GeeksforGeeks

GATE फर्स्ट

CSE

C-PROGRAMMING

SHORT NOTES

ENROLL NOW

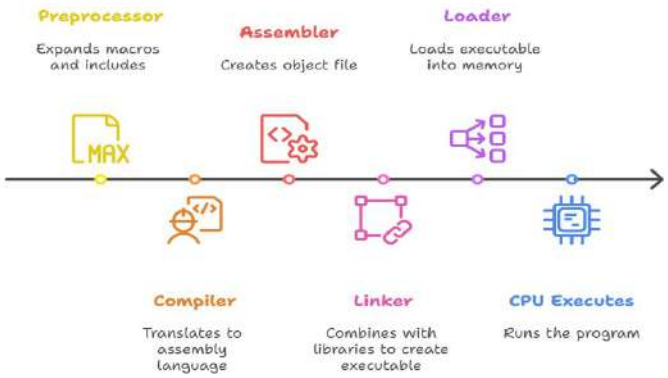
TO EXCEL IN GATE
AND ACHIEVE YOUR DREAM IIT OR PSU!

ENROLL NOW

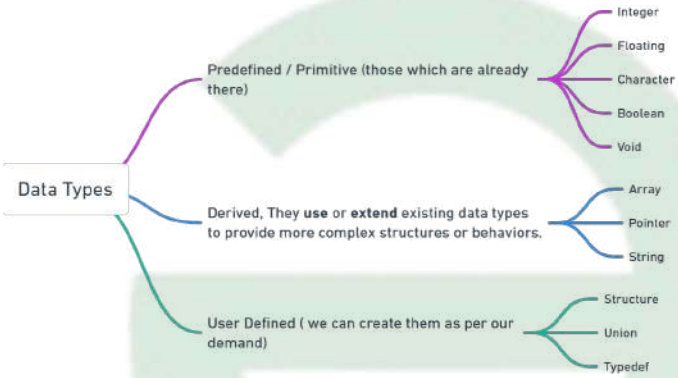
C-PROGRAMMING

GATE फर्रे

Flow chart of the C program



Data Types



Signed vs Unsigned

- Signed (default): Can store both positive and negative numbers. $-(2^{n-1}) to\ 2^{n-1} - 1$
- Unsigned: Can store only positive numbers, but with double the max range. $0\ to\ 2^{n} - 1$

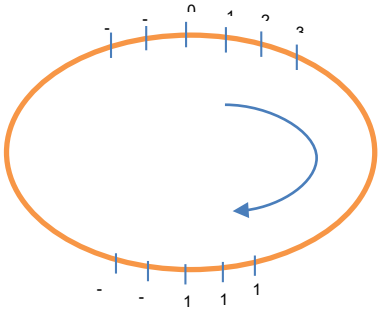
Format Specifier

Format Specifier	Data Type	Used For
%d or %i	int	Signed integer
%u	unsigned int	Unsigned integer
%f	float/double	Decimal floating-point number
%c	char	Single character
%s	char [] (string)	Null-terminated string
%p	void*	Pointer address
%x	int	Unsigned hexadecimal (lowercase)
%o	int	Unsigned octal number

\n: use to give the break in the line.
\t: move the cursor to the next available frame in the line.

Cyclic Property

This property is observed when we try to store a value larger than the range of a given data type.
For example, a char (in signed form) has a range from –128 to 127. If we try to assign a value like 129 to a char, the compiler does not show an error. Instead, it uses the cyclic (wrap-around) behaviour and stores a corresponding value within the valid range.
This happens due to integer overflow, and the excess value wraps around using modulo arithmetic based on the size of the data type.



When we try to store 129 in a char variable, it causes an overflow because the value exceeds the maximum limit of 127 (for signed char).
Since it goes beyond the range, it wraps around using the cyclic property.
Let's understand it step-by-step:

- If c = 128, it overflows by 1 step and wraps to -128 = - (129 -1)
- If c = 129, it overflows by 2, so the stored value becomes -127 = -(129-2)
- And so on...
  This is not something you need to memorize — just practice these kinds of questions more to understand the behaviour intuitively.
- When value is +ve overflow then clockwise in the circle
- When the value is -ve overflow then move anticlockwise in the circle.

Char c
(8bit)–2^{8-1} *to* $2^{8-1}-1$
Range =–128 to127

Example:
Char c = 129;
Printf ("%d", c);

This is 129, it is positive overflow. So, we need to move in the clockwise direction by 2 unit which is -127.

⚠ **Note:** This cyclic property does not apply to float and double types.
These are floating-point numbers, and they are stored in memory using the IEEE 754 format.
Because of this, overflow in floating-point types behaves differently and does not wrap around.
Hence, avoid relying on cyclic behaviour for float or double.

Negative number will be stored in the 2's complement from in the memory.
If the msb is 1 it means the number is -ve and it is stored in the 2's complement form.
Range(2^{n-1} *to* $2^{n-1}-1$)

Operators

Assignment operator (=)

left value = right value
must be variable — expression / constant variable

The assignment operator = first assigns the value, then returns the assigned value.
So if (a = 1) means:

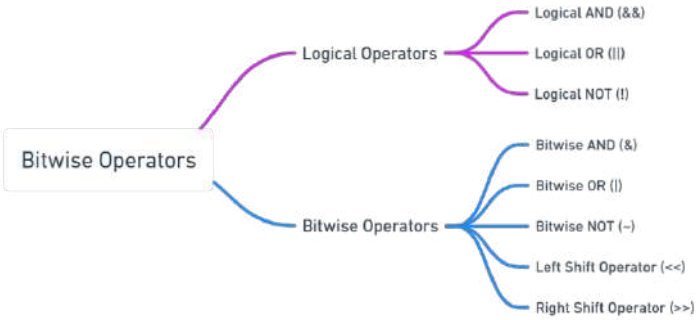
- First, 1 is assigned to a
- Then, the expression returns 1, which is true, so the if block runs

Modulus (%)

both operands must be int value types
a%b = remainder of a when divided by b.
Sign of the Result Follows Dividend (Left Operand)

5 % 3 = 2
-5 % 3 = -2
5 % -3 = 2
-5 % -3 = -2

Bitwise Operators



Logical AND (&&)

(Expression 1 && expression 2)
Returns:

- true (1) if both conditions are true
- false (0) if either condition is false

It has the short circuiting, means if the first operand output is 0 or false then second expression will not be evaluated.

Logical OR (| |)

(Expression 1 || expression 2)
Returns:

- false (0) if both conditions are false
- true (1) if either condition is true

It has the short circuiting, means if the first operand output is 1 or true then second expression will not be evaluated.

Bitwise And, OR

Bitwise And (|)

Result bit is 1 only if both bits are 1.
(5) 0101
(3) & 0011

= 0001

Bitwise OR (|)
Result bit is 1 if either of bits are 1.
(5) 0101
(3) & 0011
= 0111

Left Shift Operator (<<)

Syntax: x << n = x * (2^n)
How It Works

- Each bit shift to the left multiplies the number by 2.
- Zeros are added on the right side.
- Leftmost bits are discarded if overflow occurs.

Use Cases
Now we have the example: 1 << 3

- 1 = 0000 0001
- 1 << 3 → 0000 1000 = 8

Another method: $1*2^3 = 8$.

Right Shift Operator (>>)

Syntax: x >> n = x / (2^n)
Example: 8 >> 3

- 8 in binary: 0000 1000
- 8 >> 3 → 0000 0001 = 1

Another method:
8 / 2^3 = 8 / 8 = 1

Bitwise NOT Operator (~)

Operates bit-by-bit.
Converts each 1 to 0 and each 0 to 1.
~x = -(x+1)

~5

- 5 = 0000 0101
- ~5 = 1111 1010 = -6 (in two's complement)

Logical Not (!)

The **Logical NOT** operator is denoted by(!)

- It is a **unary operator** (works on a single operand).
- It **reverses the logical state** of its operand.

!0 = true
! true = false

Comma Operator (,)

it evaluates from left to right and discard, final value is right most

ex. int a= 9,8,7;
is **interpreted as**:
int a = 9;
8;
7;
// so the final value of a=5;

ex. int a= (9,8,7);
we apply the associativity of comma operator.

// so, the final value of a=9;

Increment (++) and Decrement (-)

Pre-Increment / Pre-Increment

Syntax: ++x, --x
We do the increment or decrement first then will use the value in the expression.
Ex. int x = 5;
int y = ++x; // x becomes 6, then y = 6

Syntax: x++, x--, First use the value and then perform the increment or decrement.
int x = 5;
int y = x++; // y = 5, then x becomes 6

Ternary Operator

Syntax: ++x, --x
We do the increment or decrement first then will use the value in the expression.
Ex. int x = 5;
int y = ++x; // x becomes 6, then y = 6

sizeof

is a compile-time operator used to determine the memory size (in bytes) of a data type or variable.
Returns size_t type (an unsigned integer).
Evaluated at compile time (except in Variable Length Arrays).
No function call — it's an operator, not a function.

If and else statements

Syntax: if**(expression)**
{statements}
else **{statements}**

There is no compulsion of the else block after if.
This means you can use if without using else.

```
if (exp) {
  statement;
} // valid syntax, there is no requirement of the else block
```

Only else block can't exist. if block is necessary for else.
You cannot **write else without if.**

```
else {
   // statement;
 } // invalid, compiler give the error of wrong syntax
```

If( )
Statement; **// expression is mandatory in the if block**
If(1) **–> this is perfect statements**
Note: **In C, if you do not use curly braces {} after an if statement, only the very next single statement is considered part of the if block.**

There should be no statement between the if and else block.
You must not insert any statements between if and else.

```
if (exp) {
   // statement1;
}
// statement2;
else {
   // statement3;
}
```

Result: **Compilation error — "else without if"**

GeeksforGeeks

GATE CSE BATCH

KEY HIGHLIGHTS:

- 300+ HOURS OF RECORDED CONTENT
- 900+ HOURS OF LIVE CONTENT
- SKILL ASSESSMENT CONTESTS
- 6 MONTHS OF 24/7 ONE-ON-ONE AI DOUBT ASSISTANCE
- SUPPORTING NOTES/DOCUMENTATION AND DPPS FOR EVERY LECTURE

COURSE COVERAGE:

- ENGINEERING MATHEMATICS
- GENERAL APTITUDE
- DISCRETE MATHEMATICS
- DIGITAL LOGIC
- COMPUTER ORGANIZATION AND ARCHITECTURE
- C PROGRAMMING
- DATA STRUCTURES
- ALGORITHMS
- THEORY OF COMPUTATION
- COMPILER DESIGN
- OPERATING SYSTEM
- DATABASE MANAGEMENT SYSTEM
- COMPUTER NETWORKS

LEARNING BENEFIT:

- GUIDANCE FROM EXPERT MENTORS
- COMPREHENSIVE GATE SYLLABUS COVERAGE
- EXCLUSIVE ACCESS TO E-STUDY MATERIALS
- ONLINE DOUBT-SOLVING WITH AI
- QUIZZES, DPPS AND PREVIOUS YEAR QUESTIONS SOLUTIONS

ENROLL NOW

TO EXCEL IN GATE
AND ACHIEVE YOUR DREAM IIT OR PSU!

ENROLL NOW

Pattern	Valid?	Explanation
if (exp) {statement;}	yes	Only if block, no else needed, but expression is mandatory in if block
If () statement;	no	Expression is missing
else {statement;}	no	Must follow an if block
if (exp) {statement;} statement; else {statement;}	no	Statement between if and else breaks the pair
if (exp) {statement;} else {statement;}	yes	Proper structure

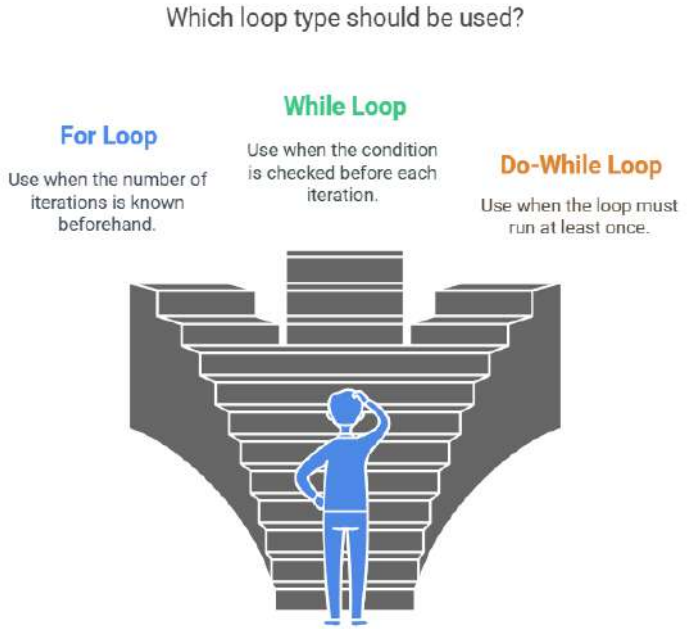
In a chain of if - else if - else if - else, only the first block whose condition is true will execute, even if other conditions are also true.

```
int x = 15;
if (x > 10)
    printf ("Condition 1: x > 10\n");
else if (x > 5)
    printf ("Condition 2: x > 5\n");
else if (x > 0)
    printf ("Condition 3: x > 0\n");
else
    printf("Condition 4: x <= 0\n");
```

There is all the blocks condition are true. But we check first condition which is if(x > 10) and it comes true then we will not execute to the next level, even if the next blocks like else if(x>5), else(x>0) are true.

Loops

For loops



Valid syntax:

```
for (exp1; exp2; exp3) {
 Statements;
}
```

for (;;) **this is valid for loop, will**

Flow: exp1 → exp2? → statements → exp3

Expression inside the for loop are optional.

But 2 semicolons are compulsory.

While

Valid syntax:

```
while(expression) {
Statements;
}
```

expression is not optional it is mandatory.

When to use

- **When number of iterations is not known in advance**

Do-while

it executes the loop body first, then checks the condition

How It Works

- Always runs once, even if the condition is initially false.
- After executing, it checks the condition.
- Continues if condition is true.

Expression is mandatory not optional.

Continue: **To skip the current iteration of a loop and go to the next iteration, used in loops, but in switch it will give**

break: **To exit a loop or switch statement immediately, skipping the remaining iterations or cases. Used in loops,**

Switch Statement

```
switch (expression) {
   case constant1:
      // code
      break;
...
   default:
      // code
}
```

- break is not a mandatory statement.
- The expression must evaluate to an integer, char, or Enum (not float, double, or string).
- Each case must use a constant value (no variables or ranges allowed).
- default is optional and executes when no cases match.
- The position of the default case doesn't matter.
- continue cannot be used inside a switch statement.
- If there is any statement between two case labels, it is ignored (not executed) unless one of the cases above falls through.
- switch is generally faster than if-else for fixed constant comparisons.
- If break is not used and one case matches, execution will continue through all subsequent cases, including default.

PRIORITY OPERATORS

Operators	Description	Associativity
(), [], . ,->	Function call, array subscript, struct	Left to Right
++, --, +, - , !, ~, *, &, (type), sizeof ()	Unary operators, type cast, dereference	Right to Left
*, /, %	Multiplication, division, modulus	Left to Right
+, -	Addition, subtraction	
<<, >>	Bitwise shift left/right	
< <=, > >=	Relational operators	
==, !=	Equality operators	
&	Bitwise AND	
^	Bitwise XOR	
|	Bitwise OR	
&&	Logical AND	
||	Logical OR	
?:	Ternary conditional	Right to Left
=	Assignment operators	
,	Comma (sequential evaluation)	Left to Right

Concept:
printf ("%d", 1 || printf ("I'll be rank 1") && 0);
In C:

- && has higher precedence than ||
- So, the expression is grouped like this:
  1 || (printf("I'll be rank 1") && 0)

But when we see the short-circuiting, (printf ("I'll be rank 1") && 0) will not execute.
Output: 1.

print in C

- **printf () is defined in the** <stdio.h> **header file.**
- **It** returns the number of characters **it prints.**
- **If an error occurs, it** returns a negative number **(usually -1).**
- Format specifiers **like %s (for strings), %d (for integers), etc., are used to** control what and how values are printed**.**

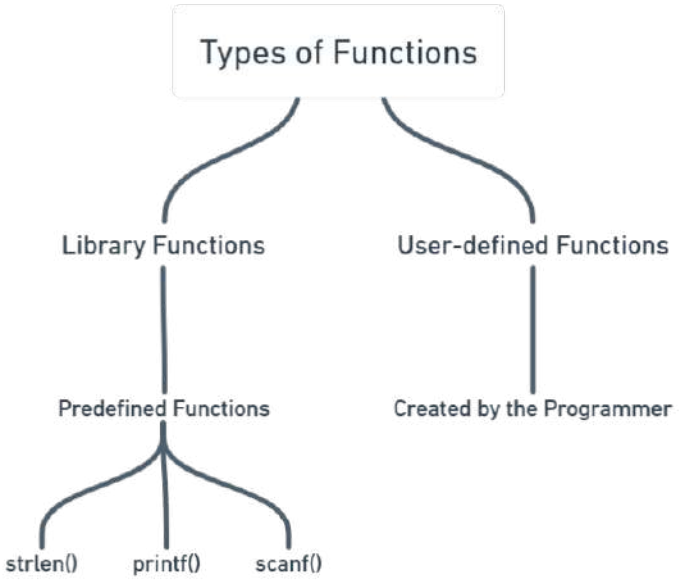
printf ("%s %d", "I'm the rank", 1);
output: I'm the rank 1
Always ensure that the number and type of format specifiers match the values you pass to printf (). Mismatches can lead to undefined behaviour or runtime errors.

Scanf in C

- **scanf () is defined in the** <stdio.h> **header file.**
- **It is used to** read input from the user**.**
- **scanf () returns the** number of input items **successfully assigned.**
- **If the input fails (like wrong format or EOF), it returns** 0 or EOF (-1)**.**
- **Format specifiers like %d, %f, %c, %s is used to** specify the type of input **expected.**
- Address-of operator & **is used to pass the memory address of variables (except for strings).**
- **Do not use &** with strings (character arrays), as the array name already represents the address.

Functions

A **function** is a block of code that performs a specific task. It promotes **code reusability** and **modular programming**.



Syntax

```
return_type function_name(parameter_list) {
   // body of function
}
```

Prototype: **A function prototype declares a function's name, return type, and parameters to the compiler before its definition, enabling type checking and early function calls.**

int add (int, int); **// Function prototype**

Note:

- **If you** do not **specify a return type, the default return type is considered as int.**
- **If the** definition or call mismatches **the prototype, the compiler throws an error.**

Definition: **it contains the code what the function does.**

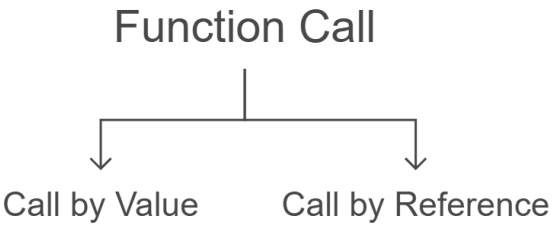
int add (int a, int b) **// Function definition**

{ ….. contains what the function does}

Things to Include in Function Definition:

1. **Return Type – What type of value the function returns (int, float, void, etc.)**
2. **Function Name – The name you'll use to call the function**
3. **Parameter List – Input values the function receives (can be empty)**
4. **Function Body – The block of code that performs the task**
5. Return Statement – (if not void) To send the result back to the caller

Function calling: function_name (arguments);



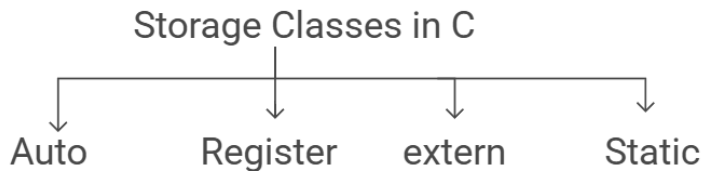
Call by Value

Formal arguments are the parameters listed in the **function definition**. When a function is called, **copies** of actual values are passed to them.

Changes made to formal arguments do not affect **actual arguments.**
This is the default behaviour **in C.**
Separate memory is allocated for the formal parameters.

Referenced Parameter Passing

In this method, instead of sending a copy, we send the **address of the variable** to the function using pointers. This allows the function to **modify the original value**.
Any change made inside the function affects **the original variable.**
This is done using pointers **in C.**
Both actual and formal parameters point to the same memory**.**



Storage classes define the **scope**, **lifetime**, **visibility**, and **memory location** of variables.

auto
- Default **for local variables.**
- **Scope: Inside block/function.**
- **Lifetime: Till the block ends.**

auto int a = 10; **// usually just written as int a = 10;**

register
1. Stored in CPU registers (if available) for faster access.
2. Cannot get the address using & register int counter;
   register int counter;

static
- Retains value between function calls.
- Scope: Local to block in which it is declared.
- Lifetime: Entire program.
  Static int count = 0;

GeeksforGeeks

GATE CSE BATCH

KEY HIGHLIGHTS:

- 300+ HOURS OF RECORDED CONTENT
- 900+ HOURS OF LIVE CONTENT
- SKILL ASSESSMENT CONTESTS
- 6 MONTHS OF 24/7 ONE-ON-ONE AI DOUBT ASSISTANCE
- SUPPORTING NOTES/DOCUMENTATION AND DPPS FOR EVERY LECTURE

COURSE COVERAGE:

- ENGINEERING MATHEMATICS
- GENERAL APTITUDE
- DISCRETE MATHEMATICS
- DIGITAL LOGIC
- COMPUTER ORGANIZATION AND ARCHITECTURE
- C PROGRAMMING
- DATA STRUCTURES
- ALGORITHMS
- THEORY OF COMPUTATION
- COMPILER DESIGN
- OPERATING SYSTEM
- DATABASE MANAGEMENT SYSTEM
- COMPUTER NETWORKS

LEARNING BENEFIT:

- GUIDANCE FROM EXPERT MENTORS
- COMPREHENSIVE GATE SYLLABUS COVERAGE
- EXCLUSIVE ACCESS TO E-STUDY MATERIALS
- ONLINE DOUBT-SOLVING WITH AI
- QUIZZES, DPPS AND PREVIOUS YEAR QUESTIONS SOLUTIONS

ENROLL NOW

TO EXCEL IN GATE
AND ACHIEVE YOUR DREAM IIT OR PSU!

ENROLL NOW

extern

- **Declares a variable defined in another file.**
- **Used for** global sharing**.**
  extern int x; // Defined elsewhere
- **If a variable is used before it is defined, you can declare it using** extern.
- **If a variable is used before it is defined, you can declare it using** extern.

```
extern int count;
int main () {
   printf ("%d",
count);
   return 0;
}
int count = 5; //
Defined later
```

1. You **cannot** initialize an extern variable during declaration.

extern int x = 10; // Error

Storage	Scope	Lifetime	Stored In	Keyword	Default Initial Value
auto	Block (local)	Till block ends	Stack	auto (default)	Garbage value (undefined)
register	Block (local)	Till block ends	CPU Register (if available)	register	Garbage value (undefined)
static	Block / File	Entire program	Data Segment	static	0 (zero)
extern	Global (across files)	Entire program	Data Segment	extern	0 (if defined globally without initialization)
global (no keyword)	Global (entire program)	Entire program	Data Segment	—	0

Array

- Array name represents the address of its first element.

int arr [5];
printf ("%p", arr); // Prints address of arr [0]);
arr means &arr[0]

- Array name is constant, so it can't be the left value.

arr = value; → Invalid

- Array size must be a constant or fixed expression (at compile time).

int arr[10]; valid
int size; int arr[size]; invalid

- variable can't be used as a size in the array.

If an array has n dimensions, and:

- You use all n dimensions → you're accessing an element.
- You use fewer than n dimensions → you're referring to an address (sub-array).

int a [2][3];
a [1][2]; // Element
a [1]; // Address of &a[1][0]
a; // Address of &a[0][0]

- Initialization at Declaration → Size is Optional (for first dimension)

If you initialize the array during declaration, the first dimension's size can be omitted — the compiler will count it automatically.

int arr [] = {1, 2, 3}; // Size = 3 (automatically)
int arr [3] = {1, 2, 3}; // Also OK

- Multidimensional Arrays → Only First Dimension Can Be Omitted

In multi-dimensional arrays:

- You can omit the first dimension if initializing.
- All other dimensions must be specified.

a[i] = *(a + i) = *(i + a) = i[a]
But remember: this is valid only for access, not for declaration.
i[a] is not valid during declaration — it will result in an error.
Also, expressions like:
a++, a--, --a, ++a are invalid if a is an array, because array names cannot be used as left-hand values.

• **Pointers** are variables that **store the address** of another variable or item.

• You can also have **pointers to pointers**, and even more levels (multiple indirection).

• We **dereference** a pointer using the * operator.

int variable; // Normal integer variable

int *ptr; // Declaration of a pointer to int

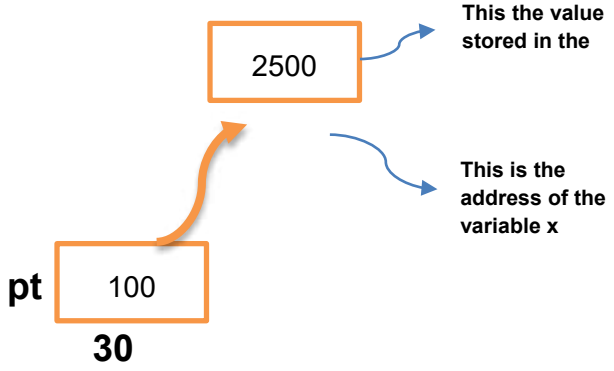
ptr = &variable; // Storing the address of variable in pointer

// Or in a single line:

int *ptr = &variable;

It Means:

- int *ptr; → Declares ptr as a pointer to int



We cannot perform arbitrary pointer arithmetic between unrelated pointers, but **pointer - pointer** is valid when both pointers point to elements of the **same array**.
In this case, the result is the **difference in element positions**, not in bytes, because the compiler automatically divides the address difference by the size of the data type.

```
int a[10];
int *p = &a[7];
int *q = &a[2];
int diff = p - q;
```

difference = (address at p - address at q) / sizeof(int)

Types of Pointers

Null Pointer

Points to nothing. Used for safety.

```
int *p = NULL;
```

Dangling Pointer

Points to memory that has been freed or is out of scope.

Wild Pointer

Uninitialized pointer that points to a random memory location.

Void Pointer (Generic Pointer)

Can point to any data type. Needs to be typecasted before dereferencing.

```
void *ptr;
int *const p = &x; → constant pointer (address can't change).
const int *p = &x; → pointer to constant (value can't change).
const int *const p = &x; → both value and address are constant.
```

Pointer Arithmetic

```
int a[5] = {1,2,3,4,5};
int *p = a;
p++; // moves to next integer (adds sizeof(int))
```

malloc(), calloc(), realloc(), and free() are used with pointers for dynamic memory.

```
int *p = (int *)malloc(sizeof(int) * 5);
free(p);
```

Dynamic memory allocation

malloc () – Memory Allocation
Definition:

malloc (memory allocation) is used to dynamically allocate a single block of memory of a specified size (in bytes). It does not initialize the memory—it contains garbage values.

```
int *ptr = (int*) malloc (5 * sizeof(int));
```

calloc () – **Contiguous Allocation**
calloc (contiguous allocation) allocates memory for an **array of elements**, initializes all bytes to **zero**.

```
void* calloc (size_t num_elements, size_t element_size);
```

realloc () – Reallocation

is used to resize a previously allocated memory block (from malloc or calloc). The contents are preserved up to the minimum of the old and new sizes.

```
void* realloc (void* ptr, size_t new_size);
```

All three functions return NULL if memory allocation fails.
Always free () the allocated memory when done, to avoid memory leaks.

String Representation in C

Strings in C can be represented in two ways:

1. Using a character array
2. Using a pointer to a string literal

Feature	Array	Pointer
Declaration	char str [] = "hello";	char *str = "hello";
Memory Location	Stored in read/write memory	Stored in read-only memory
Modifying individual characters	Allowed	Not allowed
Reassigning the whole string	Not allowed	Allowed
Array name used as l-value	Cannot be used as l-value	Pointer is an l-value

Summary of Key Points

- **Arrays** allow changing individual characters**, e.g., str [0] = 'H';**
- **Pointers** may point to string literals**, which are often** read-only
- **You** cannot assign a new string to an array **like: str = "new"; → invalid**
- **But with pointers: str = "new"; → valid**

Memory Behaviour for Strings in C

Read-Only Memory (String Literals)

- Strings stored using pointers, e.g., char *str = "hello";
- Stored in read-only (constant) memory
- Duplicate strings are not created — if two literals are the same, they share the same memory location

```
char *a = "hello";
char *b = "hello";
```

Both a and b point to the same memory address**.**
So

```
printf ("%p\n", a);
printf ("%p\n", b);
```

→ Both print the same address.
Also:

```
if (a == b)
    printf ("Same location\n");
    Output: Same location
```

Read / write memory

Strings stored using arrays, e.g., char str [] = "hello";
Stored in stack or heap, which is read/write memory
Duplicates are allowed — each array creates a separate copy of the string

Strings in C can be represented in two ways:

3. Using a character array
4. Using a pointer to a string literal

```
Example:
char a [] = "hello";
char b [] = "hello";
// a and b have different memory addresses
```

Note: "hello" is a string literal, which means it represents the address of the first character ('h') in memory.
printf ("%s", "hello" + 1);
output: ello
"hello" is a pointer to the first character ('h')
"hello" + 1 moves the pointer one character ahead — to 'e'
So printf starts printing from 'e'

string.h Functions in C

strlen

Get Length of String
unsigned int strlen(const char *str);

Returns the number of characters in the string (excluding the '\0' null terminator).
The parameter is const because strlen does not modify the string.

strcpy

char *strcpy (char *destination, const char *source);
Copies the string from source to destination including '\0'.
source is const because it should not be changed.
destination must be a writable array or memory block, i.e we should pass the array because the string will be in the r/w area.

strcat

Appends src string to the end of dest string, removing '\0' of dest and adding one at the end.

strcmp

Compares two strings character by character.
Returns:

- **0 if both strings are equal**
- **Positive or negative difference of ASCII values where mismatch occurs**

strcmp ("papu", "pake")
Compares: 'p' == 'p', 'a' == 'a', 'p' != 'k'
'p' - 'k' = 112 - 107 = 5
Return value: 5

Structure in C

A structure is a user-defined data type in C.
It allows you to group different types of variables under one name.
by default, structure contains the 0 or null values.
Useful for representing real-world entities (e.g., student, book,
employee, etc.).

```
struct Name {
    data_type member1;
    data_type member2;
    ...
};
struct Student s1; // variable of type struct Student
```

Feature	Explanation
User-defined	Unlike int, char, etc., created by the programmer
Holds multiple data types	e.g., int, float, char [], all together
Memory layout	All members stored contiguously in memory
Struct name is not an address	Unlike arrays, structure names are not pointers
Can contain nested structures	Yes

```
typedef struct Student {
    int roll;
    char name[50];
    float marks;
} Student;

Student s1;  // now no need to write 'struct' again
```

```
struct {
    int roll;
    char name[20];
} s1, s2;
```

This is anonymous and we can't create object of it but can use the variable s1 and s2 which we declared.

```
struct Student {
    int roll_num = 9; // Invalid
    char str [] = "Kunal"; // Invalid
};
```

This is incorrect**, because:**

- **A struct is only a** blueprint**, it** does not allocate memory **until an object (variable) is created.**
- **In C, you** cannot assign values **to structure members** inside the definition**.**
- **Memory is only allocated when you** create an object **like:**

struct Student s1;

Structure definition is just a template. You can't assign values inside it — you assign values only after creating a variable.
we access structure members using either the dot (.) operator or the arrow (->) operator
Union in C
A union is a special data type in C that allows storing different types of data in the same memory location.

```
union Data {
    int i;
    float f;
    char str[20];
};
```

All members share the same memory, and the size of the union is equal to the size of its largest member.

Scoping

- Scoping defines where a variable can be accessed in a program (its visibility or lifetime).

Static scoping
Variable scope is decided at compile time.
Local to the block will be preferred.

```
int x = 10;
void func () {
    int x = 20;
    printf ("%d", x); // Output: 20 (local x used)
}
```

Dynamic Scoping

Variable scope is decided at runtime.
The program searches the call stack to find the most recent variable definition.

```
x=5
foo() {
  print(x);
}
bar() {
  local x=10
  foo      # Output: 10 – because bar called foo, and
x=10 in bar
}
```

GeeksforGeeks

GATE CSE BATCH

KEY HIGHLIGHTS:

- 300+ HOURS OF RECORDED CONTENT
- 900+ HOURS OF LIVE CONTENT
- SKILL ASSESSMENT CONTESTS
- 6 MONTHS OF 24/7 ONE-ON-ONE AI DOUBT ASSISTANCE
- SUPPORTING NOTES/DOCUMENTATION AND DPPS FOR EVERY LECTURE

COURSE COVERAGE:

- ENGINEERING MATHEMATICS
- GENERAL APTITUDE
- DISCRETE MATHEMATICS
- DIGITAL LOGIC
- COMPUTER ORGANIZATION AND ARCHITECTURE
- C PROGRAMMING
- DATA STRUCTURES
- ALGORITHMS
- THEORY OF COMPUTATION
- COMPILER DESIGN
- OPERATING SYSTEM
- DATABASE MANAGEMENT SYSTEM
- COMPUTER NETWORKS

LEARNING BENEFIT:

- GUIDANCE FROM EXPERT MENTORS
- COMPREHENSIVE GATE SYLLABUS COVERAGE
- EXCLUSIVE ACCESS TO E-STUDY MATERIALS
- ONLINE DOUBT-SOLVING WITH AI
- QUIZZES, DPPS AND PREVIOUS YEAR QUESTIONS SOLUTIONS

ENROLL NOW

TO EXCEL IN GATE
AND ACHIEVE YOUR DREAM IIT OR PSU!

ENROLL NOW