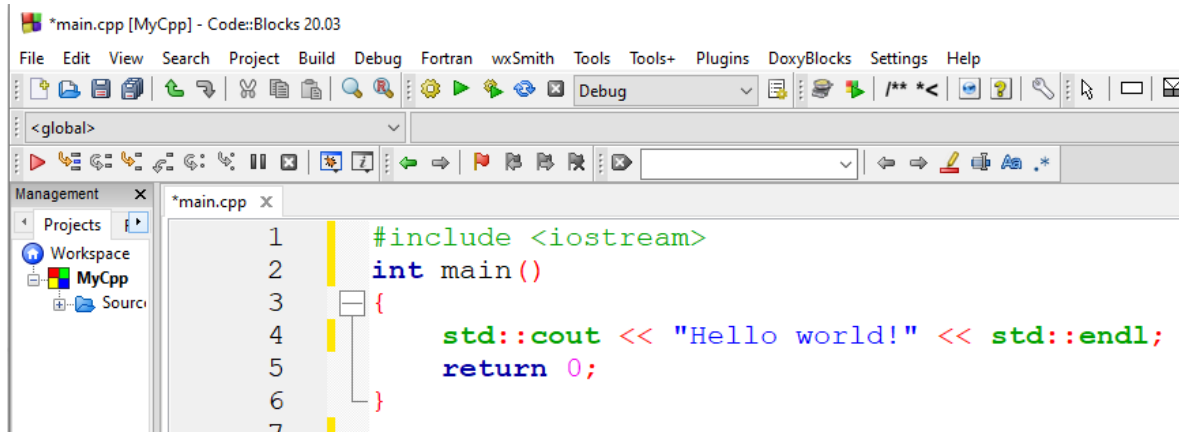


## C++ Welcome Program:

A "welcome program" in C++ is a simple program that prints a welcoming message to the user when executed. It serves as a basic example to demonstrate how to write, compile, and run a C++ program. Here's a simple welcome program in C++:



```
*main.cpp [MyCpp] - Code::Blocks 20.03
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
<global>
1 #include <iostream>
2 int main()
3 {
4     std::cout << "Hello world!" << std::endl;
5     return 0;
6 }
7
```

Let's break down the components of this program:

- **#include<iostream>:** This line includes the header file `iostream`, which stands for "input/output stream." This header provides functionality for input and output operations in C++.
- **int main() { ... }:** This is the main function of the program, which serves as the entry point for execution. It returns an integer (`int`) value to indicate the status of the program to the operating system. In this case, `0` is returned to indicate successful execution.
- **std::cout << "Hello World!" << std::endl;** This line prints the welcoming message to the console. `std::cout` is the standard output stream, and `<<` is the stream insertion operator used to output data. "Welcome to the C++ Programming World!" is the message to be printed.
- **std::endl :** It is used to insert a newline character and flush the output stream.
- **std:::** is the namespace qualifier that indicates that `cout` is part of the `std` namespace. `std` is a namespace that encapsulates many standard C++ library components. In C++, the Standard Template Library (STL) provides a set of classes and functions for various common tasks, such as input/output operations.
- **return 0;** This statement exits the main functions and returns `0` to the operating system, indicating that the program executed successfully.

Instead of using `std::` every time you can refer it by "using namespace" as:

If you want to avoid explicitly specifying the namespace every time you use a member from that namespace, you can use a `using` directive. This tells the compiler to consider all the members of the specified namespace as if they were declared in the global namespace.

```

1  /*
2      Author: Archer InfoTech, Pune
3      Contact: 9850678451
4  */
5  #include <iostream>
6  using namespace std;
7  int main() // This is starting point of program
8  {
9      cout << "Hello world!" << endl;
10     return 0;
11 }

```

## C++ Comments:

In C++, comments are non-executable lines of text intended to provide clarity and documentation for your code. While the compiler ignores them during execution, they play a crucial role in making your code more readable, maintainable, and understandable for both yourself and others.

Here's a detailed breakdown of comments in C++:

### Types of Comments:

- **Single-line comments:** Begin with `//` and extend to the end of the line. Used for brief explanations or reminders within a code block.
- **Multi-line comments:** Enclosed within `/*` and `*/`. Span multiple lines for longer descriptions, explanations, or documentation.

```

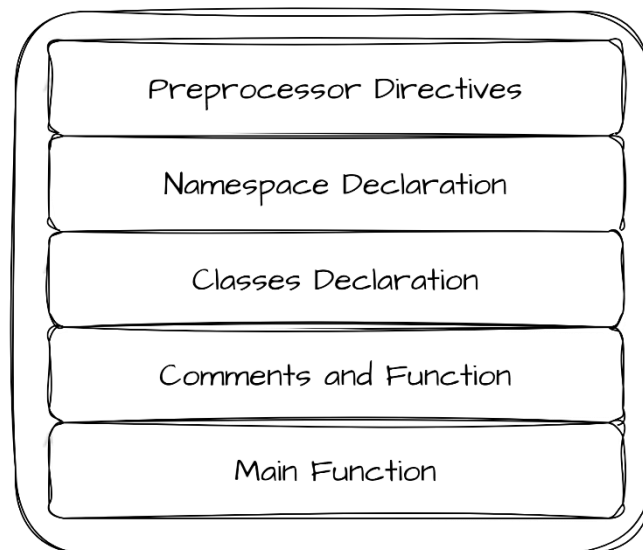
1  /**
2      Author: Archer InfoTech, Pune
3      Contact: 9850678451
4      website: https://archerinfotech.in/
5  */
6  #include <iostream>
7  using namespace std;
8  int main() /// This is starting point of program
9  {
10     cout << "Hello world!" << endl; // displays data on output screen
11     return 0;
12 }
13 /**
14     comments are non-executable lines of text intended to provide
15     clarity and documentation for your code.
16 */
17

```

In Code::Blocks you can use `///` for the single line comment and for multiline comment you are allowed to use `/** .....*/`, which will make the comment section dark.

**Benefits of Using Comments:**

- **Explain complex logic:** Break down intricate algorithms or code sections into simpler language for better understanding.
- **Document intent:** Describe the purpose of functions, variables, or code blocks to clarify their role in the program.
- **Improve readability:** Enhance code clarity by separating logic and explanations, aiding navigation and understanding.
- **Facilitate collaboration:** Make code more accessible for other developers working on the same project.
- **Remind yourself:** Use comments to remember specific details or reasons behind certain code choices, especially when revisiting older projects.

**Structure of C++ Program**

**Preprocessor Directives:** These are optional lines starting with # and tell the compiler to perform specific actions before compilation. Examples include including header files and defining macros.

**Namespace Declaration:** This is also optional and allows you to use elements from a specific namespace (like std for standard library elements) without the prefix.

**Classes:** These are blueprints for creating objects, which encapsulate data (members) and behaviours (methods). They are optional but enable object-oriented programming.

**Comments and Functions:** These are lines starting with // or /\* ... \*/ that are ignored by the compiler but provide information for humans reading the code. They are highly recommended to improve code readability and maintainability. The functions are reusable blocks of code that perform specific tasks. They are optional but very useful for organizing your code.

**Main Function:** This is the required entry point of your program. It's defined as `int main()` and contains the code that will be executed.

## C++ Tokens:

In C++, tokens are the fundamental building blocks of your code. They are the smallest meaningful units that the compiler recognizes and processes and include identifiers, keywords, literals, operators, punctuators, and preprocessors directives. Understanding different token types and their roles is crucial for writing correct and efficient C++ programs. C++ has following tokens:

- Constants and literals
- Identifiers
- Comments
- Keywords
- Punctuators
- Preprocessor Directives
- Operators

**Constants and Literals:** literals represent fixed values directly specified in the code, while constants provide symbolic names to fixed value defined using `const` keyword or `#define` directive. Literals are immutable and cannot be modified during program execution, while constants can have their values changed at compile time or runtime, depending on how they are declared. Unchangeable values represented in different forms:

- **Integer literals:** Whole numbers (e.g., 10, -5).
- **Floating-point literals:** Numbers with decimals (e.g., 3.14, -2.5e6).
- **Character literals:** Single characters enclosed in single quotes (e.g., 'a', '\$').
- **String literals:** Sequence of characters enclosed in double quotes (e.g., "Hello, world!").
- **Boolean literals:** true or false representing truth values.

**Identifiers:** Identifiers are names given to various program elements such as variables, functions, classes, etc.

- Rules for identifiers:
  - Must begin with a letter (uppercase or lowercase) or an underscore `_`.
  - It will treat uppercase or lowercase letters different or distinct.
  - Can contain letters, digits, and underscores.
  - Cannot be a keyword.
  - Must not contain spaces or special characters.
- Case-sensitive (e.g., age is different from Age).
- Choose meaningful names to enhance code readability and understandability.

**Comments:** These are Lines of text ignored by the compiler but intended for human readers. Which will enhance code readability and explain logic or intent. There are two types of comments in C++

- Single-line comments: Start with `//`, extend to the end of the line.
- Multi-line comments: Start with `/*`, end with `*/`.

**Keywords:** Keywords are reserved words with predefined meanings that cannot be used as variable or function names. Note that given list includes keywords introduced in C++20. Earlier versions may have a slightly different set of keywords and Keywords are case-

	Category	Keyword	Description
Archer	Basic types	auto, bool, char, char8_t, char16_t, char32_t, double, float, int, long, short, signed, signed, signed, unsigned, void, wchar_t	Define data types used to store different kinds of values.
Archer			
Archer	Storage specifiers	alignas, alignof, atomic_init, atomic_noexcept, co_await, co_await, co_return, concept, constexpr, decltype, dynamic_extent, explicit, extern, inline, mutable, noexcept, register, restrict, requires, static, thread_local, typedef, typename, union	Control storage duration, visibility, mutability, and other properties of variables and functions.
Archer			
Archer	Control flow	break, case, catch, continue, default, do, else, else if, for, for each, goto, if, return, switch, while	Control the flow of execution within your program.
Archer			
Archer	Function and class members	friend, operator, private, protected, public, template, this, using	Define and manage relationships between functions, classes, and their members.
Archer			
Archer	Error handling	noexcept, throw, try	Handle exceptions for robust error management.
Archer			
Archer	Memory management	delete, delete[], new, new[], placement new, placement new[]	Manage memory allocation and deallocation manually.
Arch			
Arch	Preprocessor directives	#define, #elif, #else, #endif, #error, #ifdef, #ifndef, #if, #include, #line, #pragma, #undef	Provide instructions for the compiler before compilation.
Arch			
Arch	Language extensions	alignas, alignof, atomic_init, atomic_noexcept, co_await, co_return, concept, constexpr, decltype, dynamic_extent, explicit, inline, mutable, noexcept, register, restrict, requires, static_assert, thread_local, typeid, typename	Introduce newer features and language enhancements.
C++ Keywords(C++20)			

sensitive in C++.

### Here's a list of C++ keywords:

**asm:** Specifies inline assembly code.

**auto:** Declares automatic variables.

**bool:** Represents Boolean data type.

**break:** Exits a loop or switch statement.

**case:** Defines a case in a switch statement.

**catch:** Catches exceptions in exception handling.

**char:** Declares character data type.

**class:** Declares a class.

**const:** Declares constants or read-only variables.

**const\_cast:** Performs type casting to const.

**continue:** Skips the current iteration of a loop.

**default:** Specifies default case in a switch statement.

**delete:** Deallocates memory allocated dynamically.

**do:** Starts a do-while loop.

**double:** Declares double-precision floating-point data type.

**dynamic\_cast:** Performs type casting dynamically.

**else:** Specifies alternative condition in an if statement.

**enum:** Declares an enumeration.

**explicit:** Specifies that a constructor or conversion function should not be used for implicit conversions.

**export:** Declares a function or class for use in another module.

**extern:** Specifies external linkage for variables and functions.

**false:** Represents the Boolean value false.

**float:** Declares single-precision floating-point data type.

**for:** Starts a for loop.

**friend:** Specifies a function or class as a friend.

**goto:** Transfers control to a labelled statement.

**if:** Starts an if statement.

**inline:** Specifies inline function or variable.

**int:** Declares integer data type.

**long:** Declares long integer data type.

**mutable:** Specifies that a member of a class can be modified even if the object is const.

**namespace:** Declares a namespace.

**new:** Allocates memory dynamically.

**nullptr:** Represents a null pointer.

**operator:** Specifies an operator function.

**private:** Specifies private access specifier in a class.

**protected:** Specifies protected access specifier in a class.

**public:** Specifies public access specifier in a class.

**register:** Specifies that a variable should be stored in a CPU register.

**reinterpret\_cast:** Performs type casting by reinterpreting the bit pattern of the object.

**return:** Returns a value from a function.

**short:** Declares short integer data type.

**signed:** Specifies signed data type.

**sizeof:** Returns the size of an object or data type.

**static:** Specifies static storage duration for variables and functions.

**static\_cast:** Performs type casting statically.

**struct:** Declares a structure.

**switch:** Starts a switch statement.

**template:** Specifies a template.

**this:** Pointer to the current object.

**throw:** Throws an exception.

**true:** Represents the Boolean value true.

**try:** Starts a block of code for exception handling.

**typedef:** Defines a new data type with an alias.

**typeid:** Returns type information of an expression.

**typename:** Specifies that a name represents a type.

**union:** Declares a union.

**unsigned:** Specifies unsigned data type.

**using:** Specifies using declaration or directive.

**virtual:** Specifies that a function is virtual.

**void:** Specifies no return value or empty argument list.

**volatile:** Specifies that a variable can be modified by something outside the program.



**wchar\_t:** Declares wide character data type.

**while:** Starts a while loop.

**Punctuators:** These are special symbols separating tokens and denoting structure. It Include:

- **Parentheses:** (), used for function calls and expressions.
- **Brackets:** [], used for array indexing and member access.
- **Braces:** {}, used for code blocks, function bodies, etc.
- **Semicolons:** ;, used to terminate statements.
- **Commas:** ,, used to separate elements in lists or function arguments.
- **Other symbols:** #, &, \*, ~, etc., depending on context.

**Preprocessor Directives:** Preprocessor directives provide instructions to the preprocessor, which processes the source code before compilation. They begin with a # symbol.

Examples: #include, #define, #ifdef, #endif

**Operators:** Operators are the symbols performing operations on operands like variables, constants. In C++20, there are no new operators introduced specifically as part of the language update. However, C++20 does bring improvements and changes to existing features, including some enhancements related to operators. Let's briefly review the common operators in C++, which remain fundamental in C++20:

- **Assignment Operators:** Assigns the constant, value of variable or answer of expression at its right to variable at left
  - Assignment =
  - Addition assignment +=
  - Subtraction assignment -=
  - Multiplication assignment \*=
  - Division assignment /=
  - Modulus assignment %=
  - Bitwise AND assignment &=
  - Bitwise OR assignment |=
  - Bitwise XOR assignment ^=
  - Left shift assignment <<=
  - Right shift assignment >>=
- **Unary Operator:** These operators are operated on only one operand.
  - Increment ++
  - Decrement --
  - (type)
  - sizeof()
- **Arithmetic Operators:** Perform mathematical operations such as addition, subtraction, multiplication, division, and modulus.
  - Addition +
  - Subtraction -
  - Multiplication \*
  - Division /
  - Modulus %
- **Relational Operators:** Compare values and determine the relationship between them, returning true or false.
  - Equal to ==
  - Not equal to !=
  - Less than <
  - Greater than >
  - Less than or equal to <=
  - Greater than or equal to >=
- **Logical Operators:** Perform logical operations on boolean values or expressions, such as AND, OR, and NOT. C++20 allows you to use the not, and, and or keywords as alternatives to !, &&, and || respectively
  - Logical AND &&
  - Logical OR ||
  - Logical NOT !

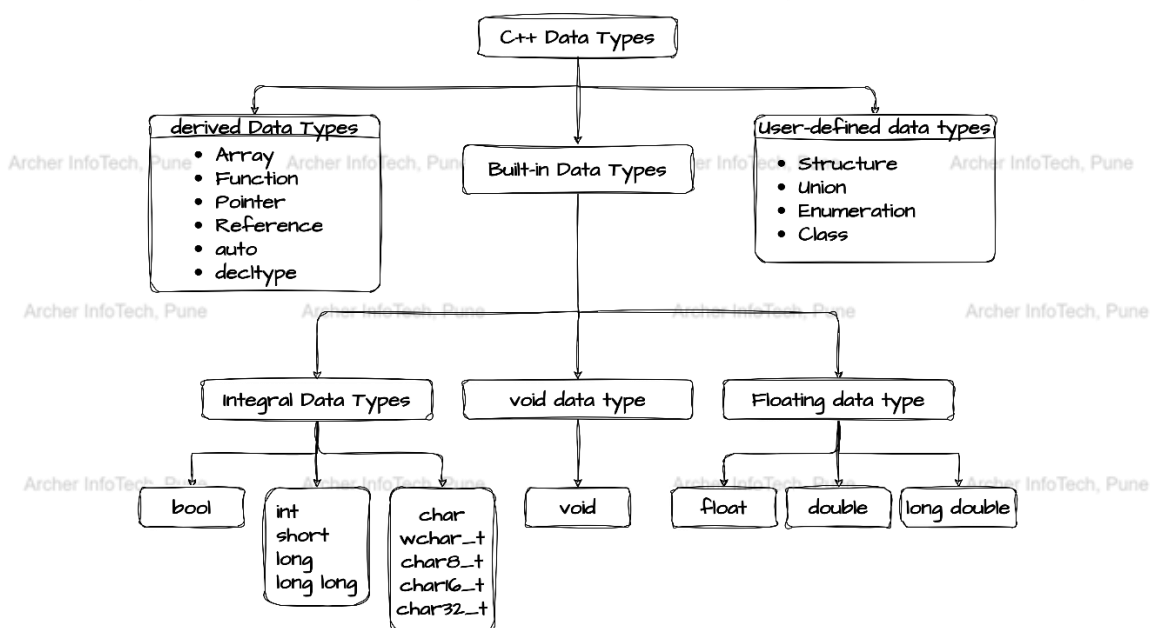
- **Bitwise Operators:** Perform operations on individual bits of integer operands, such as AND, OR, XOR, and shift operations.
  - Bitwise AND &
  - Bitwise OR |
  - Bitwise XOR ^
  - Bitwise NOT ~
  - Left shift <<
  - Right shift >>
- **Conditional Operator:** Evaluate a condition and return one of two values based on the result.
  - Conditional (ternary) operator ? :
- **Member Access Operators:** Access members of objects or pointers to objects.
  - Dot . (for accessing members of objects)
  - Arrow -> (for accessing members through pointers)
- **Other Operators:**
  - Comma , (used to separate expressions)
  - Address-of & (returns the address of a variable)
  - Dereference \* (accesses the value pointed to by a pointer)

We have seen all these operators in C programming, but initially in C++ some operators are introduced which are as listed below:

- :: - Scope resolution operator
- ::\* - Pointer-to-member declarator
- ->\* - Pointer to member operator
- .\* - Pointer to member operator
- delete - Memory release operator
- new - Memory allocation operator
- >> - Get-in / Extraction operator
- << - Put to / Insertion operator

## C++ Data Types:

The Data Type is actually the category of data which is input to computer and represented with the help of keyword in the program. The data type plays a crucial role in case of variable declaration, collecting an argument in the function and returning the value from the function. Actually, it is a tool which deals with the memory.





In C++, data types represent different kinds of values that variables can hold. These data types specify the size and format of the data. Here's a brief overview of the data types in C++20:

### ✚ Fundamental / Built-in Data Types:

- **Integer Types:** Represent whole numbers. [int, short, long (C++98), long long (C++11)]
  - **int, short, long, long long:** Signed integer types with varying sizes.
  - **unsigned int, unsigned short, unsigned long, unsigned long long:** Unsigned integer types.

**Type Aliases (C++11):** In C++, the `<cstdint>` header provides type aliases for integer types with specific characteristics regarding size and signedness. These type aliases are defined in the `std::` namespace. Type alias is a mechanism that allows you to create an alternative name for an existing type. Here are the commonly used type aliases provided by `<cstdint>` related to integer data types:

**Fixed-width Integer Types:** These types guarantee a specific size in bits regardless of the platform. Examples are:

**int8\_t:** Signed 8-bit integer.

**uint8\_t:** Unsigned 8-bit integer.

**int16\_t:** Signed 16-bit integer.

**uint16\_t:** Unsigned 16-bit integer.

**int32\_t:** Signed 32-bit integer.

**uint32\_t:** Unsigned 32-bit integer.

**int64\_t:** Signed 64-bit integer.

**uint64\_t:** Unsigned 64-bit integer.

**Minimum-width Integer Types:** These types guarantee a minimum size in bits but may have a larger size depending on the platform. Examples are:

**int\_least8\_t:** Signed integer with at least 8 bits.

**uint\_least8\_t:** Unsigned integer with at least 8 bits.

**int\_least16\_t:** Signed integer with at least 16 bits.

**uint\_least16\_t:** Unsigned integer with at least 16 bits.

**int\_least32\_t:** Signed integer with at least 32 bits.

**uint\_least32\_t:** Unsigned integer with at least 32 bits.

**int\_least64\_t:** Signed integer with at least 64 bits.

**uint\_least64\_t:** Unsigned integer with at least 64 bits.

**Fastest Integer Types:** These types are optimized for performance and may have a larger size than the minimum required. Examples are:

**int\_fast8\_t:** Fastest signed integer with at least 8 bits.

**uint\_fast8\_t:** Fastest unsigned integer with at least 8 bits.

**int\_fast16\_t:** Fastest signed integer with at least 16 bits.

**uint\_fast16\_t:** Fastest unsigned integer with at least 16 bits.

**int\_fast32\_t:** Fastest signed integer with at least 32 bits.

**uint\_fast32\_t:** Fastest unsigned integer with at least 32 bits.

**int\_fast64\_t:** Fastest signed integer with at least 64 bits.

**uint\_fast64\_t:** Fastest unsigned integer with at least 64 bits.

### Other Integer Types:

**intptr\_t:** Signed int type capable of holding a pointer.

**intmax\_t:** Largest signed integer type.

**uintptr\_t:** Unsigned integer type capable of holding a pointer.

**uintmax\_t:** Largest unsigned integer type.

These type aliases provide a standardized way to declare integer types with specific characteristics, ensuring code portability and clarity across different platforms. They are particularly useful in systems programming and when working with low-level data representations.

- **Floating-point Types:** Represent real numbers. (C++98)
  - **float:** Single-precision floating-point type.
  - **double:** Double-precision floating-point type.
  - **long double:** Extended precision floating-point type.

- **Boolean Type:** (C++98)
  - **bool:** Represents boolean values true and false.

- **Character Types:** (C++98)

- **char:** Represents a single character.
- **wchar\_t:** Represents wide characters.

*Character Types for Unicode Support (C++11 onwards):*

- **char8\_t:** Represents UTF-8 characters.
- **char16\_t:** Represents UTF-16 characters.
- **char32\_t:** Represents UTF-32 characters.

- **void Type:** (C++98)

- **void:** Represents the absence of type or value.

#### ✚ **Derived Data Types:**

- **Array Types**(C++98): `type[]` or `type[size]`: Array of values of type.
- **Pointer Types**(C++98): `type*`: Pointer to a value of type.
- **Reference Types**(C++98): `type&`: Reference to a value of type.
- **Automatic Type**(C++11): **'auto'** is used for automatic type deduction at compile time. You can use auto when declaring variables to have the compiler deduce the type based on the initializer expression.
- **decltype type**(C++11): **'decltype'** is used to obtain the type of an expression or entity at compile time. It's particularly useful when you want to declare a variable to have the same type as an existing expression or entity.
- **Smart Pointers** (C++11): `std::unique_ptr`, `std::shared_ptr`, `std::weak_ptr`: Smart pointers in C++ are a category of classes that manage the memory allocated dynamically with new. They automatically free the memory when the pointer goes out of scope, thus preventing memory leaks.

#### ✚ **User-Defined Types:** (C++98):

- **Structures (struct):** User-defined compound data type consisting of members with different data types.
- **Classes (class):** User-defined compound data type with data members and member functions.
- **Unions (union):** User-defined data type that holds only one of its data members at a time.
- **Enumerated Types(enum):** User-defined enumeration type. in C++ provide a way to define a set of named integer constants.

Understanding these data types is essential for writing efficient and robust C++ programs. Each data type has its own characteristics and usage patterns, and choosing the appropriate data type for a given situation is crucial for achieving the desired behaviour and performance in a program.

List of common data types in C++, along with their introduced versions, memory size in bytes (on typical systems), and the range of values they can represent:

Data Type	Introduced Version	Memory (bytes)	Range
<code>`bool`</code>	C++98	1	<code>`true`</code> or <code>`false`</code>
<code>`char`</code>	C++98	1	-128 to 127 (signed), 0 to 255 (unsigned)
<code>`wchar_t`</code>	C++98	2 or 4	platform-dependent, typically larger than <code>`char`</code>
<code>`char8_t`</code>	C++20	1	UTF-8 encoded character
<code>`char16_t`</code>	C++11	2	UTF-16 encoded character
<code>`char32_t`</code>	C++11	4	UTF-32 encoded character
<code>`short`</code>	C++98	2	-32,768 to 32,767 (signed), 0 to 65,535 (unsigned)
<code>`int`</code>	C++98	4	-2,147,483,648 to 2,147,483,647 (signed)
<code>`long`</code>	C++98	4 or 8	-2,147,483,648 to 2,147,483,647 (signed)
<code>`long long`</code>	C++11	8	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 (signed)
<code>`float`</code>	C++98	4	~7 significant digits
<code>`double`</code>	C++98	8	~15 significant digits
<code>`long double`</code>	C++98	8 or 12 or 16	~15 significant digits
<code>`void`</code>	C++98	N/A	N/A

Now we will proceed further with the simple program and we are going to print the student details only using the `cout`. (using online compiler: [https://www.onlinegdb.com/online\\_c++\\_compiler](https://www.onlinegdb.com/online_c++_compiler))

```

1 #include <iostream>
2 using namespace std;
3 int main() {
4     cout << "Personal Details";
5     cout << "Name: Ratan Sudarshan Sutar";
6     cout << "Age: 20";
7     cout << "Address: 123 Main Road, Pune, Maharashtra, India";
8     cout << "Phone: +91 9850 67 84 51";
9
10    cout << "Academic Details";
11    cout << "University: Pune University";
12    cout << "Major: Computer Science";
13    cout << "GPA: 3.8";
14
15    return 0;
16 }
  
```

input

```

Personal DetailsName: Ratan Sudarshan SutarAge: 20Address: 123 Main Road, Pune, Maharashtra, India
Phone: +91 9850 67 84 51Academic DetailsUniversity: Pune UniversityMajor: Computer ScienceGPA: 3.8
...Program finished with exit code 0
Press ENTER to exit console.
  
```

The above program gives the output in unformatted manner to display the output in well formatted manner we have to use the “Escape Sequence Characters”. Here is a list of commonly used escape sequence characters in C++:

Escape Sequence	Meaning	Example	Output
<code>`\n`</code>	Newline	<code>`"Line 1\nLine 2"`</code>	Line 1 Line 2
<code>`\t`</code>	Horizontal tab	<code>`"Column 1\tColumn 2\tColumn 3"`</code>	Column 1 Column 2 Column 3
<code>`\b`</code>	Backspace	<code>`"123\b45"`</code>	1245
<code>`\r`</code>	Carriage return	<code>`"1234\rAB"`</code>	AB34
<code>`\\`</code>	Backslash	<code>`"C:\\path\\to\\file"`</code>	C:\path\to\file
<code>`\'`</code>	Single quote	<code>`"It\'s raining."`</code>	It's raining.
<code>`\"`</code>	Double quote	<code>`"She said, \"Hello.\""`</code>	She said, "Hello."
<code>`\0`</code>	Null character	<code>`"Hello\0World"`</code>	Hello
<code>`\a`</code>	Alert (bell)	<code>`"Beep!\a"`</code>	(Produces an audible alert or visual bell)
<code>`\f`</code>	Form feed	<code>`"Page 1\fPage 2"`</code>	Page 1  Page 2
<code>`\v`</code>	Vertical tab	<code>`"Line 1\vLine 2"`</code>	Line 1 Line 2
<code>`\?`</code>	Question mark	<code>`"What\? Why\?"`</code>	What? Why?
<code>`\nnn`</code>	Octal representation	<code>`"\110\145\154\154\157"`</code>	Hello
<code>`\xhh`</code>	Hexadecimal representation	<code>`"\x48\x65\x6C\x6C\x6F"`</code>	Hello

These escape sequences can be used within character or string literals to represent special characters or control characters. So above program can be written as

```
main.cpp
1  #include <iostream>
2  int main() {
3      std::cout << "Escape Sequence Characters Demo:\n";
4      std::cout << "1. Newline: Line 1\nLine 2\nLine 3\n";
5      std::cout << "2. Horizontal tab: Column 1\tColumn 2\tColumn 3\n";
6      std::cout << "3. Backspace: 123\b45\n";
7      std::cout << "4. Carriage return: 1234\rAB\n";
8      std::cout << "5. Backslash: C:\\path\\to\\file\n";
9      std::cout << "6. Single quote: It\'s raining.\n";
10     std::cout << "7. Double quote: She said, \"Hello.\" \n";
11     std::cout << "8. Null character: Hello\0World\n";
12     std::cout << "9. Alert (bell): Beep!\a\n";
13     std::cout << "10. Form feed: Page 1\fPage 2\n";
14     std::cout << "11. Vertical tab: Line 1\vLine 2\n";
15     std::cout << "12. Question mark: What\? Why\?\n";
16     std::cout << "13. Octal representation: \110\145\154\154\157\n";
17     std::cout << "14. Hexadecimal representation: \x48\x65\x6C\x6C\x6F\n";
18
19     return 0;
20 }
```

**Manipulators in C++:** In C++, manipulators are special functions or objects provided by the Standard Library that are used to modify the behaviour of input and output streams. They are typically used in conjunction with the insertion (<<) and extraction (>>) operators to format input and output data. Manipulators can perform various tasks such as setting field width, formatting numbers, controlling precision, and more.

Here are some commonly used manipulators in C++:

Manipulator	Description	Example
<code>`std::setw(int n)`</code>	Sets the width of the next output field to `n` characters.	<code>`std::cout &lt;&lt; std::setw(10) &lt;&lt; "Hello";`</code>
<code>`std::setprecision(int n)`</code>	Sets the precision (number of digits) to `n` for floating-point output.	<code>`std::cout &lt;&lt; std::setprecision(3) &lt;&lt; 3.14159;`</code>
<code>`std::fixed`</code>	Ensures that floating-point numbers are displayed in fixed-point notation.	<code>`std::cout &lt;&lt; std::fixed &lt;&lt; 3.14159;`</code>
<code>`std::scientific`</code>	Ensures that floating-point numbers are displayed in scientific notation.	<code>`std::cout &lt;&lt; std::scientific &lt;&lt; 314.159;`</code>
<code>`std::left`</code>	Sets the output to be left-aligned within its field.	<code>`std::cout &lt;&lt; std::left &lt;&lt; std::setw(10) &lt;&lt; "Hello";`</code>
<code>`std::right`</code>	Sets the output to be right-aligned within its field (default behavior).	<code>`std::cout &lt;&lt; std::right &lt;&lt; std::setw(10) &lt;&lt; "Hello";`</code>
<code>`std::boolalpha`</code>	Outputs boolean values as <code>`true`</code> or <code>`false`</code> instead of <code>`1`</code> or <code>`0`</code> .	<code>`std::cout &lt;&lt; std::boolalpha &lt;&lt; true;`</code>
<code>`std::hex`</code>	Sets the output base to hexadecimal.	<code>`std::cout &lt;&lt; std::hex &lt;&lt; 255;`</code>
<code>`std::oct`</code>	Sets the output base to octal.	<code>`std::cout &lt;&lt; std::oct &lt;&lt; 255;`</code>
<code>`std::dec`</code>	Sets the output base to decimal (default behavior).	<code>`std::cout &lt;&lt; std::dec &lt;&lt; 255;`</code>

These manipulators, along with others provided by the C++ Standard Library, allow for fine control over the formatting and presentation of data during input and output operations.

Let's see one program demonstrating all above manipulators.

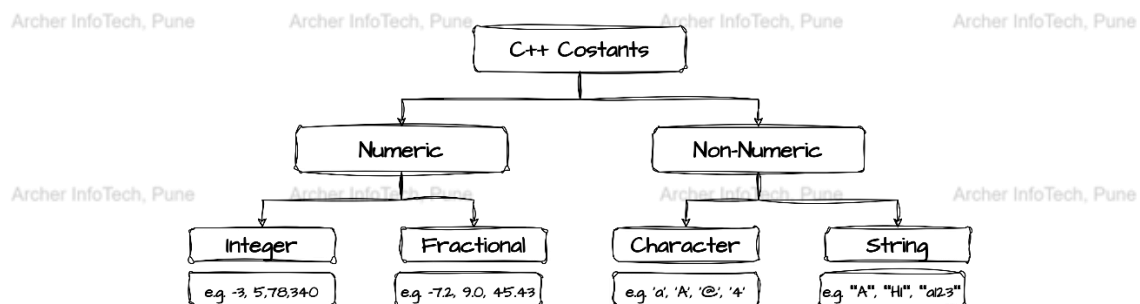
```

main.cpp
1  #include <iostream>
2  #include <iomanip> // for manipulators
3  int main() {
4      // setw(int n)
5      std::cout << std::setw(20) << "Name" << std::setw(10) << "Age" << std::setw(15) << "Salary" << std::endl;
6      std::cout << std::setw(20) << "John Doe" << std::setw(10) << 30 << std::setw(15) << 50000.0 << std::endl;
7      // setprecision(int n) and fixed
8      double pi = 3.14159265359;
9      std::cout << std::setprecision(3) << std::fixed << "Pi: " << pi << std::endl;
10     // scientific
11     double num = 123456.789;
12     std::cout << std::scientific << "Number: " << num << std::endl;
13     // left and right
14     std::cout << std::setw(10) << std::left << "Left" << std::setw(10) << std::right << "Right" << std::endl;
15     // boolalpha
16     bool flag = true;
17     std::cout << std::boolalpha << "Flag: " << flag << std::endl;
18     // hex, oct, and dec
19     int value = 255;
20     std::cout << "Hex: " << std::hex << value << std::endl;
21     std::cout << "Oct: " << std::oct << value << std::endl;
22     std::cout << "Dec: " << std::dec << value << std::endl;
23     return 0;
24 }

```

Note that, you can use "using namespace std;" to avoid std:: in the each instruction.

**Constants and Literals in C++:** In C, we have seen constants which all are valid in C++. But here in C++ the constants are represent in different aspects.



In C++, literals and constants are both used to represent fixed values, but they have different roles and characteristics:

**Literals:** Literals are fixed values, appear directly in the source code. They represent specific data types, such as integers, floating-point numbers, characters, strings, or boolean values.

Example: 42 (integer literal), 3.14 (floating-point literal), 'A' (character literal), "hello" (string literal), true (boolean literal), 054 (Octal literal), 0x3A (Hexadecimal literal), L' ()

- Literals are directly used in expressions or assignments to represent constant values.
- They are interpreted by the compiler according to their data type.
- Literals are immutable and cannot be modified.

**Constants:** Constants are named values that are declared and defined using identifiers. They represent values that remain unchanged throughout the execution of a program.

Example: `const int MAX_VALUE = 100;`      `const double PI = 3.14;`

- Constants are declared using the `const` keyword followed by a data type and an identifier.



- They are typically used to improve code readability by giving meaningful names to fixed values.
- Constants are stored in memory like variables, but their values cannot be modified once initialized.
- Constants can have global or local scope, depending on where they are declared.

Literals represent fixed values directly in the code, while constants provide named identifiers for fixed values. Literals are immutable and cannot be modified, whereas constants cannot be modified once initialized. Literals are used directly in expressions, assignments, or function calls. Constants are used to declare fixed values that are referenced multiple times in the code. In classic to modern C++ there are different ways of defining constant, which are listed below

- Using const Keyword: (C++98)
- Using constexpr Keyword: (C++11)
- Using enum: (C++98)
- Using #define Preprocessor Directive: (C++98)
- Using using Alias: (C++14)

**Using const Keyword (C++98):** Defining constants using const keyword is one of the older methods that C++ inherited from C language. In this method, we add the const keyword in the variable definition as shown: ***const DATATYPE variable\_name = value;***

- Declaring constants using the const keyword it cannot be modified after initialization.
- Constants defined with const are used when the value is known at compile time and is not expected to change during the execution of the program.
- It prevents accidental modification of constant values, which can lead to.
- It allows the compiler to perform optimizations, knowing that the value of the variable will not change.

```
main.cpp
1  #include <iostream>
2  int main() {
3      // Declaring integer constants
4      const int MAX_VALUE = 100;
5      const int MIN_VALUE = 0;
6
7      // Declaring floating-point constants
8      const double PI = 3.14159;
9      const float GRAVITY = 9.81f;
10
11     // Declaring character constants
12     const char NEWLINE = '\n';
13     const char TAB = '\t';
14
15     // Using constants in expressions
16     int range = MAX_VALUE - MIN_VALUE;
17     double circumference = 2 * PI * 5.0;
18
19     // Outputting constants
20     std::cout << "Range: " << range << NEWLINE;
21     std::cout << "Circumference: " << circumference << NEWLINE;
22
23     return 0;
24 }
```

Let's see another program and analyse the results carefully.

```
main.cpp
1  #include<iostream>
2  using namespace std;
3  int main()
4  {
5      int x=10;
6      // const int y; // error: uninitialized const 'y'
7      const int y=23;
8      cout<<endl<<"x="<<x<<"\t y="<<y;
9
10     // lets changer the values of x and y
11     x=100;
12     y=200; //error: assignment of read-only variable 'y'
13     cout<<endl<<"x="<<x<<"\t y="<<y;
14     return 0;
15 }
```

Note:

- You must initialize the const variable, otherwise it will generate the error message
- Once you initialize the const variable, it cannot be modified. If you try it; it will generate an error message.

**Using constexpr Keyword to declare a constant (C++11):** The **constexpr** keyword in C++ allows you to declare constants that are evaluated at compile time whenever possible. This keyword was introduced in C++11 and provides compile-time computation capabilities, improving performance and enabling the use of constants in contexts requiring compile-time evaluation.

```
main.cpp
1  #include<iostream>
2  constexpr int square(int x) {
3      return x * x;
4  }
5
6  int main() {
7      constexpr int side = 5;
8      constexpr int area = square(side);
9      std::cout<<"Area is: "<<area;
10
11     // here we can not change the values of variables side and area,
12     // if someone tries the compiler will generate an error message.
13     return 0;
14 }
15
```

In this example, **constexpr** is used to declare the variable **side** as a compile-time constant. The function **square** is also declared as **constexpr**, indicating that it can be evaluated at compile time when passed a **constexpr** argument.

**Using enum (C++98):** In C++, an **enum** (short for enumeration) is a user-defined type that consists of a set of named integral constants. Enums provide a way to define symbolic names for a set of related constants.

syntax:

```
enum <name>{const1, const2,...,constN};
```

e.g. `enum color {red, green, blue, black};` or `enum {red, green, blue, black};`

The general convention is the constants are represented in uppercase and if multiple words then they are separated by the underscore (\_). So above enum can be declared as

```
enum color {RED, GREEN, BLUE, BLACK}; or enum {RED, GREEN, BLUE, BLACK};
```

RED Equivalent to 0

GREEN Equivalent to 1

BLUE Equivalent to 2

BLACK Equivalent to 3

here are different styles of enum declaration in C++ along with examples:

#### Unscoped Enum:

```
// Declaration
```

```
enum Color {
```

```
    RED,
```

```
    GREEN,
```

```
    BLUE
```

```
};
```

```
// Usage
```

```
Color c = RED;
```

#### Scoped Enum:

```
// Declaration
```

```
enum class Fruit {
```

```
    APPLE,
```

```
    BANANA,
```

```
    ORANGE
```

```
};
```

```
// Usage
```

```
Fruit f = Fruit::APPLE;
```

#### Scoped Enum with Underlying Type: (C++11):

```
// Declaration
```

```
enum class Month : int {
```

```
    JAN = 1,
```

```
    FEB,
```

```
    MAR
```

```
};
```

```
// Usage
```

```
Month m = Month::JAN;
```

#### Unscoped Enum with Explicit Values:

```
// Declaration
```

```
enum Animal {
```

```
    DOG = 10,
```

```
    CAT = 20,
```

```
    BIRD = 30
```

```
};
```

```
// Usage
```

```
Animal a = DOG;
```

#### Scoped Enum with Explicit Values:

```
// Declaration
```

```
enum Animal {
```

```
    DOG = 10,
```

```
    CAT = 20,
```

```
    BIRD = 30
```

```
};
```

```
// Usage
```

```
Animal a = DOG;
```

#### Using typedef with Enum:

```
// Declaration
```

```
enum class Status : char {
```

```
    OK = 'O',
```

```
    ERROR = 'E',
```

```
    UNKNOWN = 'U'
```

```
};
```

```
// Usage
```

```
Status s = Status::ERROR;
```

**Enum Class with Inline Initialization (C++17 and later):**

// Declaration

enum class TrafficLight {

RED = 10,

YELLOW = 20,

GREEN = 30

} tl = TrafficLight::RED;

// Usage

TrafficLight currentLight = tl;

**Enum Examples:**

```

main.cpp
1  #include<iostream>
2  using namespace std;
3
4  // Define an enum named Color
5  enum Color {
6      RED,
7      GREEN,
8      BLUE
9  };
10
11 int main() {
12     // Declare variables of type Color
13     Color c1, c2;
14     // Assign values to variables
15     c1 = RED;
16     c2 = BLUE;
17     // Print the values
18     std::cout << "c1: " << c1 << std::endl; // Output: 0 (since RED is 0)
19     std::cout << "c2: " << c2 << std::endl; // Output: 2 (since BLUE is 2)
20
21     return 0;
22 }

```

```

main.cpp
1  #include<iostream>
2  // Define an enum named Weekday with explicit values
3  enum Weekday {
4      MON = 1,
5      TUE,
6      WED,
7      THU,
8      FRI,
9      SAT,
10     SUN
11 };
12 int main() {
13     // Declare a variable of type Weekday
14     Weekday today;
15     // Assign a value
16     today = WED;
17     // Print the value
18     std::cout << "Today is day " << today << std::endl; // Output: Today is day 3
19
20     return 0;
21 }

```

main.cpp

```

1  #include<iostream>
2  enum Day {
3      MON,
4      TUE,
5      WED,
6      THU,
7      FRI,
8      SAT,
9      SUN
10 };
11 void print_day(Day day) {
12     switch (day) {
13         case MON:
14             std::cout << "Monday" << std::endl;
15             break;
16         case TUE:
17             std::cout << "Tuesday" << std::endl;
18             break;
19         case WED:
20             std::cout << "Wednesday" << std::endl;
21             break;
22         case THU:
23             std::cout << "Thursday" << std::endl;
24             break;
25         case FRI:
26             std::cout << "Friday" << std::endl;
27             break;
28         case SAT:
29             std::cout << "Saturday" << std::endl;
30             break;
31         case SUN:
32             std::cout << "Sunday" << std::endl;
33             break;
34         default:
35             std::cout << "Invalid day" << std::endl;
36     }
37 }
38 int main() {
39     Day today = WED;
40     print_day(today); // Output: Wednesday
41     return 0;
42 }

```

**Using #define Preprocessor Directive (C++98):** In C++, macros refer to the #define preprocessor directive, which is used to define symbolic constants and perform simple textual substitutions in code before compilation. Macros allow you to define shorthand notations for frequently used values or expressions, as well as to conditionally compile or include code based on certain conditions. Macros can be used to define symbolic constants, making the code more readable and maintainable.

For example: #define PI 3.14159      and      #define MAX\_SIZE 100

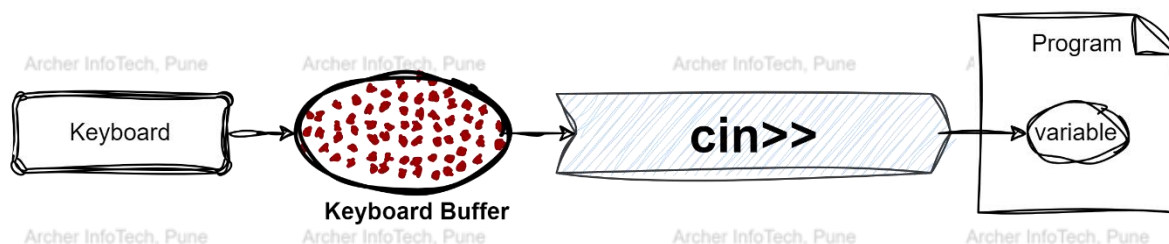
**Using using Alias (C++14):** You can use the using keyword to create an alias for a constant. This provides a more modern and type-safe alternative to using macros for defining constants. Here's how you can use using aliases to declare constants:

```
main.cpp
1 // Using alias for a constant
2 #include<iostream>
3 using PI = double;
4 constexpr PI PI_VALUE = 3.14159;
5
6 int main() {
7     // Usage of the constant
8     double radius = 5.0;
9     double area = PI_VALUE * radius * radius;
10    std::cout << "Area: " << area;
11    return 0;
12 }
```

**Input and Output in C++ using cin, cout:** cin and cout are standard input and output streams in C++ respectively, which are part of the C++ Standard Library. They are used for reading input from the user and printing output to the console.

**cin (Standard Input):** cin is a predefined object of the istream class, which represents the standard input stream. It is used to read data from the standard input device, typically the keyboard, and store it into variables. >> operator is actually bitwise right shift operator, which is overloaded here along with cin, to extract the data from stream and therefore it is known as **get-in or extraction operator**.

**Syntax:** cin >> <variable\_name>;

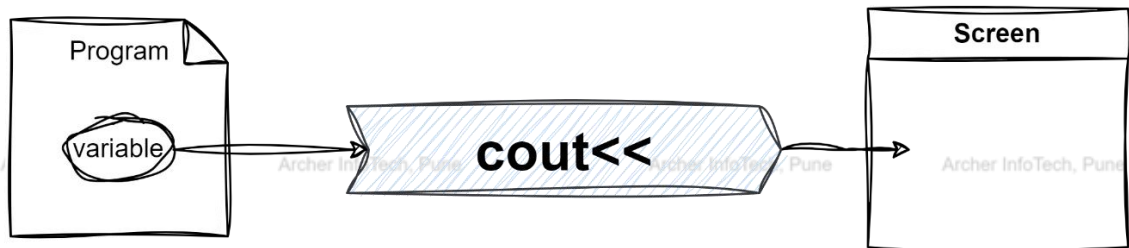


Note that, when you are using the cin >> to take the data, no need of using & operator to get the address of variable, it will be taken automatically.

**cout (Standard Output):** cout is a predefined object of the ostream class, which represents the standard output stream. It is used to output data to the standard output device, typically the console. << operator is actually bitwise left-shift operator, which when used along with cout, it will inserts / prints the data on screen and therefore it is called as **insertion operator or put-to operator**.

**Syntax:** cout << "<format\_string>" << <variable\_name>;





Let's see, how different integer values are entered and displayed.

```

main.cpp
1  #include <iostream>
2  int main() {
3      int intValue;
4      short shortValue;
5      long longValue;
6      long long longLongValue;
7
8      // Input different types of integers
9      std::cout << "Enter an integer (int): ";
10     std::cin >> intValue;
11
12     std::cout << "Enter an integer (short): ";
13     std::cin >> shortValue;
14
15     std::cout << "Enter an integer (long): ";
16     std::cin >> longValue;
17
18     std::cout << "Enter an integer (long long): ";
19     std::cin >> longLongValue;
20
21     // Display the input integers
22     std::cout << "Integer (int): " << intValue << std::endl;
23     std::cout << "Integer (short): " << shortValue << std::endl;
24     std::cout << "Integer (long): " << longValue << std::endl;
25     std::cout << "Integer (long long): " << longLongValue << std::endl;
26
27     return 0;
28 }
  
```

```

main.cpp
1  #include <iostream>
2  #include <cstdint> // Include the header file for uint32_t
3
4  int main() {
5      uint32_t uintValue; // Define a variable of type uint32_t
6
7      // Input a value of type uint32_t
8      std::cout << "Enter an unsigned integer (uint32_t): ";
9      std::cin >> uintValue;
10     // Display the input value
11     std::cout << "Unsigned integer (uint32_t): " << uintValue << std::endl;
12
13     return 0;
14 }
  
```

Now let's see, How the different types of fractional values are entered and displayed on screen. i.e. how to use float, double and long double.

```
main.cpp
1  #include <iostream>
2
3  int main() {
4      float floatValue;
5      double doubleValue;
6      long double longDoubleValue;
7
8      // Input different types of fractional values
9      std::cout << "Enter a fractional value (float): ";
10     std::cin >> floatValue;
11
12     std::cout << "Enter a fractional value (double): ";
13     std::cin >> doubleValue;
14
15     std::cout << "Enter a fractional value (long double): ";
16     std::cin >> longDoubleValue;
17
18     // Display the input fractional values
19     std::cout << "Fractional value (float): " << floatValue << std::endl;
20     std::cout << "Fractional value (double): " << doubleValue << std::endl;
21     std::cout << "Fractional value (long double): " << longDoubleValue << std::endl;
22
23     return 0;
24 }
25
```

Now, let's see how to deal with the boolean type.

Archer InfoTech, Pune

Archer InfoTech, Pune

The **Boolean data type (bool keyword)** used to hold the result of condition, it is value 1 when the condition is True and 0 when the condition is false. C++20 introduces boolean literals true and false, which can be used to represent true and false values respectively.

Archer InfoTech, Pune

Archer InfoTech, Pune

Archer InfoTech, Pune

Archer InfoTech, Pune

**std::boolalpha** is a manipulator in C++ streams that affects the formatting of boolean values. It converts any non-zero value to **'true'** and zero to **'false'**.

```
main.cpp
1  #include <iostream>
2  int main()
3  {
4      int x=10;
5      bool b= (x>10);
6      cout<<b<<endl; // 0
7
8      bool ans=(x%5==0);
9      cout<<ans<<endl; // 1
10
11     bool isTrue = true;
12     bool isFalse = false;
13
14     std::cout << "isTrue: " << isTrue << std::endl; // Output: 1
15     std::cout << "isFalse: " << isFalse << std::endl; // Output: 0
16
17     std::cout << std::boolalpha;
18     std::cout << "isTrue: " << isTrue << std::endl; // Output: true
19     std::cout << "isFalse: " << isFalse << std::endl; // Output: false
20
21     return 0;
22 }
```

**Character Data Types in C++20:** We have narrow native character type **char** and the wide native character type **wchar\_t** and additional character Types for Unicode Support (C++11 onwards) **char16\_t** and **char32\_t**, and **char8\_t** (C++20). There are no standard output streams for Unicode char types. Standard output streams exist only for **char** and **wchar\_t**. You will have to convert the **char16\_t** and **char32\_t** (and **char8\_t**) strings to one of those types in order to meaningfully write to the standard output stream. Due to lack of standard output stream support, **char16\_t** and **char32\_t** (and **char8\_t**) are only really useful for writing into files. It will work only if the source file, the compiler, and the terminal are all using same Unicode char type, which is not always the case.

```
main.cpp
1  #include <iostream>
2
3  int main() {
4      char ch;
5      std::cout << "Enter a character: ";
6      std::cin >> ch;
7      std::cout << "Character entered: " << ch << std::endl;
8      return 0;
9  }
```

```
main.cpp
1  #include <iostream>
2
3  int main() {
4      wchar_t wch;
5      std::wcout << L"Enter a wide character: ";
6      std::wcin >> wch;
7      std::wcout << L"Wide character entered: " << wch << std::endl;
8      return 0;
9  }
```

**wchar\_t** can be used to hold a wider range of characters compared to the **char** data type.

**char16\_t** and **char32\_t** (and **char8\_t**) will not execute on each system, they will work on system where editor, compiler and terminal using the same Unicode character type so on your system following program may generate the error.

```
main.cpp
1  #include <iostream>
2  int main() {
3      // Input a UTF-8 encoded character
4      std::cout << "Enter a UTF-8 character: ";
5      char8_t ch8;
6      std::cin >> ch8;
7
8      // Input a UTF-16 encoded character
9      std::wcout << L"Enter a UTF-16 character: ";
10     char16_t ch16;
11     std::wcin >> ch16;
12
13     // Input a UTF-32 encoded character
14     std::wcout << L"Enter a UTF-32 character: ";
15     char32_t ch32;
16     std::wcin >> ch32;
17
18     // Display the input characters
19     std::cout << "UTF-8 Character: " << ch8 << std::endl;
20     std::wcout << L"UTF-16 Character: " << ch16 << std::endl;
21     std::wcout << L"UTF-32 Character: " << ch32 << std::endl;
22
23     return 0;
24 }
```

**Character array input in C++:** There are two different ways of representing the character collection in C++, first by using the character array and another by using the string class. Now let's see how to deal with the character array.

We can use the `cin>> ar_nm;` to input the array but it is good for single word string. In multiword string it reads characters until it encounters whitespace and then stops, leaving the remaining characters in the input buffer. So to deal with this problem C++ std namespace provides the `std::cin.getline()` method. And can be used as shown below.

```
main.cpp
1  #include <iostream>
2
3  int main() {
4      const int MAX_SIZE = 100; // Maximum size of the character array
5      char charArray[MAX_SIZE]; // Declare a character array
6
7      // Input a character array with spaces
8      std::cout << "Enter a string with spaces: ";
9      std::cin.getline(charArray, MAX_SIZE);
10
11     // Display the character array
12     std::cout << "Entered string: " << charArray << std::endl;
13
14     return 0;
15 }
```

Sometime due to remaining character from previous input present in the buffer you may face the problem with `getline()` i.e. will proceed further without taking input, and it can be solved by calling `fflush(stdin)` or `std::cin.ignore()`.

Note that C does not consider the '\0' (NULL) as a part of String where in C++ '\0' is the part of string and therefore length of C++ character array is greater than one as compares to number of characters in it.

```
main.cpp
1  #include <iostream>
2  int main()
3  {
4      char str[]="hello";
5      int char_count=sizeof(str);
6      std::cout<<char_count;
7      return 0;
8  }
9
```

**Using void data type:** We have seen the void as a returning type of a function when function returning nothing in response to call, also it is used to indicate the function does not collect any arguments as shown below.

**`void display(void) {...}`**

This is function without arguments with any returning type.

Here in C++ the **void** type is additionally used to declare the **Generic Pointer**. The generic pointer is a pointer which don't have any type initially but it is convertible to any type.

```
main.cpp
1  #include <iostream>
2  int main ()
3  {
4      int x = 10;
5      double db = 45.67;
6      char ch = 'a';
7
8      void *gp; // here gp is a Generic Pointer
9
10     gp = &x;
11     std::cout << "\n value of x using gp: " << *(int*)gp; // Converted to integer pointer
12
13     gp = &db;
14     std::cout << "\n value of db using gp: " << *(double*)gp; // Converted to double pointer
15
16     gp = &ch;
17     std::cout << "\n value of ch using gp: " << *(char*)gp; // Converted to character pointer
18
19     return 0;
20 }
21
```

In C++, **static\_cast** is a casting operator that performs conversions between compatible types. It's a compile-time cast that is safer than traditional C-style casts because it performs checks at compile-time to ensure type safety. **static\_cast** is mainly used for implicit conversions.

```
main.cpp
1  #include <iostream>
2  void printValue(void* ptr) {
3      std::cout << "Value: " << *static_cast<int*>(ptr) << std::endl;
4  }
5
6  int main() {
7      int value = 42;
8      printValue(&value);
9      return 0;
10 }
11
```

**Structure in C++:** In C++, structures have evolved over time, gaining various properties and capabilities in different versions of the language. In the First Standardized Version C++98, a basic structure support with only member variable. C++11 introduced the additional functionalities such as member functions within structures, Initialization of member variables within the structure declaration, Uniform initialization syntax for initializing structure variables, and ability to specify access specifiers (public, private, protected) for structure members. There are no significant enhancements for structures in to C++14, C++17, C++20, and C++23.

Let's discuss some features of structure in C++,

In C++, you can specify the access level of structure members using access specifiers like public, private, and protected: