



KAMALDNP

kamaldnp.com • IT Training, Varanasi

PDF created by Raj Kamal using AI

Short Notes — Computer Programming Using C

BTEUP Diploma in Computer Science & Engineering • 3rd Semester • Short Notes
Session 2026-27

Short Revision Notes — prepared by KAMALDNP Academy from the official BTEUP 2026-27 syllabus. This is NOT a BTEUP board paper or publication.

These **short revision notes** cover all five units of Computer Programming Using C from the official BTEUP 2026-27 syllabus, in syllabus order. Each point gives the definition, syntax and example a short answer needs, in plain English.

This subject is a **practicum** in the 2026-27 scheme, so these notes target its **theory (lecture) part, lab work and viva** together. Revise one unit, practise its programs in the lab, then try the likely question at the end of the unit.

Unit 1 — Introduction to Problem Solving (2 Periods, 20%)

Topics: Computational way of thinking; flowchart and algorithm; variable, constant and representation; data types; keywords; identifier.

Computational way of thinking

Computational thinking means solving a problem in clear steps that a computer can follow. It has four habits:

- **Decomposition** — break a big problem into small parts, for example input, processing and output.
- **Pattern recognition** — spot repeated work, such as adding marks for many students in the same way.
- **Abstraction** — keep only the details that matter and ignore the rest.

- **Algorithm design** — write the exact step-by-step solution.

Example: to find the largest of three numbers, first take the numbers, then compare them one by one, then print the largest. A program is simply this thinking written in a language like C, so the computer can repeat it quickly and without tiring.

In viva, link each habit to your own program: decomposition splits a marksheet program into reading, totalling and grading parts, which can then be written and tested one by one.

Flowchart and algorithm

An **algorithm** is a finite, step-by-step procedure to solve a problem. It must be clear, ordered, and must stop after giving an output. Example — add two numbers: Step 1 Start, Step 2 Read a and b, Step 3 $\text{sum} = a + b$, Step 4 Print sum, Step 5 Stop.

A **flowchart** is the picture form of an algorithm, using standard symbols:

- **Oval** — start / stop.
- **Parallelogram** — input / output.
- **Rectangle** — processing or calculation.
- **Diamond** — decision, with Yes and No paths.
- **Arrows** — direction of flow.

Flowcharts make logic easy to check before coding, but they become messy for very large programs.

A good algorithm is written in simple numbered steps in ordinary language before any C code is typed, so that the flowchart and the program both follow exactly the same logic.

Variable, constant and representation

A **variable** is a named memory location whose value can change during the program, for example **int marks = 80;** Here marks is the name, int is its type, and 80 is the stored value. Every variable must be declared before use so the compiler reserves memory and knows its type.

A **constant** is a value that does not change. It may be a literal such as 10, 3.14 or 'A', or a named constant written with **const**, for example **const float PI = 3.14;**

Representation means how data is stored inside memory: whole numbers in binary form, decimal numbers in floating-point form, and characters through codes such as ASCII, where 'A' is 65. Correct representation decides the range and accuracy of a value.

Data types

A **data type** tells the compiler what kind of value a variable will hold, how much memory it needs, and which operations are allowed on it. The main primary types in C are:

- **int** — whole numbers, typically 4 bytes. Example: `int age = 20;`

- **float** — decimal numbers, typically 4 bytes. Example: float price = 99.5f;
- **double** — decimal numbers with more precision, typically 8 bytes.
- **char** — a single character, 1 byte. Example: char grade = 'A';
- **void** — no value; mainly used for functions that return nothing.

Choosing the right type saves memory and prevents errors — never store a price in int if paise or decimals matter, and never use double where a simple int is enough.

Keywords and identifier

Keywords are reserved words with a fixed meaning in C. They cannot be used as variable names. C has 32 standard keywords, including int, float, char, if, else, for, while, do, switch, case, break, continue, return, void, const, struct and sizeof. The compiler already knows what each keyword does.

An **identifier** is a name given by the programmer to a variable, function or array, for example total_marks or sum2. Rules for identifiers:

- Only letters, digits and underscore (_) are allowed.
- The first character must be a letter or underscore, never a digit.
- No blank space or special symbol is allowed.
- A keyword cannot be used, and names are case-sensitive — Sum and sum are different.

Choose meaningful identifier names such as total_marks instead of x1, because clear names make a program easy to read, easy to explain in viva, and easy to correct later.

Likely question: Define an algorithm and draw a flowchart to find the largest of three numbers. Also state the rules for naming an identifier in C.

Unit 2 — Operators & Expression (3 Periods, 20%)

Topics: Arithmetic, relational, logical and bitwise operators; input and output; expressions; data type casting.

Arithmetic and relational operators

Arithmetic operators do calculations: + addition, - subtraction, * multiplication, / division and % modulus, which gives the remainder. Remember that 7 / 2 is 3 with integer operands, but 7.0 / 2 is 3.5; 7 % 2 is 1. Increment ++ and decrement -- add or subtract 1; written before a variable they change it before use, written after they change it after use.

Relational operators compare two values and give true (1) or false (0): > greater than, < less than, >=, <=, == equal to and != not equal to. A common mistake is writing = (assignment) where == (comparison) is needed inside a condition.

Relational results are mostly used inside if conditions and loop tests, so practise writing full conditions such as `(a > b && b > c)` rather than comparing values in words.

Logical and bitwise operators

Logical operators join conditions: **&&** AND gives true only when both sides are true, **||** OR gives true when at least one side is true, and **!** NOT reverses the result. Example: `(age >= 18 && marks >= 40)` is true only when both conditions hold.

Bitwise operators work on the individual bits of integers: **&** AND, **|** OR, **^** XOR, **~** complement, **<<** left shift and **>>** right shift. For example, `5 & 3` is 1 because `101 & 011 = 001`, and `5 << 1` is 10 because the bits move one place left. Do not confuse **&** with **&&** or **|** with **||** — bitwise operators act on bits, logical operators act on whole conditions.

Shift operators are a quick way to multiply or divide by powers of two for integers — shifting left by one doubles the value, and shifting right by one halves it.

Input and output

C takes input mainly with **scanf()** and gives output with **printf()**, both declared in `stdio.h`. The first line of most programs is:

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

Output example:

```
printf("Marks = %d", marks);
```

Input example:

```
scanf("%d", &marks);
```

- **%d** for int, **%f** for float, **%c** for char, **%s** for a string, **%lf** for double.
- In `scanf`, **&** before a variable gives its address, so the value can be stored in it.
- **\n** inside `printf` moves the output to a new line.

Format specifiers in `printf` and `scanf` must match the data type, otherwise the result is wrong or unpredictable.

Two helper functions work with single characters: **getchar()** reads one character and **putchar()** writes one character. Clear prompts before `scanf`, such as `printf("Enter marks: ")`, make lab programs easy for the examiner to test.

Expressions

An **expression** is a combination of operands and operators that produces one value, for example `a + b * 2` or `(x > y)`. When several operators appear together, C follows **precedence** (which operator is done first) and **associativity** (the direction, usually left to right, for operators of equal rank).

Precedence order to remember, high to low:

- Brackets `( )` first.

- Unary ++, --, ! and - (minus sign).
- *, /, %
- +, -
- Relational <, >, <=, >=
- ==, !=
- && then ||
- Assignment = last.

So in $2 + 3 * 4$, multiplication is done first and the answer is 14, not 20. When in doubt, add brackets — they never harm and always make the order clear.

Data type casting

Type casting means converting a value from one data type to another. It happens in two ways:

- **Implicit casting (automatic)** — the compiler converts a smaller type to a larger type safely, for example an int used in a float calculation becomes float. In $7 / 2.0$ the 7 is promoted, so the result is 3.5.
- **Explicit casting (manual)** — the programmer writes the target type in brackets: $(float)7 / 2$ gives 3.5, while $(int)3.9$ gives 3 because the decimal part is cut off, not rounded.

Casting is useful in averages: $total / 5$ with both int gives a whole number only, but $(float)total / 5$ gives the correct decimal average. Cast carefully — converting a big value into a small type can lose data.

Likely question: Explain arithmetic, relational, logical and bitwise operators with one example each, and evaluate an expression using operator precedence.

Unit 3 — Conditional & Looping Statements (3 Periods, 20%)

Topics: if, if-else, if-else ladder, switch-case, break and continue; loops — for, while, do-while and nested loops.

if, if-else and if-else ladder

Conditional statements let a program choose what to do by testing a condition, which is either true or false.

- **if** — runs its block only when the condition is true.
if (marks >= 40) printf("Pass");
- **if-else** — runs one block when true and the other when false.
if (n % 2 == 0) printf("Even"); else printf("Odd");

- **if-else ladder** — tests a chain of conditions from top to bottom; the first true block runs and the rest are skipped. It suits grade ranges: 90 and above, then 75 and above, then 40 and above, else fail.

Conditions are usually built with relational and logical operators. Always use braces { } for multi-line blocks, and remember == for comparison, not =.

Order matters in a ladder: always test the highest range first, because once a condition becomes true the remaining else-if parts are never checked at all.

switch-case, break and continue

switch-case selects one block from many fixed choices by matching a value, usually an int or char. Skeleton:

```
switch (choice) {  
case 1: printf("Add"); break;  
case 2: printf("Subtract"); break;  
default: printf("Wrong choice"); }
```

The **break** statement ends the switch (or a loop) at once. Without break, execution **falls through** into the next case, which is a common error. The **default** case runs when no case matches.

Inside loops, break and continue differ: **break** leaves the loop completely, while **continue** skips only the rest of the current round and starts the next round. For example, continue is used to skip printing the number 5 in a 1 to 10 counting loop while still printing all other numbers.

Use switch when the choices are exact values, such as menu numbers 1, 2, 3 or characters '+', '-'; use an if-else ladder when the choices are ranges, such as marks between 60 and 75.

Loops — for, while and do-while

A **loop** repeats a block until a condition fails. Every loop needs initialisation, a test, and an update, otherwise it may run forever.

- **for** — best when the number of rounds is known.
for (i = 1; i <= 5; i++) printf("%d ", i);
- **while** — entry-controlled; the condition is checked first, so the body may run zero times.
while (i <= 5) { printf("%d ", i); i++; }
- **do-while** — exit-controlled; the body runs at least once because the condition is checked at the end, and the statement ends with a semicolon.
do { printf("%d ", i); i++; } while (i <= 5);

Typical viva programs: factorial, table of a number, and the Fibonacci series are all loop programs.

Nested loops

A loop written inside another loop is a **nested loop**. The outer loop runs once, then the inner loop runs fully, then the outer loop moves to its next round. If the outer loop runs m times and the inner loop n times, the innermost statements run $m \times n$ times.

Nested loops are the standard tool for:

- **Patterns** — the outer loop controls rows, the inner loop controls stars or numbers in each row.
- **Tables** — printing multiplication tables from 1 to 10 together.
- **Matrix work** — reading, printing and adding two-dimensional arrays, where rows and columns need two counters.

Keep counters separate for each level, for example i for rows and j for columns, and place the newline `printf` at the end of the outer loop body, not inside the inner loop.

Likely question: Differentiate between while and do-while loops with syntax and an example, and write a program to print a star pattern using nested loops.

Unit 4 — Arrays & String (3 Periods, 20%)

Topics: Introduction to arrays; array declaration; length of array; manipulating array elements; single and multi-dimensional arrays; arrays of characters; introduction to strings; string declaration and definition; string functions (`strlen`, `strcpy`, `strcmp`).

Introduction to arrays, declaration and length

An **array** is a collection of elements of the **same data type** stored in continuous memory locations under one name. Instead of writing `marks1`, `marks2` up to `marks50`, we write one array.

Declaration:

```
int marks[5];
```

This reserves five `int` locations, indexed from **0 to 4**. Elements are used as `marks[0]`, `marks[1]` and so on. Values can be given at declaration itself: `int marks[5] = {70, 85, 60, 90, 75};`

The **length** (size) of an array is the number of elements it can hold. In bytes, total size = size of one element $\times$ number of elements, so the array above occupies $4 \times 5 = 20$ bytes. The count of elements can be found as `sizeof(marks) / sizeof(marks[0])`. Crossing the last index is an error C does not check automatically.

Manipulating array elements; single and multi-dimensional arrays

Manipulating array elements means reading, printing, searching, adding or changing them, usually with a loop: `for (i = 0; i < 5; i++) sum = sum + marks[i];` The same loop pattern prints all elements, finds the largest element, or copies one array into another, element by element.

- **Single-dimensional (1-D) array** — one row of elements, such as marks of one class: `int marks[50];`
- **Multi-dimensional array** — an array of arrays. A two-dimensional array is like a table with rows and columns: `int matrix[3][3];` Its elements are reached as `matrix[row][column]`, for example `matrix[0][2]`, and it is handled with nested loops.

Matrix addition, transpose and printing a 3×3 matrix are the standard lab programs for this topic.

For searching, compare each element with the target value inside the loop and report its position when a match is found; if the loop finishes without a match, the element is not present.

Arrays of characters; introduction to strings; declaration and definition

In C, a **string** is an array of characters ending with the special null character `'\0'`, which marks the end of the string. Without `'\0'`, C cannot know where the string stops.

Declaration and definition:

```
char name[10] = "Student";
```

Here `name` holds `S-t-u-d-e-n-t` plus `'\0'` — eight locations, though only seven letters are visible. A string can also be declared as `char name[] = "Student";` and the compiler counts the size itself.

- A single character uses single quotes: `'A'`; a string uses double quotes: `"A"` (which is `'A'` plus `'\0'`).
- Strings are read with `scanf("%s", name)` — no `&` is needed because the array name itself gives the starting address — but `%s` stops at a space.
- Individual characters can still be changed like array elements, for example `name[0] = 's';`

String functions — `strlen`, `strcpy`, `strcmp`

Ready-made string functions are declared in **`string.h`**, so the program begins with `#include <string.h>`. The three functions in the syllabus are:

- **`strlen(s)`** — returns the length of string `s`, not counting `'\0'`. Signature: `int strlen(char s[]);` Example: `strlen("Student")` is 7.
- **`strcpy(dest, src)`** — copies the source string into the destination. Signature: `char *strcpy(char dest[], char src[]);` Example: `strcpy(city, "Varanasi");` The destination must be large enough to hold the copy.
- **`strcmp(s1, s2)`** — compares two strings character by character. Signature: `int strcmp(char s1[], char s2[]);` It returns 0 when both strings are equal, a negative value when `s1` comes first alphabetically, and a positive value when `s1` comes later.

Never compare strings with `==` — that compares addresses, not the characters. Always use `strcmp` for equality.

Likely question: Define an array and explain its declaration. Write a program to find the largest element in an array, and explain `strlen`, `strcpy` and `strcmp` with examples.

Unit 5 — Function & Pointer (3 Periods, 20%)

Topics: Introduction to functions; global and local variables; function declaration, call and return; types of functions; standard functions; introduction to pointers; parameters and parameter passing; call by value / reference; recursive function; I/O and formatting.

Introduction to functions; global and local variables

A **function** is a self-contained block of code that performs one task and can be used again and again. Dividing a program into functions makes it shorter, easier to test, and easier to correct — an error is fixed in one place only. `main()` is itself a function from where execution starts.

- **Local variables** — declared inside a function or block. They exist only there, are created when the function runs and destroyed when it ends, and cannot be used by other functions. Parameters are also local.
- **Global variables** — declared outside all functions, usually near the top. Every function can read and change them, and they live for the whole program.

Prefer local variables in practice problems; overuse of globals makes programs hard to trace because any function can change the value anywhere.

Function declaration, call and return; types and standard functions

A function is used in three steps:

- **Declaration (prototype)** — tells the compiler the name, return type and parameters: `int add(int, int);`
- **Definition** — the actual body: `int add(int a, int b) { return a + b; }`
- **Call** — using it in main: `result = add(4, 5);` The **return** statement sends the answer back to the caller; a void function returns nothing.

Types of functions: library (standard) functions and user-defined functions; also grouped by whether they take arguments and whether they return a value — all four combinations are possible. **Standard functions** are ready-made, such as `printf` and `scanf` (`stdio.h`), `strlen` and `strcpy` (`string.h`), and `sqrt` and `pow` (`math.h`) — just include the header and call them.

Writing the prototype before main is compulsory in practice when the definition is placed after main, otherwise the compiler does not know the function's return type or its parameters at the call.

Introduction to pointers

A **pointer** is a variable that stores the **memory address** of another variable instead of a direct value.

Declaration and use:

```
int x = 10;
int *p; /* p is a pointer to int */
p = &x; /* & gives the address of x */
printf("%d", *p); /* *p gives the value at that address: 10 */
```

- **& (address-of operator)** — finds where a variable is stored.
- *** (dereference operator)** — reaches the value stored at the address.
- Changing `*p` changes `x` itself, because both point to the same location.
- The pointer type must match the variable type — an int pointer for an int variable.

Pointers are the base of call by reference, arrays and dynamic work, so this topic is a favourite in viva.

Parameters and parameter passing — call by value / reference

Parameters are the values passed to a function. The values written in the call are **actual parameters**; the variables that receive them in the definition are **formal parameters**.

Passing can be done in two ways:

- **Call by value** — a copy of the value is passed. Changes inside the function do not affect the original variable. Simple and safe, but it cannot return changed values through the parameters.
- **Call by reference** — the address is passed using pointers. Changes inside the function change the original variable itself.

The classic proof is swapping two numbers: with call by value the swap stays inside the function and is lost, while with pointers — `swap(&a, &b)` with `*temp = *x` style steps — the original values are really exchanged. Use call by reference when a function must modify the caller's variables.

Recursive function; I/O and formatting

A **recursive function** calls itself with a smaller input until a stopping (base) condition is reached. Every recursion needs two parts: a **base case** that stops the calls, and a **recursive case** that reduces the problem. Factorial is the standard example:

```
int fact(int n) { if (n == 0) return 1; else return n * fact(n - 1); }
```

So $\text{fact}(4) = 4 \times 3 \times 2 \times 1 = 24$. Without a base case the calls never stop and the program crashes.

I/O and formatting: `printf` can control how output looks — field width such as `%5d` (right-aligned in 5 spaces), decimal places such as `%.2f` for money values, and `\t` for tab gaps. Neat

formatting matters in lab output, for example printing a proper table of marks or a matrix in rows and columns.

Likely question: Differentiate between call by value and call by reference with a swapping program, and write a recursive function to find the factorial of a number.

Quick Revision — One Line Each

- Computational thinking = decompose, spot patterns, abstract, then write the algorithm.
- Algorithm is the written steps; flowchart is its picture — oval start/stop, diamond decision.
- Variable changes, constant does not; identifier names must start with a letter or underscore.
- int 4 bytes, float 4, double 8, char 1 — and char stores a code such as 'A' = 65.
- Precedence: brackets, then *, /, %, then +, -, then comparisons, then &&, ||, assignment last.
- $7 / 2 = 3$ but $7.0 / 2 = 3.5$; $(\text{int})3.9 = 3$ — casting cuts, it does not round.
- break leaves a loop or switch; continue only skips the current round.
- do-while runs at least once; while and for may run zero times.
- Array index starts at 0; marks[5] has valid indexes 0 to 4 only.
- A string ends with '\0'; use strcmp for equality, never ==.
- *p gives the value at the address; &x gives the address of x.
- Call by value passes a copy; call by reference (pointers) changes the original.
- Recursion needs a base case — factorial: $\text{fact}(0) = 1$, $\text{fact}(n) = n \times \text{fact}(n-1)$.

KAMALDNP • kamaldnp.com • support@kamaldnp.com • +91 8318013442 • Varanasi, UP
Prepared by KAMALDNP Academy from the official BTEUP 2026-27 syllabus topics. Short explanations are the author's own plain-English summaries, not syllabus text. Please confirm with your own polytechnic for any local changes.