

Introduction_To_C

C Language Introduction

C is a procedural programming language initially developed by Dennis Ritchie in the year 1972 at Bell Laboratories of AT&T Labs. It was mainly developed as a system programming language to write the UNIX operating system.

[image]

The main features of the C language include:

- General Purpose and Portable
- Low-level Memory Access
- Fast Speed
- Clean Syntax

These features make the C language suitable for system programming like an operating system or compiler development. For beginners looking to master C along with essential data structures, the [C Programming Course Online with Data Structures](#) offers a step-by-step guide from the basics of C to more advanced topics.

Why Should We Learn C?

Many later languages have borrowed syntax/features directly or indirectly from the C language like the syntax of Java, PHP, JavaScript, and many other languages that are mainly based on the C language. C++ is nearly a superset of C language (Only a few programs may compile in C, but not in C++).

So, if a person learns C programming first, it will help them to learn any modern programming language as well. Also, learning C helps to understand a lot of the underlying architecture of the operating system like pointers, working with memory locations, etc.

Difference Between C and C++

C++ was created to add the OOPs concept into the C language so they both have very similar syntax with a few differences. The following are some of the main differences between C and C++ Programming languages.

- C++ supports OOPs paradigm while C only has the procedural concept of programming.
- C++ has exception handling capabilities. In C, we have to resolve exceptions manually.
- There are no references in C.

There are many more differences between C and C++ which are discussed here: [Difference between C and C++](#)

Beginning with C Programming

Writing the First Program in C

The following code is one of the simplest C programs that will help us understand the basic syntax structure of a C program.

Example:

```
#include <stdio.h>

int main() {
    int a = 10;
    printf("%d", a);

    return 0;
}
```

Output:

10

Let us analyze the structure of our program line by line.

Structure of the C program

After the above discussion, we can formally assess the basic structure of a C program. By structure, it is meant that any program can be written in this structure only. Writing a C program in any other structure will lead to a Compilation Error. The structure of a C program is as follows:

[image]

Components of a C Program:

1. Header Files Inclusion – Line 1 [#include < stdio.h>]

The first and foremost component is the inclusion of the Header files in a C program. A header file is a file with extension .h which contains C function declarations and macro definitions to be shared between several source files. All lines that start with # are processed by a preprocessor which is a program invoked by the compiler. In the above example, the preprocessor copies the preprocessed code of stdio.h to our file. The .h files are called header files in C.

Some of the C Header files:

- stddef.h – Defines several useful types and macros
- stdint.h – Defines exact width integer types
- stdio.h – Defines core input and output functions
- stdlib.h – Defines numeric conversion functions, pseudo-random number generator, and memory allocation
- string.h – Defines string handling functions

- `math.h` – Defines common mathematical functions.

2. Main Method Declaration – Line 2 [`int main()`]

The next part of a C program is to declare the `main()` function. It is the entry point of a C program and the execution typically begins with the first line of the `main()`. The empty brackets indicate that the `main` doesn't take any parameter (See [this](#) for more details). The `int` that was written before the `main` indicates the return type of `main()`. The value returned by the `main` indicates the status of program termination. See [this](#) post for more details on the return type.

3. Body of Main Method – Line 3 to Line 6 [enclosed in `{}`]

The body of a function in the C program refers to statements that are a part of that function. It can be anything like manipulations, searching, sorting, printing, etc. A pair of curly brackets define the body of a function. All functions must start and end with curly brackets.

4. Statement – Line 4 [`printf("Hello World");`]

Statements are the instructions given to the compiler. In C, a statement is always terminated by a semicolon (;). In this particular case, we use `printf()` function to instruct the compiler to display "Hello World" text on the screen.

5. Return Statement – Line 5 [`return 0;`]

The last part of any C function is the return statement. The return statement refers to the return values from a function. This return statement and return value depend upon the return type of the function. The return statement in our program returns the value from `main()`. The returned value may be used by an operating system to know the termination status of your program. The value 0 typically means successful termination.

How to Execute the Above Program?

In order to execute the above program, we need to first compile it using a compiler and then we can run the generated executable. There are online IDEs available for free like [GeeksforGeeksIDE](#), that can be used to start development in C without installing a compiler.

1. Windows: There are many free IDEs available for developing programs in C like [Code Blocks](#) and [Dev-CPP](#). IDEs provide us with an environment to develop code, compile it and finally execute it. We strongly recommend Code Blocks.
2. Linux: GCC compiler comes bundled with Linux which compiles C programs and generates executables for us to run. Code Blocks can also be used with Linux.
3. macOS: macOS already has a built-in text editor where you can just simply write the code and save it with a ".c" extension.

Application of C

- Operating systems: C is widely used for developing operating systems such as Unix, Linux, and Windows.

- Embedded systems: C is a popular language for developing embedded systems such as microcontrollers, microprocessors, and other electronic devices.
- System software: C is used for developing system software such as device drivers, compilers, and assemblers.
- Networking: C is widely used for developing networking applications such as web servers, network protocols, and network drivers.
- Database systems: C is used for developing database systems such as Oracle, MySQL, and PostgreSQL.
- Gaming: C is often used for developing computer games due to its ability to handle low-level hardware interactions.
- Artificial Intelligence: C is used for developing artificial intelligence and machine learning applications such as neural networks and deep learning algorithms.
- Scientific applications: C is used for developing scientific applications such as simulation software and numerical analysis tools.
- Financial applications: C is used for developing financial applications such as stock market analysis and trading systems.

Features of C Programming Language

C is a procedural programming language. It was initially developed by Dennis Ritchie in the year 1972. It was mainly developed as a system programming language to write an operating system.

The main features of C language include low-level access to memory, a simple set of keywords, and a clean style, these features make C language suitable for system programming like an operating system or compiler development.

What are the Most Important Features of C Language?

Here are some of the most important features of the C language:

1. Procedural Language
2. Fast and Efficient
3. Modularity
4. Statically Type
5. General-Purpose Language
6. Rich set of built-in Operators
7. Libraries with Rich Functions
8. Middle-Level Language
9. Portability
10. Easy to Extend

[image]

If you're looking to explore the full potential of C, especially in working with data structures, the [C Programming Course Online with Data Structures](#) provides a complete guide to mastering C's features for real-world applications.

1. Procedural Language

In a [procedural language](#) like C step by step, predefined instructions are carried out. C program may contain more than one function to perform a particular task. New people to programming will think that this is the only way a particular programming language works. There are other programming paradigms as well in the programming world. Most of the commonly used paradigm is an object-oriented programming language.

2. Fast and Efficient

Newer languages like Java, python offer more features than [c programming language](#) but due to additional processing in these languages, their performance rate gets down effectively. C programming language as the middle-level language provides programmers access to direct manipulation with the computer hardware but higher-level languages do not allow this. That's one of the reasons C language is considered the first choice to start learning programming languages. It's fast because statically typed languages are faster than dynamically typed languages.

3. Modularity

The concept of storing C programming language code in the form of libraries for further future uses is known as modularity. This programming language can do very little on its own most of its power is held by its libraries. C language has its own [library](#) to solve common problems.

4. Statically Type

C programming language is a [statically typed language](#). Meaning the type of variable is checked at the time of compilation but not at run time. This means each time a programmer types a program they have to mention the type of variables used.

5. General-Purpose Language

From system programming to photo editing software, the C programming language is used in various applications. Some of the common applications where it's used are as follows:

- [Operating systems](#): Windows, [Linux](#), iOS, [Android](#), OXS
- [Databases](#): PostgreSQL, Oracle, [MySQL](#), MS SQL Server, etc.

6. Rich set of built-in Operators

It is a diversified language with a rich set of built-in [operators](#) which are used in writing complex or simplified C programs.

7. Libraries with Rich Functions

Robust libraries and [functions in C](#) help even a beginner coder to code with ease.

8. Middle-Level Language

As it is a middle-level language so it has the combined form of both capabilities of assembly language and features of the [high-level language](#).

9. Portability

C language is lavishly portable as programs that are written in C language can run and compile on any system with either no or small changes.

10. Easy to Extend

Programs written in C language can be extended means when a program is already written in it then some more features and operations can be added to it.

Let discuss these features one by one:

C Programming Language Standard

Introduction:

The C programming language has several standard versions, with the most commonly used ones being C89/C90, C99, C11, and C18.

1. C89/C90 (ANSI C or ISO C) was the first standardized version of the language, released in 1989 and 1990, respectively. This standard introduced many of the features that are still used in modern C programming, including data types, control structures, and the standard library.
2. C99 (ISO/IEC 9899:1999) introduced several new features, including variable-length arrays, flexible array members, complex numbers, inline functions, and designated initializers. This standard also includes several new library functions and updates to existing ones.
3. C11 (ISO/IEC 9899:2011) introduced several new features, including `_Generic`, `static_assert`, and the atomic type qualifier. This standard also includes several updates to the library, including new functions for math, threads, and memory manipulation.
4. C18 (ISO/IEC 9899:2018) is the most recent standard and includes updates and clarifications to the language specification and the library.

When writing C code, it's important to know which standard version is being used and to write code that is compatible with that standard. Many compilers support multiple standard versions, and it's often possible to specify which version to use with a compiler flag or directive.

Here are some advantages and disadvantages of using the C programming language:

Advantages:

1. Efficiency: C is a fast and efficient language that can be used to create high-performance applications.
2. Portability: C programs can be compiled and run on a wide range of platforms and operating systems.
3. Low-level access: C provides low-level access to system resources, making it ideal for systems programming and developing operating systems.
4. Large user community: C has a large and active user community, which means there are many resources and libraries available for developers.
5. Widely used: C is a widely used language, and many modern programming languages are built on top of it.

To learn how to write standard-compliant programs, including data structure

implementations, the [C Programming Course Online with Data Structures](#) provides comprehensive lessons.

Disadvantages:

1. Steep learning curve: C can be difficult to learn, especially for beginners, due to its complex syntax and low-level access to system resources.
2. Lack of memory management: C does not provide automatic memory management, which can lead to memory leaks and other memory-related bugs if not handled properly.
3. No built-in support for object-oriented programming: C does not provide built-in support for object-oriented programming, making it more difficult to write object-oriented code compared to languages like Java or Python.
4. No built-in support for concurrency: C does not provide built-in support for concurrency, making it more difficult to write multithreaded applications compared to languages like Java or Go.
5. Security vulnerabilities: C programs are prone to security vulnerabilities, such as buffer overflows, if not written carefully. Overall, C is a powerful language with many advantages, but it also requires a high degree of expertise to use effectively and has some potential drawbacks, especially for beginners or developers working on complex projects.

Importance:

important for several reasons:

1. Choosing the right programming language: Knowing the advantages and disadvantages of C can help developers choose the right programming language for their projects. For example, if high performance is a priority, C may be a good choice, but if ease of use or built-in memory management is important, another language may be a better fit.
2. Writing efficient code: Understanding the efficiency advantages of C can help developers write more efficient and optimized code, which is especially important for systems programming and other performance-critical applications.
3. Avoiding common pitfalls: Understanding the potential disadvantages of C, such as memory management issues or security vulnerabilities, can help developers avoid common pitfalls and write safer, more secure code.
4. Collaboration and communication: Knowing the advantages and disadvantages of C can also help developers communicate and collaborate effectively with others on their team or in the wider programming community.

The idea of this article is to introduce C standard.

What to do when a C program produces different results in two different compilers?

For example, consider the following simple C program.

```
void main() { }
```

The above program fails in GCC as the return type of main is void, but it compiles in Turbo C. How do we decide whether it is a legitimate C program or not?

Consider the following program as another example. It produces different results in different compilers.

```
// C++ Program to illustrate the difference in different
// compiler execution
#include <stdio.h>

int main() {
    int i = 1;
    printf("%d %d %d\n", i++, i++, i);
    return 0;
}
```

```
2 1 3 - using g++ 4.2.1 on Linux.i686
1 2 3 - using SunStudio C++ 5.9 on Linux.i686
2 1 3 - using g++ 4.2.1 on SunOS.x86pc
1 2 3 - using SunStudio C++ 5.9 on SunOS.x86pc
1 2 3 - using g++ 4.2.1 on SunOS.sun4u
1 2 3 - using SunStudio C++ 5.9 on SunOS.sun4u
```

Source: [Stackoverflow](#)

Which compiler is right?

The answer to all such questions is C standard. In all such cases, we need to see what C standard says about such programs.

What is C standard?

The latest C standard is [ISO/IEC 9899:2018](#), also known as **C17** as the final draft was published in 2018. Before C11, there was [C99](#). The C11 final draft is available [here](#). See [this](#) for a complete history of C standards.

Can we know the behavior of all programs from C standard?

C standard leaves some behavior of many C constructs as [undefined](#) and some as [unspecified](#) to simplify the specification and allow some flexibility in implementation. For example, in C the use of any automatic variable before it has been initialized yields undefined behavior and order of evaluations of subexpressions is unspecified. This specifically frees the compiler to do whatever is easiest or most efficient, should such a program be submitted.

So what is the conclusion about above two examples?

Let us consider the first example which is "void main() {}", the standard says following about prototype of main().

The function called at program startup is named main. The implementation declares no prototype for this function. It shall be defined with a return type of int and with no parameters:

```
int main(void) { /* ... */ }
```

or with two parameters (referred to here as argc and argv, though any name may be used, as they are local to the function in which they are declared

```
int main(int argc, char *argv[]) { /* ... */ }
```

or equivalent;10) or in some other implementation-defined manner.

So the return type void doesn't follow the standard, and it's something allowed by certain

compilers.

Let us talk about the second example. Note the following statement in C standard is listed under unspecified behavior.

The order in which the function designator, arguments, and subexpressions within the arguments are evaluated in a function call (6.5.2.2).

What to do with programs whose behavior is undefined or unspecified in standard?

As a programmer, it is never a good idea to use programming constructs whose behavior is undefined or unspecified, such programs should always be discouraged. The output of such programs may change with the compiler and/or machine.

Data Types in C

Last Updated : 11 Oct, 2024

Each variable in C has an associated data type. It specifies the type of data that the variable can store like integer, character, floating, double, etc. Each data type requires different amounts of memory and has some specific operations which can be performed over it.

The data types in C can be classified as follows:

Types	Description	Data Types
Primitive Data Types	Primitive data types are the most basic data types that are used for representing simple values such as integers, float, double, void characters, etc.	int, char, float, double, void
Derived Types	The data types that are derived from the primitive or built-in datatypes are referred to as Derived Data Types.	array, pointers, function
User Defined Data Types	The user-defined data types are defined by the user himself.	structure, union, enum

Understanding C's data types is critical for writing efficient programs. If you're interested in how data types interact with different data structures, the [C Programming Course Online with Data Structures](#) covers everything from basic types to more complex structures in C programming.

The following are some main primitive data types in C:

Table of Content

- [Integer Data Type](#)
- [Character Data Type](#)
- [Float Data Type](#)
- [Double Data Type](#)
- [Void Data Type](#)

Integer Data Type

The integer datatype in C is used to store the integer numbers (any number including positive, negative and zero without decimal part). Octal values, hexadecimal values, and decimal values can be stored in int data type in C.

- Range: -2,147,483,648 to 2,147,483,647
- Size: 4 bytes
- Format Specifier: %d

Syntax of Integer

We use [int keyword](#) to declare the integer variable:

```
int var_name;
```

The integer data type can also be used as

1. unsigned int: Unsigned int data type in C is used to store the data values from zero to positive numbers but it can't store negative values like signed int.
2. short int: It is lesser in size than the int by 2 bytes so can only store values from -32,768 to 32,767.
3. long int: Larger version of the int datatype so can store values greater than int.
4. unsigned short int: Similar in relationship with short int as unsigned int with int.

Note: The size of an integer data type is compiler-dependent. We can use sizeof operator to check the actual size of any data type.

Example of int

C

```
// C program to print Integer data types.
#include <stdio.h>

int main()
{
    // Integer value with positive data.
    int a = 9;

    // integer value with negative data.
    int b = -9;

    // U or u is Used for Unsigned int in C.
    int c = 89U;

    // L or l is used for long int in C.
    long int d = 99998L;

    printf("Integer value with positive data: %d\n", a);
    printf("Integer value with negative data: %d\n", b);
    printf("Integer value with an unsigned int data: %u\n",
           c);
    printf("Integer value with an long int data: %ld", d);
```

```
    return 0;
}
```

Output

```
Integer value with positive data: 9
Integer value with negative data: -9
Integer value with an unsigned int data: 89
Integer value with an long int data: 99998
```

Character Data Type

Character data type allows its variable to store only a single character. The size of the character is 1 byte. It is the most basic data type in C. It stores a single character and requires a single byte of memory in almost all compilers.

- Range: (-128 to 127) or (0 to 255)
- Size: 1 byte
- Format Specifier: %c

Syntax of char

The char keyword is used to declare the variable of character type:

```
char var_name;
```

Example of char

C

```
// C program to print Integer data types.
#include <stdio.h>

int main()
{
    char a = 'a';
    char c;

    printf("Value of a: %c\n", a);

    a++;
    printf("Value of a after increment is: %c\n", a);

    // c is assigned ASCII values
    // which corresponds to the
    // character 'c'
    // a-->97 b-->98 c-->99
    // here c will be printed
    c = 99;

    printf("Value of c: %c", c);
}
```

```
    return 0;
}
```

Output

```
Value of a: a
Value of a after increment is: b
Value of c: c
```

Float Data Type

In C programming [float data type](#) is used to store floating-point values. Float in C is used to store decimal and exponential values. It is used to store decimal numbers (numbers with floating point values) with single precision.

- Range: 1.2E-38 to 3.4E+ 38
- Size: 4 bytes
- Format Specifier: %f

Syntax of float

The float keyword is used to declare the variable as a floating point:

```
float var_name;
```

Example of Float

C

```
// C Program to demonstrate use
// of Floating types
#include <stdio.h>

int main()
{
    float a = 9.0f;
    float b = 2.5f;

    // 2x10^-4
    float c = 2E-4f;
    printf("%f\n", a);
    printf("%f\n", b);
    printf("%f", c);

    return 0;
}
```

Output

9.000000
2.500000
0.000200

Double Data Type

A [Double data type](#) in C is used to store decimal numbers (numbers with floating point values) with double precision. It is used to define numeric values which hold numbers with decimal values in C.

The double data type is basically a precision sort of data type that is capable of holding 64 bits of decimal numbers or floating points. Since double has more precision as compared to that float then it is much more obvious that it occupies twice the memory occupied by the floating-point type. It can easily accommodate about 16 to 17 digits after or before a decimal point.

- Range: 1.7E-308 to 1.7E+ 308
- Size: 8 bytes
- Format Specifier: %lf

Syntax of Double

The variable can be declared as double precision floating point using the double keyword:

```
double var_name;
```

Example of Double

C

```
// C Program to demonstrate
// use of double data type
#include <stdio.h>

int main()
{
    double a = 123123123.00;
    double b = 12.293123;
    double c = 2312312312.123123;

    printf("%lf\n", a);

    printf("%lf\n", b);

    printf("%lf", c);

    return 0;
}
```

Output

```
123123123.000000
12.293123
2312312312.123123
```

Void Data Type

The void data type in C is used to specify that no value is present. It does not provide a result value to its caller. It has no values and no operations. It is used to represent nothing. Void is used in multiple ways as function return type, function arguments as void, and [pointers to void](#).

Syntax:

```
// function return type void
void exit(int check);
// Function without any parameter can accept void.
int print(void);
// memory allocation function which
// returns a pointer to void.
void *malloc (size_t size);
```

Example of Void

```
// C program to demonstrate
// use of void pointers
#include <stdio.h>

int main()
{
    int val = 30;
    void* ptr = &val;
    printf("%d", *(int*)ptr);
    return 0;
}
```

Output

30

Size of Data Types in C

The size of the data types in C is dependent on the size of the architecture, so we cannot define the universal size of the data types. For that, the C language provides the `sizeof()` operator to check the size of the data types.

Example

```
// C Program to print size of
// different data type in C
#include <stdio.h>

int main()
{
    int size_of_int = sizeof(int);
    int size_of_char = sizeof(char);
    int size_of_float = sizeof(float);
    int size_of_double = sizeof(double);

    printf("The size of int data type : %d\n", size_of_int);
    printf("The size of char data type : %d\n",
        size_of_char);
    printf("The size of float data type : %d\n",
        size_of_float);
    printf("The size of double data type : %d",
        size_of_double);

    return 0;
}
```

Output

```
The size of int data type : 4
The size of char data type : 1
The size of float data type : 4
The size of double data type : 8
```

Different data types also have different ranges up to which they can store numbers. These ranges may vary from compiler to compiler. Below is a list of ranges along with the memory requirement and format specifiers on the 32-bit GCC compiler**.

Data Type	Size (bytes)	Range	Format Specifier
short int	2	-32,768 to 32,767	%hd
unsigned short int	2	0 to 65,535	%hu
unsigned int	4	0 to 4,294,967,295	%u
int	4	-2,147,483,648 to 2,147,483,647	%d
long int	4	-2,147,483,648 to 2,147,483,647	%ld
unsigned long int	4	0 to 4,294,967,295	%lu
long long int	8	-(2 ⁶³) to (2 ⁶³)-1	%lld
unsigned long long int	8	0 to 18,446,744,073,709,551,615	%llu
signed char	1	-128 to 127	%c
unsigned char	1	0 to 255	%c
float	4	1.2E-38 to 3.4E+ 38	%f
double	8	1.7E-308 to 1.7E+ 308	%lf
long double	16	3.4E-4932 to 1.1E+ 4932	%Lf

Note: The long, short, signed and unsigned are datatype modifier that can be

used with some primitive data types to change the size or length of the datatype.

Literals in C

In C, Literals are the constant values that are assigned to the variables. Literals represent fixed values that cannot be modified. Literals contain memory but they do not have references as variables. Generally, both terms, constants, and literals are used interchangeably.

For example, "const int = 5;", is a constant expression and the value 5 is referred to as a constant integer literal.

Types of C Literals

There are 4 types of literal in C:

- Integer Literal
- Float Literal
- Character Literal
- String Literal

!<https://media.geeksforgeeks.org/wp-content/uploads/20231226085253/C-literals.png>

1. Integer Literals

[Integer literals](#) are used to represent and store the integer values only. Integer literals are expressed in two types i.e.

A) Prefixes: The Prefix of the integer literal indicates the base in which it is to be read.

For Example:

0x10 = 16 Because 0x prefix represents a HexaDecimal base. So 10 in HexaDecimal is 16 in Decimal. Hence the value 16.

There are basically represented into 4 types:

a. Decimal-literal(base 10): A non-zero decimal digit followed by zero or more decimal digits(0, 1, 2, 3, 4, 5, 6, 7, 8, 9).

Example:

56, 78

b. Octal-literal(base 8): a 0 followed by zero or more octal digits(0, 1, 2, 3, 4, 5, 6, 7).

Example:

045, 076, 06210

c. Hex-literal(base 16): 0x or 0X followed by one or more hexadecimal digits(0, 1, 2, 3, 4,

5, 6, 7, 8, 9, a, A, b, B, c, C, d, D, e, E, f, F).

Example:

0x23A, 0xb4C, 0xFEa

d. Binary-literal (base 2): 0b or 0B followed by one or more binary digits (0, 1).

Example:

0b101, 0B111

B) Suffixes: The Suffixes of the integer literal indicates the type in which it is to be read.

For example:

12345678901234LL indicates a long long integer value 12345678901234
because of the suffix LL

These are represented in many ways according to their data types.

- int: No suffix is required because integer constant is by default assigned as an int data type.
- unsigned int: character u or U at the end of an integer constant.
- long int: character l or L at the end of an integer constant.
- unsigned long int: character ul or UL at the end of an integer constant.
- long long int: character ll or LL at the end of an integer constant.
- unsigned long long int: character ull or ULL at the end of an integer constant.

Example:

C

```
#include <stdio.h>

int main()
{
    // constant integer literal
    const int intVal = 10;

    printf("Integer Literal:%d \n", intVal);
    return 0;
}
```

Output

Integer Literal:10

2. Floating-Point Literals

These are used to represent and store real numbers. The real number has an integer part, real part, fractional part, and exponential part. The floating-point literals can be stored either in

decimal form or exponential form. While representing the floating-point decimals one must keep two things in mind to produce valid literal:

- In the decimal form, one must include the integer part, or fractional part, or both, otherwise, it will lead to an error.
- In the exponential form, one must include both the significand and exponent part, otherwise, it will lead to an error.

A few floating-point literal representations are shown below:

Valid Floating Literals:

```
10.125
1.215e-10L
10.5E-3
```

Invalid Floating Literals:

```
123E
1250f
0.e879
```

Example:

```
#include <stdio.h>

int main()
{
    // constant float literal
    const float floatVal = 4.14;

    printf("Floating point literal: %.2f\n",
           floatVal);
    return 0;
}
```

Output

```
Floating point literal: 4.14
```

3. Character Literals

This refers to the literal that is used to store a single character within a single quote. To store multiple characters, one needs to use a character array. Storing more than one character within a single quote will throw a warning and display just the last character of the literal. It gives rise to the following two representations:

- char type: This is used to store normal character literal or narrow-character literals.

Example:

```
char chr = 'G';
```

Example:

C

```
#include <stdio.h>

int main()
{
    // constant char literal
    const char charVal = 'A';

    printf("Character Literal: %c\n",
           charVal);
    return 0;
}
```

[**Escape Sequences](#)** There are various special characters that one can use to perform various operations.

4. String Literals

String literals are similar to that character literals, except that they can store multiple characters and uses a double quote to store the same. It can also accommodate the special characters and escape sequences mentioned in the table above. We can break a long line into multiple lines using string literal and can separate them with the help of white spaces.

Example:

```
char stringVal[] = "GeeksforGeeks";
```

Example:

C

```
#include <stdio.h>

int main()
{
    const char str[]
        = "Welcome\nTo\nGeeks\tFor\tGeeks";
    printf("%s", str);
    return 0;
}
```

Output

```
Welcome
To
Geeks    For    Geeks
```

Escape Sequence in C

The escape sequence in C is the characters or the sequence of characters that can be used inside the string literal. The purpose of the escape sequence is to represent the characters that cannot be used normally using the keyboard. Some escape sequence characters are the part of ASCII charset but some are not.

Different escape sequences represent different characters but the output is dependent on the compiler you are using.

Escape Sequence List

The table below lists some common escape sequences in C language.

Escape Sequence	Name	Description
\a	Alarm or Beep	It is used to generate a bell sound in the C program.
\b	Backspace	It is used to move the cursor one place backward.
\f	Form Feed	It is used to move the cursor to the start of the next logical page.
\n	New Line	It moves the cursor to the start of the next line.
\r	Carriage Return	It moves the cursor to the start of the current line.
\t	Horizontal Tab	It inserts some whitespace to the left of the cursor and moves the cursor accordingly.
\v	Vertical Tab	It is used to insert vertical space.
\	Backslash	Use to insert backslash character.
\'	Single Quote	It is used to display a single quotation mark.
\"	Double Quote	It is used to display double quotation marks.
\?	Question Mark	It is used to display a question mark.
\ooo	Octal Number	It is used to represent an octal number.
\xhh	Hexadecimal Number	It represents the hexadecimal number.
\0	NULL	It represents the NULL character.

Out of all these escape sequences, \n and \0 are used the most. In fact, escape sequences like \f, \a, are not even used by programmers nowadays. To learn how to handle strings, characters, and escape sequences in larger programs, the [C Programming Course Online with Data Structures](#) offers lessons on string manipulation and formatting

Escape Sequence in C Examples

The following are the escape sequence examples that demonstrate how to use different escape sequences in C language.

1. Example to demonstrate how to use \a escape sequence in C

C

```
// C program to illustrate \a escape sequence
```

```
#include <stdio.h>

int main(void)
{
    // output may depend upon the compiler
    printf("My mobile number "
           "is 7\a8\a7\a3\a9\a2\a3\a4\a0\a8\a");
    return (0);
}
```

Output

My mobile number is 7873923408

2. Example to demonstrate how to use \b escape sequence in C

C

```
// C program to illustrate \b escape sequence
#include <stdio.h>

int main(void)
{
    // \b - backspace character transfers
    // the cursor one character back with
    // or without deleting on different
    // compilers.
    printf("Hello \b\b\b\b\b\bHi Geeks");
    return (0);
}
```

Output

Hi Geeks

3. Example to demonstrate how to use \n escape sequence in C

C

```
// C program to illustrate \n escape sequence
#include <stdio.h>
int main(void)
{
    // Here we are using \n, which is a new line character.
    printf("Hello\n");
    printf("GeeksforGeeks");
    return (0);
}
```

Output

Hello

4. Example to demonstrate how to use \t escape sequence in C

C

```
// C program to illustrate \t escape sequence
#include <stdio.h>

int main(void)
{
    // Here we are using \t, which is
    // a horizontal tab character.
    // It will provide a tab space
    // between two words.
    printf("Hello \t GFG");
    return (0);
}
```

Output

Hello GFG

The escape sequence “\t” is very frequently used in loop-based [pattern printing programs](#).

5. Example to demonstrate how to use \v escape sequence in C

C

```
// C program to illustrate \v escape sequence
#include <stdio.h>

int main(void)
{
    // Here we are using \v, which
    // is vertical tab character.
    printf("Hello friends\v");

    printf("Welcome to GFG");

    return (0);
}
```

Output

Hello friends
 Welcome to GFG

6. Example to demonstrate how to use \r escape sequence in C

```
// C program to illustrate \r escape sequence
#include <stdio.h>
```

```

int main(void)
{
    // Here we are using \r, which
    // is carriage return character.
    printf("Hello    Geeks \rGeeksfor");
    return (0);
}

```

Output

GeeksforGeeks

7. Example to demonstrate how to use \ escape sequence in C

```

// C program to illustrate \ (Backslash)
// escape sequence to print backslash.
#include <stdio.h>

```

```

int main(void)
{
    // Here we are using \,
    // It contains two escape sequence
    // means \ and \n.
    printf("Hello\\GFG");
    return (0);
}

```

Output

Hello\GFG

Explanation: It contains two '\ ' which means we want print '\ ' as output.

8. Example to demonstrate how to use \' and \" escape sequence in C

```

// C program to illustrate \' escape
// sequence/ and \" escape sequence to
// print single quote and double quote.
#include <stdio.h>

```

```

int main(void)
{
    printf("\' Hello Geeks\n");
    printf("\" Hello Geeks");
    return 0;
}

```

Output

```

' Hello Geeks
" Hello Geeks

```

9. Example to demonstrate how to use ? escape sequence in C

```
// C program to illustrate
// \? escape sequence
#include <stdio.h>

int main(void)
{
    // Here we are using \?, which is
    // used for the presentation of trigraph
    // in the early of C programming. But
    // now we don't have any use of it.
    printf("\?\?!\n");
    return 0;
}
```

Output

??!

10. Example to demonstrate how to use \ooo escape sequence in C

```
// C program to illustrate \000 escape sequence
#include <stdio.h>

int main(void)
{
    // we are using \000 escape sequence, here
    // each 0 in "000" is one to three octal
    // digits(0....7).
    char* s = "A\072\065";
    printf("%s", s);
    return 0;
}
```

Output

A:5

Explanation: Here 000 is one to three octal digits(0...7) means there must be at least one octal digit after \ and a maximum of three. Here 072 is the octal notation, first, it is converted to decimal notation which is the ASCII value of char ':'. At the place of \072, there is : and the output is A:5.

11. Example to demonstrate how to use \xhh escape sequence in C

```
// C program to illustrate \XHH escape
// sequence
#include <stdio.h>
int main(void)
{
    // We are using \xhh escape sequence.
    // Here hh is one or more hexadecimal
    // digits(0....9, a...f, A...F).
    char* s = "B\x4a";
```



```
    printf("%s", s);  
    return 0;  
}
```

Output

BJ

Explanation: Here hh is one or more hexadecimal digits(0...9, a...f, A...F). There can be more than one hexadecimal number after \x. Here, '\x4a' is a hexadecimal number and it is a single char. Firstly it will get converted into decimal notation and it is the ASCII value of the char 'J'. Therefore at the place of \x4a, we can write J. So the output is BJ.

The bool data type in C represents logical values that can be either true or false. Here's a comprehensive overview of boolean implementation in C:

Implementation Methods

Using stdbool.h (Recommended)

```
#include <stdio.h>  
#include <stdbool.h>  
  
int main() {  
    bool a = true;  
    bool b = false;  
    printf("True: %d\\n", a);  
    printf("False: %d\\n", b);  
    return 0;  
}
```

Using Enumeration

```
#include <stdio.h>  
  
typedef enum { false, true } bool;  
  
int main() {  
    bool a = true;  
    bool b = false;  
    printf("True: %d\\n", a);  
    printf("False: %d\\n", b);  
    return 0;  
}
```

Using Define

```
#define bool int
#define false 0
#define true 1

int main() {
    bool a = true;
    bool b = false;
    printf("True: %d\\n", a);
    printf("False: %d\\n", b);
    return 0;
}
```

Common Uses

Conditional Statements

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

bool is_even(int num) {
    return (num % 2 == 0);
}

int main() {
    int num = 5;
    if (is_even(num)) {
        printf("%d is even\\n", num);
    } else {
        printf("%d is odd\\n", num);
    }
    return 0;
}
```

Loop Control

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

int main() {
    bool running = true;
    int i = 0;
    while (running) {
        printf("i is %d\\n", i++);
        if (i > 5) running = false;
    }
    return 0;
}
```

Key Points

- Boolean values occupy 1 byte of memory.
- The `stdbool.h` header was introduced in the C99 standard.
- Boolean values are internally represented as integers: 1 for true and 0 for false.
- Boolean expressions can use comparison operators like `==`, `>`, `<`, `!=`.

Integer promotion in C is an automatic type conversion mechanism that occurs during arithmetic operations. Here's a detailed explanation of how it works:

Basic Concept

When operations are performed on smaller integer types (char, short int), they are automatically promoted to int or unsigned int before the calculation takes place. This promotion ensures consistent behavior and prevents overflow issues in arithmetic operations.

Promotion Rules

Standard Promotions

- Byte and short values are promoted to int
- If one operand is long, the expression is promoted to long
- Values are converted to int if they can be fully represented; otherwise, they're converted to unsigned int

Examples

Basic Arithmetic

```
#include <stdio.h>
int main() {
    char a = 30, b = 40, c = 10;
    char d = (a * b) / c;
    printf("%d", d);
    return 0;
}
```

In this case, though $(a * b)$ equals 1200 which exceeds char's range, no overflow occurs because the values are promoted to int before multiplication.

Signed vs Unsigned Comparison

```
char a = 0xfb;           // signed char
unsigned char b = 0xfb;   // unsigned char
```

```

if (a == b)                // promotion occurs here
    printf("Same");
else
    printf("Not Same");

```

Though a and b have the same binary representation as char, when promoted to int, a becomes -5 (signed) while b becomes 251 (unsigned), making them unequal[5].

Key Points

- Integer promotion happens automatically during arithmetic operations
- The promotion prevents overflow in intermediate calculations
- Different promotion rules apply based on the operands' types and ranges
- Understanding promotion rules is crucial for avoiding unexpected behavior in arithmetic operations

Character arithmetic in C allows performing mathematical operations on characters using their ASCII values. Here's how it works:

Basic Concept

Characters in C are internally stored as integers using their ASCII values, allowing arithmetic operations to be performed on them. When these operations occur, characters are automatically promoted to integers through integer promotion.

Operations and Examples

Basic Arithmetic

```

#include <stdio.h>
int main() {
    char ch1 = 125, ch2 = 10;
    ch1 = ch1 + ch2;
    printf("%d\\n", ch1);    // Prints -121
    return 0;
}

```

Character Manipulation

```

#include <stdio.h>
int main() {
    char value = 'a';
    char result = value + 3;    // Moves 3 positions forward in ASCII
}

```

```
    printf("%d\\n", result);    // Prints 100 (ASCII value)
    return 0;
}
```

Key Points

- Character values range from -128 to 127 (signed) or 0 to 255 (unsigned)
- When using %c format specifier, the value must not exceed 127
- Characters are automatically promoted to integers during arithmetic operations
- ASCII values are used for internal representation (e.g., 'A' is 65, 'B' is 66)
- Adding characters adds their ASCII values (e.g., 'A' + 'B' = 65 + 66 = 131)

Important Considerations

- Overflow can occur if results exceed the char range
- Format specifier %d prints the numeric value
- Format specifier %c prints the character representation
- Operations maintain ASCII ordering (e.g., 'b' - 1 = 'a')

Type Conversion in C

Last Updated : 11 Oct, 2024

Type conversion in C is the process of converting one data type to another. The type conversion is only performed to those data types where conversion is possible. Type conversion is performed by a compiler. In type conversion, the destination data type can't be smaller than the source data type. Type conversion is done at compile time and it is also called widening conversion because the destination data type can't be smaller than the source data type. There are two types of Conversion:

1. Implicit Type Conversion

!<https://media.geeksforgeeks.org/wp-content/cdn-uploads/Implicit-Type-Conversion-in-c.png>

Also known as 'automatic type conversion'.

- A. Done by the compiler on its own, without any external trigger from the user.
- B. Generally takes place when in an expression more than one data type is present. In such conditions type conversion (type promotion) takes place to avoid loss of data.

Implicit type conversion in C, also known as automatic type conversion, follows a hierarchical promotion system where smaller data types are automatically converted to larger ones. Here's a comprehensive explanation:

Type Promotion Hierarchy

From smallest to largest:

bool → char → short int → int → unsigned int → long → unsigned long → l

Common Occurrences

Assignment Operations

```
int x = 10;
char y = 'a';
x = x + y;    // y is promoted to int (ASCII value 97)
float z = x + 1.0; // x is promoted to float
```

Binary Operations

- When operands have different types, the smaller type is promoted to the larger type
- Integer types smaller than int are always promoted to int first
- In floating-point operations, integers are converted to floating-point

Important Considerations

Potential Issues

- Information loss can occur during conversion
- Sign loss when converting signed to unsigned values
- Overflow possible when converting long to float
- Precision loss when converting floating-point to integer

Benefits

- Ensures type safety in operations
- Improves code readability
- Enables compatibility between different data types

Limitations

- Can lead to unexpected results if not carefully managed
- May cause performance overhead
- Potential precision loss in numeric conversions

Example no 1

```
// An example of implicit conversion
#include <stdio.h>
int main()
{
    int x = 10; // integer x
    char y = 'a'; // character c

    // y implicitly converted to int. ASCII
    // value of 'a' is 97
    x = x + y;
```

```

    // x is implicitly converted to float
    float z = x + 1.0;

    printf("x = %d, z = %f", x, z);
    return 0;
}

```

Output

```
x = 107, z = 108.000000
```

2. Explicit Type Conversion

!https://media.geeksforgeeks.org/wp-content/cdn-uploads/Explicit-Type-Conversion.png

This process is also called type casting and it is user-defined. Here the user can typecast the result to make it of a particular data type. The syntax in C Programming:

```
(type) expression
```

Type indicated the data type to which the final result is converted.

Example no 2

```

// C program to demonstrate explicit type casting
#include<stdio.h>

int main()
{
    double x = 1.2;

    // Explicit conversion from double to int
    int sum = (int)x + 1;

    printf("sum = %d", sum);

    return 0;
}

```

Output

```
sum = 2
```

Example no 3

```

#include <stdio.h>

int main() {
    float a = 1.5;
    int b = (int)a;

    printf("a = %f\n", a);
}

```

```
printf("b = %d\n", b);

return 0;
}
```

Output

```
a = 1.500000
b = 1
```

Advantages of Type Conversion

- **Type safety:** Type conversions can be used to ensure that data is being stored and processed in the correct data type, avoiding potential type mismatches and type errors.
- **Improved code readability:** By explicitly converting data between different types, you can make the intent of your code clearer and easier to understand.
- **Improved performance:** In some cases, type conversions can be used to optimize the performance of your code by converting data to a more efficient data type for processing.
- **Improved compatibility:** Type conversions can be used to convert data between different types that are not compatible, allowing you to write code that is compatible with a wider range of APIs and libraries.
- **Improved data manipulation:** Type conversions can be used to manipulate data in various ways, such as converting an integer to a string, converting a string to an integer, or converting a floating-point number to an integer.
- **Improved data storage:** Type conversions can be used to store data in a more compact form, such as converting a large integer value to a smaller integer type, or converting a large floating-point value to a smaller floating-point type.

Disadvantages of type conversions in C programming:

- **Loss of precision:** Converting data from a larger data type to a smaller data type can result in loss of precision, as some of the data may be truncated.
- **Overflow or underflow:** Converting data from a smaller data type to a larger data type can result in overflow or underflow if the value being converted is too large or too small for the new data type.
- **Unexpected behavior:** Type conversions can lead to unexpected behavior, such as when converting between signed and unsigned integer types, or when converting between floating-point and integer types.
- **Confusing syntax:** Type conversions can have confusing syntax, particularly when using typecast operators or type conversion functions, making the code more difficult to read and understand.
- **Increased complexity:** Type conversions can increase the complexity of your code, making it harder to debug and maintain.
- **Slower performance:** Type conversions can sometimes result in slower performance, particularly when converting data between complex data types, such as between structures and arrays.

Basic Input and Output in C

C language provides standard libraries for input and output operations through the `stdio.h` header file.

scanf() Function

The `scanf()` function reads values from the console based on specified format specifiers.

Syntax:

```
scanf("%X", &variableOfXType);
```

Where:

- `%X` is the format specifier
- `&` is the address operator
- `variableOfXType` is the variable where the input will be stored

printf() Function

The `printf()` function displays values on the console screen.

Syntax:

```
printf("%X", variableOfXType);
```

Basic Data Type Input/Output

Integer:

```
scanf("%d", &intVariable);    // Input
printf("%d", intVariable);    // Output
```

Float:

```
scanf("%f", &floatVariable);  // Input
printf("%f", floatVariable);  // Output
```

Character:

```
scanf("%c", &charVariable);   // Input
printf("%c", charVariable);   // Output
```

Example Program for Basic Types

```
#include <stdio.h>

int main()
{
    // Variable declarations
```

```

int num;
char ch;
float f;

// Integer input/output
printf("Enter the integer: ");
scanf("%d", &num);
printf("\nEntered integer is: %d", num);

// Float input/output
while((getchar()) != '\\n'); // Clear buffer
printf("\n\\nEnter the float: ");
scanf("%f", &f);
printf("\nEntered float is: %f", f);

// Character input/output
printf("\n\\nEnter the Character: ");
scanf("%c", &ch);
printf("\nEntered character is: %c", ch);

return 0;
}

```

String Input/Output

For strings, the format specifier is %s.

Syntax:

```

scanf("%s", stringVariable); // Input
printf("%s", stringVariable); // Output

```

Example Program for Strings

```

#include <stdio.h>

int main()
{
    char str[50];

    // Reading a word
    printf("Enter the Word: ");
    scanf("%s\\n", str);
    printf("\\nEntered Word is: %s", str);

    // Reading a sentence
    printf("\\n\\nEnter the Sentence: ");
    scanf("%[^\\n]s", str);
    printf("\\nEntered Sentence is: %s", str);

    return 0;
}

```

```
}
```

Sample Output:

```
Enter the integer: 10
Entered integer is: 10
```

```
Enter the float: 2.5
Entered float is: 2.500000
```

```
Enter the Character: A
Entered Character is: A
```

```
Enter the Word: GeeksForGeeks
Entered Word is: GeeksForGeeks
```

```
Enter the Sentence: Geeks For Geeks
Entered Sentence is: Geeks For Geeks
```

Format specifiers in C are essential components used to specify data types in input/output operations. They begin with a % symbol and are used in functions like `printf()` and `scanf()` [1] .

Basic Format Specifiers

Specifier	Purpose
%c	Character type
%d, %i	Signed integer
%f	Float type
%s	String
%p	Pointer
%u	Unsigned integer
%x, %X	Hexadecimal
%o	Octal
%e, %E	Scientific notation

Advanced Format Specifiers

Specifier	Purpose
-----------	---------

%lf	Double
%Lf	Long double
%ld, %li	Long integer
%llu	Unsigned long long
%lli, %lld	Long long

Usage Examples

Character Input/Output:

```
char c;  
scanf("%c", &c);  
printf("%c", c);
```

Integer Formatting:

```
int x = 45;  
printf("Decimal: %d\\n", x);  
printf("Hexadecimal: %x\\n", x);
```

Floating Point:

```
float num = 12.67;  
printf("Float: %f\\n", num);  
printf("Scientific: %e\\n", num);
```

Formatting Options

- Use minus(-) for left alignment[7]
- Add number after % for minimum field width[7]
- Use period(.) to separate field width from precision[7]
- Precision specifies:

- Minimum digits in integers
- Maximum characters in strings
- Decimal places in floating points[7]

Printf is a fundamental C function used for formatted output to stdout. It returns the number of characters printed, or a negative value if an error occurs[8] .

Basic Syntax

```
int printf(const char *format, argument1, argument2, ...);
```

Format Specifiers

Specifier	Purpose
%d	Integer
%f	Float
%c	Character
%s	String
%x	Hexadecimal
%p	Pointer
%o	Octal
%e	Scientific notation

Formatting Options

Width and Precision:

```
printf("%10d", num);    // Minimum width of 10
printf("%.2f", num);    // 2 decimal places
printf("%10.2f", num);  // Width 10, 2 decimal places
```

Flags:

- `l` for left alignment
- `0` for zero padding
- `+` for forcing plus sign
- (space) for space before positive numbers

Examples

Basic Output:

```
int age = 25;
printf("Age: %d", age);
```

Multiple Values:

```
float height = 1.75;
char grade = 'A';
printf("Height: %.2f, Grade: %c", height, grade);
```

Return Value

The printf function returns:

- Number of characters printed on success[3]
- Negative value on error[8]

For example:

```
int count = printf("Hello"); // Returns 5
```

scanf in C

Overview

`scanf` is a function in the C programming language used to read data from the standard input stream (typically the keyboard). It allows developers to accept various types of input, including characters, strings, and numeric data, using format specifiers similar to `printf()`.

Syntax

```
int scanf(const char *format, ...);
```

- `int` is the return type
- `format` is a string containing format specifiers
- `...` indicates a variable number of arguments

Format Specifiers

Common format specifiers include:

- `%d`: Integers
- `%ld`: Long integers
- `%lld`: Long long integers
- `%f`: Real numbers
- `%c`: Characters
- `%s`: Strings

Return Values

`scanf()` can return three types of values:

- `> 0`: Number of values successfully converted and assigned
- `0`: No value assigned
- `< 0`: Read error or end-of-file (EOF) reached before assignment

The `&` Operator

The `&` (address of) operator is crucial in `scanf()` as it provides the memory location where the input will be stored.

Example Code

```
#include <stdio.h>

int main() {
    int a, b;

    printf("Enter first number: ");
    scanf("%d", &a);

    printf("Enter second number: ");
    scanf("%d", &b);

    printf("A : %d \\t B : %d", a, b);

    return 0;
}
```

Output

```
Enter first number: 5
Enter second number: 6
A : 5      B : 6
```

Additional Resources

For more in-depth learning about input handling and data structures, consider exploring [online C Programming courses](#).

Scansets in C

Introduction

Scansets are a powerful feature of the `scanf()` family of functions in C, providing a flexible way to control input reading. They are represented by the `%[]` format specifier and allow developers to precisely define which characters can be read from input.

Basic Scanset Syntax

A scanset is defined by placing characters inside square brackets `[]`. The function will only process characters that are part of this specified set.

Key Characteristics

- Scansets are case-sensitive
- You can specify individual characters or ranges of characters
- Multiple characters can be included using commas

Simple Scanset Example

```
#include <stdio.h>

int main(void)
{
    char str[128];

    printf("Enter a string: ");
    scanf("%[A-Z]s", str); // Only capital letters will be read

    printf("You entered: %s\\n", str);

    return 0;
}
```

In this example, if you enter "GEEKs_for_geeks", only "GEEK" will be stored in `str`.

Advanced Scanset Features: The Caret (^) Operator

When the first character in a scanset is `^`, the input reading stops at the first occurrence of that character.

Example

```
#include <stdio.h>

int main(void)
{
    char str[128];

    printf("Enter a string: ");
    scanf("%[^o]s", str); // Read until 'o' is encountered

    printf("You entered: %s\\n", str);

    return 0;
}
```

If you enter "<http://geeks> for geeks", the output will be "<http://geeks> f".

Implementing gets() Functionality

You can use scansets to read a full line of text, including spaces:

```
#include <stdio.h>

int main(void)
{
    char str[128];
```



```
printf("Enter a string with spaces: ");
scanf("%[^\n]s", str); // Read until newline

printf("You entered: %s\n", str);

return 0;
}
```

Important Caution

While the above example demonstrates reading a line, it's crucial to note the historical warning about `gets()`:

Warning: `gets()` is considered dangerous and should never be used due to potential buffer overflow risks. Instead, use `fgets()` for safer input handling.

Best Practices

1. Always specify a maximum field width to prevent buffer overflows
2. Use `fgets()` for line input when possible
3. Be aware of the case-sensitivity of scansets
4. Understand that scansets provide precise character filtering

Conclusion

Scansets offer a powerful, flexible method for controlling input in C, allowing developers to create more robust and precise input parsing mechanisms. By understanding and carefully using scansets, you can create more controlled and secure input handling routines.

Formatted and Unformatted Input/Output Functions in C

Overview of I/O Functions

In C programming, input/output (I/O) functions are essential for interacting with users and managing data. They are categorized into two main types:

1. Formatted I/O Functions
2. Unformatted I/O Functions

Formatted I/O Functions

Characteristics

- Allow input and output in user-specified formats
- Support multiple data types

- Use format specifiers for precise control

Common Format Specifiers

Format Specifier	Data Type	Description
%d	int/signed int	Signed integer value
%c	char	Character value
%f	float	Decimal floating-point value
%s	string	Group of characters
%ld	long int	Long signed integer
%u	unsigned int	Unsigned integer
%lf	double	Fractional or floating data

Key Formatted I/O Functions

1. printf()

Displays values on the console screen.

```
#include <stdio.h>

int main() {
    int a = 20;
    // Displaying variable value
    printf("%d", a);

    // Displaying static text
    printf("This is a string");

    return 0;
}
```

2. scanf()

Reads input from the keyboard, using the address-of & operator.

```
#include <stdio.h>

int main() {
    int num;
    printf("Enter a number: ");
    scanf("%d", &num);
    printf("You entered: %d", num);

    return 0;
}
```

3. sprintf()

Writes formatted output to a string instead of the console.

```
#include <stdio.h>

int main() {
    char str[50];
    int a = 2, b = 8;

    // Store formatted string in character array
    sprintf(str, "%d and %d are even numbers", a, b);
    printf("%s", str);

    return 0;
}
```

4. sscanf()

Reads formatted input from a string.

```
#include <stdio.h>

int main() {
    char str[50];
    int a, b;

    // Create a string with formatted data
    sprintf(str, "a = %d and b = %d", 2, 8);

    // Extract data from the string
    sscanf(str, "a = %d and b = %d", &a, &b);
    printf("a = %d and b = %d", a, b);

    return 0;
}
```

Unformatted I/O Functions

Characteristics

- Work only with character and string data types
- No format specifiers
- Limited flexibility

Key Unformatted I/O Functions

1. getch()

Reads a single character without displaying it.

```
#include <conio.h>
```

```
#include <stdio.h>

int main() {
    printf("Enter a character: ");
    getch(); // Reads without display
    return 0;
}
```

2. getche()

Reads and immediately displays a character.

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

int main() {
    printf("Enter a character: ");
    getche(); // Reads and displays
    return 0;
}
```

3. getchar()

Reads a single character after Enter key.

```
#include <stdio.h>

int main() {
    char ch;
    printf("Enter a character: ");
    ch = getchar();
    printf("You entered: %c", ch);
    return 0;
}
```

Comparison: Formatted vs Unformatted I/O

Aspect	Formatted I/O	Unformatted I/O
Format Control	Supports format specifiers	No format specifiers
Data Types	All data types	Only characters/strings
Flexibility	High	Limited
User-Friendliness	More user-friendly	Less user-friendly

Important Considerations

- Avoid deprecated functions like `gets()` due to security risks
- Prefer `fgets()` for safer string input

- Choose I/O functions based on specific requirements

Warnings

- `gets()` is considered dangerous and should never be used
- Always prioritize input validation and buffer safety
- Use modern, secure input methods

Operators in C

Overview of Operators in C

Operators in C are symbols that perform specific operations on one or more operands. They are essential in writing efficient and effective programs. Here's a comprehensive breakdown of operators in C:

Types of Operators

1. Arithmetic Operators

Used for mathematical operations.

Examples: `+`, `-`, `*`, `/`, `%`

2. Relational Operators

Used for comparison between operands.

Examples: `<`, `>`, `<=`, `>=`, `==`, `!=`

3. Logical Operators

Used to combine or invert boolean expressions.

Examples: `&&`, `||`, `!`

4. Bitwise Operators

Operate at the binary level.

Examples: `&`, `|`, `^`, `~`, `<<`, `>>`

5. Assignment Operators

Assign values to variables, often combining assignment with other operations.

Examples: `=`, `+=`, `-=`, `*=`, `/=`, `%=`

6. Miscellaneous Operators

Include special-purpose operators like `sizeof`, `?:`, `,`, `&`, `*`, `->`, and `.`

Key Operator Examples

Arithmetic Operators

```
int a = 10, b = 3;
printf("Addition: %d\n", a + b); // Output: 13
printf("Modulo: %d\n", a % b);   // Output: 1
```

Relational Operators

```
int x = 5, y = 10;
printf("x < y: %d\n", x < y);    // Output: 1 (true)
printf("x == y: %d\n", x == y);  // Output: 0 (false)
```

Logical Operators

```
int p = 1, q = 0;
printf("p && q: %d\n", p && q);   // Output: 0 (false)
printf("p || q: %d\n", p || q);  // Output: 1 (true)
```

Bitwise Operators

```
int num1 = 5, num2 = 3;
printf("Bitwise AND: %d\n", num1 & num2); // Output: 1
printf("Left shift: %d\n", num1 << 1);    // Output: 10
```

Assignment Operators

```
int z = 5;
z += 10; // Same as z = z + 10
printf("z: %d\n", z); // Output: 15
```

Miscellaneous Operators

- **sizeof**: Returns the size of a data type.
- Ternary Operator (**? :**):

```
int result = (a > b) ? a : b;
printf("Max: %d\n", result);
```

Operator Precedence

The precedence determines the order of evaluation in complex expressions. For example:

```
int result = 10 + 5 * 2; // Multiplication (*) has higher precedence t
```

```
printf("Result: %d\n", result); // Output: 20
```

Unary, Binary, and Ternary Operators

- Unary: Operate on one operand. Example: `x`, `!x`
 - Binary: Operate on two operands. Example: `x + y`, `x && y`
 - Ternary: Operate on three operands. Example: `x ? y : z`
-

Common Questions

1. Difference between `=` and `==`

- `=`: Assignment operator.
- `==`: Relational operator for equality comparison.

4. Prefix vs. Postfix Increment

- Prefix (`++x`): Increments first, then evaluates.
 - Postfix (`x++`): Evaluates first, then increments.
-

Arithmetic Operators in C

Arithmetic operators in C are fundamental tools used to perform mathematical operations in a program. They allow for defining expressions and computing mathematical formulas. These operators are divided into two categories: binary arithmetic operators and unary arithmetic operators, based on the number of operands they require.

Binary Arithmetic Operators in C

Binary arithmetic operators operate on two operands. These include:

Operator	Name	Operation	Syntax
<code>+</code>	Addition	Adds two operands.	<code>x + y</code>
<code>-</code>	Subtraction	Subtracts the second operand from the first.	<code>x - y</code>
<code>*</code>	Multiplication	Multiplies two operands.	<code>x * y</code>
<code>/</code>	Division	Divides the first operand by the second.	<code>x / y</code>
<code>%</code>	Modulus	Returns the remainder of the division.	<code>x % y</code>

Example: Binary Arithmetic Operators

```
#include <stdio.h>

int main() {
    int a = 10, b = 4, res;

    printf("a is %d and b is %d\n", a, b);
```

```

    res = a + b; // Addition
    printf("a + b is %d\n", res);

    res = a - b; // Subtraction
    printf("a - b is %d\n", res);

    res = a * b; // Multiplication
    printf("a * b is %d\n", res);

    res = a / b; // Division
    printf("a / b is %d\n", res);

    res = a % b; // Modulus
    printf("a %% b is %d\n", res);

    return 0;
}

```

Output:

```

a is 10 and b is 4
a + b is 14
a - b is 6
a * b is 40
a / b is 2
a % b is 2

```

Unary Arithmetic Operators in C

Unary arithmetic operators operate on a single operand. These include:

Operator	Name	Operation	Syntax
++	Increment Operator	Increases the value of a variable by 1.	++x or x++
--	Decrement Operator	Decreases the value of a variable by 1.	--x or x--
+	Unary Plus	Returns the value of its operand.	+x
-	Unary Minus	Returns the negative value of its operand.	-x

Example: Unary Arithmetic Operators

```

#include <stdio.h>

int main() {
    int a = 10, res;

    // Post-increment
    res = a++;
    printf("Post-Increment: a is %d, result is %d\n", a, res);

    // Post-decrement

```



```

    res = a--;
    printf("Post-Decrement: a is %d, result is %d\n", a, res);

    // Pre-increment
    res = ++a;
    printf("Pre-Increment: a is %d, result is %d\n", a, res);

    // Pre-decrement
    res = --a;
    printf("Pre-Decrement: a is %d, result is %d\n", a, res);

    return 0;
}

```

Output:

```

Post-Increment: a is 11, result is 10
Post-Decrement: a is 10, result is 11
Pre-Increment: a is 11, result is 11
Pre-Decrement: a is 10, result is 10

```

Operator Precedence and Associativity

When multiple operators are used in a single expression, the order of evaluation is determined by operator precedence and associativity.

Example: Expression with Multiple Operators

```

#include <stdio.h>

int main() {
    int var;
    var = 10 * 20 + 15 / 5; // Expression with multiple operators
    printf("Result = %d", var);
    return 0;
}

```

Output:

```
Result = 203
```

Explanation: The expression is evaluated as

```
(10 * 20) + (15 / 5)
```

Example: Application of Arithmetic Operators

Calculate the Area and Perimeter of a Rectangle

```
#include <stdio.h>

int main() {
    int length = 10, breadth = 5;
    int area, perimeter;

    area = length * breadth; // Area calculation
    perimeter = 2 * (length + breadth); // Perimeter calculation

    printf("Area = %d\nPerimeter = %d", area, perimeter);
    return 0;
}
```

Output:

```
Area = 50
Perimeter = 30
```

FAQs on C Arithmetic Operators

1. List all arithmetic operators in C.

- Binary: +, -, *, /, %
- Unary: ++, --, ~, &&

4. Difference between unary minus and subtraction operators:

- Unary minus (`-x`) works on a single operand, negating its value.
- Subtraction (`x - y`) works on two operands, returning their difference.

Arithmetic operators are essential for mathematical operations, and mastering them builds a solid foundation in C programming.

Unary Operators in C

Last Updated: 11 Oct, 2024

Introduction

Unary operators are operators that perform operations on a single operand to produce a new value.

Types of Unary Operators

1. Unary minus (-)

2. Increment (+ +)
3. Decrement (--)
4. NOT (!)
5. Address-of operator (&)
6. sizeof()

1. Unary Minus (-)

Changes the sign of its argument:

```
int a = 10;  
int b = -a;    // b = -10
```

2. Increment (+ +)

Two types:

2.1 Prefix Increment

- Operator precedes operand (+ + a)
- Value altered before use

```
int a = 1;  
int b = ++a;    // b = 2
```

2.2 Postfix Increment

- Operator follows operand (a+ +)
- Value altered after use

```
int a = 1;  
int b = a++;    // b = 1  
int c = a;      // c = 2
```

3. Decrement (--)

Two types:

3.1 Prefix Decrement

- Operator precedes operand (--a)
- Value altered before use

```
int a = 1;  
int b = --a;    // b = 0
```

3.2 Postfix Decrement

- Operator follows operand (a--)

- Value altered after use

```
int a = 1;
int b = a--;    // b = 1
int c = a;      // c = 0
```

4. NOT (!)

- Reverses the logical state of its operand
- If x is true, !x is false
- If x is false, !x is true

5. Address-of Operator (&)

- Returns memory address of a variable
- Used with pointers

```
int a;
int *ptr;
ptr = &a; // address of a is copied to ptr
```

6. sizeof()

- Returns size of operand in bytes
- Machine dependent
- Example:

```
printf("Size of double: %d\\n", sizeof(double)); // typically 8
printf("Size of int: %d\\n", sizeof(int));        // typically 4
```

Example Programs

Various code examples are provided in the original text demonstrating each operator's usage, including:

- Increment/decrement operations
- Logical NOT operations
- Address-of operator usage
- sizeof() operator implementation

Note: The exact size returned by sizeof() may vary depending on the machine architecture (32-bit vs 64-bit systems).

Relational Operators in C

Relational operators in C are used to compare two values, determining the relationship between them. The result of a relational operation is a boolean value: true (non-zero) or false (zero). These operators are commonly used in conditional statements and loops.

Types of Relational Operators in C

There are six relational operators in C:

Operator	Name	Description	Example Result
==	Equal to	Checks if two operands are equal.	5 == 5 true
!=	Not equal to	Checks if two operands are not equal.	5 != 5 false
>	Greater than	Checks if the first operand is greater than the second operand.	6 > 5 true
<	Less than	Checks if the first operand is less than the second operand.	6 < 5 false
>=	Greater than or equal to	Checks if the first operand is greater than or equal to the second operand.	5 >= 5 true
<=	Less than or equal to	Checks if the first operand is less than or equal to the second operand.	5 <= 5 true

Examples of Relational Operators

Example: Using Relational Operators

```
#include <stdio.h>

int main() {
    int a = 10, b = 4;

    // Greater than
    if (a > b)
        printf("a is greater than b\n");
    else
        printf("a is less than or equal to b\n");

    // Greater than or equal to
    if (a >= b)
        printf("a is greater than or equal to b\n");
    else
        printf("a is lesser than b\n");

    // Less than
    if (a < b)
        printf("a is less than b\n");
    else
        printf("a is greater than or equal to b\n");

    // Less than or equal to
    if (a <= b)
        printf("a is lesser than or equal to b\n");
    else
        printf("a is greater than b\n");

    // Equal to
```

```

    if (a == b)
        printf("a is equal to b\n");
    else
        printf("a and b are not equal\n");

    // Not equal to
    if (a != b)
        printf("a is not equal to b\n");
    else
        printf("a is equal to b\n");

    return 0;
}

```

Output:

```

a is greater than b
a is greater than or equal to b
a is greater than or equal to b
a is greater than b
a and b are not equal
a is not equal to b

```

Applications

1. Conditional Statements: Relational operators are essential in `if`, `else if`, and `else` statements to evaluate conditions.
 2. Loops: These operators help determine loop termination conditions in `for`, `while`, and `do-while` loops.
-

Key Points

1. Result Type: Relational operators always return a boolean value: `true` (non-zero) or `false` (zero).
 2. Binary Operators: All relational operators are binary, meaning they require two operands.
 3. Use with Different Data Types: Relational operators can be used with integers, floating-point numbers, and characters, but not directly with strings or arrays.
 4. Operator Precedence: Relational operators have lower precedence than arithmetic operators but higher precedence than logical operators.
-

FAQs

1. What is the difference between `==` and `=` in C?
 - `==` is a relational operator used for comparison.
 - `=` is an assignment operator used to assign values to variables.

4. Can relational operators be used with strings?

- No, relational operators cannot directly compare strings in C. For string comparison, functions like `strcmp()` from `<string.h>` are used.

6. What happens if two floating-point numbers are compared using relational operators?

- The result depends on their values. Due to floating-point precision issues, results might not always be as expected.

Bitwise Operators in C

Bitwise operators in C perform operations directly on bits, offering a low-level way to manipulate data. These operators are often used for tasks such as setting flags, optimizing code, and implementing algorithms requiring direct hardware control.

List of Bitwise Operators

Operator	Name	Description
&	Bitwise AND	Performs AND operation on each pair of corresponding bits. Result is 1 if both bits are 1.
	Bitwise OR	Performs OR operation on each pair of corresponding bits. Result is 1 if at least one bit is 1.
^	Bitwise XOR	Performs XOR operation. Result is 1 if corresponding bits are different.
~	Bitwise NOT	Inverts all bits of the operand (one's complement).
<<	Left Shift	Shifts bits of the first operand left by the number of positions specified in the second operand.
>>	Right Shift	Shifts bits of the first operand right by the number of positions specified in the second operand.

Truth Table for Bitwise Operations

X	Y	X & Y	X Y	X ^ Y	~X	~Y	X & ~Y	~X & Y	X ^ ~Y	~X ^ Y
0	0	0	0	0	1	1	0	1	1	1
0	1	0	1	1	1	0	1	0	0	1
1	0	0	1	1	0	1	0	1	0	0
1	1	1	1	0	0	0	0	0	0	0

Examples of Bitwise Operators in C

Basic Operations

```
#include <stdio.h>
int main() {
    unsigned int a = 5, b = 9; // Binary: a = 00000101, b = 00001001

    printf("a & b = %u\n", a & b); // AND: 00000001
    printf("a | b = %u\n", a | b); // OR:  00001101
}
```

```

printf("a ^ b = %u\n", a ^ b); // XOR: 00001100
printf("~a = %u\n", ~a);        // NOT: 11111111111111111111111111111111
printf("b << 1 = %u\n", b << 1); // Left shift: 00010010
printf("b >> 1 = %u\n", b >> 1); // Right shift: 00000100

return 0;
}

```

Output:

```

a & b = 1
a | b = 13
a ^ b = 12
~a = 4294967290
b << 1 = 18
b >> 1 = 4

```

Interesting Facts

1. Left Shift and Right Shift Operations

- << is equivalent to multiplying by 2^n .
- >> is equivalent to dividing by 2^n (only for positive numbers).

Example:

```

int x = 19;
printf("x << 1 = %d\n", x << 1); // Result: 38
printf("x >> 1 = %d\n", x >> 1); // Result: 9

```

4. Checking Odd or Even

The expression $(x \& 1)$ is non-zero if x is odd and zero if x is even.

Example:

```

int x = 19;
printf("%s\n", (x & 1) ? "Odd" : "Even");

```

Output: Odd

5. Efficient XOR Usage

The XOR operator is useful for finding unique elements in a set.

Problem: Find the element in an array where all other elements appear twice.

Example:

```

int findOdd(int arr[], int n) {

```



```

    int res = 0;
    for (int i = 0; i < n; i++)
        res ^= arr[i];
    return res;
}

```

6. Bitwise Operators vs Logical Operators

Logical operators (&, ||, !) always return 0 or 1. Bitwise operators work on each bit and can return any integer.

Example:

```

int x = 2, y = 5;
printf("%s ", (x & y) ? "True" : "False"); // False
printf("%s\n", (x && y) ? "True" : "False"); // True

```

7. Caution with ~ Operator

The result of ~ depends on whether the variable is signed or unsigned.

Example:

```

unsigned int x = 1;
printf("Signed Result: %d\n", ~x); // Signed: -2
printf("Unsigned Result: %u\n", ~x); // Unsigned: 4294967294

```

Common Problems Using Bitwise Operators

1. Find the missing number in an array.
2. Swap two numbers without using a temporary variable.
3. Add two numbers without using arithmetic operators.
4. Check if two integers have opposite signs.

Logical Operators in C

Logical operators are used in C to combine conditions for decision-making. They return either 1 (true) or 0 (false), depending on the evaluation result.

Types of Logical Operators

1. Logical AND (&&)

- Returns true if both operands are non-zero.
- Returns false if any operand is zero.
- Syntax: (operand1 && operand2)

- Truth Table:

X	Y	X && Y
1	1	1
1	0	0
0	1	0
0	0	0

Example:

```
#include <stdio.h>
int main() {
    int a = 10, b = 20;
    if (a > 0 && b > 0) {
        printf("Both values are greater than 0\n");
    }
    return 0;
}
```

Output:

Both values are greater than 0

1. Logical OR (||)

- Returns true if any one operand is non-zero.
- Returns false only if both operands are zero.
- Syntax: (operand1 || operand2)
- Truth Table:

X	Y	X Y
1	1	1
1	0	1
0	1	1
0	0	0

Example:

```
#include <stdio.h>
int main() {
    int a = -1, b = 20;
    if (a > 0 || b > 0) {
        printf("At least one value is greater than 0\n");
    }
    return 0;
}
```

Output:

At least one value is greater than 0

1. Logical NOT (!)

- Inverts the result of the operand.
- Returns true if operand is zero.
- Returns false if operand is non-zero.
- Syntax: !(operand)
- Truth Table:

X	!X
1	0
0	1

Example:

```
#include <stdio.h>
int main() {
    int a = 10;
    if (!(a > 0)) {
        printf("Condition is false\n");
    } else {
        printf("Condition is true\n");
    }
    return 0;
}
```

Output:

Condition is true

Short-Circuiting in Logical Operators

Logical operators in C short-circuit the evaluation when the result can be determined early.

1. Short-Circuiting with &&

If the first operand is false, subsequent operands are not evaluated because the result will always be false.

Example:

```
int x = -5;
if (x > 0 && x % 2 == 0) {
    printf("Condition satisfied\n");
} else {
    printf("Condition not satisfied\n");
}
```

Output:

Condition not satisfied

2. Short-Circuiting with ||

If the first operand is true, subsequent operands are not evaluated because the result will always be true.

Example:

```
int y = 10;
if (y > 0 || y % 2 == 0) {
    printf("At least one condition satisfied\n");
}
```

Output:

At least one condition satisfied

Common Questions

1. Precedence of Logical Operators

The precedence is:

- ! (Highest)
- &&
- || (Lowest)

Use parentheses to explicitly define the evaluation order.

1. Chaining Logical Operators Logical operators can be chained to evaluate multiple conditions, e.g., (a > 0 && b > 0) || c < 0.
-

Interesting Cases

1. Code Example 1:

```
#include <stdio.h>
void main() {
    int a = 1, b = 0, c = 5;
    int d = a && b || c++;
    printf("%d", c);
}
```

Output:

2. Code Example 2:

```
#include <stdio.h>
int main() {
    int i = 1;
    if (i++ && (i == 1)) {
        printf("GeeksforGeeks\n");
    } else {
        printf("Coding\n");
    }
    return 0;
}
```

Output:

Coding

Logical operators are a foundational concept in C programming, and understanding their behavior—including short-circuiting—is critical for building efficient conditional statements.

Assignment Operators in C

Assignment operators are used to assign values to variables. In C, the left-hand operand must be a variable, while the right-hand operand is the value or expression to be assigned.

Types of Assignment Operators

1. = (Simple Assignment Operator):

- Assigns the value on the right to the variable on the left.
- Example:

```
int a = 10;
char ch = 'y';
```

- Explanation: Assigns 10 to a and 'y' to ch.
-

1. += (Add and Assign Operator):

- Adds the current value of the variable on the left to the value on the right, then assigns the result to the variable.
- Syntax: `a += b` equivalent to `a = a + b`.
- Example:

```
int a = 5;
a += 6;    // Now a = 5 + 6 = 11
```

1. = (Subtract and Assign Operator):

- Subtracts the value on the right from the current value of the variable on the left, then assigns the result to the variable.
- Syntax: `a -= b` equivalent to `a = a - b`.
- Example:

```
int a = 8;  
a -= 6; // Now a = 8 - 6 = 2
```

1. = (Multiply and Assign Operator):

- Multiplies the current value of the variable on the left by the value on the right, then assigns the result to the variable.
- Syntax: `a *= b` equivalent to `a = a * b`.
- Example:

```
int a = 5;  
a *= 6; // Now a = 5 * 6 = 30
```

1. /= (Divide and Assign Operator):

- Divides the current value of the variable on the left by the value on the right, then assigns the result to the variable.
- Syntax: `a /= b` equivalent to `a = a / b`.
- Example:

```
int a = 6;  
a /= 2; // Now a = 6 / 2 = 3
```

Example Program:

Here is a C program demonstrating various assignment operators:

```
#include <stdio.h>  
  
int main() {  
    int a = 10; // Using "=" operator  
    printf("Value of a is %d\n", a);  
  
    a += 10; // Using "+=" operator
```

```
printf("Value of a is %d\n", a);

a -= 10; // Using "--" operator
printf("Value of a is %d\n", a);

a *= 10; // Using "*=" operator
printf("Value of a is %d\n", a);

a /= 10; // Using "/=" operator
printf("Value of a is %d\n", a);

return 0;
}
```

Output:

```
Value of a is 10
Value of a is 20
Value of a is 10
Value of a is 100
Value of a is 10
```

Key Points:

- Assignment operators allow for shorthand operations, saving time and improving code readability.
 - The right-hand side must be compatible with the data type of the left-hand variable.
 - Using compound assignment operators (+, =, etc.) ensures consistency and simplifies expressions.
-

Increment and Decrement Operators in C

Last Updated: 28 Aug, 2023

Introduction

The increment (+ +) and decrement (--) operators are unary operators used to increase or decrease numeric values by 1. These operations are commonly used in:

- Looping
- Array traversal
- Pointer arithmetic

Increment Operator (+ +)

Syntax

```
// Prefix
++m
// Postfix
m++
```

Types of Increment

1. Pre-Increment (Prefix)

- Operator used as prefix
- Value incremented first

```
result = ++var1;
// Equivalent to:
var = var + 1;
result = var;
```

2. Post-Increment (Postfix)

- Operator used as suffix
- Increment performed after other operations

```
result = var1++;
// Equivalent to:
result = var;
var = var + 1;
```

Decrement Operator (--)

Syntax

```
// Prefix
--m
// Postfix
m--
```

Types of Decrement

1. Pre-Decrement

- Decreases value immediately

```
result = --m;
// Equivalent to:
m = m - 1;
result = m;
```


2. Post-Decrement

- Value decreased after expression evaluation

```
result = m--;  
// Equivalent to:  
result = m;  
m = m - 1;
```

Differences Between Increment and Decrement Operators

Increment Operator	Decrement Operator
Adds 1 to operand	Subtracts 1 from operand
Postfix: evaluates expression first, then increments	Postfix: evaluates expression first, then decrements
Prefix: increments first, then evaluates expression	Prefix: decrements first, then evaluates expression
Common in decision-making and looping	Common in decision-making and looping

Example Code

```
#include <stdio.h>  
  
void increment()  
{  
    int a = 5;  
    int b = 5;  
  
    // PREFIX  
    int prefix = ++a;  
    printf("Prefix Increment: %d\\n", prefix);  
  
    // POSTFIX  
    int postfix = b++;  
    printf("Postfix Increment: %d", postfix);  
}  
  
int main()  
{  
    increment();  
    return 0;  
}
```

Output:

```
Prefix Increment: 6  
Postfix Increment: 5
```

Note: Post-increment has higher precedence than pre-increment as it is a postfix operator, while pre-increment is categorized as a unary operator.

Conditional or Ternary Operator (?:) in C

The conditional or ternary operator in C is a concise way to perform conditional checks, functioning similarly to an `if-else` statement but using less space. It operates on three operands, hence the name "ternary operator."

Syntax of the Conditional/Ternary Operator

The conditional operator can take any of the following forms:

1. Basic Syntax:

```
variable = Expression1 ? Expression2 : Expression3;
```

2. With Parentheses:

```
variable = (condition) ? Expression2 : Expression3;
```

3. Variable Assignment Inside Conditional:

```
(condition) ? (variable = Expression2) : (variable = Expression3);
```

How it Translates to **if-else**:

The ternary operator can be rewritten as:

```
if (Expression1) {  
    variable = Expression2;  
} else {  
    variable = Expression3;  
}
```

Working of the Ternary Operator

1. Step 1: Evaluate `Expression1` (the condition).
 2. Step 2A: If `Expression1` is true, execute `Expression2`.
 3. Step 2B: If `Expression1` is false, execute `Expression3`.
 4. Step 3: Return the result of `Expression2` or `Expression3`.
-

Key Notes

- The ternary operator has a lower precedence compared to most other operators. Use

parentheses to avoid ambiguity.

- It is best suited for short conditional assignments or checks to keep the code compact.
-

Examples

Example 1: Find the Greatest of Two Numbers

```
#include <stdio.h>

int main() {
    int m = 5, n = 4;

    (m > n)
        ? printf("m is greater than n: %d > %d", m, n)
        : printf("n is greater than m: %d > %d", n, m);

    return 0;
}
```

Output:

m is greater than n: 5 > 4

Example 2: Check Leap Year

```
#include <stdio.h>

int main() {
    int year = 1900;

    (year % 4 == 0)
        ? (year % 100 != 0
            ? printf("The year %d is a leap year", year)
            : (year % 400 == 0
                ? printf("The year %d is a leap year", year)
                : printf("The year %d is not a leap year", year)))
        : printf("The year %d is not a leap year", year);

    return 0;
}
```

Output:

The year 1900 is not a leap year

Advantages of the Ternary Operator

- Conciseness: Reduces the number of lines in simple conditional assignments.
 - Improved Readability: For short conditions, the ternary operator makes the code more compact.
-

When to Avoid

- Avoid using the ternary operator for complex conditions as it can make the code harder to read and debug.
-

FAQs on Ternary Operator

Q1: What is the ternary operator in C?

The ternary operator in C is a conditional operator (`?:`) that evaluates a condition and executes one of two expressions based on whether the condition is true or false.

Q2: What is the advantage of using the ternary operator?

The ternary operator simplifies short conditional assignments and improves code readability for small conditions.

sizeof Operator in C

The `sizeof` operator in C is a compile-time unary operator used to calculate the size (in bytes) of its operand. It is widely used to determine memory requirements for data types, variables, or expressions.

Syntax

```
sizeof(Expression);
```

- Expression: Can be a data type, a variable, or an expression.
-

Characteristics

- Returns the size (in bytes) of the operand.
 - The result is of type **size_t**, an unsigned integral type.
 - Time Complexity: $O(1)$
 - Auxiliary Space: $O(1)$
-

Use Cases of **sizeof** Operator

1. With Data Types: Output (on a 64-bit machine):

- Determines the size of primitive data types like `int`, `float`, `char`, etc.

```
#include <stdio.h>
int main() {
    printf("Size of char: %lu bytes\n", sizeof(char));
    printf("Size of int: %lu bytes\n", sizeof(int));
    printf("Size of float: %lu bytes\n", sizeof(float));
    printf("Size of double: %lu bytes\n", sizeof(double));
    printf("Size of pointer: %lu bytes\n", sizeof(void *));
    return 0;
}
```

```
Size of char: 1 bytes
Size of int: 4 bytes
Size of float: 4 bytes
Size of double: 8 bytes
Size of pointer: 8 bytes
```

Note: Results may vary depending on the system architecture (e.g., 32-bit vs 64-bit).

1. With Expressions: Output:

- Determines the size of the result of an expression.

```
#include <stdio.h>
int main() {
    int a = 0;
    double d = 10.21;
    printf("%lu", sizeof(a + d));
    return 0;
}
```

```
8
```

- The result is 8 because `a + d` evaluates to double.
-

1. Compile-Time Nature: Output:

- The `sizeof` operator does not evaluate expressions; it computes the size during compilation.

```
#include <stdio.h>
int main() {
    int y;
    int x = 11;
    y = sizeof(x++); // x is not incremented
    printf("%d %d", y, x);
    return 0;
}
```

- `x` remains 11 because `sizeof` only checks the type of `x` without executing the increment.

1. For Arrays: Output:

- To calculate the number of elements in an array:

```
#include <stdio.h>
int main() {
    int arr[] = {1, 2, 3, 4, 7, 98, 0, 12, 35, 99, 14};
    printf("Number of elements: %lu\n", sizeof(arr) / sizeof(arr[0])
    return 0;
}
```

Number of elements: 11

1. For Dynamic Memory Allocation:

- Determines the memory size needed for `malloc` dynamically.

```
int *ptr = (int *)malloc(10 * sizeof(int));
```

- This ensures portability, as `sizeof(int)` varies across platforms.

Need for **sizeof**

1. Portability: Ensures compatibility across different systems with varying type sizes.
2. Memory Optimization: Helps allocate only the required memory.
3. Dynamic Programming: Useful in dynamic memory allocation.
4. Array Operations: Simplifies calculation of array sizes.

Key Notes

- `sizeof` does not evaluate the value of its operand, only its type.
- The results may vary depending on the platform or compiler.
- Use `sizeof` to write portable and efficient code.

Operator Precedence and Associativity in C

Operator precedence and associativity are essential concepts in C programming, helping determine the order in which operators are evaluated in expressions. Understanding these rules is crucial for writing clear, efficient, and error-free code.

Operator Precedence

Operator precedence refers to the priority of operators in expressions. The operator with higher precedence is evaluated first. When multiple operators have the same precedence, associativity determines the evaluation order.

Operator Precedence Table

Here's a summary of operator precedence and associativity from highest to lowest:

Precedence	Operator	Description	Associativity
1	<code>()</code> , <code>[]</code> , <code>.</code> , <code>-></code> , <code>++</code> , <code>--</code>	Function call, array subscript, dot operator, structure pointer, postfix increment/decrement	Left-to-Right
2	<code>++</code> , <code>--</code>	Prefix increment/decrement	Right-to-Left
	<code>+</code> , <code>-</code> , <code>!</code> , <code>~</code> , <code>(type)</code> , <code>*</code> , <code>&</code> , <code>sizeof</code>	Unary plus, minus, NOT, bitwise complement, type cast, dereference, address-of, size of	Right-to-Left
3	<code>*</code> , <code>/</code> , <code>%</code>	Multiplication, division, modulus	Left-to-Right
4	<code>+</code> , <code>-</code>	Addition, subtraction	Left-to-Right
5	<code><<</code> , <code>>></code>	Bitwise shift left, right	Left-to-Right
6	<code><</code> , <code><=</code> , <code>></code> , <code>>=</code>	Relational operators	Left-to-Right
7	<code>==</code> , <code>!=</code>	Equality and inequality	Left-to-Right
8	<code>&</code>	Bitwise AND	Left-to-Right
9	<code>^</code>	Bitwise XOR	Left-to-Right
10	<code>~</code>	Bitwise OR	Left-to-Right
11	<code>&&</code>	Logical AND	Left-to-Right
12	<code> </code>	Logical OR	Left-to-Right
13	<code>?:</code>	Ternary conditional	Right-to-Left
14	<code>=</code> , <code>+=</code> , <code>-=</code> , <code>*=</code> , <code>/=</code> , <code>=</code> , <code><</code> , <code><=</code> , <code>></code> , <code>>=</code> , <code>%=</code> , <code>&=</code> , <code>^=</code> , <code>~</code>	Assignment operators	Left-to-Right
15	<code>,</code>	Comma (expression separator)	Left-to-Right

Key Points:

- Parentheses `()`: Highest precedence and can be used to alter the default order of evaluation.
- Postfix operators `++`, `--`: These have higher precedence than prefix operators `++`, `--`, and their associativity is Left-to-Right.
- Unary operators `+`, `~`, `!`, `~`, `sizeof`: These operators have Right-to-Left associativity.
- Arithmetic and bitwise operators `+`, `-`, `*`, `/`, `%`, `&`, `^`, `~`, `<<`, `>>`, etc.): These operators generally have Left-to-Right associativity.

Example of Operator Precedence

Consider the expression:

$$10 + 20 * 30$$

According to the precedence table, the multiplication operator (*) has higher precedence than addition (+). So the expression is evaluated as:

$$10 + (20 * 30) = 10 + 600 = 610$$

Example of Operator Associativity

When evaluating an expression like:

$$100 / 5 \% 2$$

Both / and % have the same precedence, but since their associativity is Left-to-Right, the division is evaluated first:

$$(100 / 5) \% 2 = 20 \% 2 = 0$$

Complex Example

For an expression like:

$$100 + 200 / 10 - 3 * 10$$

1. Division and Multiplication are evaluated first because they have higher precedence than addition and subtraction. The order follows Left-to-Right associativity.

- $200 / 10 = 20$
- $3 * 10 = 30$

The expression becomes:

$$100 + 20 - 30$$

4. Addition and Subtraction are evaluated from left to right:

- $100 + 20 = 120$
- $120 - 30 = 90$

Thus, the final result is 90.

Important Points to Remember:

1. Associativity is only relevant when operators have the same precedence.
2. Parentheses () can be used to force a specific evaluation order, regardless of precedence.
3. Comma operator , has the lowest precedence, often used in expressions to separate multiple sub-expressions.
4. No chaining of comparison operators like $c > b > a$ in C (unlike Python). It is

evaluated as $(c > b) > a$, which may give unexpected results.

Basic if Statements

```
// Example 1: Check if a number is positive
#include <stdio.h>
int main() {
    int num = 10;
    if (num > 0) {
        printf("Number is positive\\n");
    }
    return 0;
}
```

```
// Example 2: Check if a number is even
#include <stdio.h>
int main() {
    int num = 4;
    if (num % 2 == 0) {
        printf("Number is even\\n");
    }
    return 0;
}
```

If-Else Statements

```
// Example 3: Check voting eligibility
#include <stdio.h>
int main() {
    int age = 20;
    if (age >= 18) {
        printf("Eligible to vote\\n");
    } else {
        printf("Not eligible to vote\\n");
    }
    return 0;
}
```

```
// Example 4: Grade system
#include <stdio.h>
int main() {
    int marks = 75;
    if (marks >= 50) {
        printf("Passed\\n");
    } else {
        printf("Failed\\n");
    }
    return 0;
}
```

```
}
```

Nested If Statements

```
// Example 5: Check number properties
#include <stdio.h>
int main() {
    int num = 6;
    if (num > 0) {
        if (num % 2 == 0) {
            printf("Positive even number\\n");
        } else {
            printf("Positive odd number\\n");
        }
    }
    return 0;
}
```

```
// Example 6: Temperature classification
#include <stdio.h>
int main() {
    float temp = 38.5;
    if (temp > 37) {
        if (temp >= 40) {
            printf("High fever\\n");
        } else {
            printf("Mild fever\\n");
        }
    }
    return 0;
}
```

If-Else-If Ladder

```
// Example 7: Grade calculator
#include <stdio.h>
int main() {
    int score = 85;
    if (score >= 90) {
        printf("Grade A\\n");
    } else if (score >= 80) {
        printf("Grade B\\n");
    } else if (score >= 70) {
        printf("Grade C\\n");
    } else {
        printf("Grade F\\n");
    }
    return 0;
}
```

```
// Example 8: Season determiner
#include <stdio.h>
int main() {
    int month = 7;
    if (month == 12 || month <= 2) {
        printf("Winter\\n");
    } else if (month <= 5) {
        printf("Spring\\n");
    } else if (month <= 8) {
        printf("Summer\\n");
    } else {
        printf("Autumn\\n");
    }
    return 0;
}
```

Complex Nested If-Else

```
// Example 9: BMI calculator
#include <stdio.h>
int main() {
    float bmi = 23.5;
    if (bmi < 18.5) {
        printf("Underweight\\n");
    } else {
        if (bmi < 25) {
            printf("Normal weight\\n");
        } else {
            if (bmi < 30) {
                printf("Overweight\\n");
            } else {
                printf("Obese\\n");
            }
        }
    }
    return 0;
}
```

```
// Example 10: Triangle type checker
#include <stdio.h>
int main() {
    int a = 3, b = 4, c = 5;
    if (a + b > c && b + c > a && a + c > b) {
        if (a == b && b == c) {
            printf("Equilateral triangle\\n");
        } else if (a == b || b == c || a == c) {
            printf("Isosceles triangle\\n");
        } else {
            printf("Scalene triangle\\n");
        }
    }
}
```

```

    }
} else {
    printf("Not a triangle\\n");
}
return 0;
}

```

Multiple Conditions

```

// Example 11: Leap year checker
#include <stdio.h>
int main() {
    int year = 2024;
    if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0) {
        printf("Leap year\\n");
    } else {
        printf("Not a leap year\\n");
    }
    return 0;
}

```

```

// Example 12: Character type checker
#include <stdio.h>
int main() {
    char ch = 'A';
    if (ch >= 'A' && ch <= 'Z') {
        printf("Uppercase letter\\n");
    } else if (ch >= 'a' && ch <= 'z') {
        printf("Lowercase letter\\n");
    } else if (ch >= '0' && ch <= '9') {
        printf("Digit\\n");
    } else {
        printf("Special character\\n");
    }
    return 0;
}

```

```

// Example 13: Password validator
#include <stdio.h>
int main() {
    int length = 8;
    int hasUpper = 1;
    int hasNumber = 1;
    if (length >= 8) {
        if (hasUpper && hasNumber) {
            printf("Strong password\\n");
        } else if (hasUpper || hasNumber) {
            printf("Medium strength password\\n");
        } else {
            printf("Weak password\\n");
        }
    }
}

```

```

    }
} else {
    printf("Password too short\\n");
}
return 0;
}

```

// Example 14: Calculator

```

#include <stdio.h>
int main() {
    char op = '+';
    int a = 10, b = 5;
    if (op == '+') {
        printf("Sum: %d\\n", a + b);
    } else if (op == '-') {
        printf("Difference: %d\\n", a - b);
    } else if (op == '*') {
        printf("Product: %d\\n", a * b);
    } else if (op == '/') {
        if (b != 0) {
            printf("Quotient: %d\\n", a / b);
        } else {
            printf("Division by zero\\n");
        }
    }
    return 0;
}

```

// Example 15: Time period classifier

```

#include <stdio.h>
int main() {
    int hour = 14;
    if (hour >= 0 && hour < 24) {
        if (hour < 12) {
            printf("Morning\\n");
        } else if (hour < 17) {
            printf("Afternoon\\n");
        } else if (hour < 20) {
            printf("Evening\\n");
        } else {
            printf("Night\\n");
        }
    } else {
        printf("Invalid hour\\n");
    }
    return 0;
}

```

// Example 1: Password System with Multiple Attempts

```

#include <stdio.h>

```

```

#include <string.h>

int main() {
    char password[20];
    int attempts = 3;

    while(attempts > 0) {
        printf("Enter password: ");
        scanf("%s", password);

        if(strcmp(password, "SecurePass123") == 0) {
            printf("Access granted!\\n");
            break;
        } else {
            attempts--;
            printf("Wrong password. %d attempts remaining\\n", attempts);
        }
    }
    if(attempts == 0) {
        printf("Account locked!\\n");
    }
    return 0;
}

```

// Example 2: Number Range Validator with Input Cleaning

```

#include <stdio.h>

int main() {
    int num;
    char c;

    printf("Enter a number between 1 and 100: ");
    while((scanf("%d%c", &num, &c) != 2 || c != '\\n') ||
        (num < 1 || num > 100)) {
        printf("Invalid input! Try again: ");
        while(getchar() != '\\n');
    }

    if(num >= 1 && num <= 30) {
        printf("Range: Low\\n");
    } else if(num <= 70) {
        printf("Range: Medium\\n");
    } else {
        printf("Range: High\\n");
    }
    return 0;
}

```

// Example 3: Student Grade Calculator with Input Validation

```

#include <stdio.h>

```

```

int main() {
    float marks[3];
    int i;
    float average = 0;

    for(i = 0; i < 3; i++) {
        printf("Enter mark %d (0-100): ", i+1);
        while(scanf("%f", &marks[i]) != 1 || marks[i] < 0 || marks[i] >
            printf("Invalid mark! Enter again: ");
            while(getchar() != '\\n');
        }
        average += marks[i];
    }

    average /= 3;
    if(average >= 90) {
        printf("Grade: A+ (%.2f)\\n", average);
    } else if(average >= 80) {
        printf("Grade: A (%.2f)\\n", average);
    } else if(average >= 70) {
        printf("Grade: B (%.2f)\\n", average);
    } else {
        printf("Grade: F (%.2f)\\n", average);
    }
    return 0;
}

```

// Example 4: Date Validator
#include <stdio.h>

```

int main() {
    int day, month, year;
    int isLeap;

    printf("Enter date (DD MM YYYY): ");
    while(scanf("%d %d %d", &day, &month, &year) != 3) {
        printf("Invalid format! Try again: ");
        while(getchar() != '\\n');
    }

    isLeap = (year % 4 == 0 && year % 100 != 0) || (year % 400 == 0);

    if(month < 1 || month > 12) {
        printf("Invalid month!\\n");
    } else if(day < 1) {
        printf("Invalid day!\\n");
    } else if((month == 2 && isLeap && day > 29) ||
        (month == 2 && !isLeap && day > 28) ||
        ((month == 4 || month == 6 || month == 9 || month == 11)
            day > 31) {
        printf("Invalid day for given month!\\n");
    } else {

```

```

        printf("Valid date: %02d/%02d/%04d\\n", day, month, year);
    }
    return 0;
}

```

// Example 5: Complex Calculator with Error Handling

```

#include <stdio.h>
#include <math.h>

int main() {
    double num1, num2;
    char operator;

    printf("Enter calculation (number operator number): ");
    while(scanf("%lf %c %lf", &num1, &operator, &num2) != 3) {
        printf("Invalid format! Try again: ");
        while(getchar() != '\\n');
    }

    if(operator == '+') {
        printf("Result: %.2lf\\n", num1 + num2);
    } else if(operator == '-') {
        printf("Result: %.2lf\\n", num1 - num2);
    } else if(operator == '*') {
        printf("Result: %.2lf\\n", num1 * num2);
    } else if(operator == '/') {
        if(num2 == 0) {
            printf("Error: Division by zero!\\n");
        } else {
            printf("Result: %.2lf\\n", num1 / num2);
        }
    } else if(operator == '^') {
        printf("Result: %.2lf\\n", pow(num1, num2));
    } else {
        printf("Invalid operator!\\n");
    }
    return 0;
}

```

The `if` statement in C is a fundamental decision-making statement that allows the program to execute specific blocks of code based on whether a given condition is true or false. Here's a summary of key concepts:

Syntax:

```

if (condition) {
    // Code block to execute if condition is true
}

```


How it works:

1. Condition Evaluation: The condition inside the `if` statement is evaluated.
2. True Condition: If the condition is true, the statements inside the `if` block execute.
3. False Condition: If the condition is false, the `if` block is skipped.

Example:

```
#include <stdio.h>

int main() {
    int gfg = 9;

    // if statement with true condition
    if (gfg < 10) {
        printf("%d is less than 10", gfg);
    }

    // if statement with false condition
    if (gfg > 20) {
        printf("%d is greater than 20", gfg);
    }

    return 0;
}
```

Output:

9 is less than 10

Flowchart:

The flowchart for an `if` statement demonstrates that the condition is evaluated, and if true, the code inside the block is executed. If false, the block is skipped.

Examples of **if** Statements:

1. Even or Odd Check:

```
#include <stdio.h>

int main() {
    int n = 4956;

    if (n % 2 == 0) {
        printf("%d is Even", n);
    } else {
        printf("%d is Odd", n);
    }

    return 0;
}
```

```
}
```

Output: 4956 is Even

1. Prime Number Check:

```
#include <math.h>

int main() {
    int n = 19;
    int flag = 0;

    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) {
            flag = 1;
            break;
        }
    }

    printf("%d is ", n);
    if (flag == 1) {
        printf("not ");
    }
    printf("a prime number.\n");

    return 0;
}
```

Output: 19 is a prime number.

Advantages of **if** Statement:

- Simple and easy to understand.
- Can evaluate various data types (e.g., int, char, bool).

Disadvantages:

- Only allows a single block of code, which may lead to inefficient checks if multiple conditions exist.
- Can become complex and hard to read with many conditions.

FAQs:

1. Define **if** statement: A control structure that executes a block of code if a given condition is true.
2. Types of decision-making statements in C:

- if
- if-else
- if-else-if ladder
- switch

- Conditional operator (ternary)

8. Multiple conditions: You can combine multiple conditions using logical operators like `&&` (AND) or `||` (OR):

- Valid: `if (a < b && a < c)`
- Invalid: `if (a < b, a < c)` (Comma operator is incorrect here).

The `if` statement is essential in C programming for making decisions based on conditions, and understanding its behavior allows for creating more dynamic and responsive programs.

C **if-else** Statement

The `if-else` statement in C is a control flow statement that allows a program to execute one block of code if a condition is true, and another block if the condition is false. It's an extension of the basic `if` statement, which provides an alternative for when the condition is false.

Syntax:

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

How it works:

1. Condition Evaluation: The program first checks the condition inside the `if` statement.
2. True Condition: If the condition is true (non-zero or non-null), the `if` block executes.
3. False Condition: If the condition is false (zero or null), the `else` block executes.

After one of the blocks executes, the program moves on to the code after the `if-else` statement.

Example 1: Check if a number is even or odd

```
#include <stdio.h>

int main() {
    int num = 9911234;

    if (num % 2 == 0) {
        printf("Number is even");
    } else {
        printf("Number is odd");
    }

    return 0;
}
```

Output: Number is even

Example 2: Check if a person is eligible to vote

```
#include <stdio.h>

int main() {
    int p1_age = 15;
    int p2_age = 25;

    if (p1_age < 18)
        printf("Person 1 is not eligible to vote.\n");
    else
        printf("Person 1 is eligible to vote.\n");

    if (p2_age < 18)
        printf("Person 2 is not eligible to vote.\n");
    else
        printf("Person 2 is eligible to vote.");

    return 0;
}
```

Output:

```
Person 1 is not eligible to vote.
Person 2 is eligible to vote.
```

Key Points:

1. Skipping Braces: You can omit braces `{ }` around the `if` and `else` blocks when there is only one statement inside them. However, if there are multiple statements, braces are required.
2. Condition Evaluation: Any non-zero value is considered true, while zero is considered false.

Advantages of **if-else**:

- Allows decision-making based on conditions.
- Easy to use and understand.
- Can evaluate multiple data types (int, char, bool, etc.).

Disadvantages of **if-else**:

- The code can become complex and unreadable if there are many `if-else` statements.
- Can be slower than the `switch` statement when there are many conditions.

FAQs:

1. Can we skip braces around the body of the **if-else** block in C? Yes, if there is only one statement in the block. Otherwise, braces are required.
2. What are the types of **if-else** statements in C?
 - if Statement
 - if-else Statement
 - if-else-if Ladder

The if-else statement is essential in C programming for branching logic and managing different execution paths based on conditions.

C **if-else-if** Ladder

The if-else-if ladder in C is used to evaluate multiple conditions sequentially. Unlike a simple if-else statement, the if-else-if ladder allows you to check several conditions one after the other and execute the corresponding block of code for the first true condition.

Syntax:

```
if (condition1) {
    // code executed if condition1 is true
}
else if (condition2) {
    // code executed if condition2 is true (and condition1 is false)
}
else if (condition3) {
    // code executed if condition3 is true (and previous conditions are
}
// Additional else if blocks can be added as needed
else {
    // code executed if all previous conditions are false
}
```

Working Flow:

1. Start with **if**: The first condition is evaluated.
2. Evaluate **else if** conditions: If the first condition is false, the next **else if** condition is evaluated, and so on.
3. Execute the first true block: As soon as one condition evaluates to true, the corresponding block is executed.
4. Exit the ladder: Once a block is executed, the rest of the **else if** or **else** blocks are skipped.
5. Final **else** block: If none of the conditions are true, the **else** block (if present) is executed.

Example 1: Check whether a number is positive, negative, or zero

```
#include <stdio.h>

int main() {
    int n = 0;
```

```

    if (n > 0) {
        printf("Positive");
    }
    else if (n < 0) {
        printf("Negative");
    }
    else {
        printf("Zero");
    }

    return 0;
}

```

Output:

Zero

Example 2: Calculate grade according to marks

```

#include <stdio.h>

int main() {
    int marks = 91;

    if (marks <= 100 && marks >= 90)
        printf("A+ Grade");
    else if (marks < 90 && marks >= 80)
        printf("A Grade");
    else if (marks < 80 && marks >= 70)
        printf("B Grade");
    else if (marks < 70 && marks >= 60)
        printf("C Grade");
    else if (marks < 60 && marks >= 50)
        printf("D Grade");
    else
        printf("F Failed");

    return 0;
}

```

Output:

A+ Grade

Key Notes:

- The `else` block is optional. If no conditions match and there's no `else`, the program simply moves on.

- The conditions are evaluated sequentially, and once a true condition is found, the corresponding block is executed, and the rest of the ladder is skipped.
- The `else` block at the end is executed if none of the `if` or `else if` conditions are true.

Advantages of **if-else-if** Ladder:

- It allows multiple conditions to be checked without nesting too many `if` statements.
- It provides a clear structure for decision-making when there are multiple possible outcomes.

Disadvantages:

- If there are many conditions to check, it may reduce the readability of the code.
- The execution can become slower as more conditions are added.

Switch Statement in C

The `switch` statement in C is used to evaluate an expression and execute the code corresponding to the matching case value. It is a cleaner and more efficient alternative to a long `if-else-if` ladder, especially when you need to compare a variable to several possible values. The `switch` statement allows you to dispatch execution to different parts of code based on the value of the expression.

Syntax of **switch** Statement:

```
switch (expression)
{
    case value1:
        statement_1;
        break;
    case value2:
        statement_2;
        break;
    // More cases can be added as needed
    default:
        default_statement;
}
```

Key Points:

1. Expression: The `switch` statement evaluates an expression. It can only evaluate values of types `int` or `char`.
2. Cases: Each case corresponds to a specific value that the expression can match. If the expression evaluates to `value1`, `statement_1` is executed.
3. Break Statement: The `break` keyword is used to exit the `switch` statement after executing the matching case's code. If omitted, the program will "fall through" and execute all subsequent cases until a `break` is encountered or the `switch` block ends.
4. Default Case: The `default` case is optional and executes if no case matches the value.

of the expression.

Rules for Using the **switch** Statement:

- The expression evaluated in a `switch` statement must result in a constant value.
- Case values must be unique and can only be of type `int` or `char`.
- The `break` statement is optional but recommended to prevent fall-through.
- The default case is optional.

Example 1: Basic Switch Example

```
#include <stdio.h>

int main() {
    int var = 1;

    switch (var) {
        case 1:
            printf("Case 1 is Matched.\n");
            break;
        case 2:
            printf("Case 2 is Matched.\n");
            break;
        case 3:
            printf("Case 3 is Matched.\n");
            break;
        default:
            printf("Default case is Matched.\n");
    }

    return 0;
}
```

Output:

Case 1 is Matched.

How the Switch Statement Works:

1. The switch variable is evaluated.
2. The evaluated value is matched against the values in the cases.
3. If a match is found, the corresponding code block is executed.
4. If the `break` statement is present, control exits the switch block.
5. If no match is found, the `default` case (if present) is executed.

Example 2: Switch Case Without Break

```
#include <stdio.h>

int main() {
    int var = 2;
```



```

switch (var) {
    case 1:
        printf("Case 1 is executed.\n");
    case 2:
        printf("Case 2 is executed.\n");
    case 3:
        printf("Case 3 is executed.\n");
    case 4:
        printf("Case 4 is executed.\n");
}

return 0;
}

```

Output:

```

Case 2 is executed.
Case 3 is executed.
Case 4 is executed.

```

In this example, since no `break` statements are used, the program continues to execute all subsequent cases after case 2 is matched.

Example 3: Switch Case with Default

```

#include <stdio.h>

int main() {
    int day = 2;

    switch (day) {
        case 1:
            printf("Monday\n");
            break;
        case 2:
            printf("Tuesday\n");
            break;
        case 3:
            printf("Wednesday\n");
            break;
        default:
            printf("Invalid Input\n");
            break;
    }

    return 0;
}

```

Output:

Advantages of **switch** Statement:

1. Ease of Use: It is easier to read and maintain than using a long chain of `if-else-if` statements.
2. Faster Execution: `switch` can be more efficient than `if-else-if` for a large number of conditions.
3. Clarity: It simplifies multi-way branching, making the code easier to understand.

Disadvantages of **switch** Statement:

1. Limited Expression Types: It can only evaluate `int` or `char` values.
2. No Support for Logical Conditions: Unlike `if-else-if`, `switch` cannot evaluate complex logical expressions (like `x > 5`).
3. Fall-through: If the `break` is omitted, execution will continue through all subsequent cases, which may lead to unintended behavior.

Conclusion:

The `switch` statement is a powerful tool in C programming for handling multiple possible conditions in an organized and efficient manner. It is particularly useful when evaluating a single expression against several values, reducing the need for multiple `if-else-if` conditions. However, care must be taken with the `break` statement to prevent fall-through, and it can only handle certain types of expressions.

Using Range in **switch** Case in C

In standard C, `switch` cases only work with specific values (like integers or characters). However, with GNU C compiler (GCC), there is an extension that allows you to specify ranges of values in a `switch` case. This can be useful when you want to execute the same code for a range of values, saving you from writing separate `case` statements for each individual value.

Syntax for Range in **switch** Case:

To use a range, the syntax is:

```
case low ... high:
```

Where:

- `low` is the starting value of the range.
- `high` is the ending value of the range.
- Note: There must be spaces around the ellipsis (`...`), like `case 1 ... 6:`. Writing `case 1...6:` without spaces will cause an error.

This feature is not part of the C standard, but it works in GCC and other compilers that support this extension.

Example Program Using Range in **switch** Case:

```
#include <stdio.h>

int main() {
    int arr[] = {1, 5, 15, 20};

    for (int i = 0; i < 4; i++) {
        switch (arr[i]) {
            // Range 1 to 6
            case 1 ... 6:
                printf("%d in range 1 to 6\n", arr[i]);
                break;
            // Range 19 to 20
            case 19 ... 20:
                printf("%d in range 19 to 20\n", arr[i]);
                break;
            default:
                printf("%d not in range\n", arr[i]);
                break;
        }
    }

    return 0;
}
```

Output:

```
1 in range 1 to 6
5 in range 1 to 6
15 not in range
20 in range 19 to 20
```

Explanation:

- Range Case: The case 1 ... 6: will match all values between 1 and 6, inclusive.
- Another Range Case: The case 19 ... 20: will match 19 and 20.
- For values that do not match any of the specified ranges, the default case will be executed.

Complexity Analysis:

- Time Complexity: $O(n)$, where n is the size of the array. We are looping through the array once, and for each element, the switch statement takes constant time.
- Auxiliary Space: $O(1)$, as the program uses a constant amount of extra space.

Error Conditions:

1. Invalid Range: If $\text{low} > \text{high}$ (e.g., case 6 ... 1:), the compiler will produce an error.

2. Overlapping Case Values: If you have ranges that overlap (e.g., case 1 ... 5 and case 4 ... 7), the compiler will also produce an error. Make sure that ranges do not overlap.

Exercise:

You can modify this program to work with a char array and define ranges for ASCII values. For example, checking if characters fall within the ranges for uppercase ('A' to 'Z') or lowercase ('a' to 'z') letters.

Here's a modified version using characters:

```
#include <stdio.h>

int main() {
    char arr[] = {'A', 'b', 'z', 'D'};

    for (int i = 0; i < 4; i++) {
        switch (arr[i]) {
            // Range A to Z
            case 'A' ... 'Z':
                printf("%c is an uppercase letter\n", arr[i]);
                break;
            // Range a to z
            case 'a' ... 'z':
                printf("%c is a lowercase letter\n", arr[i]);
                break;
            default:
                printf("%c is not a letter\n", arr[i]);
                break;
        }
    }

    return 0;
}
```

Output for the Exercise:

```
A is an uppercase letter
b is a lowercase letter
z is a lowercase letter
D is an uppercase letter
```

This demonstrates how to handle a range of characters using the range syntax in GCC's switch case extension.

Loops in C Programming

Loops in C are used to repeatedly execute a block of code until a specified condition is met.

This is useful when you need to perform an action multiple times without writing repetitive code. There are several types of loops in C, each suited to different scenarios. Below are explanations and examples of the main types of loops in C.

Types of Loops in C

1. Entry-Controlled Loops:

- The condition is checked before entering the loop. If the condition is false initially, the loop body may not execute at all.
- Examples: **for** loop, **while** loop.

4. Exit-Controlled Loops:

- The condition is checked after executing the loop body. The loop will always execute at least once.
 - Example: **do-while** loop.
-

1. **for** Loop

The `for` loop is used when you know the number of iterations in advance. It has three main parts: initialization, condition check, and update expression.

Syntax:

```
for (initialization; test_condition; update_expression)
{
    // body of the loop
}
```

Example:

```
#include <stdio.h>

int main()
{
    for (int i = 1; i <= 10; i++) {
        printf("Hello World\n");
    }
    return 0;
}
```

Output:

```
Hello World
Hello World
Hello World
Hello World
Hello World
```

```
Hello World
Hello World
Hello World
Hello World
Hello World
```

Explanation:

- The loop starts with `i = 1`.
 - It checks if `i <= 10`, and if true, it prints "Hello World".
 - Then, `i` is incremented by 1 after each iteration, until the condition becomes false.
-

2. **while** Loop

The `while` loop is used when you do not know the number of iterations in advance. It keeps executing the loop body as long as the condition is true.

Syntax:

```
initialization;
while (test_condition)
{
    // body of the loop
    update_expression;
}
```

Example:

```
#include <stdio.h>

int main()
{
    int i = 2;
    while (i < 10) {
        printf("Hello World\n");
        i++;
    }
    return 0;
}
```

Output:

```
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
```

Hello World

Explanation:

- The loop starts with `i = 2`.
 - The condition `i < 10` is checked, and as long as it is true, the loop body is executed.
 - After each iteration, `i` is incremented.
-

3. **do-while** Loop

The `do-while` loop is similar to the `while` loop, but the key difference is that the condition is checked after the loop body is executed. This guarantees that the loop body will execute at least once, even if the condition is false initially.

Syntax:

```
initialization;
do {
    // body of the loop
    update_expression;
} while (test_condition);
```

Example:

```
#include <stdio.h>

int main()
{
    int i = 2;
    do {
        printf("Hello World\n");
        i++;
    } while (i < 1); // The condition is false, but the loop runs once
    return 0;
}
```

Output:

Hello World

Explanation:

- The loop body runs once, even though the condition `i < 1` is false, because the test is done after the first execution.
-

4. Infinite Loops

An infinite loop occurs when the loop condition is always true, causing the loop to execute indefinitely. You can create infinite loops using any of the three loops: `for`, `while`, or `do-while`.

Infinite **for** loop example:

```
#include <stdio.h>

int main()
{
    for (;;) {
        printf("This loop will run forever.\n");
    }
    return 0;
}
```

Infinite **while** loop example:

```
#include <stdio.h>

int main()
{
    while (1) {
        printf("This loop will run forever.\n");
    }
    return 0;
}
```

Infinite **do-while** loop example:

```
#include <stdio.h>

int main()
{
    do {
        printf("This loop will run forever.\n");
    } while (1);
    return 0;
}
```

Output:

```
This loop will run forever.
This loop will run forever.
This loop will run forever.
...
```

Loop Control Statements

Loop control statements are used to alter the flow of control inside loops:

1. **break**: Exits the loop or switch statement immediately.
2. **continue**: Skips the rest of the loop body and jumps to the next iteration.
3. **goto**: Transfers control to a labeled statement.

Example of **break** and **continue**:

```
#include <stdio.h>

int main()
{
    for (int i = 1; i <= 10; i++) {
        if (i == 5) {
            continue; // Skips the iteration when i equals 5
        }
        if (i == 8) {
            break; // Exits the loop when i equals 8
        }
        printf("%d\n", i);
    }
    return 0;
}
```

Output:

```
1
2
3
4
6
7
```

Explanation:

- When `i == 5`, the `continue` statement skips the current iteration.
- When `i == 8`, the `break` statement exits the loop.

Summary

- **for** loop: Best when the number of iterations is known beforehand.
- **while** loop: Best when the number of iterations is unknown, but the loop needs to execute while a condition is true.
- **do-while** loop: Similar to `while`, but guarantees at least one execution of the loop body.
- Infinite loops: Loops that run indefinitely if the condition is always true, useful in certain scenarios like event handling or server-side programming.
- Loop control statements: `break` to exit, `continue` to skip iterations, and `goto` for

jumping to a labeled statement.

By mastering these loop structures, you can write efficient programs in C that handle repetitive tasks with ease.

for Loop in C Programming

The `for` loop in C is used to repeat a block of code a fixed number of times. It is an entry-controlled loop, meaning the condition is checked before the loop starts executing.

Syntax of the **for** Loop

```
for (initialization; condition; update)
{
    // Body of the loop (set of statements to be executed)
}
```

Components of a **for** Loop:

1. Initialization:

- This is the starting point of the loop where a loop control variable is initialized.
- Example: `int i = 0;`

4. Condition:

- The condition is tested before each iteration. If the condition evaluates to `true`, the loop body executes. If it's `false`, the loop terminates.
- Example: `i < 10;`

7. Update:

- After each iteration, the update expression is executed to modify the loop control variable.
- Example: `i++` (incrementing the loop control variable).

10. Body:

- This is the block of code that gets executed for each iteration as long as the condition is true.
-

How the **for** Loop Works

1. Step 1: Initialization occurs once before the loop starts.
 2. Step 2: The condition is checked before entering the loop body. If true, the body is executed.
 3. Step 3: The body of the loop executes.
 4. Step 4: The loop control variable is updated, and the condition is checked again.
 5. Step 5: The loop continues until the condition is false.
-

Example of a **for** Loop

```
#include <stdio.h>

int main() {
    int gfg = 0;
    for (gfg = 1; gfg <= 5; gfg++) {
        printf("GeeksforGeeks\n");
    }
    return 0;
}
```

Output:

```
GeeksforGeeks
GeeksforGeeks
GeeksforGeeks
GeeksforGeeks
GeeksforGeeks
```

Explanation:

- The loop starts with `gfg = 1`, checks if `gfg <= 5`, prints "GeeksforGeeks", increments `gfg`, and repeats until `gfg` becomes 6.
-

Nested **for** Loop

A nested **for** loop is a `for` loop inside another `for` loop. This is useful when dealing with multi-dimensional arrays or other complex data structures.

Syntax:

```
for (initialization; condition; update) {
    for (initialization; condition; update) {
        // Inner loop body
    }
}
```

Special Cases

1. **for** Loop Without Curly Braces

If the `for` loop contains only one statement, the curly braces can be omitted.

```
#include <stdio.h>

int main() {
    int i;
```

```

    for (i = 1; i <= 10; i++)
        printf("%d ", i); // One statement after the loop
    printf("\nThis statement executes after the for loop end!!!!");
    return 0;
}

```

Output:

```

1 2 3 4 5 6 7 8 9 10
This statement executes after the for loop end!!!!

```

- Here, the `for` loop runs the `printf` statement 10 times, and the statement after the loop runs only once.

2. Infinite **for** Loop

An infinite **for** loop is created when there are no conditions or the condition is always true.

```

#include <stdio.h>

int main() {
    for (;;) { // No condition provided, creating an infinite loop
        printf("GeeksforGeeks to Infinite");
    }
    return 0;
}

```

Output:

```

GeeksforGeeks to InfiniteGeeksforGeeks to InfiniteGeeksforGeeks to Infi

```

- The loop will print "GeeksforGeeks to Infinite" endlessly unless interrupted.

Advantages of **for** Loop

- **Code Reusability:** Loops allow you to reuse code without repetition.
- **Compact Code:** The `for` loop has a compact syntax, making code easier to write and read.
- **Efficient Traversal:** Ideal for iterating through arrays, strings, or other data structures.

Disadvantages of **for** Loop

- **Limited Condition Control:** You can only specify a single condition (though multiple variables can be used).
- **No Skipping of Elements:** Unlike some other loops, `for` cannot skip certain elements unless explicitly programmed.

FAQs on **for** Loops

1. What is a loop in C?

- A loop repeats a block of code multiple times until a condition is met.

3. How can I iterate over elements in C?

- You can iterate using a loop to process each element of an array, string, or data structure.

5. How many types of loops exist in C?

- There are three types of loop control statements: `for`, `while`, and `do-while` loops.

7. Can I use multiple variables in a **for** loop?

- Yes, you can use multiple variables in the initialization and update expressions.

9. How do I create an infinite loop in C?

- An infinite loop can be created by leaving the condition in the `for` loop empty or setting it to always evaluate as `true`.

while Loop in C Programming

The `while` loop in C is an entry-controlled loop that executes a block of code as long as a specified condition is true. It is commonly used when the number of iterations is not known in advance, and the loop needs to run until a particular condition is met.

Syntax of the **while** Loop

```
while (test_expression)
{
    // Body of the loop (set of statements)
}
```

Components of a **while** Loop:

1. Test Expression:

- The condition is evaluated before each iteration. If the condition is true, the loop executes the body; otherwise, it exits.
- Example: `i < 5`

4. Body:

- The block of code to be executed repeatedly as long as the condition is true.
- Example: `printf("GeeksforGeeks\n");`

7. Updation:

- The loop control variable must be updated within the loop body (not part of the syntax, but necessary to avoid infinite loops).
 - Example: `i++`
-

How the **while** Loop Works

1. Step 1: The condition is evaluated before entering the loop body.
 2. Step 2: If the condition is true, the loop body executes.
 3. Step 3: After executing the body, the loop control variable is updated.
 4. Step 4: The condition is checked again. If still true, the body executes again. This process repeats until the condition becomes false.
-

Example of a **while** Loop

```
#include <stdio.h>

int main() {
    int i = 0;    // Initialization of loop variable

    // Test expression: the loop will run as long as i < 5
    while (i < 5) {
        // Loop body
        printf("GeeksforGeeks\n");

        // Update: Increment i after each iteration
        i++;
    }

    return 0;
}
```

Output:

```
GeeksforGeeks
GeeksforGeeks
GeeksforGeeks
GeeksforGeeks
GeeksforGeeks
```

Explanation:

- The loop starts with `i = 0`, and checks if `i < 5`. Since the condition is true, the loop body executes, printing "GeeksforGeeks". The loop variable `i` is then incremented, and the condition is checked again. The loop continues until `i` reaches 5.
-

Infinite **while** Loop

An infinite **while** loop occurs when the test condition is always true or when the loop control variable is never updated in the body, causing the condition to never become false.

Example:

```
#include <stdio.h>

int main() {
    int gfg1 = 1;
    int gfg2 = 1;

    // Condition is always true as gfg1 is always less than 10
    while (gfg1 < 10) {
        gfg2 = gfg2 + 1;
        printf("GeeksforGeeks to Infinity");
    }

    return 0;
}
```

Output:

GeeksforGeeks to InfinityGeeksforGeeks to InfinityGeeksforGeeks to Infi

- Since `gfg1` is never updated in the body of the loop, the condition `gfg1 < 10` remains true, causing the loop to run indefinitely.

Key Points about the **while** Loop

- Entry-Controlled: The condition is evaluated before the loop body executes, meaning the body may not execute at all if the condition is false initially.
- Pre-Tested Loop: The condition is checked before each iteration, making it ideal when the number of iterations is uncertain.
- Preferred Usage: It is generally used when the number of iterations is unknown, and the loop needs to continue until a specific condition is met.

Advantages of **while** Loop

- Flexibility: The loop can run an indefinite number of times, based on the condition.
- Useful for Unknown Iterations: Suitable for situations where the number of iterations is not known in advance.
- Simplicity: It is straightforward for scenarios requiring an indefinite loop or when the condition is checked dynamically.

Disadvantages of **while** Loop

- Infinite Loops: If the condition is not updated or if the condition is always true, the loop can become infinite.

- **Requires Manual Update:** Unlike the `for` loop, the `while` loop does not handle initialization, condition checking, and updates in one place. You must manage them manually in the loop body.
-

By using the `while` loop, you can effectively handle scenarios where the iteration count is unknown or dynamic, ensuring more control over the execution flow.

do...while Loop in C

The `do...while` loop is an exit-controlled (or post-tested) loop in C, which means that the body of the loop is executed at least once before the condition is checked. This behavior distinguishes it from other loops like the `while` loop, where the condition is checked before execution.

Syntax of the `do...while` Loop

```
do {  
    // body of do-while loop  
} while (condition);
```

- **Condition:** After executing the body of the loop, the condition is checked. If it is true, the loop continues; otherwise, it exits.
 - **Body:** The set of statements that will execute at least once, and repeatedly if the condition remains true.
-

How the `do...while` Loop Works

1. Step 1: The body of the loop is executed first.
 2. Step 2: The condition is checked after executing the body.
 3. Step 3: If the condition is true, the body executes again.
 4. Step 4: If the condition becomes false, the loop exits, and control moves to the next statement.
-

Example of a `do...while` Loop

```
#include <stdio.h>  
  
int main() {  
    int i = 0; // Initialization  
  
    do {  
        printf("Geeks\n"); // Body of the loop  
        i++; // Update loop variable  
    } while (i < 3); // Condition to check after executing the body  
  
    return 0;
```



```
}
```

Output:

```
Geeks
Geeks
Geeks
```

Explanation:

- The loop runs three times because the condition ($i < 3$) is true initially, and the body executes before the condition is checked.
-

Example 2: **do...while** with a False Condition from the Start

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

int main() {
    bool condition = false; // False condition

    do {
        printf("This is loop body.\n"); // Body of the loop
    } while (condition); // False condition will not affect the first

    return 0;
}
```

Output:

```
This is loop body.
```

Explanation:

- Even though the condition is false from the start, the body of the loop is executed once before the condition is checked.
-

Example 3: Multiplication Table of **N**

```
#include <stdio.h>

int main() {
    int N = 5, i = 1;

    do {
        printf("%d x %d = %d\n", N, i, N * i);
        i++; // Update
    } while (i <= N);
}
```

```

    } while (i <= 10); // Continue until i reaches 10

    return 0;
}

```

Output:

```

5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
5 x 6 = 30
5 x 7 = 35
5 x 8 = 40
5 x 9 = 45
5 x 10 = 50

```

Explanation:

- The multiplication table for `N` is printed by repeatedly multiplying `N` with `i` and incrementing `i` until it reaches 10.

Differences between **while** and **do...while** Loop

Feature	while Loop	do...while Loop
Condition check	Condition checked before the loop body.	Condition checked after the loop body.
Guaranteed Execution	Body may not execute if condition is false initially.	Body always executes at least once.
Type of loop	Pre-tested (entry-controlled).	Post-tested (exit-controlled).
Semicolon	Not required after the condition.	Required after the condition.

Infinite **do...while** Loop

An infinite loop can be created by specifying a condition that is always true. For example, using `while(1)` will always be true, resulting in an infinite loop.

```

#include <stdio.h>

int main() {
    do {
        printf("gfg "); // Infinite loop body
    } while (1); // Condition always true

    return 0;
}

```

Output:

gfg gfg gfg gfg gfg ...

Explanation:

- Since the condition 1 (true) is always true, the loop will continue indefinitely.
-

Nested **do...while** Loops

A do...while loop can also be nested within another do...while loop. For example:

```
#include <stdio.h>

int main() {
    int i = 0, j;

    do {
        j = 0;
        do {
            printf("%d ", i * 3 + j);
            j++;
        } while (j < 3);
        printf("\n");
        i++;
    } while (i < 3);

    return 0;
}
```

Output:

```
0 1 2
3 4 5
6 7 8
```

Explanation:

- The outer loop runs three times, and the inner loop also runs three times, printing a sequence of numbers.
-

Conclusion

The do...while loop is useful when you need to ensure that the loop body runs at least once, even if the condition is false initially. It is typically used in scenarios where the first iteration must occur regardless of the condition, such as traversing data structures or printing tables.

Key Difference Between **for** and **while** Loops in C

The main difference between `for` and `while` loops in C comes into play when a `continue` statement is used within the loop body. This difference results in varied behavior in certain cases, as shown in the following example.

Example with **for** Loop

```
#include <stdio.h>

int main() {
    for (int i = 0; i < 5; i++) {
        if (i == 2) {
            continue; // Skip the rest of the loop body when i == 2
        }
        printf("Loop\n");
    }
    return 0;
}
```

Output:

```
Loop
Loop
Loop
Loop
```

Explanation:

- In the `for` loop, the `continue` statement causes the loop to skip the remaining code (i.e., the `printf` statement) for the current iteration. However, the loop's increment (`i++`) still happens as part of the loop structure, and the loop proceeds to the next iteration.
-

Example with **while** Loop (Problematic Version)

```
#include <stdio.h>

int main() {
    int i = 0;

    while (i < 5) {
        if (i == 2) {
            continue; // Will skip the rest of the loop, but i is not
        }
        printf("Loop\n");
    }
}
```

```
    }  
    return 0;  
}
```

Output:

Time Limit Exceeded (Infinite Loop)

Explanation:

- In this `while` loop, the `continue` statement skips the `printf` but does not increment `i`. As a result, when `i == 2`, the loop continues to execute from the top with `i` still equal to 2, causing an infinite loop.
-

Correcting the **while** Loop

To make the `while` loop behave like the `for` loop (i.e., to increment `i` properly and avoid an infinite loop), we can manually increment `i` before the `continue` statement:

```
#include <stdio.h>  
  
int main() {  
    int i = 0;  
  
    while (i < 5) {  
        if (i == 2) {  
            i++; // Increment i before continue to avoid infinite loop  
            continue;  
        }  
        printf("Loop\n");  
        i++; // Increment i for normal iteration  
    }  
    return 0;  
}
```

Output:

```
Loop  
Loop  
Loop  
Loop
```

Explanation:

- Now, when `i == 2`, the `continue` statement is properly handled. By incrementing `i` before continuing, the loop will no longer get stuck at `i == 2` and will proceed correctly to print "Loop" 4 times.
-

Summary of the Difference

- **for** loop: The `continue` statement works as expected, skipping the body of the loop for the current iteration but still performing the increment as part of the loop structure.
- **while** loop: The `continue` statement causes the loop to skip the body, but if the increment is not explicitly included in the loop body, it can lead to an infinite loop.

Thus, in a `while` loop, care must be taken to ensure the loop variable is updated properly within the body to avoid issues like infinite loops.

The **continue** Statement in C

The `continue` statement is a jump statement in C that alters the flow of control within loops. It is used to skip the current iteration of a loop and proceed directly to the next iteration.

Syntax of **continue** in C:

```
continue;
```

The `continue` statement can be used in all types of loops in C, including:

- `for` loop
- `while` loop
- `do-while` loop

How **continue** Works:

When the `continue` statement is encountered inside a loop:

1. In a **for** loop: The loop immediately jumps to the next iteration of the loop, executing the increment or decrement statement.
2. In a **while** or **do-while** loop: The loop immediately evaluates the condition for the next iteration.

Example 1: **continue** in a **for** loop

```
#include <stdio.h>

int main() {
    for (int i = 1; i <= 8; i++) {
        if (i == 4) {
            continue; // Skip when i is 4
        }
        printf("%d ", i);
    }
    printf("\n");

    return 0;
}
```

Output:

1 2 3 5 6 7 8

Explanation:

- When `i == 4`, the `continue` statement is triggered, skipping the `printf` statement for this iteration.
 - The loop continues with the next value of `i`.
-

Example 2: **continue** in a **while** loop

```
#include <stdio.h>

int main() {
    int i = 0;
    while (i < 8) {
        i++;
        if (i == 4) {
            continue; // Skip when i is 4
        }
        printf("%d ", i);
    }
    printf("\n");

    return 0;
}
```

Output:

1 2 3 5 6 7 8

Explanation:

- The `continue` statement causes the loop to skip the current iteration when `i == 4`.
 - After the `continue`, the loop checks the condition (`i < 8`) again, and the incrementing continues for subsequent iterations.
-

Example 3: **continue** in Nested Loops

```
#include <stdio.h>

int main() {
    for (int i = 1; i <= 3; i++) {
        for (int j = 0; j <= 4; j++) {
            if (j == 3) {
```

```

        continue; // Skip when j is 3 in the inner loop
    }
    printf("%d ", j);
}
printf("\n");
}
return 0;
}

```

Output:

```

0 1 2 4
0 1 2 4
0 1 2 4

```

Explanation:

- The `continue` statement only affects the inner loop. When `j == 3`, it skips the rest of the inner loop's body, but the outer loop continues with the next iteration.

Key Differences: **continue** vs **break**

- **continue**: Skips the current iteration and moves on to the next iteration of the loop.
- **break**: Exits the loop entirely, terminating the loop's execution.

Example: **break** vs **continue**

```

#include <stdio.h>

int main() {
    printf("The loop with break produces output as: \n");
    for (int i = 1; i <= 7; i++) {
        if (i == 3) {
            break; // Exit the loop when i equals 3
        }
        printf("%d ", i);
    }

    printf("\nThe loop with continue produces output as: \n");
    for (int i = 1; i <= 7; i++) {
        if (i == 3) {
            continue; // Skip the iteration when i equals 3
        }
        printf("%d ", i);
    }

    return 0;
}

```


Output:

The loop with break produces output as:

```
1 2
```

The loop with continue produces output as:

```
1 2 4 5 6 7
```

Explanation:

- The first loop prints numbers from 1 to 2, then exits the loop when `i == 3` because of the `break` statement.
 - The second loop skips the number 3, continuing with the next iteration after the `continue` statement, printing all other numbers.
-

The **break** Statement in C

The `break` statement in C is used to unconditionally exit from a loop or switch case. When executed, it immediately transfers control out of the loop or switch block, and the program continues with the next statement following the loop or switch.

Syntax of **break** in C:

```
break;
```

You place the `break` statement wherever you want to terminate the execution of the loop or switch statement.

Uses of **break** in C:

The `break` statement is typically used in:

- Simple loops: `for`, `while`, and `do-while` loops
- Nested loops: To break out of the current loop
- Infinite loops: To exit a loop that would otherwise run indefinitely
- Switch case: To exit a specific case of a switch statement

Example 1: **break** in Simple Loops

```
#include <stdio.h>

int main() {
    // Using break inside a for loop
    printf("Break in for loop\n");
    for (int i = 1; i < 5; i++) {
        if (i == 3) {
            break; // Exit the loop when i is 3
        }
        printf("%d ", i);
    }
}
```

```

    }

    // Using break inside a while loop
    printf("\nBreak in while loop\n");
    int i = 1;
    while (i < 20) {
        if (i == 3) {
            break; // Exit the loop when i is 3
        }
        printf("%d ", i);
        i++;
    }

    return 0;
}

```

Output:

```

Break in for loop
1 2
Break in while loop
1 2

```

Explanation:

- In both loops, the `break` statement is executed when `i` equals 3, causing an early exit from the loop.

Example 2: **break** in Nested Loops

```

#include <stdio.h>

int main() {
    for (int i = 1; i <= 6; ++i) {
        for (int j = 1; j <= i; ++j) {
            if (i > 4) {
                break; // Break the inner loop when i > 4
            }
            printf("%d ", j);
        }
        printf("\n");
    }
    return 0;
}

```

Output:

```

1
1 2

```

```
1 2 3
1 2 3 4
```

Explanation:

- The `break` statement only breaks out of the inner loop (when `i > 4`), and control is passed to the next iteration of the outer loop.
-

Example 3: **break** in Infinite Loops

```
#include <stdio.h>

int main() {
    int i = 0;

    // Infinite while loop
    while (1) {
        printf("%d ", i);
        i++;
        if (i == 5) {
            break; // Exit the infinite loop when i equals 5
        }
    }

    return 0;
}
```

Output:

```
0 1 2 3 4
```

Explanation:

- The loop condition is always true (an infinite loop), but the `break` statement stops the loop when `i` reaches 5.
-

How **break** Works in C:

1. The loop starts with the test condition being evaluated.
2. When the break condition is met, the program control reaches the `break` statement, which immediately exits the loop.
3. If the break condition is not met, the loop continues executing as normal.

Flowchart of **break** in C:

1. Loop Start: Evaluate the loop's condition.
2. Condition Check: If the break condition is true, jump out of the loop using `break`.
3. Normal Execution: If the condition is false, continue normal execution of the loop.

break in Switch Case:

In a switch case, the **break** statement is used to exit the case after the matching case is executed. Without **break**, execution will "fall through" to the next case.

Example of **break** in Switch Case:

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

int main() {
    char c;
    float x, y;

    while (1) {
        printf("Enter an operator (+, -), if you want to exit press x:");
        scanf(" %c", &c);

        if (c == 'x')
            exit(0);

        printf("Enter Two Values:\n ");
        scanf("%f %f", &x, &y);

        switch (c) {
            case '+':
                printf("%.1f + %.1f = %.1f\n", x, y, x + y);
                break;
            case '-':
                printf("%.1f - %.1f = %.1f\n", x, y, x - y);
                break;
            default:
                printf("Error! Please enter a valid operator\n");
        }
    }
    return 0;
}
```

Output:

```
Enter an operator (+, -), if you want to exit press x: +
Enter Two Values:
10
20
10.0 + 20.0 = 30.0
```

Explanation:

- The **break** statement ensures that once a case is executed, control exits the switch statement, preventing the program from executing the subsequent cases.

break VS continue:

- **break:** Exits the loop entirely and transfers control to the next statement after the loop.
- **continue:** Skips the current iteration and moves to the next iteration of the loop without exiting the loop.

Summary Table:

Statement	Function	Can be Used in
<code>break</code>	Exits the loop or switch block entirely.	Loops and switch cases
<code>continue</code>	Skips the current iteration and continues with the next one.	Only in loops

goto Statement in C

The `goto` statement in C is an unconditional jump statement that transfers control to a specific part of the program. It is considered a jump statement because it allows the program to jump from one point in the function to another.

The `goto` statement can be used for transferring control within the same function, and it is often used in situations where structured programming might not be easily applied.

Syntax of `goto` Statement:

```
goto label;    // Jump to the label
...
label:         // Define a label
```

- **`goto label;`**: This instructs the program to jump to the statement marked by `label`.
- **`label:`**: This is the target for the jump and can be placed anywhere in the function.

The label is a user-defined identifier and marks the destination of the jump. The statement that follows the label is the target that execution will jump to when the `goto` statement is executed.

Flowchart of `goto` Statement:

1. Program starts.
 2. The `goto` statement is encountered, and the program jumps to the specified label.
 3. Control is transferred to the statement following the label.
 4. Program execution continues from that point onward.
-

Examples of Using `goto`:

Example 1: Check if a Number is Even or Odd

This program uses the `goto` statement to jump to different parts of the function depending on whether the number is even or odd.

```
#include <stdio.h>

void checkEvenOrNot(int num) {
    if (num % 2 == 0)
        goto even; // Jump to even if the number is even
    else
        goto odd;  // Jump to odd if the number is odd

even:
    printf("%d is even\n", num);
    return;

odd:
    printf("%d is odd\n", num);
}

int main() {
    int num = 26;
    checkEvenOrNot(num);
    return 0;
}
```

Output:

26 is even

Explanation:

- The program checks if the number is even using a conditional check. If it is, control jumps to the even label; otherwise, it jumps to the odd label.

Example 2: Print Numbers from 1 to 10

This program uses the `goto` statement to print numbers from 1 to 10.

```
#include <stdio.h>

void printNumbers() {
    int n = 1;
label:
    printf("%d ", n);
    n++;
    if (n <= 10)
        goto label; // Jump to the label to print next number
}

int main() {
```

```

        printNumbers();
        return 0;
}

```

Output:

1 2 3 4 5 6 7 8 9 10

Explanation:

- A label is defined at `label:`. The program prints the number and increments it. If the number is less than or equal to 10, it jumps back to `label:` to print the next number.

Disadvantages of Using **goto**:

1. Complex Logic: The `goto` statement can make the program's flow difficult to follow. It might lead to spaghetti code, where the flow of execution jumps all over the program, making it harder to understand.
2. Debugging Difficulty: It complicates tracing the flow of execution, as control can jump unpredictably.
3. Verification of Correctness: Verifying the correctness of a program using `goto` becomes challenging, especially when multiple `goto` statements are used in complex loops or nested structures.
4. Alternatives: The use of structured programming constructs like `if`, `while`, `for`, `break`, and `continue` is usually preferred. These provide more readable, maintainable, and error-free code compared to `goto`.

Basic Loop Exercises

```

// Exercise 1: Print First 10 Natural Numbers
#include <stdio.h>
int main() {
    int i = 1;
    while(i <= 10) {
        printf("%d ", i);
        i++;
    }
    return 0;
}

```

```

// Exercise 2: Sum of Positive Numbers Until Zero
#include <stdio.h>
int main() {
    int num, sum = 0;
    printf("Enter numbers (0 to stop):\n");
    while(1) {
        scanf("%d", &num);
        if(num == 0) break;
    }
}

```

```

        if(num > 0) sum += num;
    }
    printf("Sum = %d\\n", sum);
    return 0;
}

```

Number Pattern Exercises

```

// Exercise 3: Print Multiplication Table
#include <stdio.h>
int main() {
    int num;
    printf("Enter number: ");
    scanf("%d", &num);
    for(int i = 1; i <= 10; i++) {
        printf("%d x %d = %d\\n", num, i, num * i);
    }
    return 0;
}

```

```

// Exercise 4: Factorial Calculator
#include <stdio.h>
int main() {
    int n, fact = 1;
    printf("Enter number: ");
    scanf("%d", &n);
    for(int i = 1; i <= n; i++) {
        fact *= i;
    }
    printf("Factorial = %d\\n", fact);
    return 0;
}

```

Mathematical Series

```

// Exercise 5: Fibonacci Series
#include <stdio.h>
int main() {
    int n, first = 0, second = 1, next;
    printf("Enter number of terms: ");
    scanf("%d", &n);
    for(int i = 0; i < n; i++) {
        printf("%d ", first);
        next = first + second;
        first = second;
        second = next;
    }
    return 0;
}

```



```
// Exercise 6: Sum of Squares
#include <stdio.h>
int main() {
    int n, sum = 0;
    printf("Enter limit: ");
    scanf("%d", &n);
    for(int i = 1; i <= n; i++) {
        sum += (i * i);
    }
    printf("Sum of squares = %d\\n", sum);
    return 0;
}
```

Number Manipulation

```
// Exercise 7: Reverse a Number
#include <stdio.h>
int main() {
    int num, reversed = 0;
    printf("Enter number: ");
    scanf("%d", &num);
    while(num != 0) {
        reversed = reversed * 10 + num % 10;
        num /= 10;
    }
    printf("Reversed number = %d\\n", reversed);
    return 0;
}
```

```
// Exercise 8: Armstrong Number Checker
#include <stdio.h>
#include <math.h>
int main() {
    int num, original, remainder, result = 0, digits = 0;
    printf("Enter number: ");
    scanf("%d", &num);
    original = num;

    while(original != 0) {
        original /= 10;
        digits++;
    }
    original = num;
    while(original != 0) {
        remainder = original % 10;
        result += pow(remainder, digits);
        original /= 10;
    }
    if(result == num)
```

```

        printf("Armstrong number\\n");
    else
        printf("Not Armstrong number\\n");
    return 0;
}

```

Pattern Printing

```

// Exercise 9: Print Triangle Pattern
#include <stdio.h>
int main() {
    int rows;
    printf("Enter rows: ");
    scanf("%d", &rows);
    for(int i = 1; i <= rows; i++) {
        for(int j = 1; j <= i; j++) {
            printf("* ");
        }
        printf("\\n");
    }
    return 0;
}

```

```

// Exercise 10: Number Pattern
#include <stdio.h>
int main() {
    int rows;
    printf("Enter rows: ");
    scanf("%d", &rows);
    for(int i = 1; i <= rows; i++) {
        for(int j = 1; j <= i; j++) {
            printf("%d ", j);
        }
        printf("\\n");
    }
    return 0;
}

```

Advanced Exercises

```

// Exercise 11: Prime Numbers in Range
#include <stdio.h>
int main() {
    int start, end, isPrime;
    printf("Enter range (start end): ");
    scanf("%d %d", &start, &end);
    for(int i = start; i <= end; i++) {
        if(i == 1) continue;
        isPrime = 1;
    }
}

```

```

        for(int j = 2; j * j <= i; j++) {
            if(i % j == 0) {
                isPrime = 0;
                break;
            }
        }
        if(isPrime) printf("%d ", i);
    }
    return 0;
}

```

// Exercise 12: Perfect Number Checker

```

#include <stdio.h>
int main() {
    int num, sum = 0;
    printf("Enter number: ");
    scanf("%d", &num);
    for(int i = 1; i < num; i++) {
        if(num % i == 0) sum += i;
    }
    if(sum == num)
        printf("Perfect number\\n");
    else
        printf("Not perfect number\\n");
    return 0;
}

```

// Exercise 13: GCD Calculator

```

#include <stdio.h>
int main() {
    int a, b;
    printf("Enter two numbers: ");
    scanf("%d %d", &a, &b);
    while(b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    printf("GCD = %d\\n", a);
    return 0;
}

```

// Exercise 14: Power Calculator

```

#include <stdio.h>
int main() {
    int base, exp, result = 1;
    printf("Enter base and exponent: ");
    scanf("%d %d", &base, &exp);
    for(int i = 0; i < exp; i++) {
        result *= base;
    }
}

```

```

    }
    printf("Result = %d\\n", result);
    return 0;
}

```

// Exercise 15: Sum of Series $1/1! + 2/2! + 3/3! + \dots$

```

#include <stdio.h>
int main() {
    int n;
    float sum = 0, fact = 1;
    printf("Enter number of terms: ");
    scanf("%d", &n);
    for(int i = 1; i <= n; i++) {
        fact *= i;
        sum += i/fact;
    }
    printf("Sum = %f\\n", sum);
    return 0;
}

```

// Exercise 1: Traffic Toll Calculator

```

#include <stdio.h>
int main() {
    int motorcycles = 0, cars = 0, trucks = 0;
    float moto_total = 0, car_total = 0, truck_total = 0;
    char vehicle_type;

    printf("Enter vehicle type (m/c/t, x to end):\\n");
    printf("m - motorcycle ($2)\\n");
    printf("c - car ($4)\\n");
    printf("t - truck ($8)\\n");

    while(1) {
        scanf(" %c", &vehicle_type);
        if(vehicle_type == 'x') break;

        switch(vehicle_type) {
            case 'm':
                motorcycles++;
                moto_total += 2;
                break;
            case 'c':
                cars++;
                car_total += 4;
                break;
            case 't':
                trucks++;
                truck_total += 8;
                break;
            default:
                printf("Invalid vehicle type!\\n");
        }
    }
}

```

```

    }
}

printf("\nTraffic Summary:\n");
printf("Motorcycles: %d (Total: $%.2f)\n", motorcycles, moto_total);
printf("Cars: %d (Total: $%.2f)\n", cars, car_total);
printf("Trucks: %d (Total: $%.2f)\n", trucks, truck_total);
printf("Total Revenue: $%.2f\n", moto_total + car_total + truck_to);
return 0;
}

```

// Exercise 2: Restaurant Order System

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

```

int main() {
    int table_num, item_code, quantity;
    float total = 0, table_totals[10] = {0};
    const float prices[] = {10.99, 8.50, 15.75, 12.25, 7.99};

    printf("Menu:\n");
    printf("1. Steak ($10.99)\n");
    printf("2. Pasta ($8.50)\n");
    printf("3. Seafood ($15.75)\n");
    printf("4. Pizza ($12.25)\n");
    printf("5. Salad ($7.99)\n");

    while(1) {
        printf("\nEnter table number (1-10, 0 to end): ");
        scanf("%d", &table_num);

        if(table_num == 0) break;
        if(table_num < 1 || table_num > 10) {
            printf("Invalid table number!\n");
            continue;
        }

        printf("Enter item code (1-5): ");
        scanf("%d", &item_code);
        if(item_code < 1 || item_code > 5) {
            printf("Invalid item code!\n");
            continue;
        }

        printf("Enter quantity: ");
        scanf("%d", &quantity);
        if(quantity < 1) {
            printf("Invalid quantity!\n");
            continue;
        }

        table_totals[table_num-1] += prices[item_code-1] * quantity;
        total += prices[item_code-1] * quantity;
    }
}

```

```

    }

    printf("\nTable Summary:\n");
    for(int i = 0; i < 10; i++) {
        if(table_totals[i] > 0) {
            printf("Table %d: $%.2f\n", i+1, table_totals[i]);
        }
    }
    printf("Total Revenue: $%.2f\n", total);
    return 0;
}

```

// Exercise 3: Student Grade Analysis System
#include <stdio.h>

```

int main() {
    int num_students, num_subjects;
    float grade, student_avg, class_avg = 0;
    int highest_student = 0, lowest_student = 0;
    float highest_avg = 0, lowest_avg = 100;

    printf("Enter number of students: ");
    scanf("%d", &num_students);
    printf("Enter number of subjects: ");
    scanf("%d", &num_subjects);

    for(int i = 1; i <= num_students; i++) {
        printf("\nStudent %d:\n", i);
        student_avg = 0;

        for(int j = 1; j <= num_subjects; j++) {
            do {
                printf("Enter grade for subject %d (0-100): ", j);
                scanf("%f", &grade);
            } while(grade < 0 || grade > 100);

            student_avg += grade;
        }

        student_avg /= num_subjects;
        class_avg += student_avg;

        if(student_avg > highest_avg) {
            highest_avg = student_avg;
            highest_student = i;
        }
        if(student_avg < lowest_avg) {
            lowest_avg = student_avg;
            lowest_student = i;
        }

        printf("Average for Student %d: %.2f\n", i, student_avg);
    }
}

```

```

    }

    class_avg /= num_students;
    printf("\nClass Analysis:\n");
    printf("Class Average: %.2f\n", class_avg);
    printf("Highest Average: Student %d (%.2f)\n", highest_student, hi
    printf("Lowest Average: Student %d (%.2f)\n", lowest_student, lowe
    return 0;
}

```

```

// Exercise 4: Bank Transaction Processor
#include <stdio.h>

```

```

int main() {
    int account_num;
    char trans_type;
    float amount, balances[100] = {0};
    int num_deposits[100] = {0}, num_withdrawals[100] = {0};
    float total_deposits = 0, total_withdrawals = 0;

    printf("Transaction Types:\n");
    printf("D - Deposit\n");
    printf("W - Withdrawal\n");

    while(1) {
        printf("\nEnter account number (1-100, 0 to end): ");
        scanf("%d", &account_num);

        if(account_num == 0) break;
        if(account_num < 1 || account_num > 100) {
            printf("Invalid account number!\n");
            continue;
        }

        printf("Enter transaction type (D/W): ");
        scanf(" %c", &trans_type);
        printf("Enter amount: $");
        scanf("%f", &amount);

        switch(trans_type) {
            case 'D':
            case 'd':
                balances[account_num-1] += amount;
                num_deposits[account_num-1]++;
                total_deposits += amount;
                break;
            case 'W':
            case 'w':
                if(amount > balances[account_num-1]) {
                    printf("Insufficient funds!\n");
                    continue;
                }

```

```

        balances[account_num-1] -= amount;
        num_withdrawals[account_num-1]++;
        total_withdrawals += amount;
        break;
    default:
        printf("Invalid transaction type!\n");
    }
}

printf("\nAccount Summary:\n");
for(int i = 0; i < 100; i++) {
    if(balances[i] > 0 || num_deposits[i] > 0 || num_withdrawals[i])
        printf("Account %d:\n", i+1);
        printf("    Balance: %.2f\n", balances[i]);
        printf("    Deposits: %d\n", num_deposits[i]);
        printf("    Withdrawals: %d\n", num_withdrawals[i]);
    }
}
printf("\nTotal Deposits: %.2f\n", total_deposits);
printf("Total Withdrawals: %.2f\n", total_withdrawals);
return 0;
}

```

// Exercise 5: Inventory Management System
#include <stdio.h>

```

int main() {
    int product_id, quantity;
    char operation;
    int stock[100] = {0};
    int low_stock_threshold = 10;
    int transactions = 0;
    float prices[100] = {0};
    float total_value = 0;

    // Initialize product prices
    printf("Enter prices for products 1-100:\n");
    for(int i = 0; i < 100; i++) {
        printf("Product %d: $", i+1);
        scanf("%f", &prices[i]);
    }

    printf("\nOperations:\n");
    printf("I - Increase stock\n");
    printf("D - Decrease stock\n");
    printf("X - Exit\n");

    while(1) {
        printf("\nEnter operation (I/D/X): ");
        scanf(" %c", &operation);

        if(operation == 'X' || operation == 'x') break;
    }
}

```



```

printf("Enter product ID (1-100): ");
scanf("%d", &product_id);

if(product_id < 1 || product_id > 100) {
    printf("Invalid product ID!\n");
    continue;
}

printf("Enter quantity: ");
scanf("%d", &quantity);

switch(operation) {
    case 'I':
    case 'i':
        stock[product_id-1] += quantity;
        transactions++;
        break;
    case 'D':
    case 'd':
        if(quantity > stock[product_id-1]) {
            printf("Insufficient stock!\n");
            continue;
        }
        stock[product_id-1] -= quantity;
        transactions++;
        break;
    default:
        printf("Invalid operation!\n");
}

}

printf("\n\nInventory Summary:\n");
printf("Low stock items (below %d units):\n", low_stock_threshold)
for(int i = 0; i < 100; i++) {
    total_value += stock[i] * prices[i];
    if(stock[i] < low_stock_threshold && stock[i] > 0) {
        printf("Product %d: %d units\n", i+1, stock[i]);
    }
}

printf("\n\nTotal Transactions: %d\n", transactions);
printf("Total Inventory Value: $%.2f\n", total_value);
return 0;
}

```

Programs Using Goto

```

// Exercise 1: ATM System with Transaction Limits
#include <stdio.h>

```

```

int main() {
    float balance = 1000.00;
    int pin = 1234, attempts = 3;
    int input_pin;
    float amount;
    char choice;

start:
    printf("\nEnter PIN: ");
    scanf("%d", &input_pin);

    if(input_pin != pin) {
        attempts--;
        if(attempts == 0) {
            printf("Card blocked! Please contact bank.\n");
            goto end;
        }
        printf("Wrong PIN! %d attempts remaining.\n", attempts);
        goto start;
    }

menu:
    printf("\n1. Withdraw (w)\n2. Deposit (d)\n3. Balance (b)\n4. E
    scanf(" %c", &choice);

    switch(choice) {
        case 'w':
            printf("Enter amount: $");
            scanf("%f", &amount);
            if(amount > balance || amount > 500) {
                printf("Invalid amount or limit exceeded!\n");
                goto menu;
            }
            balance -= amount;
            break;
        case 'd':
            printf("Enter amount: $");
            scanf("%f", &amount);
            if(amount > 1000) {
                printf("Deposit limit exceeded!\n");
                goto menu;
            }
            balance += amount;
            break;
        case 'b':
            printf("Current balance: $%.2f\n", balance);
            goto menu;
        case 'x':
            goto end;
        default:
            printf("Invalid choice!\n");
            goto menu;
    }
}

```

```

    }
    goto menu;

end:
    printf("Thank you for using our ATM!\\n");
    return 0;
}

// Exercise 2: Inventory System with Error Handling
#include <stdio.h>

int main() {
    int items[5] = {100, 150, 200, 250, 300};
    int item_id, quantity;
    char operation;

input:
    printf("\\nEnter item ID (1-5): ");
    scanf("%d", &item_id);

    if(item_id < 1 || item_id > 5) {
        printf("Invalid item ID!\\n");
        goto input;
    }

operation_input:
    printf("Add or Remove (a/r): ");
    scanf(" %c", &operation);

    if(operation != 'a' && operation != 'r') {
        printf("Invalid operation!\\n");
        goto operation_input;
    }

quantity_input:
    printf("Enter quantity: ");
    scanf("%d", &quantity);

    if(quantity <= 0) {
        printf("Invalid quantity!\\n");
        goto quantity_input;
    }

    if(operation == 'r' && quantity > items[item_id-1]) {
        printf("Insufficient stock!\\n");
        goto quantity_input;
    }

    items[item_id-1] = (operation == 'a') ?
                        items[item_id-1] + quantity :
                        items[item_id-1] - quantity;

```

```

    printf("\nUpdated stock for item %d: %d\n", item_id, items[item_i]);

continue_prompt:
    printf("\nContinue? (y/n): ");
    scanf(" %c", &operation);

    if(operation == 'y') goto input;
    else if(operation == 'n') goto end;
    else {
        printf("Invalid choice!\n");
        goto continue_prompt;
    }

end:
    printf("\nFinal Stock Levels:\n");
    for(int i = 0; i < 5; i++) {
        printf("Item %d: %d\n", i+1, items[i]);
    }
    return 0;
}

```

// Exercise 3: Student Grade Calculator with Validation
#include <stdio.h>

```

int main() {
    int num_subjects;
    float marks, total = 0, percentage;

input_subjects:
    printf("Enter number of subjects (1-10): ");
    scanf("%d", &num_subjects);

    if(num_subjects < 1 || num_subjects > 10) {
        printf("Invalid number of subjects!\n");
        goto input_subjects;
    }

    for(int i = 1; i <= num_subjects; i++) {
mark_input:
        printf("Enter marks for subject %d (0-100): ", i);
        scanf("%f", &marks);

        if(marks < 0 || marks > 100) {
            printf("Invalid marks!\n");
            goto mark_input;
        }
        total += marks;
    }

    percentage = total / num_subjects;
    printf("\nTotal Marks: %.2f\n", total);
    printf("Percentage: %.2f%\n", percentage);
}

```

```

    if(percentage >= 90) goto grade_a;
    if(percentage >= 80) goto grade_b;
    if(percentage >= 70) goto grade_c;
    goto grade_f;

grade_a:
    printf("Grade: A\\n");
    goto end;
grade_b:
    printf("Grade: B\\n");
    goto end;
grade_c:
    printf("Grade: C\\n");
    goto end;
grade_f:
    printf("Grade: F\\n");

end:
    return 0;
}

```

```

// Exercise 4: Temperature Conversion System
#include <stdio.h>

```

```

int main() {
    float temp;
    char scale, choice;

start:
    printf("\\nSelect conversion:\\n");
    printf("1. Celsius to Fahrenheit (c)\\n");
    printf("2. Fahrenheit to Celsius (f)\\n");
    printf("3. Exit (x)\\n");

    scanf(" %c", &choice);

    if(choice == 'x') goto end;
    if(choice != 'c' && choice != 'f') {
        printf("Invalid choice!\\n");
        goto start;
    }

input_temp:
    printf("Enter temperature: ");
    scanf("%f", &temp);

    if(choice == 'c') {
        temp = (temp * 9/5) + 32;
        printf("%.2f°C = %.2f°F\\n", (temp-32) * 5/9, temp);
    } else {
        temp = (temp - 32) * 5/9;
    }
}

```

```

        printf("%.2f°F = %.2f°C\\n", (temp * 9/5) + 32, temp);
    }

continue_prompt:
    printf("\\nContinue? (y/n): ");
    scanf(" %c", &choice);

    if(choice == 'y') goto start;
    if(choice == 'n') goto end;

    printf("Invalid choice!\\n");
    goto continue_prompt;

end:
    printf("Thank you for using the converter!\\n");
    return 0;
}

// Exercise 5: Bank Account Management
#include <stdio.h>

int main() {
    float accounts[3] = {1000.0, 2000.0, 3000.0};
    int acc_num;
    float amount;
    char operation;

login:
    printf("Enter account number (1-3): ");
    scanf("%d", &acc_num);

    if(acc_num < 1 || acc_num > 3) {
        printf("Invalid account!\\n");
        goto login;
    }

menu:
    printf("\\nBalance: $%.2f\\n", accounts[acc_num-1]);
    printf("1. Deposit (d)\\n2. Withdraw (w)\\n3. Exit (x)\\n");
    scanf(" %c", &operation);

    if(operation == 'x') goto end;
    if(operation != 'd' && operation != 'w') {
        printf("Invalid operation!\\n");
        goto menu;
    }

amount_input:
    printf("Enter amount: $");
    scanf("%f", &amount);

    if(amount <= 0) {

```

```

        printf("Invalid amount!\\n");
        goto amount_input;
    }

    if(operation == 'w' && amount > accounts[acc_num-1]) {
        printf("Insufficient funds!\\n");
        goto amount_input;
    }

    accounts[acc_num-1] = (operation == 'd') ?
        accounts[acc_num-1] + amount :
        accounts[acc_num-1] - amount;

    goto menu;

end:
    printf("Transaction complete!\\n");
    return 0;
}

```

Programs Without Goto

```

// Exercise 6: Library Management System
#include <stdio.h>

int main() {
    int books[100] = {0}; // 0=available, 1=borrowed
    int book_id, member_id;
    char operation;
    int total_borrowed = 0;

    while(1) {
        printf("\\n1. Borrow Book (b)\\n2. Return Book (r)\\n");
        printf("3. Check Status (c)\\n4. Exit (x)\\n");
        scanf(" %c", &operation);

        if(operation == 'x') break;

        switch(operation) {
            case 'b':
                printf("Enter book ID (1-100): ");
                scanf("%d", &book_id);
                if(book_id < 1 || book_id > 100) {
                    printf("Invalid book ID!\\n");
                    continue;
                }
                if(books[book_id-1] == 1) {
                    printf("Book already borrowed!\\n");
                    continue;
                }
                printf("Enter member ID: ");

```

```

        scanf("%d", &member_id);
        books[book_id-1] = 1;
        total_borrowed++;
        break;

    case 'r':
        printf("Enter book ID (1-100): ");
        scanf("%d", &book_id);
        if(book_id < 1 || book_id > 100) {
            printf("Invalid book ID!\n\n");
            continue;
        }
        if(books[book_id-1] == 0) {
            printf("Book already returned!\n\n");
            continue;
        }
        books[book_id-1] = 0;
        total_borrowed--;
        break;

    case 'c':
        printf("\n\nLibrary Status:\n\n");
        printf("Total borrowed: %d\n", total_borrowed);
        printf("Available: %d\n", 100 - total_borrowed);
        break;

    default:
        printf("Invalid operation!\n\n");
}
}
return 0;
}

```

// Exercise 7: Restaurant Order System

```

#include <stdio.h>

int main() {
    float prices[] = {10.99, 8.50, 15.75, 12.25};
    int quantities[4] = {0};
    char choice;
    float total = 0;

    while(1) {
        printf("\n\nMenu:\n\n");
        printf("1. Burger ($10.99)\n\n");
        printf("2. Pizza ($8.50)\n\n");
        printf("3. Steak ($15.75)\n\n");
        printf("4. Salad ($12.25)\n\n");
        printf("5. Finish Order (f)\n\n");

        scanf(" %c", &choice);
    }
}

```



```

        if(choice == 'f') break;

        int item = choice - '1';
        if(item < 0 || item > 3) {
            printf("Invalid choice!\\n");
            continue;
        }

        printf("Enter quantity: ");
        int qty;
        scanf("%d", &qty);

        if(qty <= 0) {
            printf("Invalid quantity!\\n");
            continue;
        }

        quantities[item] += qty;
        total += prices[item] * qty;
    }

    printf("\\nOrder Summary:\\n");
    for(int i = 0; i < 4; i++) {
        if(quantities[i] > 0) {
            printf("Item %d: %d x $%.2f = $%.2f\\n",
                i+1, quantities[i], prices[i],
                prices[i] * quantities[i]);
        }
    }
    printf("Total: $%.2f\\n", total);
    return 0;
}

```

// Exercise 8: Employee Attendance System
#include <stdio.h>

```

int main() {
    int emp_id, hours;
    float rate = 15.0;
    float overtime_rate = 22.5;
    float total_pay = 0;
    int total_employees = 0;

    while(1) {
        printf("\\n\\nEnter Employee ID (0 to finish): ");
        scanf("%d", &emp_id);

        if(emp_id == 0) break;

        printf("Enter hours worked: ");
        scanf("%d", &hours);
    }
}

```

```

    if(hours < 0 || hours > 168) {
        printf("Invalid hours!\n");
        continue;
    }

    float pay;
    if(hours <= 40) {
        pay = hours * rate;
    } else {
        pay = (40 * rate) + ((hours - 40) * overtime_rate);
    }

    printf("Employee %d pay: $%.2f\n", emp_id, pay);
    total_pay += pay;
    total_employees++;
}

if(total_employees > 0) {
    printf("\nPayroll Summary:\n");
    printf("Total Employees: %d\n", total_employees);
    printf("Total Payroll: $%.2f\n", total_pay);
    printf("Average Pay: $%.2f\n", total_pay/total_employees);
}
return 0;
}

```