Q1. What is a Preprocessor? Give Examples.

Preprocessor: Preprocessor is a program in C that processes our program before it is passed to the compiler.

When the source code is passed through the 'Preprocessor', it creates 'Expanded Source code' as per the preprocessor directives used in the source code begins with a # symbol.

For example: #include, #define etc.

Q2. In C what is the hashed line use for?

The hashed line in C, typically starting with a `#`, is used for preprocessor directives. These directives provide instructions to the compiler to preprocess the information before actual compilation starts. Common examples include `#include`, `#define`, and `#ifdef`.

Q3. How can a String can be converted to a number and vice versa in C (Write two code snippet)?

```
// Convert String to Number

#include <stdio.h>

#include <stdlib.h>

int main() {

   char str[] = "1234";

   int num = atoi(str);

   printf("String to Number: %d\n", num);

   return 0;

}
```

```
// Convert Number to String

#include <stdio.h>

int main() {

   int num = 1234;

   char str[10];

   sprintf(str, "%d", num);

   printf("Number to String: %s\n", str);

   return 0;

}
```

Q4. What is Recursion in C? Give an example.

Recursion in C is a process where a function calls itself directly or indirectly to solve a problem. It is used to solve problems that can be broken down into smaller, similar problems.

Example:

```
#include <stdio.h>

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

int main() {
    int num = 5;
    printf("Factorial of %d is %d\n", num, factorial(num));
    return 0;
}
```

In this example, the `factorial` function calls itself to calculate the factorial of a number.

Q5. What is iteration in C? Give an example.

Iteration in C refers to the process of executing a set of instructions repeatedly until a certain condition is met. This is typically achieved using loops.

An example of iteration in C is the `for` loop:

```
#include <stdio.h>

int main() {
    for (int i = 0; i < 5; i++) {
        printf("Iteration %d\n", i);
    }
    return 0;
}
```

In this example, the `for` loop iterates five times, printing the iteration number each time.

Q6. Why C does not support function overloading?

C does not support function overloading because it does not have the capability to differentiate functions solely based on their parameter types or number. Function names in C are not mangled, meaning they are not encoded with parameter information, which is necessary for distinguishing overloaded functions.

Q7. What is Storage class in C? Give example.

Storage classes in C define the scope, visibility, and lifetime of variables or functions within a C program. There are four types of storage classes in C:

1. Automatic (auto): The default storage class for local variables. They are created when the function is called and destroyed when the function exits.

2. External (extern): Used to declare a global variable or function that is accessible across multiple files.

3. Static: Retains the value of a variable between function calls. For global variables, it restricts the scope to the file in which they are declared.

4. Register: Suggests that the variable be stored in a CPU register instead of RAM for faster access.

Example:

```
#include <stdio.h>

// Global variable with external storage class

extern int globalVar;

void function() {

   // Local variable with automatic storage class

   auto int localVar = 5;

   printf("Local Variable: %d\n", localVar);

   // Static variable retains its value between function calls

   static int staticVar = 0;

   staticVar++;

   printf("Static Variable: %d\n", staticVar);

   // Register variable

   register int regVar = 10;

   printf("Register Variable: %d\n", regVar);}

int main() {

   function();

   function();}
```

In this example, `localVar` is an automatic variable, `staticVar` is a static variable, and `regVar` is a register variable. The `globalVar` is declared as an external variable, which would be defined elsewhere in the program.

Q8. Write the difference between global int and static int declaration in c.

Feature	Global int	Static int (at global scope)
Scope	Available throughout the entire program.	Limited to the file in which it is declared.
Lifetime	Exists for the entire duration of the program.	Exists for the entire duration of the program.
Default Value	Initialized to 0 if not explicitly initialized.	Initialized to 0 if not explicitly initialized.
Visibility	Externally visible by default (can be accessed in other files with extern).	Not externally visible (file-local linkage).
Access Modifiers	Can be explicitly made static or accessed via extern.	Always static and cannot be accessed with extern.
Use Case	Used when variables need to be shared across multiple files.	Used when the variable should be confined to a single file for encapsulation.

Q9. Define pointers in c.

Definition of Pointers in C

A pointer in C is a variable that stores the memory address of another variable. Pointers are a powerful feature in C that allow direct memory access and manipulation.

Types of Pointers in C

Type	Description
Null Pointer	A pointer that doesn't point to any memory location (`NULL` or `0`).
Void Pointer	A pointer that can point to any data type.
Wild Pointer	A pointer that is uninitialized and points to a random memory location (should be avoided).
Dangling Pointer	A pointer that points to memory that has been deallocated or is out of scope.
Function Pointer	A pointer that stores the address of a function and is used to invoke it.
Array Pointer	A pointer that points to the first element of an array.

Type	Description
Constant Pointer	A pointer whose value (memory address it points to) cannot be changed after initialization.
Pointer to Constant	A pointer that points to a value that cannot be modified through the pointer.
Double Pointer	A pointer that stores the address of another pointer.

Examples of Pointers in C

Basic Pointer Example

```
#include <stdio.h>
int main() {
    int num = 10;
    int *ptr = &num;  // Pointer to an integer
    printf("Value of num: %d\n", num);
    printf("Address of num: %p\n", &num);
    printf("Value stored in ptr (address of num): %p\n", ptr);
    printf("Value pointed to by ptr: %d\n", *ptr);
    return 0;
}
```

Null Pointer

```
int *ptr = NULL;  // Pointer initialized to NULL
if (ptr == NULL) {
    printf("Pointer is NULL\n");
}
```

Void Pointer

```
void *ptr;
int num = 10;
ptr = &num;
printf("Void pointer value: %p\n", ptr);
```

Pointer to Pointer (Double Pointer)

```
#include <stdio.h>
int main() {
    int num = 20;
    int *ptr = &num;  // Pointer to an integer
    int **dptr = &ptr;  // Pointer to a pointer
    printf("Value of num: %d\n", num);
    printf("Value through ptr: %d\n", *ptr);
    printf("Value through dptr: %d\n", **dptr);
    return 0;
}
```

Function Pointer

```
#include <stdio.h>
void display() {
    printf("Hello, Function Pointer!\n");
```

```
}
int main() {
    void (*funcPtr)() = &display;  // Function pointer
    funcPtr();  // Call the function using the pointer
    return 0;
}
```

Q10. Write the difference between constant char *p and char constant *p.

In C, there is a subtle difference between const char *p and char const *p. However, const char *p and char const *p are **equivalent**; the position of const doesn't matter. But, if you mean char *const p versus const char *p, they differ in **what is constant**. Here's the breakdown:

Declaration	Description
const char *p or char const *p	The pointer can point to different memory locations, but the value being pointed to cannot be modified.
char *const p	The pointer is constant and cannot point to a different memory location, but the value being pointed to can be modified.
const char *const p	Both the pointer and the value being pointed to are constant (neither can be changed).

Q11. What is a pointer to pointer in c? Give an example.

A pointer to a pointer in C is a form of multiple indirection or a chain of pointers. It is a pointer that points to another pointer, which in turn points to a data value. This is useful in scenarios where you need to modify the pointer itself, such as when dealing with dynamic memory allocation or passing pointers to functions.

Example:

```
#include <stdio.h>
int main() {
   int value = 10;
   int *ptr = &value;      // Pointer to an integer
   int **ptr_to_ptr = &ptr; // Pointer to a pointer
   printf("Value: %d\n", value);
   printf("Pointer to value: %p\n", (void*)ptr);
   printf("Pointer to pointer: %p\n", (void*)ptr_to_ptr);
   printf("Value via pointer to pointer: %d\n", **ptr_to_ptr);
   return 0;
}
```

In this example, `ptr` is a pointer to `value`, and `ptr_to_ptr` is a pointer to `ptr`. The value of `value` can be accessed using `**ptr_to_ptr`.

Q12. Why n++ faster than n+1?

n++ is generally faster than n+1 because n++ is a post-increment operation that directly increments the value of n in place, often using a single machine instruction. In contrast, n+1 creates a new value by adding 1 to n, which may involve additional steps such as creating a temporary variable to store the result.

Q13. What is type casting in C? Give an example.

Type casting in C is the process of converting a variable from one data type to another.

Example:

```
int a = 10;

double b = (double)a; // Type casting from int to double
```

Q14. What is macro in C? Give an example.

A macro in C is a fragment of code that is given a name. Whenever the name is used, it is replaced by the contents of the macro. Macros are defined using the `#define` directive.

Example:

```
#define PI 3.14

int main() {
   float area, radius = 5;
   area = PI * radius * radius;
   return 0;
}
```

Q15. Write the advantages of macro over function.

Automation: Macros can automate repetitive tasks, saving time and reducing errors.

Complexity: Macros can handle more complex tasks and workflows compared to functions.

User Interaction: Macros can interact with users through forms and dialogs, enhancing user experience.

Integration: Macros can integrate with other applications and systems, providing more flexibility.

Customization: Macros allow for greater customization and can be tailored to specific needs.

Efficiency: Macros can execute multiple functions and commands in a single action, improving efficiency.

Q16. What is enumeration in c? Give example.

Enumeration in C is a user-defined data type that consists of integral constants. It is used to assign names to the integral constants, which makes the code more readable and maintainable.

Example:

```
#include <stdio.h>

enum Days {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};

int main() {
   enum Days today;
   today = Wednesday;
   printf("Day %d\n", today);
   return 0;
}
```

In this example, `Days` is an enumeration with the days of the week as its members. Each member is assigned an integer value starting from 0.

Q17. What is r value and l value in C language?

In C language, an "r value" is a value that appears on the right side of an assignment operator and represents a temporary data value. An "l value" is an object with a specific memory location, meaning it has an address, and can appear on the left side of an assignment operator.

Example:

```
int x = 10; // 'x' is an l value, '10' is an r value
int y = x + 5; // 'x + 5' is an r value, 'y' is an l value
```

Q18. Define the memory allocation function & write the difference between calloc and malloc.

Definition of Memory Allocation Functions

In C, memory allocation functions dynamically allocate memory at runtime from the heap. The two commonly used functions are **malloc** and **calloc**, declared in the <stdlib.h> header.

malloc Function

- **Definition:** Allocates a single block of memory of a specified size in bytes and returns a pointer to the first byte of the block.
- **Syntax:** void *malloc(size_t size);
- **Key Points:**
    - Does not initialize the allocated memory; contains garbage values.
    - If allocation fails, it returns NULL.

calloc Function

- **Definition:** Allocates multiple blocks of memory, each of a specified size, and initializes all bytes to zero.
- **Syntax:** void *calloc(size_t num, size_t size);
- **Key Points:**
    - Initializes allocated memory to zero.
    - If allocation fails, it returns NULL.

Differences Between malloc and calloc

Feature	**malloc**	**calloc**
Syntax	malloc(size)	calloc(num, size)
Memory Initialization	Does not initialize memory (contains garbage values).	Initializes allocated memory to zero.
Arguments	Takes one argument: the total size of memory in bytes.	Takes two arguments: number of blocks and size of each block.
Use Case	Suitable for single block allocation.	Suitable for allocating multiple blocks of equal size.
Speed	Slightly faster as it doesn't initialize memory.	Slightly slower due to memory initialization.

Q19. What is Union? Write the difference between Structure and Union.

Union: A "Union" in programming languages like C is a user-defined data type that allows you to store different data types in the same memory location, meaning only one member of the union can hold a value at a time, unlike a "Structure" where each member has its own separate memory space and can store a value independently.

Here is a tabular representation of the differences between a Structure and a Union in programming:

Feature	**Structure**	**Union**
Definition	A structure is a user-defined data type that groups variables of different data types.	A union is a user-defined data type where all members share the same memory location.
Memory Allocation	Allocates memory for all members individually.	Allocates a single shared memory space large enough for the largest member.

Feature	Structure	Union
Memory Size	The size of a structure is the sum of the sizes of all its members (with padding, if applicable).	The size of a union is equal to the size of its largest member.
Access	All members of a structure can be accessed at any time independently.	Only one member can be accessed at a time. Writing to one member overwrites others.
Usage	Used when storing related data elements of different types is required.	Used when memory conservation is critical and only one member is used at a time.
Initialization	All members of a structure can be initialized separately.	Only the first member of a union can be initialized during declaration.
Use Case Example	To define a Student structure with fields like name, age, and grade.	To define a Data union that stores either an integer, a float, or a character (but only one at a time).

Q20. Define Call by Reference and Pass by Reference in function with example.

Call by Reference: Call by Reference is a method of passing the address (or reference) of a variable to a function. In this method, any changes made to the formal parameters in the function directly affect the actual parameters because the function operates on the same memory location.

Pass by Reference: Pass by Reference is a broader concept used in programming languages that allow the function to receive the reference of an argument (rather than its value). This approach is typically implemented using pointers or references, enabling direct modification of the original data.

Key Differences:
Although both terms are often used interchangeably, "Call by Reference" specifically refers to how arguments are passed during a function call, while "Pass by Reference" is the overall mechanism or concept enabling reference-based parameter passing.

Q21. What is memory leak? How we avoid them?

Memory Leak: A memory leak in C occurs when a program allocates memory using functions like malloc but fails to deallocate it using free when it's no longer needed, leading to a gradual buildup of unused memory and potential performance issues

Cause: Losing track of pointers to dynamically allocated memory, meaning the program can't access the memory to free it.

How to avoid:

- **Always free allocated memory:** Use free() to release memory when it's no longer needed.
- **Proper pointer management:** Maintain a clear understanding of where pointers are pointing and ensure they are updated when necessary.
- **Null checks:** Set pointers to NULL after freeing them to prevent accidental access to invalid memory.
- **Error handling:** Free memory correctly even when errors occur during program execution.
- **Use RAII (Resource Acquisition Is Initialization):** Design classes where memory is automatically managed during object creation and destruction

Q22. What is the difference between #include “” and #include <>?

Feature	#include ""	#include <>
Usage	Used to include user-defined or custom header files.	Used to include standard library header files.
Search Path	Searches for the file in the current directory first, then in the standard directories.	Searches for the file only in the standard system directories.
Primary Use Case	For including project-specific or locally developed headers.	For including standard C library headers like <stdio.h> or <stdlib.h>.
Example	#include "myheader.h"	#include <stdio.h>
Flexibility	Allows specifying paths explicitly, like "../headers/myheader.h".	Does not allow specifying custom paths.
Compiler Behaviour	Gives priority to the local directory before moving to the standard paths.	Looks directly in the system's standard include paths.

Key Points

- Use #include "" for your project's custom headers.
- Use #include <> for standard library headers.

Q23. What is Dangling pointer? Give example.

A dangling pointer is a pointer that does not point to a valid object of the appropriate type. It occurs when an object is deleted or deallocated, without modifying the value of the pointer, so the pointer still points to the memory location of the deallocated memory.

Example:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
   int *ptr = (int *)malloc(sizeof(int)); // Allocate memory
   *ptr = 42; // Assign a value
   free(ptr); // Deallocate memory
   // ptr is now a dangling pointer
   printf("%d\n", *ptr); // Undefined behaviour
   return 0;
}
```

In this example, after `free(ptr)`, `ptr` becomes a dangling pointer because it still holds the address of the deallocated memory. Accessing it leads to undefined behaviour.

Q24. How the dangling pointer differ from memory leak?

A dangling pointer is a pointer that continues to reference a memory location after the memory has been deallocated, leading to potential undefined behaviour if accessed. A memory leak, on the other hand, occurs when a program allocates memory but fails to release it back to the system, causing a gradual loss of available memory.

Q25. Explain near pointer, far pointer, huge pointer, null pointer in c with an example.

Near Pointer: A near pointer in C is a pointer that can access memory within the current segment. It is typically 16 bits in size and is used in 16-bit architectures.
Example:
char *nearPtr;

Far Pointer: A far pointer can access memory outside the current segment. It is 32 bits in size, consisting of a segment and an offset.
Example:
char far *farPtr;

Huge Pointer: A huge pointer is similar to a far pointer but can be normalized, meaning it can be adjusted to point to the same memory location regardless of segment boundaries.
Example:
char huge *hugePtr;

Null Pointer: A null pointer is a pointer that does not point to any valid memory location. It is often used to indicate that the pointer is not initialized.
Example:
char *nullPtr = NULL;

Q26. Explain which is better between #define and enum.

In C programming, `#define` and `enum` serve different purposes and have their own advantages:

1. #define:

- It is a preprocessor directive used to define macros or constants.
- It does not have a type, so it can be used for any kind of constant value.
- It is replaced by the preprocessor before compilation, which means it does not consume memory.
- It is simple and straightforward for defining constants.

2. enum:

- It is a user-defined data type that consists of named integer constants.
- It provides better type safety and can be used in switch statements.

- It is more readable and maintainable, especially when dealing with a list of related constants.
- It can be debugged more easily compared to `#define`.

Which is better?
- **enum** is generally better for defining a set of related constants because it provides type safety, better readability, and easier debugging. It is more suitable for cases where you need a group of related named constants.
- **#define** is better for simple constant definitions where type safety is not a concern, or when you need to define macros.
In summary, use `enum` for related constants and `#define` for simple, standalone constants.

Q27. Write a program in c to remove duplicate in an array.

```
#include <stdio.h>
void removeDuplicates(int arr[], int n) {
    if (n == 0) return;
    int temp[n];
    int j = 0;
    for (int i = 0; i < n - 1; i++) {
        if (arr[i] != arr[i + 1]) {
            temp[j++] = arr[i];
        }
    }
    temp[j++] = arr[n - 1];

    for (int i = 0; i < j; i++) {
        arr[i] = temp[i];
    }

    for (int i = 0; i < j; i++) {
        printf("%d ", arr[i]);
    }
}
int main() {
    int arr[] = {1, 2, 2, 3, 4, 4, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    removeDuplicates(arr, n);
    return 0;
}
```

Q28. Can we compile a c program without main function? Explain.

No, we cannot compile a C program without a main function.
The main function serves as the entry point for program execution in C. Without it, the compiler does not know where to begin executing the program.

Q29. Define all the operators in c.

Here is a detailed list of all operators in C with definitions and examples:

1. Arithmetic Operators

These are used for basic mathematical operations.

Operator	Description	Example
+	Addition	a + b (adds a and b)
-	Subtraction	a - b (subtracts b from a)
*	Multiplication	a * b (multiplies a and b)
/	Division	a / b (divides a by b)
%	Modulus	a % b (remainder of a divided by b)

Example:

```
#include <stdio.h>
int main() {
   int a = 10, b = 3;
   printf("Addition: %d\n", a + b);
   printf("Division: %d\n", a / b);
   printf("Modulus: %d\n", a % b);
   return 0;
}
```

2. Relational (Comparison) Operators

These compare two values and return a Boolean value (true/false).

Operator	Description	Example
==	Equal to	a == b
!=	Not equal to	a != b
>	Greater than	a > b
<	Less than	a < b
>=	Greater than or equal to	a >= b
<=	Less than or equal to	a <= b

Example:

```
#include <stdio.h>
int main() {
   int a = 5, b = 10;
   printf("a < b: %d\n", a < b);
   printf("a == b: %d\n", a == b);
   return 0;
}
```

3. Logical Operators

Used to perform logical operations.

Operator	Description	Example
&&	Logical AND	(a > 0 && b > 0)
`		`
!	Logical NOT	!(a > 0)

Example:

```
#include <stdio.h>
int main() {
   int a = 1, b = 0;
   printf("a && b: %d\n", a && b);
   printf("a || b: %d\n", a || b);
   printf("!a: %d\n", !a);
   return 0;
}
```

4. Bitwise Operators

Operate on bits of data.

Operator	Description	Example
&	Bitwise AND	a & b
`	`	Bitwise OR
^	Bitwise XOR	a ^ b
~	Bitwise NOT	~a
<<	Left shift	a << 1
>>	Right shift	a >> 1

Example:

```
#include <stdio.h>
int main() {
   int a = 5, b = 3; // Binary: a = 0101, b = 0011
   printf("a & b: %d\n", a & b); // 0001 = 1
   printf("a | b: %d\n", a | b); // 0111 = 7
   printf("a ^ b: %d\n", a ^ b); // 0110 = 6
   return 0;
}
```

5. Assignment Operators

Used to assign values.

Operator	Description	Example
=	Assign	a = b
+=	Add and assign	a += b (a = a + b)
-=	Subtract and assign	a -= b (a = a - b)
*=	Multiply and assign	a *= b (a = a * b)
/=	Divide and assign	a /= b (a = a / b)
%=	Modulus and assign	a %= b (a = a % b)

Example:

```
#include <stdio.h>
int main() {
   int a = 10, b = 5;
   a += b; // a = a + b
   printf("Updated a: %d\n", a);
   return 0;
}
```

6. Increment and Decrement Operators

Used to increase or decrease the value of a variable.

Operator	Description	Example
++	Increment	a++ or ++a
--	Decrement	a-- or --a

Example:

```
#include <stdio.h>
int main() {
   int a = 10;
   printf("a++: %d\n", a++); // Prints 10, then increments
   printf("++a: %d\n", ++a); // Increments, then prints 12
   return 0;
}
```

7. Conditional (Ternary) Operator

Used for short conditional expressions.

Operator	Description	Example
? :	Ternary conditional operator	a = (b > c) ? b : c;

Example:

```
#include <stdio.h>
int main() {
   int a = 10, b = 20;
   int max = (a > b) ? a : b;
   printf("Max: %d\n", max);
   return 0;
}
```

Q30. Write the difference between source code and object code in c.

Aspect	Source Code	Object Code
Definition	The high-level code written by the programmer in a human-readable format.	The machine-readable, intermediate code generated after compilation.
Representation	Written in a programming language like C (e.g., .c file).	Typically in binary or assembly format (e.g., .o or .obj file).
Human Readability	Readable and understandable by humans.	Not human-readable; intended for the machine.
Purpose	Represents the logic and instructions for the program.	Acts as the compiled form of the source code, ready for linking and execution.
Creation	Written manually by the developer.	Generated automatically by the compiler.
File Extension	Commonly .c for C programs.	Commonly .o, .obj, or .out.
Modification	Can be directly edited by the programmer.	Cannot be modified directly; changes require editing the source code and recompilation.
Example	```c	Binary instructions for machine execution.

Q31. Why C is called middle level language?

C is called a middle-level language because it combines the features of both high-level and low-level languages.
It provides high-level constructs for structured programming and abstraction, while also allowing low-level manipulation of hardware and memory, making it versatile for system programming and application development.

Q32. What is dynamic Data Structure in c?

Dynamic data structures in C are data structures that can grow and shrink in size during the execution of a program.
They are implemented using pointers and memory allocation functions such as `malloc()`, `calloc()`, `realloc()`, and `free()`.
Examples of dynamic data structures include linked lists, stacks, queues, and trees. These structures allow for efficient memory usage and flexibility in managing data.

Q33. Explain the return type of printf() and scanf() in c with example.

Return Type of printf() and scanf() in C:

1. printf() Function

- **Return Type:** int
- **Description:**
    - Returns the **number of characters successfully written** to the output stream.
    - Returns a **negative value** if an error occurs.

Example:

```
int count = printf("Hello, World!\n");
printf("Characters printed: %d\n", count);
```

2. scanf() Function

- **Return Type:** int
- **Description:**
    - Returns the **number of input items successfully assigned** to the variables.
    - Returns **EOF (typically -1)** if an error occurs or if the end of input is reached.

Example:

```
int num, result;
result = scanf("%d", &num);
if (result == 1) {
   printf("You entered: %d\n", num);
} else {
   printf("Input error. scanf() returned: %d\n", result);
}
```

Q34. What is the utilization of seek() in c?

The `seek()` function in C is used to move the file pointer to a specific location within a file, enabling random access to file contents.

For example, if you have a file and you want to skip the first 100 bytes and start reading from the 101st byte, you can use `fseek(file, 100, SEEK_SET);` to set the file pointer to the desired position.

Q35. What is the utilization of sprintf() in c?

The `sprintf()` function in C is used to format and store a series of characters and values in a string. It works similarly to `printf()`, but instead of printing the output to the console, it stores the output in a string buffer.

Example:

```
#include <stdio.h>
int main() {
   char buffer[50];
   int age = 25;
   sprintf(buffer, "I am %d years old.", age);
   printf("%s\n", buffer);
   return 0;
}
```

In this example, `sprintf()` formats the string "I am 25 years old." and stores it in the `buffer` array.

Q36. Write the difference between getch() and getche() in c.

Feature	getch()	getche()
Functionality	Reads a character from the keyboard without echoing it on the screen.	Reads a character from the keyboard and echoes it on the screen.
Echo	Does not display the input character on the console.	Displays the input character on the console.
Header File	<conio.h>	<conio.h>
Use Case	Useful for password input or hidden data entry.	Useful for interactive input where feedback is needed.
Return Type	Returns the ASCII value of the input character.	Returns the ASCII value of the input character.
Example Usage	char ch = getch();	char ch = getche();