RELATIONAL

< <= > >=

Closely related with the relational operators are the following two equality operators:

== !=

LOGICAL

x	y		x	y	&&	z	!z
0	0	0	0	0	0	0	1
0	1	1	0	1	0		
1	0	1	1	0	0	1	0
1	1	1	1	1	1		

ASSIGNMENT

It is used to assign a value or expression etc. to a variable.

e.g., `variable_name/identifier = expression`

CONDITIONAL OPERATORS

Conditional operators, often referred to as the *Ternary Operator*, are a concise way to express conditional (if-else) statements. They take three operands and return a value based on a condition. The format is typically:

`condition ? value_if_true : value_if_false`

Precedence and Associativity of Operators

	Parentheses	()
	Postfix increment/decrement	++ --
	Unary increment/decrement, logical negation, bitwise negation	++ -- ! ~
Arithmetic	Multiplication, division, and remainder	× ÷ %
	Addition and subtraction	+ -
Bitwise	Left shift and Right shift	<< >>
	Relational and equality operators	== != < <= > >=
Bitwise	AND	&
	XOR	^
	OR	

Logical	AND	&&
	OR	
Conditional operator		? :
Assignment operators		= += -= *= /= %= <<= >>= &= ^= =
Comma operator		,

Both determine the order in which operators are evaluated in an expression. Precedence defines the priority of operators, while associativity determines the order of evaluation when multiple operators of the same precedence appear consecutively.

Associativity determines the grouping of operators when they have the same precedence. In C, most operators are left-associative, meaning they are evaluated from left to right.

The *Right-Associative* operators in C are evaluated from right to left:

the assignment operators

the unary operators

the conditional operators

the comma operator

Expression

In C, expressions are combinations of variables, constants, operators, and functions that yield a single value when evaluated. They are used for calculations, comparisons, and decision-making within the program.

Type Conversions

2 primary types of Type Castings

a) Implicit Type Casting (Type Coercion)

Also known as automatic type casting, it occurs when the compiler automatically converts one data type into another without the programmer's intervention.

This usually happens when performing operations between different data types, and the compiler promotes or demotes one of them to ensure compatibility.

Rules for Implicit Type Casting:

- Smaller data types are implicitly converted to larger data types to avoid loss of data.
- For arithmetic operations, if operands are of different data types, the lower-ranked type is promoted to the higher-ranked type.
- When assigning a value of one data type to a variable of another data type, the compiler may perform implicit casting if it's safe.

b) Explicit Type Casting (Type Conversion)

Also known as manual type casting, it occurs when the programmer explicitly specifies the conversion from one data type to another.

This is done using casting operators, such as `(type)` in C, to ensure that a value is treated as a different data type.

Rules for Explicit Type Casting:

- The programmer uses casting operators like `(int)`, `(float)`, etc., to specify the desired data type for conversion.
- Explicit type casting can lead to data loss and truncation if the destination type can't represent the full range of values from the source type.

<i>Data Type</i>	<i>Description</i>	<i>Typical Memory Requirements</i>
int	Integer quantity	2 bytes or 1 word (varies from one computer to another)
short	Short integer quantity (may contain fewer digits than int)	2 bytes or 1 word (varies from one computer to another)
long	Long integer quantity (may contain more digits than int)	1 or 2 words (varies from one computer to another)
unsigned	Unsigned (positive) integer quantity (maximum permissible quantity is approximately twice as large as int)	2 bytes or 1 word (varies from one computer to another)
char	Single character	1 byte
signed char	Single character, with numerical values ranging from -128 to +127	1 byte
unsigned char	Single character, with numerical values ranging from 0 to 255	1 byte
float	Floating-point number (i.e., a number containing a decimal point and/or an exponent)	1 word
double	Double-precision floating-point number (i.e., more significant figures, and an exponent that may be larger in magnitude)	2 words
long double	Double-precision floating-point number (may be higher precision than double)	2 or more words (varies from one computer to another)
void	Special data type for functions that do not return any value	(not applicable)
enum	Enumeration constant (special type of int)	2 bytes or 1 word (varies from one computer to another)

Note: The qualifier **unsigned** may appear with **short int** or **long int**, e.g., **unsigned short int** (or **unsigned short**), or **unsigned long int** (or **unsigned long**).

(Extras)

CONVERSION RULES

These rules apply to arithmetic operations between two operators with dissimilar data types. There may be some variation from one version of C to another.

1. If one of the operands is long double, the other will be converted to long double and the result will be long double.
2. Otherwise, if one of the operands is double, the other will be converted to double and the result will be double.
3. Otherwise, if one of the operands is float, the other will be converted to float and the result will be float.
4. Otherwise, if one of the operands is unsigned long int, the other will be converted to unsigned long int and the result will be unsigned long int.
5. Otherwise, if one of the operands is long int and the other is unsigned int, then:
 - If unsigned int can be converted to long int, the unsigned int operand will be converted as such and the result will be long int.
 - Otherwise, both operands will be converted to unsigned long int and the result will be unsigned long int.
6. Otherwise, if one of the operands is long int, the other will be converted to long int and the result will be long int.
7. Otherwise, if one of the operands is unsigned int, the other will be converted to unsigned int and the result will be unsigned int.
8. If none of the above conditions applies, then both operands will be converted to int (if necessary), and the result will be int.

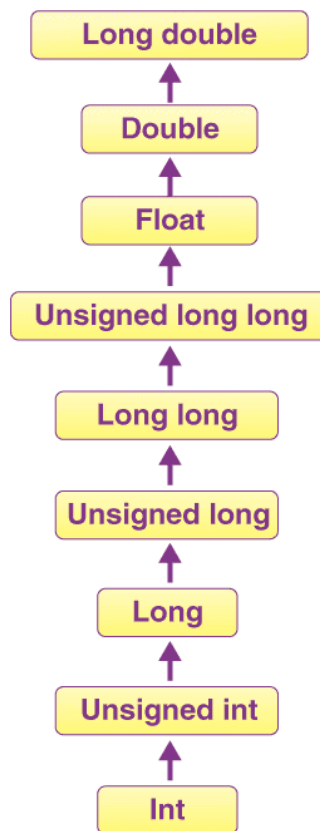
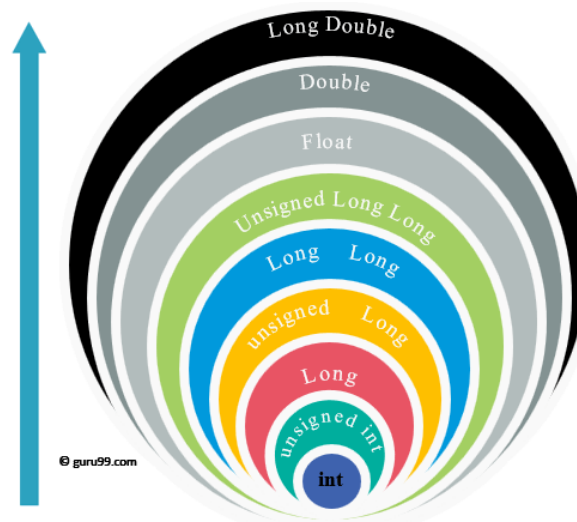
Note that some versions of C automatically convert all floating-point operands to double-precision.

ASSIGNMENT RULES

If the two operands in an assignment expression are of different data types, then the value of the right-hand operand will automatically be converted to the type of the operand on the left. The entire assignment expression will then be of this same data type. In addition,

1. A floating-point value may be truncated if assigned to an integer identifier.
2. A double-precision value may be rounded if assigned to a floating-point (single-precision) identifier.
3. An integer quantity may be altered (some high-order bits may be lost) if it is assigned to a shorter integer identifier or to a character identifier.

Hierarchy of Type Conversion/Casting.



Library Functions

<i>Function</i>	<i>Type</i>	<i>Purpose</i>	<i>include File</i>
<code>abs(i)</code>	int	Return the absolute value of <code>i</code> .	<code>stdlib.h</code>
<code>acos(d)</code>	double	Return the arc cosine of <code>d</code> .	<code>math.h</code>
<code>asin(d)</code>	double	Return the arc sine of <code>d</code> .	<code>math.h</code>
<code>atan(d)</code>	double	Return the arc tangent of <code>d</code> .	<code>math.h</code>
<code>atan2(d1,d2)</code>	double	Return the arc tangent of <code>d1/d2</code> .	<code>math.h</code>
<code>atof(s)</code>	double	Convert string <code>s</code> to a double-precision quantity.	<code>stdlib.h</code>
<code>atoi(s)</code>	int	Convert string <code>s</code> to an integer.	<code>stdlib.h</code>
<code>atol(s)</code>	long	Convert string <code>s</code> to a long integer.	<code>stdlib.h</code>
<code>calloc(u1,u2)</code>	void*	Allocate memory for an array having <code>u1</code> elements, each of length <code>u2</code> bytes. Return a pointer to the beginning of the allocated space.	<code>malloc.h</code> , or <code>stdlib.h</code>
<code>ceil(d)</code>	double	Return a value rounded up to the next higher integer.	<code>math.h</code>
<code>cos(d)</code>	double	Return the cosine of <code>d</code> .	<code>math.h</code>
<code>cosh(d)</code>	double	Return the hyperbolic cosine of <code>d</code> .	<code>math.h</code>
<code>difftime(l1,l2)</code>	double	Return the time difference <code>l1 - l2</code> , where <code>l1</code> and <code>l2</code> represent elapsed times beyond a designated base time (see the <code>time</code> function).	<code>time.h</code>
<code>exit(u)</code>	void	Close all files and buffers, and terminate the program. (Value of <code>u</code> is assigned by function, to indicate termination status.)	<code>stdlib.h</code>
<code>exp(d)</code>	double	Raise e to the power <code>d</code> ($e = 2.7182818 \dots$ is the base of the natural (Naperian) system of logarithms).	<code>math.h</code>
<code>fabs(d)</code>	double	Return the absolute value of <code>d</code> .	<code>math.h</code>
<code>fclose(f)</code>	int	Close file <code>f</code> . Return 0 if file is successfully closed.	<code>stdio.h</code>
<code>feof(f)</code>	int	Determine if an end-of-file condition has been reached. If so, return a nonzero value; otherwise, return 0.	<code>stdio.h</code>
<code>fgetc(f)</code>	int	Enter a single character from file <code>f</code> .	<code>stdio.h</code>
<code>fgets(s,i,f)</code>	char*	Enter string <code>s</code> , containing <code>i</code> characters, from file <code>f</code> .	<code>stdio.h</code>
<code>floor(d)</code>	double	Return a value rounded down to the next lower integer.	<code>math.h</code>
<code>fmod(d1,d2)</code>	double	Return the remainder of <code>d1/d2</code> (with same sign as <code>d1</code>).	<code>math.h</code>
<code>fopen(s1,s2)</code>	file*	Open a file named <code>s1</code> of type <code>s2</code> . Return a pointer to the file.	<code>stdio.h</code>
<code>fprintf(f,...)</code>	int	Send data items to file <code>f</code> (remaining arguments are complicated — see Appendix G).	<code>stdio.h</code>
<code>fputc(c,f)</code>	int	Send a single character to file <code>f</code> .	<code>stdio.h</code>
<code>fputs(s,f)</code>	int	Send string <code>s</code> to file <code>f</code> .	<code>stdio.h</code>

<code>fread(s,i1,i2,f)</code>	int	Enter <code>i2</code> data items, each of size <code>i1</code> bytes, from file <code>f</code> to string <code>s</code> .	<code>stdio.h</code>
<code>free(p)</code>	void	Free a block of allocated memory whose beginning is indicated by <code>p</code> .	<code>malloc.h</code> , or <code>stdlib.h</code>
<code>fscanf(f,...)</code>	int	Enter data items from file <code>f</code> (remaining arguments are complicated — see Appendix G)	<code>stdio.h</code>
<code>fseek(f,l,i)</code>	int	Move the pointer for file <code>f</code> a distance <code>l</code> bytes from location <code>i</code> (<code>i</code> may represent the beginning of the file, the current pointer position, or the end of the file).	<code>stdio.h</code>
<code>ftell(f)</code>	long int	Return the current pointer position within file <code>f</code> .	<code>stdio.h</code>
<code>fwrite(s,i1,i2,f)</code>	int	Send <code>i2</code> data items, each of size <code>i1</code> bytes, from string <code>s</code> to file <code>f</code> .	<code>stdio.h</code>
<code>getc(f)</code>	int	Enter a single character from file <code>f</code> .	<code>stdio.h</code>
<code>getchar()</code>	int	Enter a single character from the standard input device.	<code>stdio.h</code>
<code>gets(s)</code>	char*	Enter string <code>s</code> from the standard input device.	<code>stdio.h</code>
<code>isalnum(c)</code>	int	Determine if argument is alphanumeric. Return a nonzero value if true; 0 otherwise.	<code>ctype.h</code>
<code>isalpha(c)</code>	int	Determine if argument is alphabetic. Return a nonzero value if true; 0 otherwise.	<code>ctype.h</code>
<code>isascii(c)</code>	int	Determine if argument is an ASCII character. Return a nonzero value if true; 0 otherwise.	<code>ctype.h</code>
<code>iscntrl(c)</code>	int	Determine if argument is an ASCII control character. Return a nonzero value if true; 0 otherwise.	<code>ctype.h</code>
<code>isdigit(c)</code>	int	Determine if argument is a decimal digit. Return a nonzero value if true; 0 otherwise.	<code>ctype.h</code>
<code>isprint(c)</code>	int	Determine if argument is a printing ASCII character (hex 0x20–0x7e; octal 040–176). Return a nonzero value if true; 0 otherwise.	<code>ctype.h</code>
<code>ispunct(c)</code>	int	Determine if argument is a punctuation character. Return a nonzero value if true; 0 otherwise.	<code>ctype.h</code>
<code>isspace(c)</code>	int	Determine if argument is a whitespace character. Return a nonzero value if true; 0 otherwise.	<code>ctype.h</code>
<code>isupper(c)</code>	int	Determine if argument is uppercase. Return a nonzero value if true; 0 otherwise.	<code>ctype.h</code>
<code>isxdigit(c)</code>	int	Determine if argument is a hexadecimal digit. Return a nonzero value if true; 0 otherwise.	<code>ctype.h</code>
<code>labs(l)</code>	long int	Return the absolute value of <code>l</code> .	<code>math.h</code>
<code>log(d)</code>	double	Return the natural logarithm of <code>d</code> .	<code>math.h</code>

<code>log10(d)</code>	double	Return the logarithm (base 10) of <code>d</code> .	<code>math.h</code>
<code>malloc(u)</code>	void*	Allocate <code>u</code> bytes of memory. Return a pointer to the beginning of the allocated space.	<code>malloc.h</code> , or <code>stdlib.h</code>
<code>pow(d1,d2)</code>	double	Return <code>d1</code> raised to the <code>d2</code> power.	<code>math.h</code>
<code>printf(...)</code>	int	Send data items to the standard output device (arguments are complicated — see Appendix G).	<code>stdio.h</code>
<code>putc(c,f)</code>	int	Send a single character to file <code>f</code> .	<code>stdio.h</code>
<code>putchar(c)</code>	int	Send a single character to the standard output device.	<code>stdio.h</code>
<code>puts(s)</code>	int	Send string <code>s</code> to the standard output device.	<code>stdio.h</code>
<code>rand()</code>	int	Return a random positive integer.	<code>stdlib.h</code>
<code>rewind(f)</code>	void	Move the pointer to the beginning of file <code>f</code> .	<code>stdio.h</code>
<code>scanf(...)</code>	int	Enter data items from the standard input device (arguments are complicated — see Appendix G).	<code>stdio.h</code>
<code>sin(d)</code>	double	Return the sine of <code>d</code> .	<code>math.h</code>
<code>sinh(d)</code>	double	Return the hyperbolic sine of <code>d</code> .	<code>math.h</code>
<code>sqrt(d)</code>	double	Return the square root of <code>d</code> .	<code>math.h</code>
<code>srand(u)</code>	void	Initialize the random number generator.	<code>stdlib.h</code>
<code>strcmp(s1,s2)</code>	int	Compare two strings lexicographically. Return a negative value if <code>s1 < s2</code> ; 0 if <code>s1</code> and <code>s2</code> are identical; and a positive value if <code>s1 > s2</code> .	<code>string.h</code>
<code>strcmpi(s1,s2)</code>	int	Compare two strings lexicographically, without regard to case. Return a negative value if <code>s1 < s2</code> ; 0 if <code>s1</code> and <code>s2</code> are identical; and a positive value if <code>s1 > s2</code> .	<code>string.h</code>
<code>strcpy(s1,s2)</code>	char*	Copy string <code>s2</code> to string <code>s1</code> .	<code>string.h</code>
<code>strlen(s)</code>	int	Return the number of characters in a string.	<code>string.h</code>
<code>strset(s,c)</code>	char*	Set all characters within <code>s</code> to <code>c</code> (excluding the terminating null character <code>\0</code>).	<code>string.h</code>
<code>system(s)</code>	int	Pass command <code>s</code> to the operating system. Return 0 if the command is successfully executed; otherwise, return a nonzero value, typically <code>-1</code> .	<code>stdlib.h</code>
<code>tan(d)</code>	double	Return the tangent of <code>d</code> .	<code>math.h</code>
<code>tanh(d)</code>	double	Return the hyperbolic tangent of <code>d</code> .	<code>math.h</code>
<code>time(p)</code>	long int	Return the number of seconds elapsed beyond a designated base time.	<code>time.h</code>
<code>toascii(c)</code>	int	Convert value of argument to ASCII.	<code>ctype.h</code>
<code>tolower(c)</code>	int	Convert letter to lowercase.	<code>ctype.h</code> , or <code>stdlib.h</code>
<code>toupper(c)</code>	int	Convert letter to uppercase.	<code>ctype.h</code> , or <code>stdlib.h</code>

Notes: *Type* refers to the data type of the quantity that is returned by the function. An asterisk (*) denotes a pointer.
c denotes a character-type argument
d denotes a double-precision argument

<i>Function</i>	<i>Type</i>	<i>Purpose</i>
<code>abs(i)</code>	int	Return the absolute value of <i>i</i> .
<code>ceil(d)</code>	double	Round up to the next integer value (the smallest integer that is greater than or equal to <i>d</i>).
<code>cos(d)</code>	double	Return the cosine of <i>d</i> .
<code>cosh(d)</code>	double	Return the hyperbolic cosine of <i>d</i> .
<code>exp(d)</code>	double	Raise <i>e</i> to the power <i>d</i> (<i>e</i> = 2.7182818... is the base of the natural (Naperian) system of logarithms).
<code>fabs(d)</code>	double	Return the absolute value of <i>d</i> .
<code>floor(d)</code>	double	Round down to the next integer value (the largest integer that does not exceed <i>d</i>).
<code>fmod(d1, d2)</code>	double	Return the remainder (i.e., the noninteger part of the quotient) of <i>d1</i> / <i>d2</i> , with same sign as <i>d1</i> .
<code>getchar()</code>	int	Enter a character from the standard input device.
<code>log(d)</code>	double	Return the natural logarithm of <i>d</i> .
<code>pow(d1, d2)</code>	double	Return <i>d1</i> raised to the <i>d2</i> power.
<code>printf(...)</code>	int	Send data items to the standard output device (arguments are complicated — see Chap. 4).
<code>putchar(c)</code>	int	Send a character to the standard output device.
<code>rand()</code>	int	Return a random positive integer.
<code>sin(d)</code>	double	Return the sine of <i>d</i> .
<code>sqrt(d)</code>	double	Return the square root of <i>d</i> .
<code>srand(u)</code>	void	Initialize the random number generator.
<code>scanf(...)</code>	int	Enter data items from the standard input device (arguments are complicated — see Chap. 4).
<code>tan(d)</code>	double	Return the tangent of <i>d</i> .
<code>toascii(c)</code>	int	Convert value of argument to ASCII.
<code>tolower(c)</code>	int	Convert letter to lowercase.
<code>toupper(c)</code>	int	Convert letter to uppercase.

Input and Output

MANAGING DATA INPUT		Scanf	Getchar	Gets
	Purpose	Reads formatted input from the standard input stream (usually the keyboard) into variables.	Reads a single character from the standard input stream (keyboard).	Reads a line of text from the standard input stream (keyboard) into a string.
	Differences	<ul style="list-style-type: none"> Requires format specifiers to specify the data type. Can read multiple values in a single call. 	<ul style="list-style-type: none"> Reads one character at a time. No format specifiers are needed. 	<ul style="list-style-type: none"> Reads a whole line of text, including spaces. Considered unsafe due to potential buffer overflows.
MANAGING DATA OUTPUT		Printf	Putchar	Puts
	Purpose	Prints formatted output to the standard output stream (usually the screen or console).	Writes a single character to the standard output stream (usually the screen).	Writes a string to the standard output stream (usually the screen).
	Differences	<ul style="list-style-type: none"> Requires format specifiers to format and display data. Allows complex formatting with placeholders. 	<ul style="list-style-type: none"> Outputs one character at a time. No format specifiers are needed. 	<ul style="list-style-type: none"> Outputs a string with a newline character at the end. Simpler for printing strings but limited formatting.

FORMATTED INPUT AND OUTPUT

scanf Conversion Characters

<i>Conversion Character</i>	<i>Meaning</i>
c	Data item is a single character
d	Data item is a decimal integer
e	Data item is a floating-point value
f	Data item is a floating-point value
g	Data item is a floating-point value
h	Data item is a short integer
i	Data item is a decimal, hexadecimal, or octal integer
o	Data item is an octal integer
s	Data item is a string followed by a whitespace character (the null character '\0' will automatically be added at the end)
u	Data item is an unsigned decimal integer
x	Data item is a hexadecimal integer
[. . .]	Data item is a string which may include whitespace characters

A *prefix* may precede certain conversion characters.

<i>Prefix</i>	<i>Meaning</i>
h	Short data item (short integer or short unsigned integer)
l	Long data item (long integer, long unsigned integer or double)
L	Long data item (long double)

Example:

```
int a;  
short b;  
long c;  
unsigned d;  
double x;  
char str[80];  
  
scanf("%5d %3hd %12ld %12lu %15lf", &a, &b, &c, &d, &x);  
  
scanf("%[^\n]", str);
```

***printf* Conversion Characters**

<i>Conversion Character</i>	<i>Meaning</i>
c	Data item is displayed as a single character
d	Data item is displayed as a signed decimal integer
e	Data item is displayed as a floating-point value with an exponent
f	Data item is displayed as a floating-point value without an exponent
g	Data item is displayed as a floating-point value using either e -type or f -type conversion, depending on value; trailing zeros, trailing decimal point will not be displayed.
i	Data item is displayed as a signed decimal integer
o	Data item is displayed as an octal integer, without a leading zero
s	Data item is displayed as a string
u	Data item is displayed as an unsigned decimal integer
x	Data item is displayed as a hexadecimal integer, without leading 0x

Some of these characters are interpreted differently than with the `scanf` function.

A *prefix* may precede certain conversion characters.

<i>Prefix</i>	<i>Meaning</i>
h	Short data item (short integer or short unsigned integer)
l	Long data item (long integer, long unsigned integer or double)
L	Long data item (long double)

Example:

```
int a;
short b;
long c;
unsigned d;
double x;
char str[80];

printf("%5d %3hd %12ld %12lu %15.7le\n", a, b, c, d, x);
printf("%40s\n", str);
```

Flags

<i>Flag</i>	<i>Meaning</i>
-	Data item is left justified within the field (blank spaces required to fill the minimum field-width will be added <i>after</i> the data item rather than <i>before</i> the data item.)
+	A sign (either + or -) will precede each signed numerical data item. Without this flag, only negative data items are preceded by a sign.
0	Causes leading zeros to appear instead of leading blanks. Applies only to data items that are right justified within a field whose minimum size is larger than the data item. (<i>Note:</i> Some compilers consider the zero flag to be a part of the field-width specification rather than an actual flag. This assures that the 0 is processed last, if multiple flags are present.)
' '	A blank space will precede each positive signed numerical data item. This flag is overridden by the + flag if both are present.
# (blank space) with o- and x- type conversion)	Causes octal and hexadecimal data items to be preceded by 0 and 0x, respectively.
# , f- and g- type conversion)	Causes a decimal point to be present in all floating-point numbers, even if the data item (<i>with</i> is a whole number. Also prevents the truncation of trailing zeros in g-type conversion.

Example:

```
int a;
short b;
long c;
unsigned d;
double x;

printf("%+5d %+5hd %+12ld %-12lu %#15.7le\n", a, b, c, d, x);
```

Control Statements

BRANCHING

It refers to the ability of a program to make decisions and execute different code blocks based on certain conditions. It allows the program to follow different paths depending on whether specific conditions are true or false, enabling dynamic and flexible behaviour.

Branching statements are:

If

Executes a block of code if a specified condition is true.

Syntax:

```
if (condition) { /* code to execute */ }
```

Else

Executes a block of code when the previous 'if' condition is false.

Syntax:

```
if (condition) {  
    // Code to execute if the condition is true  
} else {  
    // Code to execute if the condition is false  
}
```

Else...If

It adds an extra condition to evaluate if the preceding 'if' is false, enabling sequential testing of multiple conditions for complex decision-making.

Syntax:

```
if (condition1) {  
    // Code to execute if condition1 is true  
} else if (condition2) {  
    // Code to execute if condition2 is true (only if  
condition1 is false)  
} else {  
    // Code to execute if neither condition1 nor condition2 is  
true }
```

Nested If

Allows multiple levels of conditional statements inside one another.

Syntax:

```
if (condition1) {  
    if (condition2) {  
        /* code to execute if both conditions are true */  
    }  
}
```

Switch

Provides a way to select one of many code blocks to execute based on the value of an expression.

Syntax:

```
switch (expression) {  
    case value1:  
        // Code to execute if expression matches value1  
        break;  
    case value2:  
        // Code to execute if expression matches value2  
        break;  
    default:  
        // Code to execute if no cases match  
}
```

LOOPING

The repetitive execution of a block of code based on a specified condition or for a predetermined number of iterations. It's a fundamental control structure for automating repetitive tasks.

For

Executes code for a fixed number of times or based on a specified condition. It typically consists of initialization, condition, and update expressions.

```
for(Expression 1; Expression 2; Expression 3){  
    //code to be executed  
}
```

While

Repeatedly executes code as long as a specified condition remains true. It checks the condition before each iteration.

```
while(condition){  
    //code to be executed  
}
```

Do...While

It is similar to the 'while' loop but ensures that the code block is executed at least once before checking the condition.

```
do{  
    //code to be executed  
} while(condition);
```