[Introduction, Variables and Operators]

HARSH SRIVASTAVA

INDEX

Table of Contents

Introduction to C++

To create an object file and a binary file in C++, you need to follow these steps:

main.cpp

```cpp
#include <iostream>

int main() {
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

Compile the source code to generate an object file:

```
g++ -c main.cpp -o main.o
```

Link the object file with C++ standard libraries to create the binary executable

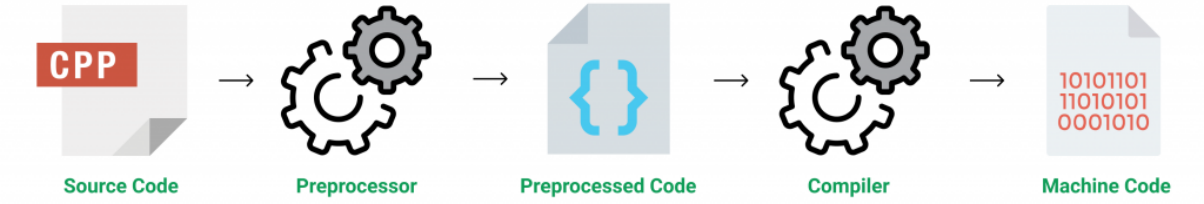
```
g++ main.o -o main
```

Run the binary executable

```
./main
```

Preprocessor Directives in C++

A preprocessor directive is a statement which gets processed by the C++ preprocessor before compilation.



For basic programming in C++, we only need to understand the **#include** and the **#define** directives.

#include is a preprocessor directive used to include external files or libraries into your code. It allows you to use code from other files in your current source code, which enables code reusability and modularity.

The syntax for #define is as follows:

```
#include <header_file>
```

Or

```
#include "header_file"
```

<header_file>: This form is used to include standard library header files.

"header_file": This form is used to include user-defined header files or files located in the same directory as the source file. The compiler will first search for the header file in the current directory and then in the standard system directories if it's not found in the current directory.

For example, if you want to use functionalities from the iostream library, you can include it in your C++ code using #include <iostream> like this:

```
#include <iostream>

int main() {
    // Your code using iostream functions and classes goes here
    return 0;
}
```

#define is a preprocessor directive used to define constants or macros. It allows you to give a name to a specific value or expression, making it easier to use and maintain throughout your code.

The syntax for #define is as follows:

```
#define identifier value
```

When the preprocessor encounters a #define directive, it replaces all occurrences of identifier with the corresponding value in the source code before the actual compilation takes place. This process is known as macro substitution.

Here's an example of using #define to define a constant:

```
#define PI 3.14159

int main() {
    double radius = 5.0;
    double area = PI * radius * radius;

    // Rest of your code
    return 0;
}
```

In this example, #define PI 3.14159 defines a constant named "PI" with the value 3.14159. Whenever the preprocessor encounters the identifier "PI" in the code, it will replace it with the value 3.14159.

Basic Input/Output in C++

Standard output stream (cout):

standard input stream (cin):

```cpp
#include <iostream>

int main() {
    int num;
    std::cout << "Enter a number: ";
    std::cin >> num;

    std::cout << "You entered: " << num << std::endl;

    return 0;
}
```

Un-buffered standard error stream (cerr):

```cpp
#include <iostream>

int main() {
    int num = 10;

    if (num <= 0) {
        // Output an error message using std::cerr
        std::cerr << "Error: The number must be greater than zero." << std::endl;
        return 1; // Indicate an error by returning a non-zero value
    }

    // Rest of the code
    // ...

    return 0; // Indicate successful execution by returning zero
}
```

buffered standard error stream (clog):

```cpp
#include <iostream>

int main() {
    int num = 42;

    // Log a message using clog
    std::clog << "The program is running..." << std::endl;

    // Log some debugging information
    std::clog << "Value of num: " << num << std::endl;

    // Rest of the code
    // ...

    return 0;
}
```

C/C++ Preprocessors

In C and C++, the preprocessor is a separate phase of the compilation process that occurs before the actual compilation of the source code. It is responsible for performing various text manipulations on the source code before it is passed to the compiler. Preprocessor directives start with the # symbol and are processed before the code is compiled.

Here are some common preprocessor directives and their functionalities:

1. **#include:** Used to include header files into the source code. It allows you to use functions and features from other files in your program.
2. **#define:** Used to define constants or macros. It allows you to give a name to a specific value or expression, making it easier to use and maintain throughout your code.
3. **#ifdef, #ifndef, #else, #endif:** Used for conditional compilation. These directives allow you to include or exclude blocks of code based on certain conditions.
4. **#if, #elif, #else, #endif:** Used for more complex conditional compilation. These directives allow you to evaluate expressions to include or exclude blocks of code.
5. **#pragma:** Used to provide compiler-specific instructions or control certain compilation behaviors. Pragmas are implementation-specific and may vary between different compilers.
6. **#error:** Used to generate a compilation error with a custom error message. It is often used to enforce certain conditions in the code.
7. **#warning:** Used to generate a compilation warning with a custom warning message. It is helpful for flagging potential issues in the code.

For example, here's a simple C++ code that uses some preprocessor directives:

```cpp
#include <iostream>

#define PI 3.14159

#ifdef DEBUG_MODE
    #define LOG(msg) std::cout << "[DEBUG] " << msg << std::endl
#else
    #define LOG(msg)
#endif

int main() {
    LOG("Starting the program...");

    int radius = 5;
    double area = PI * radius * radius;

    LOG("Area calculated.");

    std::cout << "The area of the circle is: " << area << std::endl;

    return 0;
}
```

O/P

```
The area of the circle is: 78.5397
```

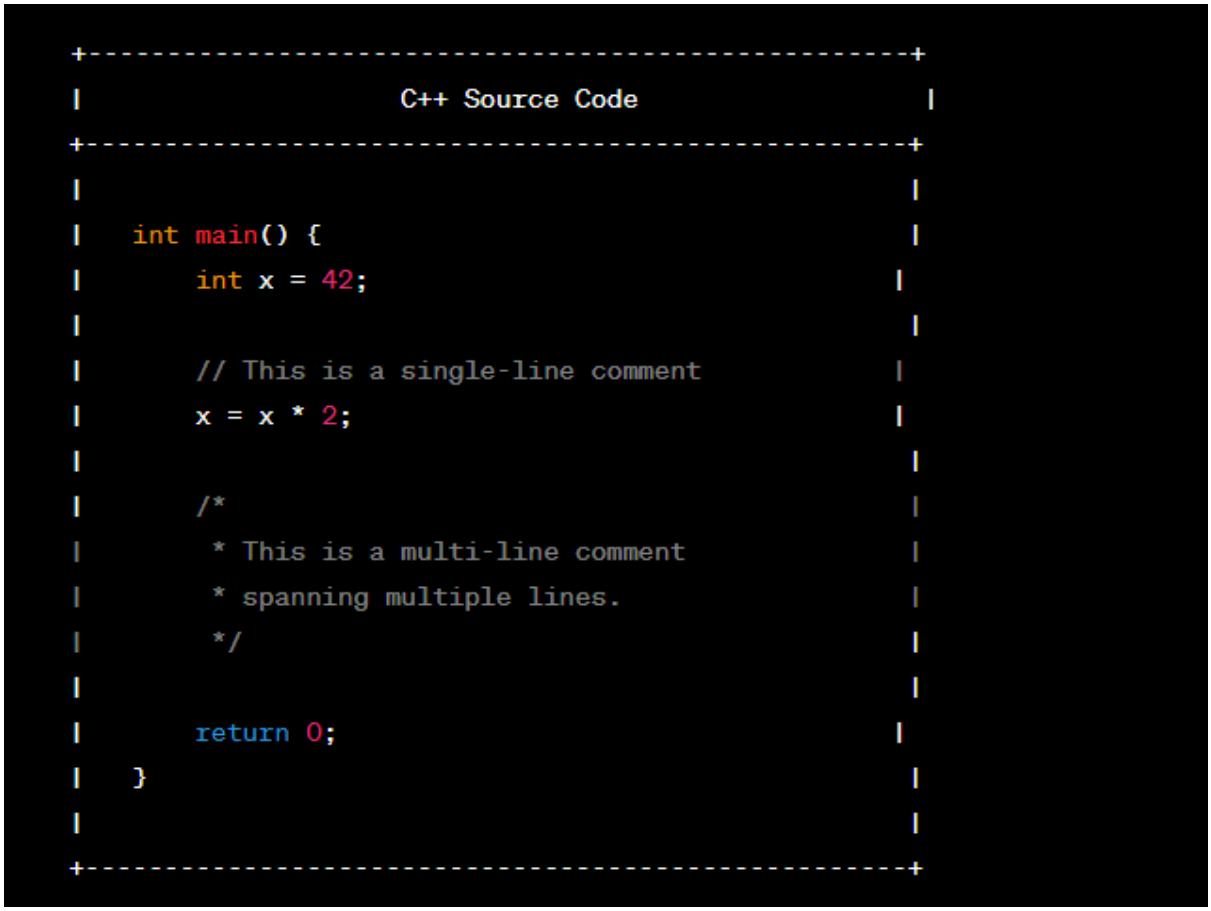
O/P: if we define DEBUG_MODE

```
[DEBUG] Starting the program...
[DEBUG] Area calculated.
The area of the circle is: 78.5397
```


Object File (.obj) 1
Object File (.obj) 2
Object File (.obj) N
Library Files
Linker
Executable File (.exe)

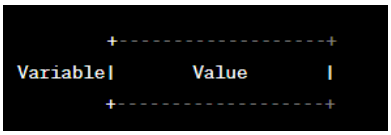
Comments in C++

comments are text annotations within the source code that provide explanatory or informative notes to programmers. Comments are ignored by the compiler and do not affect the program's functionality.



Data Types and Variables in C++

Variable: A variable is a named storage location in a computer's memory that holds a value.



Data Type: A data type in programming defines the type of data that a variable can hold. It specifies the size and format of the data, which determines how the data is stored in memory and how it can be manipulated.

Fundamental Data Types	Size	Range
int	4 bytes	-2,147,483,648 to 2,147,483,647
float	4 bytes	3.4e-38 to 3.4e+38
double	8 bytes	1.7e-308 to 1.7e+308
char	1 byte	-128 to 127
bool	1 byte	true or false

Modifier Data Types	Size	Range
signed int	4 bytes	-2,147,483,648
unsigned int	4 bytes	0 to 4,294,967,295
signed char	1 byte	-128 to 127
unsigned char	1 byte	0 to 255
signed short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
signed long	8 bytes	-9,223,372,036,854,775,808
unsigned long	8 bytes	0 to 18,446,744,073,709,551,615

Extended Data Types	Size	Range
long int	8 bytes	-9,223,372,036,854,775,808
long double	12 or 16 bytes	1.1e-4932 to 1.1e+4932

Miscellaneous Data Types	Size	Range
void	undefined	undefined
wchar_t	2 or 4 bytes	0 to 65,535

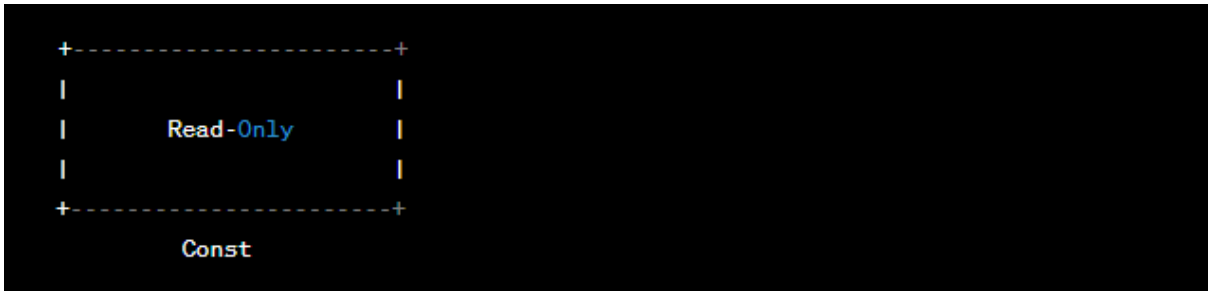
Qualifiers, Modifiers & Specifiers

Qualifiers:

Const: The const keyword in C++ is used to declare constants, which are values that **cannot be modified after initialization**, attempting to modify it will result in a **compilation error**.

There are a certain set of rules for the declaration and initialization of the constant variables:

- The const variable cannot be left un-initialized at the time of the declaration.
- It cannot be assigned value anywhere in the program.



There are two primary ways the const keyword can be used:

1. **Constant Variables**: In this example, x is a constant variable of type int, and its value is initialized to 10. Once initialized, you cannot change the value of x throughout the program.

```
const int x = 10;
```

2. **Constant Pointers**: In this example, **ptr is a pointer to a constant integer**. It can point to an integer (y in this case), but **it cannot modify the value it points to**. The **pointer ptr can be changed** to point to another constant integer, but it cannot modify the integer it currently points to.

```
int y = 20;
const int* ptr = &y;
```

   Here, **ptr2 is a constant pointer to an integer**. It can point to an integer (z in this case), but it **cannot be changed to point to another integer**. However, it can modify the value of the integer it points to (z).

```
int z = 30;
int* const ptr2 = &z;
```

   Here, **ptr is a constant pointer to a constant integer**. We have a constant integer variable x with the value 10. Once initialized, the value of x cannot be modified throughout the program and the pointer itself is constant and cannot be changed to point to another memory address.

```
const int x = 10;
const int* const ptr = &x;
```

3. **Const method:**

Volatile: Volatile variables can be read or written by external factors such as hardware, other threads, or interrupts.

Consider a scenario where we have a simple program that involves an interrupt service routine (ISR) that updates a variable representing the number of times the interrupt has occurred. The main program then reads and displays the value of this variable.

```cpp
#include <iostream>
#include <chrono>
#include <thread>
#include <csignal>

// Volatile variable to track the number of interrupts
volatile int interruptCount = 0;

// Simulate an interrupt service routine (ISR)
void interruptHandler(int signum) {
    ++interruptCount;
}

int main() {
    // Register the interrupt handler with SIGINT (Ctrl+C)
    std::signal(SIGINT, interruptHandler);

    std::cout << "Press Ctrl+C to simulate interrupts...\n";

    while (true) {
        // Display the current interrupt count
        std::cout << "Interrupt count: " << interruptCount << "\n";
        std::this_thread::sleep_for(std::chrono::seconds(1));
    }

    return 0;
}
```

A thread-based example that demonstrates the significance of using the volatile keyword when working with shared variables in a multi-threaded environment. Here we update a shared flag variable in a separate thread, and the main thread waits for the flag to be updated:

```cpp
#include <iostream>
#include <thread>
#include <chrono>

// Global volatile flag variable
volatile bool flag = false;

// Function to update the flag in a separate thread
void updateFlag() {
    std::this_thread::sleep_for(std::chrono::seconds(2));
    flag = true;
}

int main() {
    std::cout << "Starting a separate thread to update the flag...\n";

    // Start a separate thread to update the flag
    std::thread t(updateFlag);

    // Wait for the flag to be updated
    while (!flag) {
        // Do nothing, just wait
    }

    std::cout << "Flag has been updated!\n";

    // Wait for the thread to finish
    t.join();

    return 0;
}
```

The **volatile** keyword is important here to ensure that the main thread correctly observes the updated value of **flag** set by the separate thread. Without **volatile**, the compiler might optimize the read operation inside the loop and keep using a cached value of **flag**, leading to an infinite loop.

By using volatile, we ensure that the main thread reads the latest value of flag from memory, reflecting the updates made by the separate thread.

Modifiers: Used to alter the characteristics of data types or variables, typically by changing their size, range, or storage behavior.


Character Data Types
char
(Typically 8 bits, signed or unsigned)
signed char
(Typically 8 bits, signed)
unsigned char
(Typically 8 bits, unsigned)


Wide Character Data Types
wchar_t
(Platform-dependent, typically 16 bits or more)
signed wchar_t
(Platform-dependent, typically 16 bits or more, signed)
unsigned wchar_t
(Platform-dependent, typically 16 bits or more, unsigned)

Declaration, Definition & Initialization:

1. **Declaration:** A declaration **introduces the existence of a variable** to the compiler **without allocating memory** for it or assigning an initial value. It informs the compiler about the variable's name and data type.
2. **Definition:** A definition not only **introduces the existence of a variable** but also **allocates memory to store its value**. It reserves a memory space for the variable to hold data of the specified type.

   In the case of **int x;** the line of code both declares and defines the variable x

   - It is a declaration because it tells the compiler that there is a variable named x of type int.
   - It is also a definition because it allocates memory for the variable x to store an integer value.
3. **Initialization:** Initialization sets the initial value of the variable after it has been defined and allocated memory. It assigns a specific value to the variable, so it's no longer in an undefined state.

```
// Declaration: Tells the compiler about the existence of the variable
extern int x;

// Definition: Allocates memory for the variable 'x'
int x;

// Initialization: Assigns a specific value to the variable 'x'
x = 42;
```

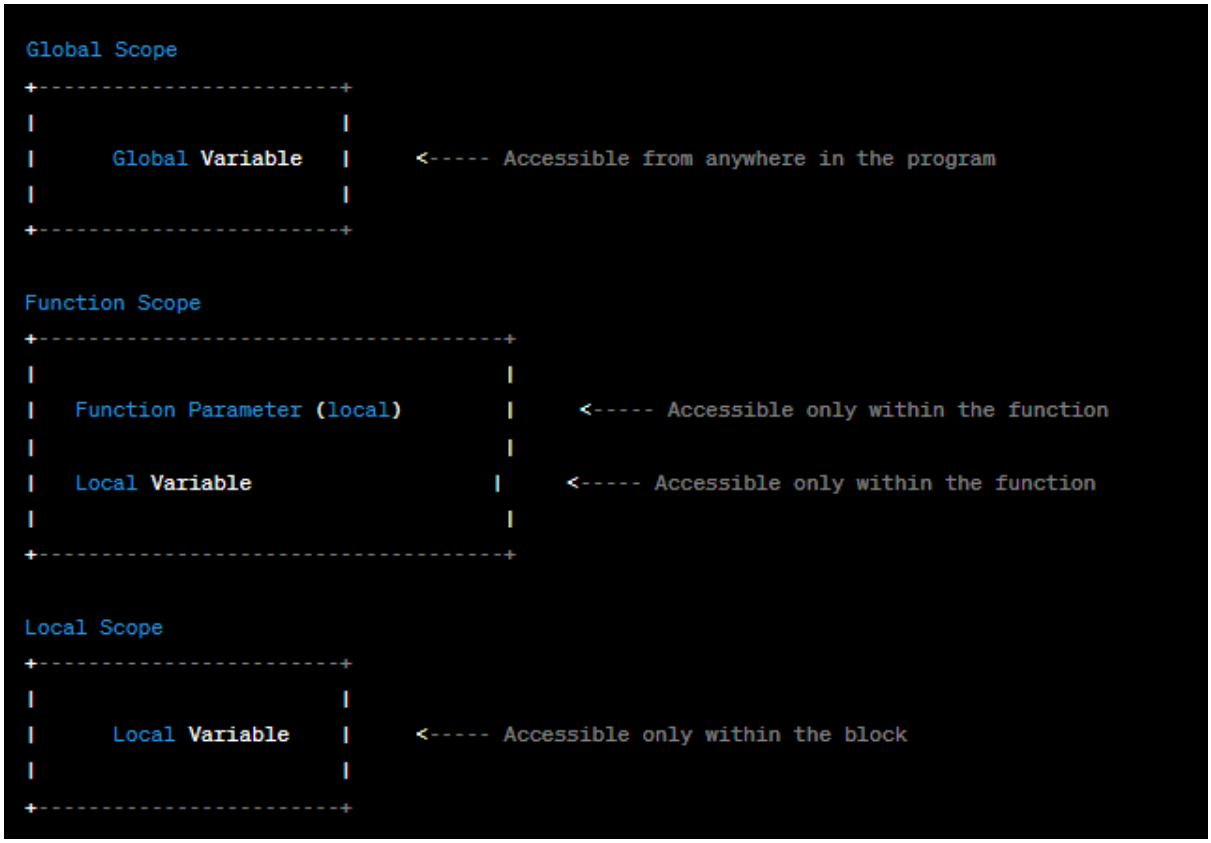
Here's a breakdown of the line **extern int x;**

extern: The extern keyword informs the compiler that the variable is **declared elsewhere** and should not allocate memory for it in the current file.

int: The data type of the variable is int, indicating that x is an integer variable.

x: x is the name of the variable being declared.

Scope of Variables:



sizeof Operator in C++

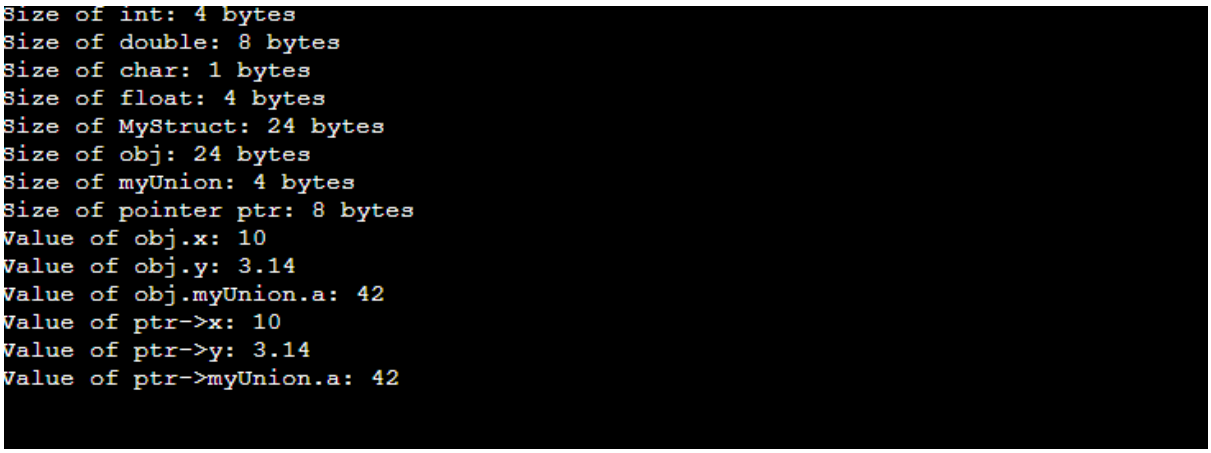
sizeof is an operator in C++ that allows you to determine the size in bytes, of a data type or an object. Internally, **sizeof** is evaluated during the **compilation phase** rather than at runtime. Compiler already knows their sizes based on the target architecture and compiler settings. So, when the compiler encounters the sizeof operator, it performs the following actions:

1. For **built-in data types** like int and double, the compiler knows their sizes. It **directly replaces** sizeof(int) with the size of an int data type.
2. For **user-defined types** like MyStruct, the compiler calculates the size by summing up the sizes of its members. In this case, MyStruct has an int, a double and an union. The compiler calculates the size of MyStruct based on the sizes of these members (considering alignment requirements and padding) and replaces sizeof(MyStruct) with the result.

   sizeof **obj.myUnion** will be 4 bytes, which is the size of the union (int a) since it is the largest member.

```cpp
#include <iostream>

struct MyStruct {
    int x;
    double y;
    union {
        int a;
        char b;
        float c;
    } myUnion;
};

int main() {
    MyStruct obj;
    MyStruct* ptr = &obj;

    obj.x = 10;
    obj.y = 3.14;
    obj.myUnion.a = 42;

    std::cout << "Size of int: " << sizeof(int) << " bytes\n";
    std::cout << "Size of double: " << sizeof(double) << " bytes\n";
    std::cout << "Size of char: " << sizeof(char) << " bytes\n";
    std::cout << "Size of float: " << sizeof(float) << " bytes\n";
    std::cout << "Size of MyStruct: " << sizeof(MyStruct) << " bytes\n";
    std::cout << "Size of obj: " << sizeof obj << " bytes\n";
    std::cout << "Size of myUnion: " << sizeof obj.myUnion << " bytes\n";
    std::cout << "Size of pointer ptr: " << sizeof ptr << " bytes\n";

    std::cout << "Value of obj.x: " << obj.x << std::endl;
    std::cout << "Value of obj.y: " << obj.y << std::endl;
    std::cout << "Value of obj.myUnion.a: " << obj.myUnion.a << std::endl;

    std::cout << "Value of ptr->x: " << ptr->x << std::endl;
    std::cout << "Value of ptr->y: " << ptr->y << std::endl;
    std::cout << "Value of ptr->myUnion.a: " << ptr->myUnion.a << std::endl;

    return 0;
}
```

O/P:


Size of int: 4 bytes
Size of double: 8 bytes
Size of char: 1 bytes
Size of float: 4 bytes
Size of MyStruct: 24 bytes
Size of obj: 24 bytes
Size of myUnion: 4 bytes
Size of pointer ptr: 8 bytes
Value of obj.x: 10
Value of obj.y: 3.14
Value of obj.myUnion.a: 42
Value of ptr->x: 10
Value of ptr->y: 3.14
Value of ptr->myUnion.a: 42

Literals in C++

Literals in C++ are fixed values that are directly written in the source code.

1. Numeric Literals: Represent numeric values.
   - Integer Literals: Whole numbers without a fractional part.
   - Decimal: Base 10 (e.g., 123).
   - Octal: Base 8 (e.g., 075).
   - Hexadecimal: Base 16 (e.g., 0x1A).
2. Floating-Point Literals: Numbers with a fractional part (e.g., 3.14).
3. Character Literals: Represent single characters enclosed in single quotes (e.g., 'A').
4. String Literals: Represent sequences of characters enclosed in double quotes (e.g., "Hello").
5. Boolean Literals: Boolean literals are written directly as true or false without the need for any special syntax or keywords.

```cpp
#include <iostream>
#include <string>

int main() {
    int x = 10;                          // Integer literal, value is 10
    double pi = 3.14159;                 // Floating-point literal, value is 3.14159
    char ch = 'A';                       // Character literal, value is 'A'
    const char* message = "Hello, World!"; // String literal, value is "Hello, World!"
    std::string str = "C++ Programming"; // String variable, value is "C++ Programming"
    bool flag = true;                    // Boolean literal, value is true

    std::cout << "x: " << x << std::endl;                  // Output integer literal
    std::cout << "pi: " << pi << std::endl;                // Output floating-point literal
    std::cout << "ch: " << ch << std::endl;                // Output character literal
    std::cout << "message: " << message << std::endl;      // Output string literal
    std::cout << "str: " << str << std::endl;              // Output the string variable
    std::cout << "flag: " << std::boolalpha << flag << std::endl; // Output boolean literal

    return 0;
}
```

O/P

```
x: 10
pi: 3.14159
ch: A
message: Hello, World!
str: C++ Programming
flag: true
```

Type Casting in C++