Last Moment Tuitions

Introduction to C++

Prepare for CDAC CCAT Exam with LMT Study Materials

Get Complete Study Notes, Practice Problems and Previous Year Question Papers, Syllabus and More

1. Introduction To C++

★ Limitations Of C Language

The limitations of C programming languages are as follows:

- Difficult to debug.
- C allows a lot of freedom in writing code, and that is why you can put an empty line or white space anywhere in the program. And because there is no fixed place to start or end the line, so it isn't easy to read and understand the program.
- C compilers can only identify errors and are incapable of handling exceptions (run-time errors).
- C provides no data protection.
- It also doesn't feature the reusability of source code extensively.
- It does not provide strict data type checking (for example, an integer value can be passed for floating datatype).

Categorisation Of High Level Languages

High-level programming languages

The **high-level programming languages** can be categorized into different types on the **basis of the application area in which they are employed** as well as the **different design paradigms** supported by them. The high-level programming languages are designed for use in a number of areas. Each high-level language is designed by keeping its target application area in mind. Some of the high-level languages are best suited for business domains, while others are apt in the scientific domain only. The high-level language can be categorized on the basis of the various programming paradigms approved by them. The programming paradigms refer to the approach employed by the programming language for solving the different types of problem.

1. Categorisation based on Application

On the basis of application area the high level language can be divided into the following types:

i) Commercial languages

These programming languages are dedicated to the commercial domain and are specially designed for solving business-related problems. These languages can be used in organizations for processing handling the data related to payroll, accounts payable and tax building applications. COBOL is the best example of the commercial based high-level programming language employed in the business domain.

ii) Scientific languages

These programming languages are dedicated to the scientific domain and are specially designed for solving different scientific and mathematical problems. These languages can be used to develop programs for performing complex calculation during scientific research. FORTRAN is the best example of scientific based language.

iii) Special purpose languages

These programming languages are specially designed for performing some dedicated functions. For example, SQL is a high-level language specially designed to interact with the database programs only. Therefore we can say that the special purpose high-level language is designed to support a particular domain area only.

iv) General purpose languages

These programming languages are used for developing different types of software applications regardless of their application area. The various examples of general purpose high-level programming languages are BASIC, C, C++, and java.

2. Categorisation based on Design paradigm

On the basis of design paradigms the high level programming languages can be categorised into the following types:

i) Procedure-oriented languages

These programming languages are also called an imperative programming language. In this language, a program is written as a sequence of procedures. Each procedure contains a series of instructions for performing a specific task. Each procedure can be called by the other procedures during the program execution. In this type of programming paradigms, a code once written in the form of a procedure can be used any number of times in the program by only specifying the corresponding procedure name. Therefore the procedure-oriented language allows the data to move freely around the system. The various examples of procedure-oriented language are FORTRAN, ALGOL, C, BASIC, and ADA.

ii) Logic-oriented languages

These languages use logic programming paradigms as the design approach for solving various computational problems. In this programming paradigm predicate logic is used to describe the nature of a problem by defining the relationship between rules and facts. Prolog is the best example of the logic-oriented programming language.

iii) Object-oriented languages

These languages use object-oriented programming paradigms as the design approach for solving a given problem. In this programming language, a problem is divided into a number of objects which can interact by passing messages to each other. C++ and C# are examples of object-oriented programming languages.

★ A Brief History of C++

- The history of C++ begins with C. The reason for this is easy to understand: C++ is built upon the foundation of C. Thus, C++ is a superset of C.
- C++ expanded and enhanced the C language to support object-oriented programming (which is described later in this module).
- C++ also added several other improvements to the C language, including an extended set of library routines. However, much of the spirit and flavor of C++ is directly inherited from C. Therefore, to fully understand and appreciate C++, you need to understand the "how and why" behind C.
- C++ is regarded as a middle-level language, as it comprises a combination of both high-level and low-level language features.
- The C++ programming language was created by Bjarne Stroustrup and his team at Bell Laboratories (AT&T, USA) to help implement simulation projects in an object-oriented and efficient way. The earliest versions, which were originally referred to as "C with classes," date back to 1980. As the name C++ implies, C++ was derived from the C programming language: ++ is the increment operator in C.

Characteristics of C++

C++ is not a purely object-oriented language but a hybrid that contains the functionality of the C programming language. This means that you have all the features that are available in C:

- Universally usable modular programs
- Efficient, close to the machine programming

- Portable programs for various platforms.

★ C++ Organization

- C++ is designed as a bridge between the programmer and the raw computer.
- The idea is to let the programmer organize a program in a way that he or she can easily understand.
- The compiler then translates the language into something the machine can use.
- Computer programs consist of two main parts: data and instructions. The computer imposes little or no organization on these two parts.
- After all, computers are designed to be as general as possible. The idea is for the programmer to impose his or her own organization on the computer and not the other way around.

C++ Compilation Process

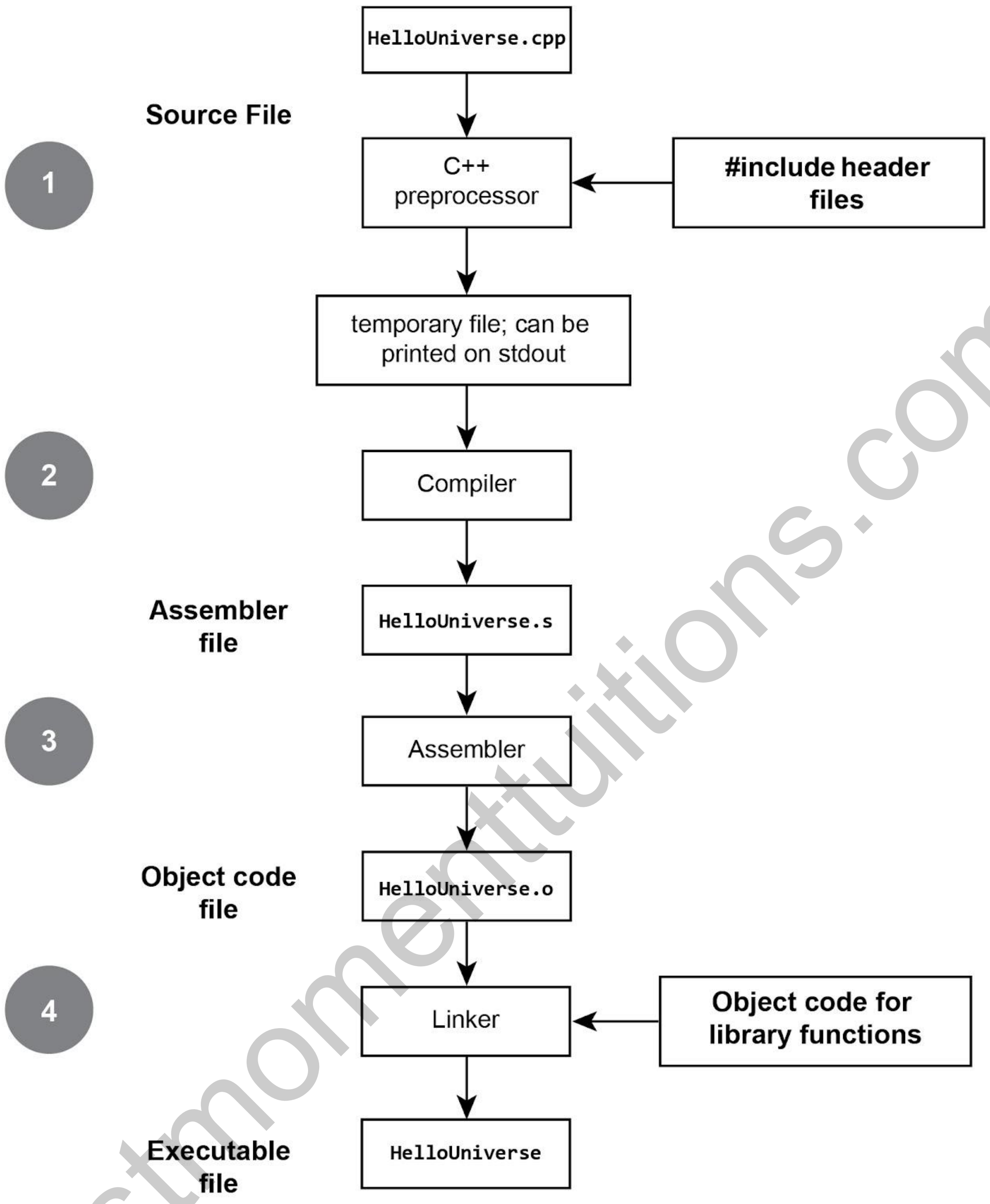
It is fundamental to know how C++ compilation works to understand how programs are compiled and executed. Compiling C++ source code into machine-readable code consists of the following four processes:

1. Preprocessing the source code.
2. Compiling the source code.
3. Assembling the compiled file.
4. Linking the object code file to create an executable file.

Let's start with a simple C++ program to understand how compilation happens.

```
#include <iostream>
int main(){
    // This is a single line comment
    /* This is a multi-line
        comment */
    std::cout << "Hello Universe" << std::endl;
    return 0;
}
```

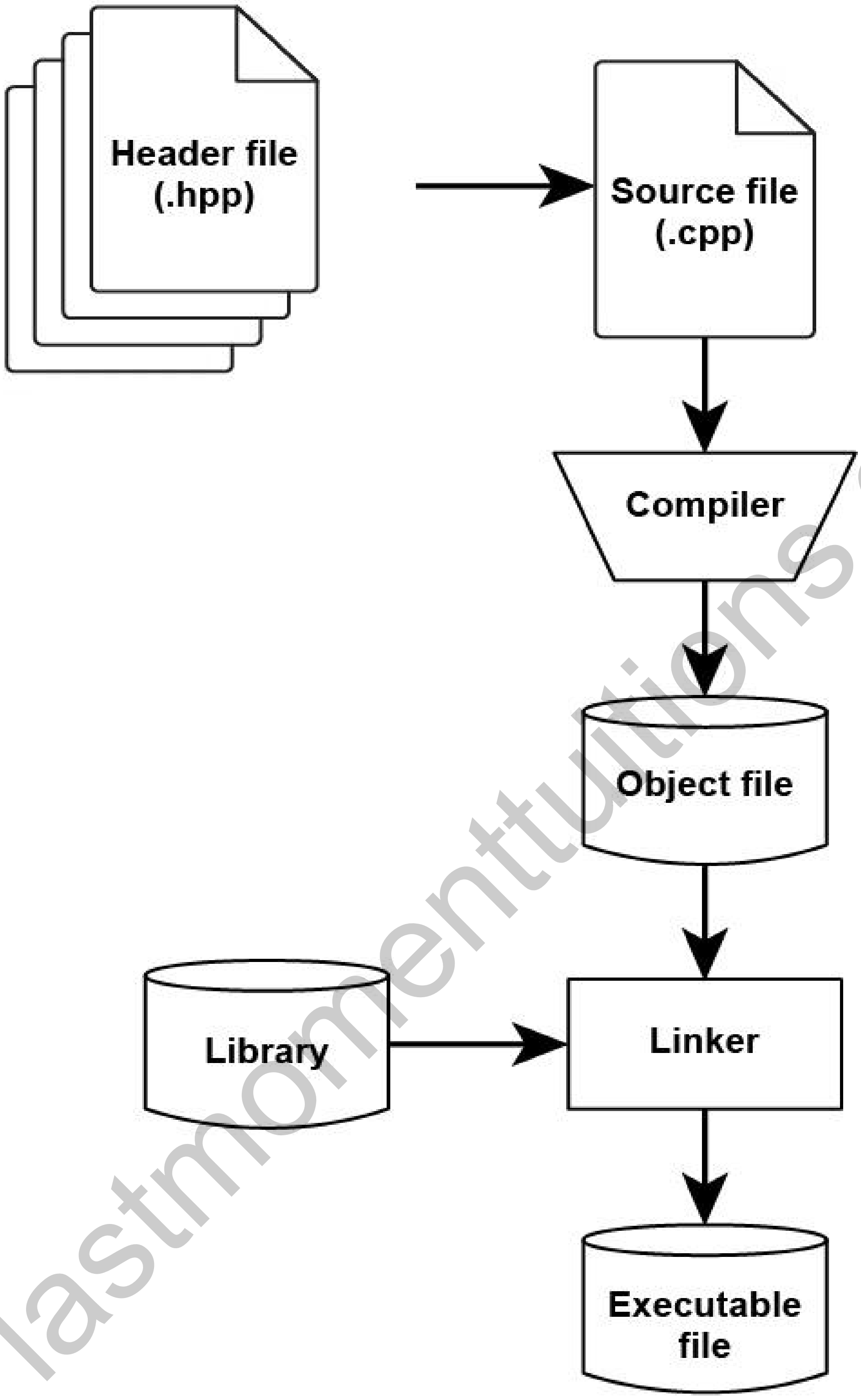
Let's demystify the C++ compilation process using the following diagram:



1. When the C++ preprocessor encounters the **#include <file>** directive, it replaces it with the content of the file creating an expanded source code file.
2. Then, this expanded source code file is compiled into an assembly language for the platform.
3. The assembler converts the file that's generated by the compiler into the object code file.

4. This object code file is linked together with the object code files for any library functions to produce an executable file.

Difference Between Header and Source Files


Header file (.hpp)
Source file (.cpp)
Compiler
Object file
Library
Linker
Executable file

Source files contain the actual implementation code. Source files typically have the extension **.cpp**, although other extensions such as **.cc**, **.ccx**, or **.c++** are also quite common.

On the other hand, header files contain code that describes the functionalities that are available. These functionalities can be referred to and used by the executable code in the source files, allowing source files to know what functionality is defined in other source files.

Compiling And Linking Process

Compilation is a process that ensures that a program is syntactically correct, but it does not perform any checks regarding its logical correctness. This means that a program that compiles correctly might still produce undesired results:

Every C++ program needs to define a starting point, that is, the part of the code the execution should start from. The convention is to have a uniquely named main function in the source code, which will be the first thing to be executed. This function is called by the operating system, so it needs to return a value that indicates the status of the program; for this reason, it is also referred to as the **exit status code**.

★ Object-Oriented Programming

Central to C++ is object-oriented programming (OOP). As just explained, OOP was the impetus for the creation of C++. Because of this, it is useful to understand OOP's basic principles before you write even a simple C++ program.

Object-oriented programming offers several major advantages to software development:

- **Reduced susceptibility to errors:** an object controls access to its own data.

More specifically, an object can reject erroneous access attempts.

- **Easy re-use:** objects maintain themselves and can therefore be used as building

blocks for other programs.

- **Low maintenance requirement:** an object type can modify its own internal

data representation without requiring changes to the application.

A First Simple C++ Program

```
/* This is a simple C++ program. Call this file Sample.cpp. */
#include<iostream>
```

```
using namespace std;
 // A C++ program begins at main().
 int main()
 {
   cout << "C++ is power programming.";
   return 0;
 }
```

Output

```
/* When run, the program displays the following output:*/
 C++ is power programming.
```

★ Basic I/O in C++

The C++ standard libraries provide an extensive set of input/output capabilities which we will see in subsequent chapters. This chapter will discuss very basic and most common I/O operations required for C++ programming.

C++ I/O occurs in streams, which are sequences of bytes. If bytes flow from a device like a keyboard, a disk drive, or a network connection etc. to main memory, this is called **input operation** and if bytes flow from main memory to a device like a display screen, a printer, a disk drive, or a network connection, etc., this is called **output operation**.

I/O Library Header Files

There are following header files important to C++ programs −

Sr.No	Header File & Function and Description
1	**<iostream>** This file defines the **cin, cout, cerr** and **clog** objects, which correspond to the standard input stream, the standard output stream, the un-buffered standard error stream and the buffered standard error stream, respectively.
2	**<iomanip>** This file declares services useful for performing formatted I/O with so-called parameterized stream manipulators, such as **setw** and **setprecision**.
3	**<fstream>** This file declares services for user-controlled file processing. We will discuss about it in detail in File and Stream related chapter.

The Standard Output Stream (cout)

The predefined object cout is an instance of ostream class. The cout object is said to be "connected to" the standard output device, which usually is the display screen. The cout is used in conjunction with the stream insertion operator, which is written as << which are two less than signs as shown in the following example.

```
#include <iostream>

using namespace std;

int main() {
   char str[] = "Hello C++";

   cout << "Value of str is : " << str << endl;
}
```

When the above code is compiled and executed, it produces the following result −

```
Value of str is : Hello C++
```

The C++ compiler also determines the data type of variable to be output and selects the appropriate stream insertion operator to display the value. The << operator is overloaded to output data items of built-in types integer, float, double, strings and pointer values.

The insertion operator << may be used more than once in a single statement as shown above and **endl** is used to add a new-line at the end of the line.

The Standard Input Stream (cin)

The predefined object **cin** is an instance of **istream** class. The cin object is said to be attached to the standard input device, which usually is the keyboard. The **cin** is used in conjunction with the stream extraction operator, which is written as >> which are two greater than signs as shown in the following example.

```
#include <iostream>
using namespace std;
int main() {
   char name[50];
   cout << "Please enter your name: ";
   cin >> name;
   cout << "Your name is: " << name << endl;
}
```

When the above code is compiled and executed, it will prompt you to enter a name. You enter a value and then hit enter to see the following result −

```
Please enter your name: cplusplus
Your name is: cplusplus
```

The C++ compiler also determines the data type of the entered value and selects the appropriate stream extraction operator to extract the value and store it in the given variables.

The stream extraction operator >> may be used more than once in a single statement. To request more than one datum you can use the following −

```
cin >> name >> age;
```

This will be equivalent to the following two statements −

```
cin >> name;
cin >> age;
```

The Standard Error Stream (cerr)

The predefined object **cerr** is an instance of **ostream** class. The cerr object is said to be attached to the standard error device, which is also a display screen but the object **cerr** is un-buffered and each stream insertion to cerr causes its output to appear immediately.

The **cerr** is also used in conjunction with the stream insertion operator as shown in the following example.

```
#include <iostream>

using namespace std;

int main() {
   char str[] = "Unable to read....";

   cerr << "Error message : " << str << endl;
}
```

When the above code is compiled and executed, it produces the following result −

```
Error message : Unable to read....
```

The Standard Log Stream (clog)

The predefined object **clog** is an instance of **ostream** class. The clog object is said to be attached to the standard error device, which is also a display screen but the object **clog** is buffered. This means that each insertion to clog could cause its output to be held in a buffer until the buffer is filled or until the buffer is flushed.

The **clog** is also used in conjunction with the stream insertion operator as shown in the following example.

```
#include <iostream>

using namespace std;

int main() {
   char str[] = "Unable to read....";
```

```
    clog << "Error message : " << str << endl;
}
```

When the above code is compiled and executed, it produces the following result −

Error message : Unable to read....

You would not be able to see any difference in cout, cerr and clog with these small examples, but while writing and executing big programs the difference becomes obvious. So it is good practice to display error messages using cerr stream and while displaying other log messages then clog should be used.

★ Datatypes In C++

There are 4 types of data types in C++ language.

Types	Data Types
Basic Data Type	int, char, float, double, etc
Derived Data Type	array, pointer, etc
Enumeration Data Type	enum
User Defined Data Type	structure

Basic Data Types

The basic data types are integer-based and floating-point based. C++ language supports both signed and unsigned literals.

The memory size of basic data types may change according to the 32 or 64 bit operating system.

Let's see the basic data types. Its size is given according to 32 bit OS.

Data Types	Memory Size	Range
char	1 byte	-128 to 127
signed char	1 byte	-128 to 127
unsigned char	1 byte	0 to 127
short	2 byte	-32,768 to 32,767
signed short	2 byte	-32,768 to 32,767
unsigned short	2 byte	0 to 32,767
int	2 byte	-32,768 to 32,767
signed int	2 byte	-32,768 to 32,767
unsigned int	2 byte	0 to 32,767
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 32,767
long int	4 byte	
signed long int	4 byte	
unsigned long int	4 byte	
float	4 byte	
double	8 byte	
long double	10 byte	

Derived Data Types

The data-types that are derived from the primitive or built-in datatypes are referred to as Derived Data Types. These can be of four types namely:

1. Function
2. Array
3. Pointer
4. Reference

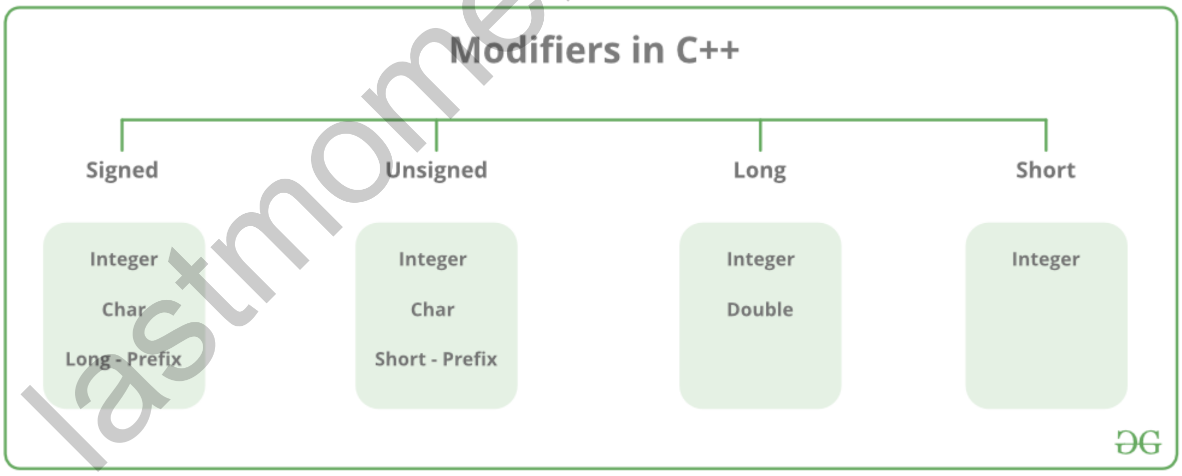
Abstract or User-Defined Data Types

These data types are defined by user itself. Like, defining a class in C++ or a structure. C++ provides the following user-defined datatypes:

1. Class
2. Structure
3. Union
4. Enumeration
5. Typedef defined DataType

Datatype Modifiers

As the name implies, datatype modifiers are used with the built-in data types to modify the length of data that a particular data type can hold.



Data type modifiers available in C++ are:

Signed

- **Unsigned**
- **Short**

- **Long**

Below table summarizes the modified size and range of built-in datatypes when combined with the type modifiers:

Data Type	Size (in bytes)	Range
short int	2	-32,768 to 32,767
unsigned short int	2	0 to 65,535
unsigned int	4	0 to 4,294,967,295
int	4	-2,147,483,648 to 2,147,483,647
long int	4	-2,147,483,648 to 2,147,483,647
unsigned long int	8	0 to 4,294,967,295
long long int	8	-(2^63) to (2^63)-1
unsigned long long int	8	0 to 18,446,744,073,709,551,615
signed char	1	-128 to 127
unsigned char	1	0 to 255
float	4	
double	8	
long double	12	
wchar_t	2 or 4	1 wide character

★ Variables

A variable is a name of memory location. It is used to store data. Its value can be changed and it can be reused many times.

It is a way to represent memory location through symbol so that it can be easily identified.

Let's see the syntax to declare a variable:

1. int x;
2. float y;
3. char z;

Here, x, y, z are variables and int, float, char are data types.

We can also provide values while declaring the variables as given below:

1. int x=5,b=10; //declaring 2 variable of integer type
2. float f=30.8;
3. char c='A';

Rules for defining variables

1. A variable can have alphabets, digits and underscore.
2. A variable name can start with an alphabet and underscore only. It can't start with a digit.
3. No white space is allowed within the variable name.
4. A variable name must not be any reserved word or keyword e.g. char, float etc.
5. Valid variable names:
   - int a;
   - int _ab;
   - int a30;
6. Invalid variable names:
   - int 4;
   - int x y;
   - int double;

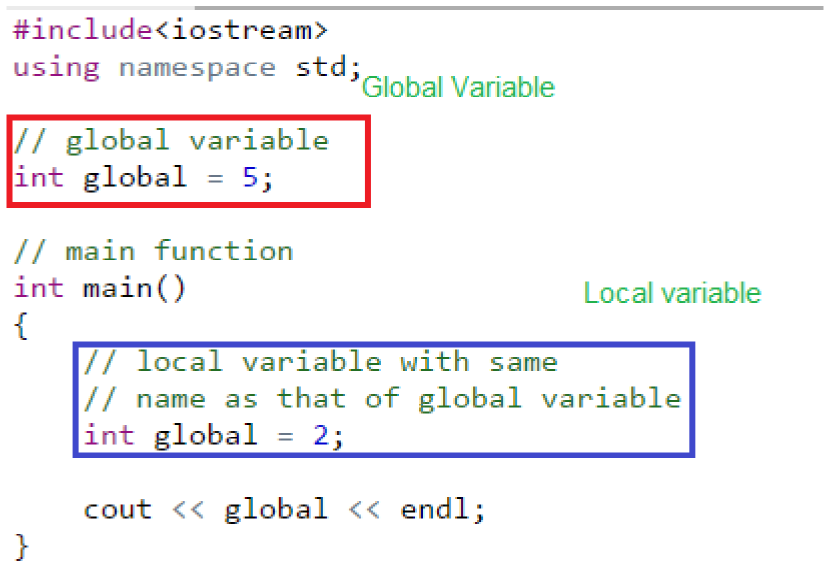
Scope Of Variables

In general, the scope is defined as the extent up to which something can be worked with. In programming also the scope of a variable is defined as the extent of the program code within which the variable can be accessed or declared or worked with.

There are mainly two types of variable scopes:

1. Local Variables

2. Global Variables



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

// global variable
int global = 5;

// main function
int main()
{
    // local variable with same
    // name as that of global variable
    int global = 2;

    cout << global << endl;
}
```

Now let's understand each of the scope at a greater detail:

Local Variables

Variables defined within a function or block are said to be local to those functions.

- Anything between '{' and '}' is said to inside a block.
- Local variables do not exist outside the block in which they are declared, i.e. they **can not** be accessed or used outside that block.
- **Declaring local variables**: Local variables are declared inside a block.

```
// CPP program to illustrate
// usage of local variables
#include<iostream>
using namespace std;
void func(){
    // this variable is local to the
    // function func() and cannot be
    // accessed outside this function
    int age=18;
}
int main() {
    cout<<"Age is: "<<age;
    return 0;
}
```

Output:

Error: age was not declared in this scope

The above program displays an error saying “age was not declared in this scope”. The variable age was declared within the function func() so it is local to that function and not visible to portion of program outside this function.

Rectified Program : To correct the above error we have to display the value of variable age from the function func() only. This is shown in the below program:

```
// CPP program to illustrate
// usage of local variables
#include<iostream>
using namespace std;

void func()
{
    // this variable is local to the
    // function func() and cannot be
    // accessed outside this function
    int age=18;
    cout<<age;
}

int main()
{
    cout<<"Age is: ";
    func();

    return 0;
}
```

Output:

Age is: 18

Global Variables

As the name suggests, Global Variables can be accessed from any part of the program.

- They are available throughout the lifetime of a program.
- They are declared at the top of the program outside all of the functions or blocks.
- **Declaring global variables**: Global variables are usually declared outside of all of the functions and blocks, at the top of the program. They can be accessed from any portion of the program.

```
// CPP program to illustrate
// usage of global variables
#include<iostream>
using namespace std;

// global variable
int global = 5;

// global variable accessed from
// within a function
void display()
{
    cout<<global<<endl;
}

// main function
int main()
{
    display();
    // changing value of global
    // variable from main function
    global = 10;
    display();
}
```

Output:

```
5
10
```

In the program, the variable “global” is declared at the top of the program outside all of the functions so it is a global variable and can be accessed or updated from anywhere in the program.

What if there exists a local variable with the same name as that of global variable inside a function?

Let us repeat the question once again. The question is : if there is a variable inside a function with the same name as that of a global variable and if the function tries to access the variable with that name, then which variable will be given precedence? Local variable or Global variable? Look at the below program to understand the question:

```
// CPP program to illustrate
// scope of local variables
// and global variables together
#include<iostream>
using namespace std;

// global variable
int global = 5;

// main function
int main()
{
    // local variable with same
    // name as that of global variable
    int global = 2;
    cout << global << endl;
}
```

Look at the above program. The variable "global" declared at the top is global and stores the value 5 whereas that declared within the main function is local and stores a value 2. So, the question is when the value stored in the variable named "global" is printed from the main function then what will be the output? 2 or 5?

- Usually when two variable with the same name are defined then the compiler produces a compile time error. But if the variables are defined in different scopes then the compiler allows it.
- Whenever there is a local variable defined with same name as that of a global variable then the **compiler will give precedence to the local variable**

★ Escape Sequences

Escape sequences are used in the programming languages C and C++, and their design was copied in many other languages such as Java and C#. Escape sequences are the special characters used in control string to modify the format of the output. These characters are not displayed in output. These characters are used with combination of **backslash \\.** This **backslash** \ is called **escape character**.

Table of **Escape sequence** of C & C++ are as follows:

Escape Sequences	**Purpose**
\a	Alarm (Beep, bell)
\t	tab
\n	new line
\f	Form Feed
\r	Carriage Return
\'	Single quote mark
\"	Double quote mark
\?	Question mark
\b	Backspace

★ Constants

Constants refer to fixed values that the program may not alter and they are called **literals**.

Constants can be of any of the basic data types and can be divided into Integer Numerals, Floating-Point Numerals, Characters, Strings and Boolean Values.

Again, constants are treated just like regular variables except that their values cannot be modified after their definition.

Keywords

This is a list of reserved keywords in C++. Since they are used by the language, these keywords are not available for re-definition or overloading.

A – C	D – P	R – Z
alignas (since C++11)	decltype (since C++11)	reflexpr (reflection TS)
alignof (since C++11)	default (1)	register (2)
and	delete (1)	reinterpret_cast
and_eq	do	requires (since C++20)
asm	double	return
atomic_cancel (TM TS)	dynamic_cast	short
atomic_commit (TM TS)	else	signed
atomic_noexcept (TM TS)	enum	sizeof (1)
auto (1)	explicit	static
bitand	export (1) (3)	static_assert (since C++11)
bitor	extern (1)	static_cast
bool	false	struct (1)
break	float	switch
case	for	synchronized (TM TS)
catch	friend	template
char	goto	this
char8_t (since C++20)	if	thread_local (since C++11)
char16_t (since C++11)	inline (1)	throw
char32_t (since C++11)	int	true
class (1)	long	try
compl	mutable (1)	typedef
concept (since C++20)	namespace	typeid

const	new	typename
consteval (since C++20)	noexcept (since C++11)	union
constexpr (since C++11)	not	unsigned
constinit (since C++20)	not_eq	using (1)
const_cast	nullptr (since C++11)	virtual
continue	operator	void
co_await (since C++20)	or	volatile
co_return (since C++20)	or_eq	wchar_t
co_yield (since C++20)	private	while
	protected	xor
	public	xor_eq