SASTRA

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

DEEMED TO BE UNIVERSITY

(U/S 3 of the UGC Act,1956)

THINK MERIT | THINK TRANSPARENCY | THINK SASTRA

CSE213

OBJECT ORIENTED PROGRAMMING

Unit-1

Unit-1

Procedural programming, **An Overview of C:** Types- Fundamental types, boolean, Character type, Integer types, Float types, prefixes and suffixes, void, **Declarations**-scope, Reference data types, Variables, Declared Constants, enumerated constants, the typecasting operator, Operator and Expressions, **Input and Output (C-way),** Statements- Statement Summary, Declarations as Statements, Selection Statements, Iteration Statements, goto Statements, Pointers, Arrays, and References-Pointers, arrays, pointers into arrays, pointers and const, Pointers and Ownership, References, **Functions** - Function Declarations, argument passing-reference argument, array arguments, overloaded functions, Error handling, Namespaces, **Preprocessor directive:** Trigraph sequence, Digraph Sequence, #define, #undef, #ifdef, #ifndef, #if, #endif, #else, #elif and #line

What is C & C++?

- C is a structural or procedural oriented programming language which is machine-independent and extensively used in various applications.
- It can be used to develop from the operating systems (like Windows) to complex programs like Oracle database, Git, Python interpreter, and many more.
- C++ is special-purpose programming language **Developed by Bjarne Stroustrup** at Bell Labs circa 1980.
- C++ is an object-oriented programming language.
- Well-structured programming language than C.

The following are the differences between C and C++:

- **Definition**
  C is a structural programming language, and it does not support classes and objects, while C++ is an object-oriented programming language that supports the concept of classes and objects.
- **Type of programming language**
  - C is the structural programming language the code is checked line by line
  - C++ is an object-oriented programming language that supports the concept of classes and objects.

The following are the differences between C and C++:

- **Developer of the language**
- Dennis Ritchie developed C language at Bell Laboratories while Bjarne Stroustrup developed the C++ language at Bell Labs circa 1980.
- **Subset**
  C++ is a superset of C programming language. C++ can run 99% of C code but C language cannot run C++ code.
- **Type of Approach**
- C follows the top-down approach, while C++ follows the bottom-up approach. The top-down approach breaks the main modules into tasks; these tasks are broken into sub-tasks, and so on. The bottom-down approach develops the lower level modules first and then the next level modules.

The following are the differences between C and C++:

- **Security**
  In C, the data can be easily manipulated by the outsiders as it does not support the encapsulation and information hiding while C++ is a very secure language, i.e., no outsiders can manipulate its data as it supports both encapsulation and data hiding. In C language, functions and data are the free entities, and in C++ language, all the functions and data are encapsulated in the form of objects.
- **Function Overloading**
- Function overloading is a feature that allows you to have more than one function with the same name but varies in the parameters. C does not support the function overloading, while C++ supports the function overloading.
- **Function Overrriding**
- Function overriding is a feature that provides the specific implementation to the function, which is already defined in the base class. C does not support the function overriding, while C++ supports the function overriding.

The following are the differences between C and C++:

- **Reference variables**
- C does not support the reference variables, while C++ supports the reference variables.
- **Keywords**
  C contains 32 keywords, and C++ supports 52 keywords.
- **Namespace feature**
- A namespace is a feature that groups the entities like classes, objects, and functions under some specific name. C does not contain the namespace feature, while C++ supports the namespace feature that avoids the name collisions.
- **Exception Handling**
- C does not provide direct support to the exception handling; it needs to use functions that support exception handling. C++ provides direct support to exception handling by using a try-catch block.

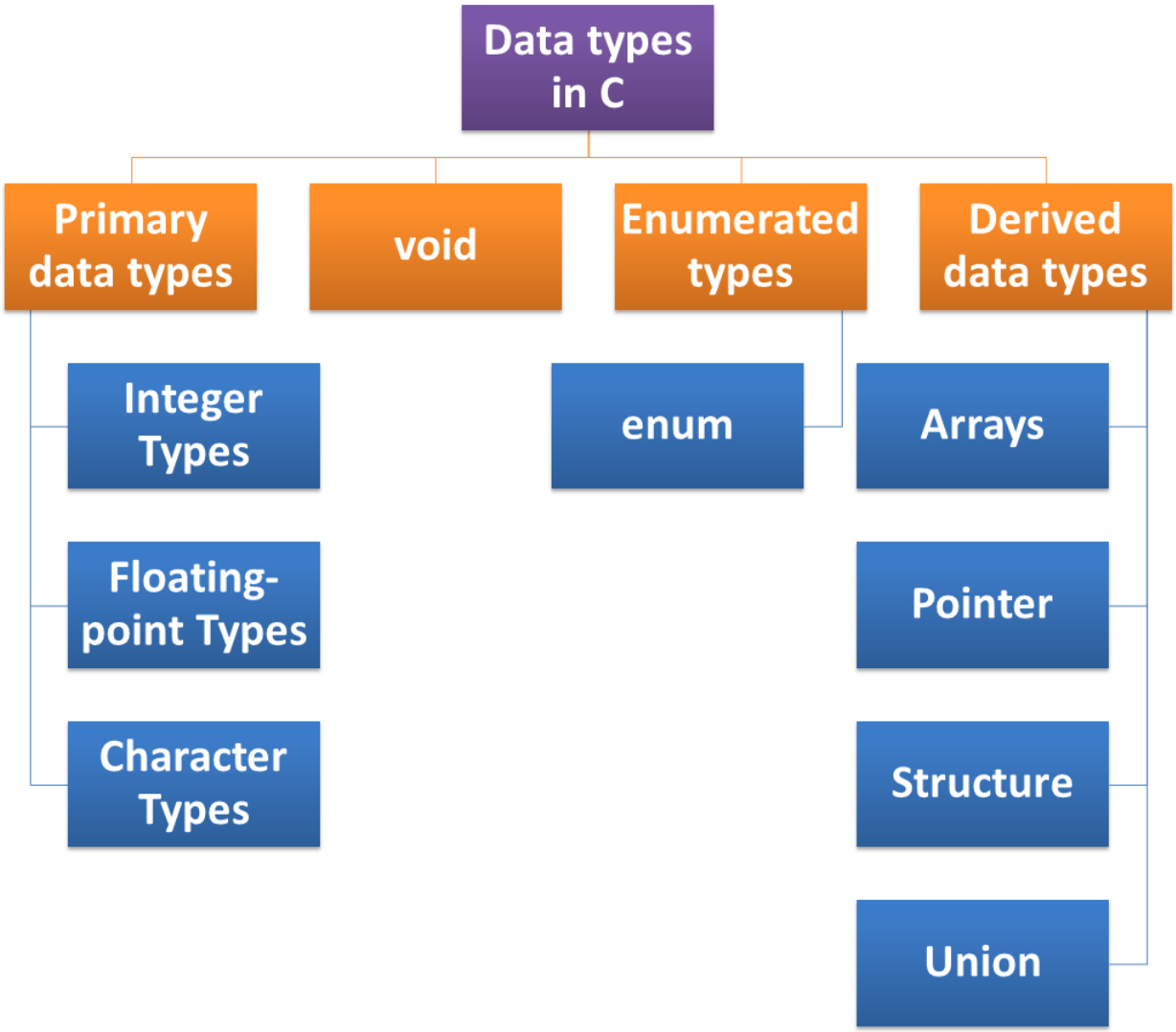
The following are the differences between C and C++:

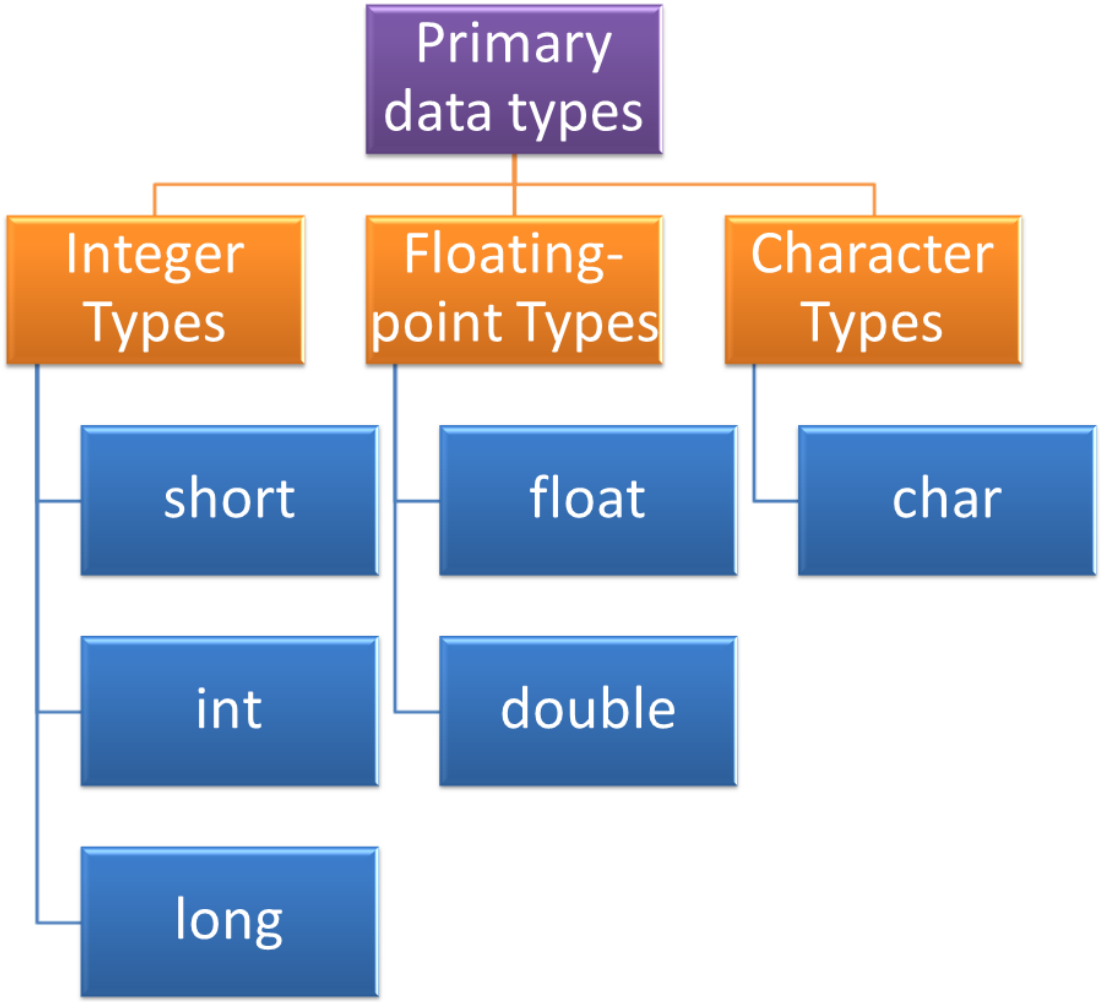
- **Input/Output functions**
  In C, scanf and printf functions are used for input and output operations, respectively, while in C++, cin and cout are used for input and output operations, respectively.
- **Memory allocation and Deallcoation**
- C supports calloc() and malloc() functions for the memory allocation, and free() function for the memory de-allocation. C++ supports a new operator for the memory allocation and delete operator for the memory de-allocation.
- **Inheritance**
  Inheritance is a feature that allows the child class to reuse the properties of the parent class. C language does not support the inheritance while C++ supports the inheritance.
- **Header file**
- C program uses **<stdio.h>** header file while C++ program uses **<iostream.h>** header file.

Let's summarize the above differences in a tabular form.

No.	C	C++
1)	C follows the procedural style programming.	C++ is multi-paradigm. It supports both procedural and object oriented.
2)	Data is less secured in C.	In C++, you can use modifiers for class members to make it inaccessible for outside users.
3)	C follows the top-down approach.	C++ follows the bottom-up approach.
4)	C does not support function overloading.	C++ supports function overloading.
5)	In C, you can't use functions in structure.	In C++, you can use functions in structure.
6)	C does not support reference variables.	C++ supports reference variables.
7)	In C, scanf() and printf() are mainly used for input/output.	C++ mainly uses stream cin and cout to perform input and output operations.
8)	Operator overloading is not possible in C.	Operator overloading is possible in C++.
9)	C programs are divided into procedures and modules	C++ programs are divided into functions and classes.
10)	C does not provide the feature of namespace.	C++ supports the feature of namespace.
11)	Exception handling is not easy in C. It has to perform using other functions.	C++ provides exception handling using Try and Catch block.
12)	C does not support the inheritance.	C++ supports inheritance.

Data Types


Data types in C
Primary data types
void
Enumerated types
Derived data types
Integer Types
Floating-point Types
Character Types
enum
Arrays
Pointer
Structure
Union


Primary data types
Integer Types
Floating-point Types
Character Types
short
int
long
float
double
char

- Primary Datatype or Basic Datatype
- The memory size of basic data types may change according to 32 or 64 bit operating system. Let's see the basic data types. Its size is given according to 32 bit architecture.
- **Integer Type Datatype**

Data Types	Memory Size	Range
int	2 byte	-32,768 to 32,767
signed int	2 byte	-32,768 to 32,767
unsigned int	2 byte	0 to 65,535
short int	2 byte	-32,768 to 32,767
signed short int	2 byte	-32,768 to 32,767
unsigned short int	2 byte	0 to 65,535
long int	4 byte	-2,147,483,648 to 2,147,483,647
signed long int	4 byte	-2,147,483,648 to 2,147,483,647
unsigned long int	4 byte	0 to 4,294,967,295

Floating-points Datatypes

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

Character Types Datatypes

Data Types	Memory Size	Range
char	1 byte	-128 to 127
signed char	1 byte	-128 to 127
unsigned char	1 byte	0 to 255

sizeof(DataType) Operator

To get the exact size of a type or a variable on a particular platform, you can use the sizeof operator.

```
#include <stdio.h>
int main()
{
   printf("Storage size for int : %d \n", sizeof(int));
    return 0;
}
```

A simple Explanation of Basic data types with the help of code :

```
#include <stdio.h>

int main() {
  // Create variables
  int rollno = 15;                    // Integer (whole number)
  char classSection = 'A';              //character
  float gradepoints = 8.5f;             // Floating point number
  double calculations = 228.5659895;      // double

  // Print variables
  printf("The Value of int is : %d\  n", rollno);
  printf("The Value of char is : %c\  n", classSection);
  printf("The Value of float is : %f\  n", gradepoints);
  printf("The Value of double is : %lf\n", calculations);

  return 0;
}
```

Output

- The Value of int is : 15
  The Value of char is : A
  The Value of float is : 8.500000
  The Value of double is : 228.565990

Void Data types

- In C, the void data type is used to indicate that no value is present. It does not return a value to the receiver. It lacks both values and functions. It is used to indicate nothing. Void is used in a variety of contexts, including function return type, function parameters as void, and pointers to void.
- **Function returns as void :** There are various functions in C which do not return any value or you can say they return void. A function with no return value has the return type as void. For example, **void exit (int status)**;
- **Function arguments as void :** There are various functions in C which do not accept any parameter. A function with no parameter can accept a void. For example, **int rand(void)**;
- **Pointers to void :** A pointer of type void * represents the address of an object, but not its type. For example, a memory allocation function **void *malloc(size_t size)**; returns a pointer to void which can be casted to any data type.
- void sum (int a, int b);

- The above example shows that function will not return any value to the calling function.

Enumerated data type

enumerations" = "specifically listed"

An enumerated data type is a user-defined data type made up of integer constants, each with its own identifier. The enumerated data type is defined by the term "enum".

Syntax

In C, an enum is specified by using the 'enum' keyword, and the constants within are separated by a comma. The fundamental syntax for defining an enum is as follows:

In the below syntax, the default value of int_const1 is 0, int_const2 is 1, int_const3 is 2, and so on.

However, you can alter these default numbers after the enum is declared.

enum variable{int_const1, int_const2, int_const3, …. int_constN};

Sample Code

```
#include <stdio.h>
int main()
{
enum week{Sun, Mon, Tue, Wed, Thu, Fri, Sat};
printf("Sun = %d", Sun);
printf("\nMon = %d", Mon);
printf("\nTue = %d", Tue);
printf("\nWed = %d", Wed);
printf("\nThu = %d", Thu);
printf("\nFri = %d", Fri);
printf("\nSat = %d", Sat);
return 0;
}
```

Output : Sun = 0 Mon = 1 Tue = 2 Wed = 3 Thu = 4 Fri = 5 Sat = 6

Find the output for below code

1. #include <stdio.h>

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

int main()

{

// creating today variable of enum week type

enum week today;

today = Wednesday;

printf("Day %d",today+1);

return 0;

}

Find the output for below code

2) #include <stdio.h>

```
enum suit {
  club = 0,
  diamonds = 10,
  hearts = 20,
 spades = 3
} card;

int main()
{
   card = diamonds;

printf( "%d ", card);

return 0;
  }
```

3)CPP Code to demonstrate the enum datatype

```
#include <bits/stdc++.h>
using namespace std;
// Defining enum Year
enum year { Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct,
   Nov, Dec };
  // Driver Code
int main()
{
   int i;

   // Traversing the year enum
   for (i = Jan; i <= Dec; i++)
      cout << i << " ";

   return 0;
}
```

- **Derived data type**

Derived data types are collections of main data types. Many elements of comparable data types can be grouped together. The user defines these data types. The derived data structures in C are Array, Pointer, Union and Structure.

- **Array :**

In C, an array is a collection of numerous values of the same data format that are stored in a single memory location. An array can contain chars, integers, doubles, and so on.

- **Syntax**

datatype arrayname[arraysize];

- **Pointer :**
- The pointer data type is used to store the address of another variable. A pointer can hold the address of any data type variable. Users can conduct dynamic memory allocation using pointers. They are also useful for passing values by reference.
- A null pointer is a pointer that has no location. A void Pointer is a reference that has no data type. The '*' operator is used to describe it.

Structure :

It is a data type that can contain variables of either comparable or dissimilar data types. Structures, for example, can be used to hold information about an employee, uch as the employee's name, employee ID, salary, and so on. Each employee's record will be depicted by a structure object. The structure's size is the sum of the storage sizes needed by each variable. The keyword struct describes a structure.

Union :

The union type is in charge of storing objects of various types in the same place in memory. It implies that in any program, different types of union members can occupy the same location at different times. A union declaration contains all of the union's members. It lists all of the possible object types that a union can contain. A typical union can hold any one of its members at any given moment. Any later assignment of another union member would result in the overwriting of the existing object in the same specified storage area.

C Boolean

In C, Boolean is a data type that contains two types of values, i.e., 0 and 1. Basically, the bool type value represents two types of behavior, either true or false. Here, '0' represents false value, while '1' represents true value.
In C Boolean, '0' is stored as 0, and another integer is stored as 1. We do not require to use any header file to use the Boolean data type in C++, but in C, we have to use the header file, i.e., **stdbool.h**. If we do not use the header file, then the program will not compile.

Syntax

```
bool variable_name;
```

In the above syntax, **bool** is the data type of the variable, and **variable_name** is the name of the variable.

Let's understand through an example.

```
#include <stdio.h>
#include<stdbool.h>
int main()
{
bool x=false; // variable initialization.
if(x==true) // conditional statements
{
printf("The value of x is true");
}
else
printf("The value of x is FALSE");
return 0;
}
```

In the above code, we have used **<stdbool.h>** header file so that we can use the bool type variable in our program. After the declaration of the header file, we create the bool type variable '**x**' and assigns a '**false**' value to it. Then, we add the conditional statements, i.e., **if..else**, to determine whether the value of 'x' is true or not.

Output

```
The value of x is FALSE
```

Let's understand through another example.

```
#include <stdio.h>
#include <stdbool.h> // Include the header for the bool datatype

int main() {
    int number;
    bool isEven; // Declare a bool variable

    printf("Enter an integer: ");
    scanf("%d", &number);

    // Check if the number is even
    if (number % 2 == 0) {
        isEven = true; // Assign true to the bool variable
    } else {
        isEven = false; // Assign false to the bool variable
    }
    // Print the result
    if (isEven) {
        printf("%d is even.\n", number);
    } else {
        printf("%d is odd.\n", number);
    }

    return 0;
}
```

Reference Data Type:

A reference data type is a type of data that stores a reference or memory address rather than the actual value. This reference points to the location in memory where the data is stored.

Example:

```
#include <stdio.h>
void incrementByReference(int *numPtr)
 {
   (*numPtr)++;
}

int main()
{
   int value = 5;

   printf("Original value: %d\n", value);

   incrementByReference(&value); // Pass the address of the variable

   printf("Incremented value: %d\n", value);

   return 0;
}
```

SCOPE:

In programming, the scope refers to the region of the code where a variable, function, or other named entity can be accessed and manipulated. The scope determines the visibility and lifetime of these entities. There are generally two types of scopes:

Global Scope: Variables or functions defined in the global scope are accessible from anywhere in the code, including within functions and other blocks. They have a global lifetime, existing throughout the program's execution.

Local Scope: Variables or functions defined within a block, such as a function, have local scope. They are only accessible within that specific block and have a lifetime that corresponds to the duration of the block's execution.

Example of Scope data type

```
#include <stdio.h>
int globalVar = 10; // Global variable
void foo()
{
    int localVar = 20; // Local variable
    printf("Local variable: %d\n", localVar);
}
int main()
{
    printf("Global variable: %d\n", globalVar);
    foo();
    printf("Local variable: %d\n", localVar);
    return 0;
}
```

C- TypeCasting

Typecasting in C is the process of converting one data type to another data type by the programmer using the casting operator during program design.

In typecasting, the destination data type may be smaller than the source data type when converting the data type to another data type, that's why it is also called **narrowing conversion**.

Syntax:

```
int x;
float y;
y = (float) x;
```

Types of Type Casting in C

In C there are two major types to perform type casting.
Implicit type casting
Explicit type casting

1. Implicit Type Casting

Implicit type casting in C is used to convert the data type of any variable without using the actual value that the variable holds. It performs the conversions without altering any of the values which are stored in the data variable. Conversion of lower data type to higher data type will occur automatically.
Integer promotion will be performed first by the compiler. After that, it will determine whether two of the operands have different data types. Using the hierarchy below, the conversion would appear as follows if they both have varied data types:

2. Explicit Type Casting

There are some cases where if the datatype remains unchanged, it can give incorrect output. In such cases, typecasting can help to get the correct output and reduce the time of compilation. In explicit type casting, we have to force the conversion between data types. This type of casting is explicitly defined within the program.

C- TypeCasting -Implicit

```
#include <stdio.h>
 int main() {
 int numInt = 10;
 float numFloat;

 numFloat = numInt; // Implicit type casting from int to float

 printf("Integer: %d\n", numInt);
 printf("Float: %f\n", numFloat);

 return 0;
}
```

C- TypeCasting -Explicit

```
#include <stdio.h>

    int main() {
    float numFloat = 5.67;
    int numInt;

    numInt = (int) numFloat; // Explicit type casting from float to int

    printf("Float: %f\n", numFloat);
    printf("Integer: %d\n", numInt);

    return 0;
}
```

Operators and Expresssions

Operators are symbols or special keywords that are used to perform various operations on variables and values. Expressions, on the other hand, are combinations of variables, constants, and operators that can be evaluated to produce a single value.

Operators can be classified into different categories based on their functionality. Here are some common categories of operators along with brief explanations:

Arithmetic Operators: These operators perform basic mathematical operations.

+ Addition
- Subtraction
* Multiplication
/ Division
% Modulus (remainder after division)

Assignment Operators: These operators assign values to variables.

= Assignment
+= Add and assign
-= Subtract and assign
*= Multiply and assign
/= Divide and assign
%= Modulus and assign

Comparison Operators: These operators compare values.

== Equal to
!= Not equal to
< Less than
> Greater than
<= Less than or equal to
>= Greater than or equal to

Logical Operators: These operators perform logical operations.

&& Logical AND
|| Logical OR
! Logical NOT

Increment and Decrement Operators:

++ Increment
-- Decrement
Bitwise Operators: These operators perform operations at the bit level.

& Bitwise AND
| Bitwise OR
^ Bitwise XOR
~ Bitwise NOT
<< Left shift
>> Right shift
Conditional (Ternary) Operator: An operator that allows for conditional expressions.

condition ? expr1 : expr2
Comma Operator: An operator that separates expressions and evaluates them from left to right.

Pointer Operators: Used with pointers to access memory addresses and values.

*** Dereference operator** (to access value through a pointer)
& Address-of operator (to get the address of a variable)
Member Access Operators: Used to access members of structures and classes.

. Dot operator (for structures)
-> Arrow operator (for pointers to structures)
Sizeof Operator: Used to determine the size (in bytes) of a data type or expression.

These are just a few examples of operators in programming. Expressions, as mentioned earlier, are combinations of operators and operands (variables, constants) that produce a single value. Expressions can involve multiple operators and follow a specific order of evaluation, known as operator precedence and associativity.

For example, in the expression 2 + 3 * 4, the multiplication (*) is evaluated before the addition (+), resulting in the value 14. To change the order of evaluation, parentheses can be used: (2 + 3) * 4 would give the value 20.

Program to check given number is Odd of Even

```
#include <stdio.h>

int main() {
    int num;

    // Input a number from the user
    printf("Enter an integer: ");
    scanf("%d", &num);

    // Using the ternary operator to check if the number is even or odd
    (num % 2 == 0) ? printf("%d is even.\n", num) : printf("%d is odd.\n", num);

    return 0;
}
```

Maximum of two numbers

```
#include <stdio.h>

int main() {
    int num1, num2, max;

    // Input two numbers from the user
    printf("Enter the first number: ");
    scanf("%d", &num1);

    printf("Enter the second number: ");
    scanf("%d", &num2);

    // Using the ternary operator to find the maximum of the two numbers
    max = (num1 > num2) ? num1 : num2;

    printf("The maximum of %d and %d is %d.\n", num1, num2, max);

    return 0;
}
```

Sample Program to Demonstrate Various Operators and Expression

```
include <stdio.h>
int main() {
    // Arithmetic operators
    int a = 10, b = 5;
    int sum = a + b;
    int difference = a - b;
    int product = a * b;
    int quotient = a / b;
    int remainder = a % b;

    // Comparison operators
    int x = 7, y = 3;
    int isEqual = x == y;
    int isGreaterThan = x > y;
    int isLessThan = x < y;

    // Logical operators
    int p = 1, q = 0;
    int logicalAnd = p && q;
    int logicalOr = p || q;
    int logicalNotP = !p;
    int logicalNotQ = !q;

// Increment and decrement operators
    int num = 5;
    num++; // Increment by 1
    num--; // Decrement by 1

    // Ternary (conditional) operator
    int age = 18;
    char* result = (age >= 18) ? "Adult" : "Minor";

    // Bitwise operators
    int num1 = 5, num2 = 3;
    int bitwiseAnd = num1 & num2;
    int bitwiseOr = num1 | num2;
    int bitwiseXor = num1 ^ num2;
    int bitwiseComplement = ~num1;
    int leftShift = num1 << 1;
    int rightShift = num1 >> 1;
```

```
// Sizeof operator
    int sizeOfInt = sizeof(int);
    printf("Sum: %d\n", sum);
    printf("Difference: %d\n", difference);
    printf("Product: %d\n", product);
    printf("Quotient: %d\n", quotient);
    printf("Remainder: %d\n", remainder);
    printf("Is Equal: %d\n", isEqual);
    printf("Is Greater Than: %d\n", isGreaterThan);
    printf("Is Less Than: %d\n", isLessThan);
    printf("Logical AND: %d\n", logicalAnd);
    printf("Logical OR: %d\n", logicalOr);
    printf("Logical NOT P: %d\n", logicalNotP);
    printf("Logical NOT Q: %d\n", logicalNotQ);
    printf("Num: %d\n", num);
    printf("Result: %s\n", result);
    printf("Bitwise AND: %d\n", bitwiseAnd);
    printf("Bitwise OR: %d\n", bitwiseOr);
    printf("Bitwise XOR: %d\n", bitwiseXor);
    printf("Bitwise Complement: %d\n", bitwiseComplement);
    printf("Left Shift: %d\n", leftShift);
    printf("Right Shift: %d\n", rightShift);
    printf("Size of int: %d bytes\n", sizeOfInt);
     return 0;
}
```

Explanations for each section:

Arithmetic operators:

Sum: 10 + 5 = 15

Difference: 10 - 5 = 5

Product: 10 * 5 = 50

Quotient: 10 / 5 = 2

Remainder: 10 % 5 = 0

Comparison operators:

Is Equal: 7 is not equal to 3, so the result is 0.

Is Greater Than: 7 is greater than 3, so the result is 1.

Is Less Than: 7 is not less than 3, so the result is 0.

Logical operators:

Logical AND: 1 (true) AND 0 (false) is false (0).

Logical OR: 1 (true) OR 0 (false) is true (1).

Logical NOT P: NOT 1 (true) is false (0).

Logical NOT Q: NOT 0 (false) is true (1).

Increment and decrement operators:

Num: Initially 5, then incremented by 1, and finally decremented by 1.

Ternary (conditional) operator:

Result: Since age is 18 (greater than or equal to 18), the result is "Adult".

Bitwise operators:

Bitwise AND: 5 AND 3 = 1 (binary: 0101 & 0011 = 0001)

Bitwise OR: 5 OR 3 = 7 (binary: 0101 | 0011 = 0111)

Bitwise XOR: 5 XOR 3 = 6 (binary: 0101 ^ 0011 = 0110)

Bitwise Complement: NOT 5 = -6 (binary: ~0101 = 1010, two's complement representation)

Left Shift: 5 << 1 = 10 (binary: 0101 << 1 = 1010)

Right Shift: 5 >> 1 = 2 (binary: 0101 >> 1 = 0010)

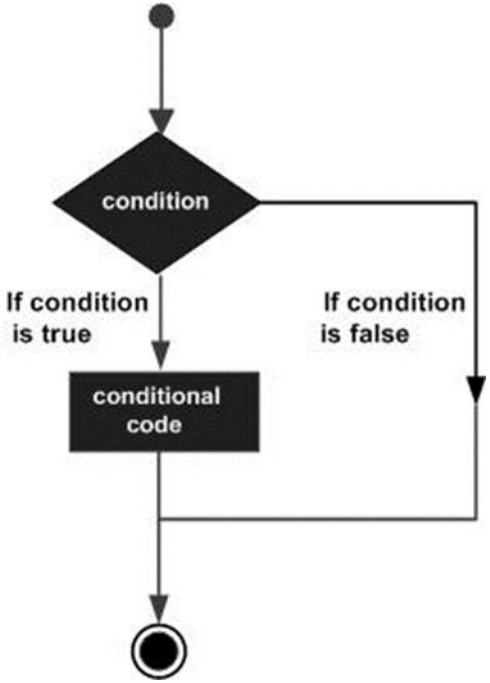
Sizeof operator:

Size of int: The size of an int is 4 bytes on most systems.

Decision Making Statement

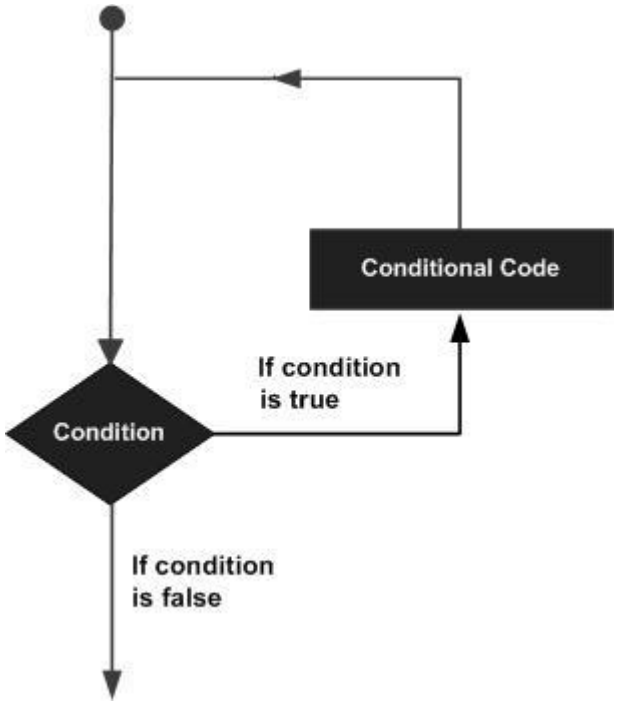
Decision making structures require that the programmer specifies one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Show below is the general form of a typical decision making structure found in most of the programming languages −



Loop Statement

A loop statement allows us to execute a statement or group of statements multiple times. Given below is the general form of a loop statement in most of the programming languages −



1	while loopRepeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
2	for loopExecutes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
3	do...while loopIt is more like a while statement, except that it tests the condition at the end of the loop body.
4	nested loopsYou can use one or more loops inside any other while, for, or do..while loop.

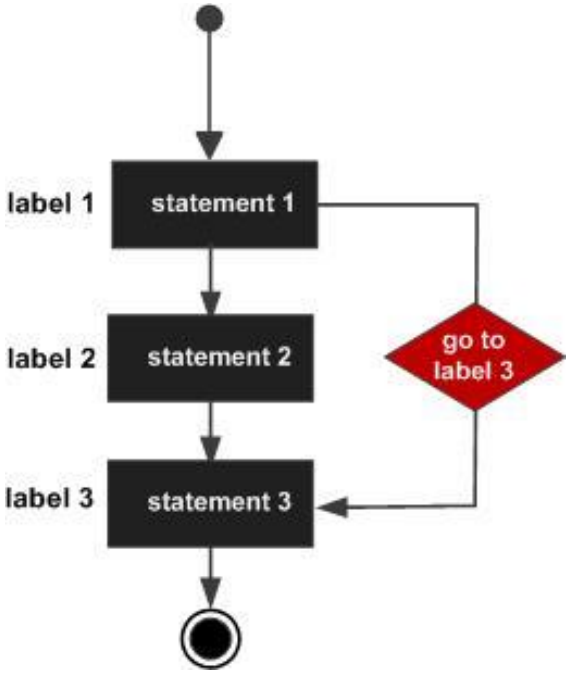
goto

A **goto** statement in C programming provides an unconditional jump from the 'goto' to a labeled statement in the same function.

Syntax:

```
goto label;
 ..
 ..
label: statement;
```

Flow Diagram



//Sample Program

```
#include <stdio.h>
 int main () {
   /* local variable definition */
   int a = 10;
   /* do loop execution */
   LOOP: do {

      if( a == 15) {
         /* skip the iteration */
         a = a + 1;
         goto LOOP;
      }
    printf("value of a: %d\n", a);
      a++;
   }while( a < 20 );
    return 0;
}
```

Output:

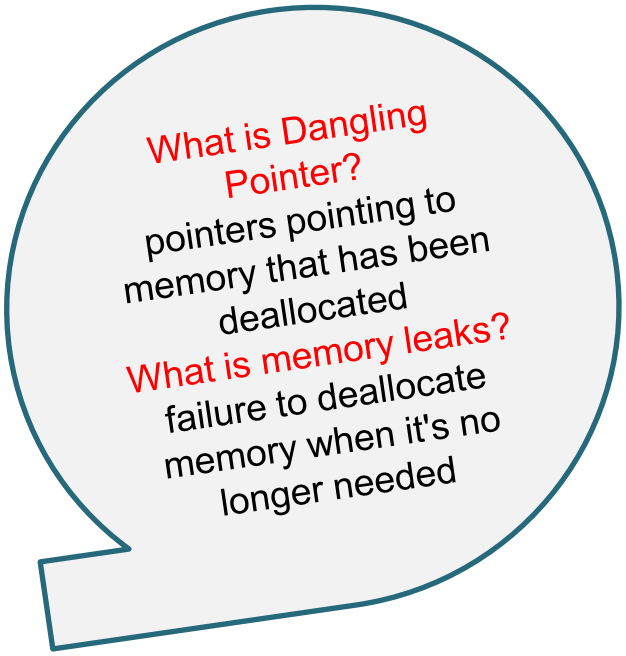
```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
 value of a: 16
value of a: 17
 value of a: 18
 value of a: 19
```

Pointers and Array

```
// C program to understand difference between pointer to an integer and pointer to an array
 of integers.
#include<stdio.h>
int main(){
 // Pointer to an integer
 int *p;
  // Pointer to an array of 5 integers
 int (*ptr)[5];
 int arr[5];
  // Points to 0th element of the arr.
 p = arr;
  // Points to the whole array arr.
 ptr = &arr;
  printf("p = %p, ptr = %p\n", p, ptr);
  p++;
 ptr++;
  printf("p = %p, ptr = %p\n", p, ptr);
  return 0;
}
```

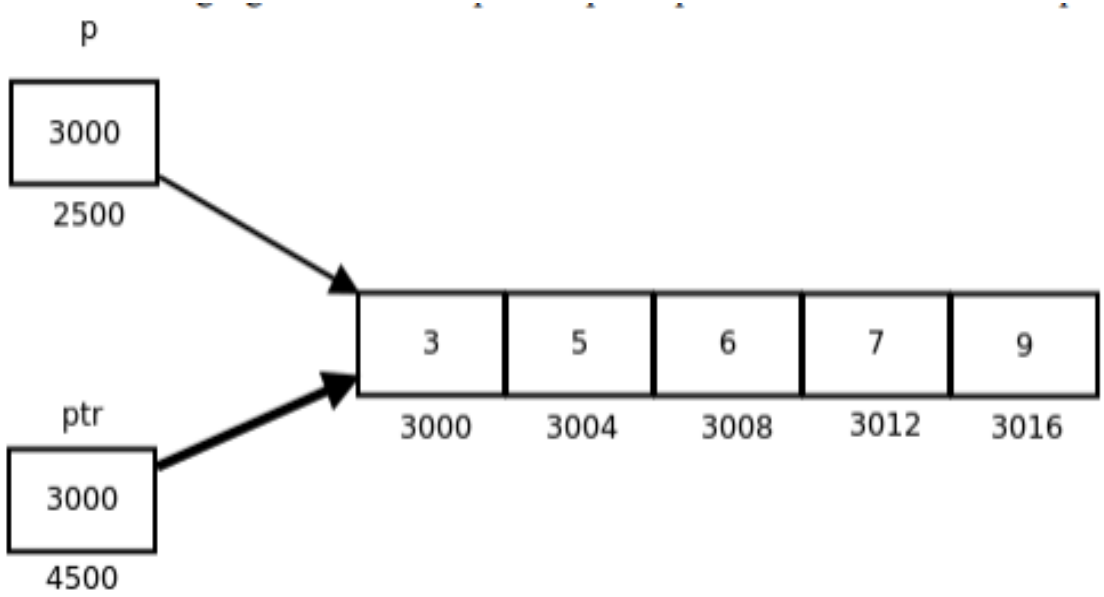
Output:

```
p = 0x7fff4f32fd50, ptr = 0x7fff4f32fd50
p = 0x7fff4f32fd54, ptr = 0x7fff4f32fd64
```



p: is pointer to 0^{th} element of the array arr, while **ptr** is a pointer that points to the whole array arr.

•The base type of p is int while base type of ptr is ‘an array of 5 integers’.
•We know that the pointer arithmetic is performed relative to the base size, so if we write ptr++, then the pointer ptr will be shifted forward by 20 bytes.



```
// C program to illustrate sizes of pointer of array
#include<stdio.h>

int main()
{
        int arr[] = { 3, 5, 6, 7, 9 };
        int *p = arr;
        int (*ptr)[5] = &arr;

        printf("p = %p, ptr = %p\n", p, ptr);
        printf("*p = %d, *ptr = %p\n", *p, *ptr);

        printf("sizeof(p) = %lu, sizeof(*p) = %lu\n",sizeof(p), sizeof(*p));
        printf("sizeof(ptr) = %lu, sizeof(*ptr) = %lu\n",sizeof(ptr), sizeof(*ptr));
        return 0;
}
```

Output:

p = 0x7ffde1ee5010, ptr = 0x7ffde1ee5010
*p = 3, *ptr = 0x7ffde1ee5010
sizeof(p) = 8, sizeof(*p) = 4
sizeof(ptr) = 8, sizeof(*ptr) = 20

Array of Pointers

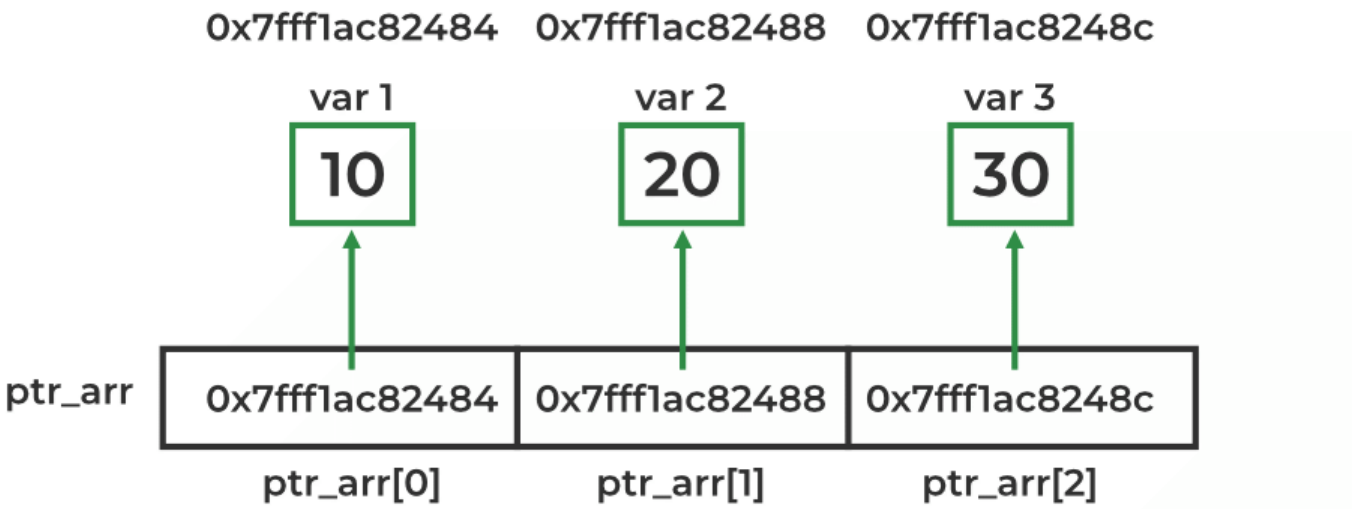
In C, a pointer array is a homogeneous collection of indexed pointer variables that are references to a memory location. It is generally used in C Programming when we want to point at multiple memory locations of a similar data type in our C program. We can access the data by dereferencing the pointer pointing to it.

Syntax: Pointer_type * array_name[array_size];

```
// C program to demonstrate the use of array of pointers
#include <stdio.h>
int main()
{
        // declaring some temp variables
        int var1 = 10;
        int var2 = 20;
        int var3 = 30;
        // array of pointers to integers
        int* ptr_arr[3] = { &var1, &var2, &var3 };
        for (int i = 0; i < 3; i++) {
        printf("Value of var%d: %d\tAddress: %p\n", i + 1, *ptr_arr[i], ptr_arr[i]);
        }
        return 0;
}
```

Output

Value of var1: 10 Address: 0x7fff1ac82484
Value of var2: 20 Address: 0x7fff1ac82488
Value of var3: 30 Address: 0x7fff1ac8248c



// C++ program to illustrate concept of the pointers to constant

```
#include <iostream>
using namespace std;

// Driver Code
int main()
{
     int high =100;
    int low=66; //Syntax const datatype* pointer name
      const int* score=&high;
      cout << *score << "\n"; // print 100
 score = &low;
cout << *score << "\n";// o/p 60

    return 0;
}
```

This code will not work when we add a line *score=70(some value) and print *score.
But it will work when we remove **const** keyword .
Values to the pointer variable cannot be changed.

// C++ program to illustrate concept of the constant pointers

```
#include <iostream>
using namespace std;
// Driver Code
int main()
{           int a{ 90 };
            int b{ 50 };
            int* const ptr{ &a };
            cout << *ptr << "\n";
            cout << ptr << "\n";
            // Address what it points to
            *ptr = 56;
            // Acceptable to change the value of a

            // Error: assignment of read-only
            // variable ‘ptr’
            // ptr = &b;

            cout << *ptr << "\n";
            cout << ptr << "\n";

            return 0;
}
```

Address cannot be changed.

// C++ program to illustrate concept of the **constant pointers to constant**

```
#include <iostream>
using namespace std;

// Driver Code
int main()
{
        const int a{ 50 };
        const int b{ 90 };
        // ptr points to a
        const int* const ptr{ &a };
        // *ptr = 90;
        // Error: assignment of read-only
        // location '*(const int*)ptr'
        // ptr = &b;
        // Error: assignment of read-only
        // variable 'ptr'
        // Address of a
        cout << ptr << "\n";

        // Value of a
        cout << *ptr << "\n";

        return 0;
}
```

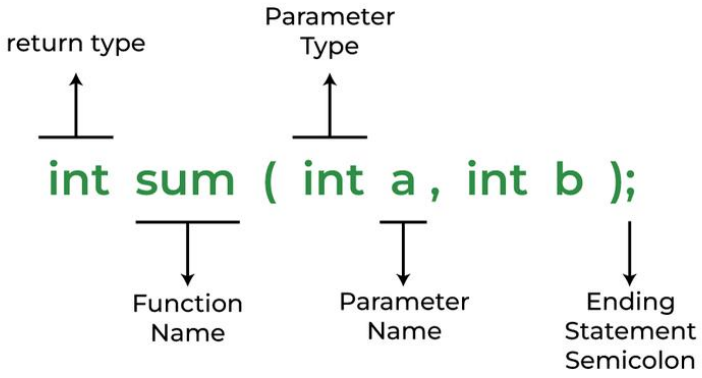
You cant change value and also address

Functions

Defining a Function

```
return_type function_name( parameter list )

{
 body of the function
 }
```



•**Return Type** − A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.

•**Function Name** − This is the actual name of the function. The function name and the parameter list together constitute the function signature.

•**Parameters** − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

•**Function Body** − The function body contains a collection of statements that define what the function does.

Example

Given below is the source code for a function called **max()**. This function takes two parameters num1 and num2 and returns the maximum value between the two −

```
/* function returning the max between two numbers */

int max(int num1, int num2)

{ /* local variable declaration */

 int result; if (num1 > num2)

result = num1;

else

result = num2;

return result;

 }
```

Conditions of Return Types and Arguments

In C programming language, functions can be called either with or without arguments and might return values. They may or might not return values to the calling functions.

1.Function with no arguments and no return value
2.Function with no arguments and with return value
3.Function with argument and with no return value
4.Function with arguments and with return value

We can pass arguments to the C function in two ways:
Pass by Value
Pass by Reference

1. Pass by Value

Parameter passing in this method copies values from actual parameters into formal function parameters. As a result, any changes made inside the functions do not reflect in the caller's parameters.

```
// C program to show use of call by value
#include <stdio.h>

void swap(int var1, int var2)
{
int temp = var1;
var1 = var2;
var2 = temp;
}

// Driver code
int main()
{
int var1 = 3, var2 = 2;
printf("Before swap Value of var1 and var2 is: %d, %d\n",var1, var2);
swap(var1, var2);
printf("After swap Value of var1 and var2 is: %d, %d",var1, var2);
return 0;
}
```

Output:

```
Before swap Value of var1 and var2 is: 3, 2
After swap Value of var1 and var2 is: 3, 2
```

2. Pass by Reference

The caller's actual parameters and the function's actual parameters refer to the same locations, so any changes made inside the function are reflected in the caller's actual parameters.

```
// C program to show use of call by Reference
#include <stdio.h>
void swap(int *var1, int *var2)
{
int temp = *var1;
*var1 = *var2;
*var2 = temp;
}

// Driver code
int main()
{
int var1 = 3, var2 = 2;
printf("Before swap Value of var1 and var2 is: %d, %d\n", var1, var2);
swap(&var1, &var2);
printf("After swap Value of var1 and var2 is: %d, %d", var1, var2);
return 0;
}
```

Output:

Before swap Value of var1 and var2 is: 3, 2

After swap Value of var1 and var2 is: 2, 3

Passing Array as parameter

```
// C++ Program to display marks of 5 students
#include <iostream>
using namespace std;
// declare function to display marks take a 1d array as parameter
void display(int m[5]) {
    cout << "Displaying marks: " << endl;

    // display array elements
    for (int i = 0; i < 5; ++i) {
        cout << "Student " << i + 1 << ": " << m[i] << endl;
    }
}
int main() {

    // declare and initialize an array
    int marks[5] = {88, 76, 90, 61, 69};

    // call display function
    // pass array as argument
    display(marks);

    return 0;
}
```

Output:

```
Displaying Values:
num[0][0]: 3
num[0][1]: 4
num[1][0]: 9
num[1][1]: 5
num[2][0]: 7
num[2][1]: 1
```

Function Overloading

Function overloading is a feature in C++ that allows you to define multiple functions with the same name but different parameter lists. The compiler determines which function to call based on the number, types, and order of the arguments passed to the function. Function overloading is a form of compile-time polymorphism.

```
#include <iostream>
// Function with different parameter types
void print(int num) {
    std::cout << "Integer: " << num << std::endl;
}
void print(double num) {
    std::cout << "Double: " << num << std::endl;
}
// Function with different number of parameters
void print(const char* str) {
    std::cout << "String: " << str << std::endl;
}
int main() {
    print(5);          // Calls the first print function
    print(3.14);       // Calls the second print function
    print("Hello");    // Calls the third print function
    return 0;
}
```

Function overloading rules:

1.Functions must have the same name.
2.Functions must have different parameter lists (number, types, or both).
3.Return type does not play a role in function overloading.
Function overloading can also involve default arguments:

```
#include <iostream>
void print(int num) {
    std::cout << "Integer: " << num << std::endl;
}
void print(double num, int precision = 2) {
    std::cout << "Double with precision " << precision << ": " << num <<
std::endl;
}
int main() {
    print(5);          // Calls the first print function
    print(3.14159);    // Calls the second print function with default precision
    print(3.14159, 4); // Calls the second print function with specified precision
    return 0;
}
```

Output:

```
Integer: 5
Double with precision 2: 3.14159
Double with precision 4: 3.14159
```

Inline Function: The primary purpose of an inline function in C++ is to provide a hint to the compiler to perform function inlining. Inlining is an optimization technique where the compiler replaces a function call with the actual code of the function at the call site. This can result in improved performance by reducing the overhead of function calls, especially for small and frequently called functions.

Inline Function

```
#include <iostream>

// Inline function to calculate the square of a number
inline int square(int num) {
    return num * num;
}

int main() {
    int number;

    std::cout << "Enter a number: ";
    std::cin >> number;

    // Calling the inline function
    int result = square(number);

    std::cout << "Square of " << number << " is: " << result <<
std::endl;

    return 0;
}
```

Name Spaces

In C++, a namespace is a collection of related names or identifiers (functions, class, variables) which helps to separate these identifiers from similar identifiers in other namespaces or the global namespace.
The identifiers of the C++ standard library are defined in a namespace called std.
In order to use any identifier belonging to the standard library, we need to specify that it belongs to the std namespace. One way to do this is by using the **scope resolution** operator **::** For example,

- std::cout << "Hello World!";

Here, we have used the code std:: before cout. This tells the C++ compiler that the cout object we are using belongs to the std namespace.

// C++ program to illustrate the use of namespace with same name of function and variable

```
#include <iostream>
using namespace std;
 // Namespace n1
namespace n1 {
int x = 2;
 // Function to display the message
// for namespace n1
void fun()
{`
    cout << "This is fun() of n1"
         << endl;
}
}
 // Namespace n2
namespace n2 {
int x = 5;
 // Function to display the message
// for namespace n2
void fun()
{
    cout << "This is fun() of n2"
         << endl;
} }
```

```
// Driver Code
int main()
{
    // The methods and variables called
    // using scope resolution(::)
    cout << n1::x << endl;

    // Function call
    n1::fun();
     cout << n2::x << endl;
     // Function call;
    n2::fun();
     return 0;
}
```

Output:

```
2
This is fun() of n1
5
This is fun() of n2
```

// C++ program to demonstrate the use of "using" directive

```
 #include <iostream>
using namespace std;
 // Namespace n1
namespace n1 {
int x = 2;
void fun()
{
    cout << "This is fun() of n1"
         << endl;
}
}
 // Namespace is included
using namespace n1;
 int main()
{
    cout << x << endl;
     // Function Call
    fun();
     return 0;
}
```

Output:

```
2

This is fun() of n1
```

C/C++ Preprocessors

Preprocessors are programs that process the source code before compilation. A number of steps are involved between writing a program and executing a program in C / C++.

Preprocessor Directives	Description
#define	Used to define a macro
#undef	Used to undefine a macro
#include	Used to include a file in the source code program
#ifdef	Used to include a section of code if a certain macro is defined by #define
#ifndef	Used to include a section of code if a certain macro is not defined by #define
#if	Check for the specified condition
#else	Alternate code that executes when #if fails
#endif	Used to mark the end of #if, #ifdef, and #ifndef

Types of C/C++ Preprocessors
There are 4 Main Types of Preprocessor Directives:

Macros
File Inclusion
Conditional Compilation
Other directives
Let us now learn about each of these directives in detail.

1. Macros
In C/C++, Macros are pieces of code in a program that is given some name. Whenever this name is encountered by the compiler, the compiler replaces the name with the actual piece of code. The '#define' directive is used to define a macro.
Syntax of Macro Definition

```
#define token value
// C++ Program to illustrate the macros
#include <iostream>

// macro definition
#define LIMIT 5

int main()
{
        for (int i = 0; i < LIMIT; i++) {
                std::cout << i << "\n";
        }

        return 0;
}
```

Macros with arguments

We can also pass arguments to macros. Macros defined with arguments work similarly to functions.

Example

```
#define foo(a, b) a + b
#define func(r) r * r
// C++ Program to illustrate the function like macro
#include <iostream>
// macro with parameter
#define AREA(l, b) (l * b)
int main()
{       int l1 = 10, l2 = 5, area;
        area = AREA(l1, l2);
        std::cout << "Area of rectangle is: " << area;
        return 0;
}
```

3. Conditional Compilation

Conditional Compilation in C/C++ directives is a type of directive that helps to compile a specific portion of the program or to skip the compilation of some specific part of the program based on some conditions. There are the following preprocessor directives that are used to insert conditional code:

#if Directive
#ifdef Directive
#ifndef Directive
#else Directive
#elif Directive
#endif Directive

#endif directive is used to close off the #if, #ifdef, and #ifndef opening directives which means the preprocessing of these directives is completed.

Syntax

```
#ifdef macro_name
    statement1;
    statement2;
    .
    .
    statementN;
#endif
```

If the macro with the name ‘macro_name‘ is defined, then the block of statements will execute normally, but if it is not defined, the compiler will simply skip this block of statements.

Other Directives
Apart from the above directives, there are two more directives that are not commonly used. These are:
#undef Directive
#pragma Directive

1. **#undef Directive**

The #undef directive is used to undefine an existing macro. This directive works as:

#undef LIMIT
Using this statement will undefine the existing macro LIMIT. After this statement, every "#ifdef LIMIT" statement will evaluate as false.

2. #pragma Directive
This directive is a special purpose directive and is used to turn on or off some features. These types of directives are compiler-specific, i.e., they vary from compiler to compiler. Some of the #pragma directives are discussed below:

#pragma startup: These directives help us to specify the functions that are needed to run before program startup (before the control passes to main()).
#pragma exit: These directives help us to specify the functions that are needed to run just before the program exit (just before the control returns from main()).