Introduction to CPP (C++)

Overview of Introduction to CPP (C++)

C++ is a powerful and versatile programming language that builds upon the foundation of the C programming language. Introduction to CPP (C++) developed in the early 1980s by Bjarne Stroustrup and has since become one of the most widely used programming languages in the world. C++ offers a wide range of features and functionalities, making it suitable for various domains such as system programming, game development, embedded systems, and more.

Advantages of C++ Programming

C++ offers numerous advantages that make it a popular choice among developers:

- **High Performance:** C++ allows for low-level memory manipulation and direct hardware access, enabling developers to write efficient and fast code.
- **Object-Oriented Paradigm:** C++ supports the object-oriented programming (OOP) paradigm, which promotes modular and reusable code, making it easier to manage complex projects.
- **Platform Compatibility:** C++ code can be compiled and executed on different platforms, including Windows, macOS, Linux, and embedded systems.
- **Extensibility:** C++ allows the integration of existing C code and provides features like namespaces and templates, enabling code reuse and extensibility.
- **Vast Community and Libraries:** C++ has a large and active community of developers, offering extensive libraries and frameworks that simplify development and accelerate the coding process.

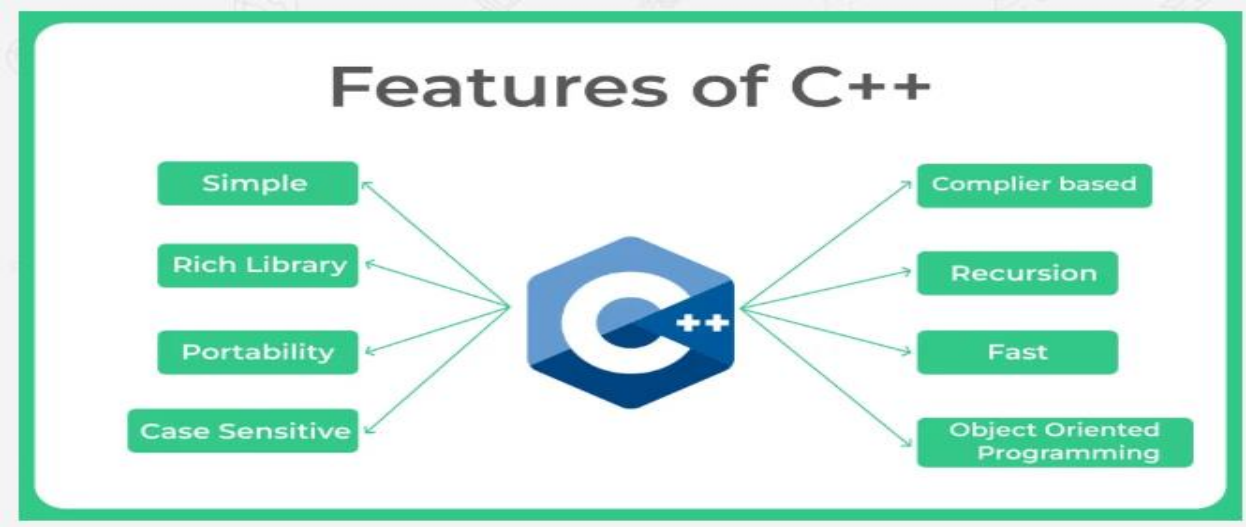
C++ Applications and Use Cases:

- C++ finds applications in various domains, including:
- System programming: Operating systems, device drivers, and firmware development
- Game development: High-performance game engines and graphics programming
- Embedded systems: Microcontrollers, IoT devices, and real-time systems
- Scientific computing: Numerical simulations, data analysis, and modeling
- Financial systems: High-frequency trading, risk analysis, and algorithmic trading

Key Features of C++ Programming Language

C++ incorporates several key features that contribute to its power and flexibility:

- **Strong Typing:** C++ enforces strong typing, ensuring type safety and reducing runtime errors.
- **Memory Management:** C++ provides manual memory management through features like pointers, allowing precise control over memory allocation and deallocation.
- **Operator Overloading:** C++ allows operators to be overloaded, enabling custom behavior for built-in operators.
- **Inheritance and Polymorphism:** C++ supports inheritance, allowing classes to inherit properties and behaviors from other classes. Polymorphism enables objects of different types to be treated uniformly.
- **Templates:** C++ templates facilitate generic programming, allowing algorithms and data structures to be written in a generic way, independent of specific data types.
- **Exception Handling:** C++ provides exception handling mechanisms to gracefully handle runtime errors and exceptional conditions.
- **Standard Library:** C++ includes a comprehensive standard library, offering a wide range of pre-defined functions, data structures, and algorithms.



Object-Oriented Programming in C++

Object-oriented programming (OOP) is a fundamental paradigm in C++. It promotes the organization of code into modular, reusable, and interconnected objects. Key concepts in OOP include:

- **Classes and Objects:** Classes define the blueprint for creating objects, encapsulating data and behaviors. Objects are instances of classes.
- **Encapsulation:** Encapsulation hides the internal implementation details of a class, allowing access only through well-defined interfaces.
- **Inheritance:** Inheritance enables the creation of new classes based on existing ones, inheriting their properties and behaviors.
- **Polymorphism:** Polymorphism allows objects of different classes to be treated as objects of a common base class, facilitating code reusability and flexibility.
- **Abstraction:** Abstraction focuses on representing essential features and hiding unnecessary details to simplify complex systems.

Data Types and Variables

IIntroduction to C++, data types define the kind of data a variable can store. C++ offers various built-in data types, including integers, floating-point numbers, characters, booleans, and more. Variables are named storage locations that hold values of a particular data type. Declaration and initialization of variables are essential concepts in C++ programming.

```
// Example of variable declaration and initialization
int age = 25;
float pi = 3.14;
char grade = 'A';
```

Functions and Libraries

Functions are blocks of reusable code that perform specific tasks. C++ supports both predefined functions from libraries and user-defined functions. Functions provide modularity, reusability, and better code organization.

```
// Example of a user-defined function
int sum(int a, int b) {
  return a + b;
}
```

Pointers and Memory Management Pointers in C++ hold memory addresses as their values. They provide direct access to memory locations and are essential for dynamic memory allocation and

deallocation. Memory management in C++ involves concepts like stack and heap memory, new and delete operators, and smart pointers.

Input and Output StreamsC++ supports input and output operations through input streams (e.g., cin) and output streams (e.g., cout). These streams facilitate interaction between the user and the program, allowing data input and output.

Standard Template Library (STL) The Standard Template Library (STL) is a powerful library in C++ that provides a collection of data structures and algorithms. It includes containers like vectors, lists, and maps, as well as algorithms for sorting, searching, and manipulating data.

Multithreading and ConcurrencyC++ supports multithreading and concurrency, allowing programs to perform multiple tasks concurrently. This feature enhances performance and responsiveness, especially in applications that require parallel execution.

Control Structures

Control structures in C++ allow developers to control the flow of execution based on specific conditions. The three primary control structures are:

- **Conditional Statements:** C++ provides if, else if, and else statements to execute different blocks of code based on logical conditions.
- **Loops:** C++ offers different loop structures like while, do-while, and for loops to repeat a block of code until a certain condition is met.
- **Switch Statements:** Switch statements evaluate an expression and execute a specific block of code based on the expression's value.

Exception Handling

- Exception handling in C++ allows developers to gracefully handle errors and exceptional conditions that may occur during program execution. It involves the use of try, catch, and throw statements.

File Handling

- File handling in C++ enables reading from and writing to files. It involves operations like opening and closing files, reading data from files, writing data to files, and error handling.

Templates and Generic Programming

- C++ templates enable the creation of generic classes and functions that can work with different data types. This feature allows for code reuse and provides flexibility in handling various data structures and algorithms.

Debugging and Optimization Techniques

- C++ offers various tools and techniques for debugging and optimizing code. Debuggers, profilers, and optimization flags help identify and resolve errors, improve performance, and enhance the efficiency of C++ programs.

Best Practices in C++ Programming

To write high-quality C++ code, it is important to follow best practices, including:

- Writing clear and concise code with meaningful variable and function names
- Using comments to document code and improve its readability
- Following naming conventions and coding style guidelines
- Testing code thoroughly and handling errors gracefully
- Considering performance implications and optimizing critical sections
- Adhering to the principles of object-oriented design and modularity

History of C++ Language

The Birth of C++

In the early 1970s, Bjarne Stroustrup, a Danish computer scientist, developed a language called "C with Classes" as an extension to the C programming language. Stroustrup aimed to enhance C's capabilities by adding support for object-oriented programming (OOP) concepts. This new language formed the foundation of what would later become C++.

The Evolution of C++

In 1983, the language was renamed to C++ to reflect the new features and improvements introduced over time. Stroustrup published "The C++ Programming Language," a book that served as a comprehensive guide to the language and gained widespread popularity among programmers. The evolution of C++ continued with the release of various standards, each bringing new features, optimizations, and bug fixes to the language.

Definition of C++

According to Stroustrup C++ is a general-purpose, high-level, compiler-based and objected-oriented Programming Language.

Stroustrup is very interested in the C language because of its powerful features like –

- Multipurpose programming,
- Portability
- Pointers
- Dynamic memory allocation,
- Midlevel flavour

Motivation from Simula67

- The Major disadvantage of the C language is, data is freely moving around the functions.
- Due to this, It is not able to design super secure applications because of the absence of data hiding features
- To avoid this problem the concept of classes is added to traditional C language which was picked from Simula67
- Hence It was named as C with classes initially.

Naming as C++

Initially, people call this language as

- C with classes
- Superset of C
- Extension of C
- Advanced version of C
- Increment of C and many names…

To avoid this confusion in 1983 it was renamed as C++ from C with classes

Object-Oriented Programming

- C++ main objective is to mix the flavors of data hiding with C to design super secure applications
- Object Oriented programming is a programming style that is associated with the concept of Class, Objects and various other concepts revolving around these two, like Inheritance, Polymorphism, Abstraction, Encapsulation etc.
- These ideas made C++ powerful to design both enterprise and system applications

The increment(++) operator in its name: It indicates that C++ is an extension of C language. due to this all the operators, data types, functions can be used in C++ as well

Trivia50 - 70% of windows OS is still written in C++ since its very fast and provides very close interaction with hardware while still being a high level language

Trivia 2 Majority of the games are created and written in C++/C# because of the same reason again that it fast, efficient and provides concept of objects and classes to support real life coding scenarios .

Unity (game engine) and unreal engines which are two most popular platforms to write games are codes in C++/C#

Standardization and Portability

To ensure consistency and compatibility across different platforms, the International Organization for Standardization (ISO) started working on standardizing the C++ language. The first standardized version, known as C++98 or C++03, was released in 1998. Subsequent revisions, such as C++11, C++14, C++17, and the latest C++20, brought numerous improvements, including enhanced language features, libraries, and performance optimizations.

C++'s standardized nature has contributed to its portability, allowing developers to write code that can run on various operating systems and architectures. This characteristic has made C++ a popular choice for building applications ranging from desktop software to embedded systems and game development.

C++ vs. Other Languages

When comparing C++ with other programming languages, several factors come into play. C++ excels in terms of performance, as it allows fine-grained control over system resources. It also provides a low-level programming model that enables developers to optimize code for speed and memory usage.

However, C++ complexity and syntax can be challenging for beginners or developers coming from higher-level languages. Moreover, languages like Python and JavaScript offer faster development cycles and greater simplicity for certain applications.

Structure of C++ Program

```
#include<iostream.h>

using namespace std;

int main()

{

cout << "Hello, World!";

return 0;

}
```

Header Files
In the program above, #include is an example of a header file.

Definition – **Header files** contain definitions of Functions and Variables, which is imported or used into any **C++**program by using the pre-processor #include statement. **Header file** have an extension ".h" which contains **C++** function declaration and macro definition.

Here iostream contains the definitions of input output statements, that are used to print or scan at run time.

NameSpaces
Namespace is a feature added in C++ and not present in C. A namespace is a declarative region that provides a scope to the identifiers (names of the types, function, variables etc) inside it. Multiple namespace blocks with the same name are allowed. All declarations within those blocks are declared in the named scope.

Just remember, name spaces are used to allow re-declaration of variables with same name in a scope.

using namespace std; – enables this.

Main Function

By design, every C++ program must be enclosed in a function int main() by default it is the first function that is called automatically at run time.

Here the function main() is of return type int, thus return an integer value; at the very end causes the program to execute without any error.

Cout function

cout <<“Hello, World!”;

Here, << which is called as insertion operator, it basically inserts the object on its right to its left and cout is predefined function to print value on the screen.

Keywords in C++

Keywords are **predefined reserved identifiers** that have special meanings. They cannot be used as identifiers in your program. The following keywords are reserved for C++. Names with leading underscores.

Keywords in C++ program

asm	else	namespace	template
auto	enum	new	this
bool	explicit	operator	throw
break	export	private	true
case	extern	private	try
catch	false	public	typedef
char	float	register	typeid
class	for	reinterpret_cast	typename
const	friend	return	union
const_cast	goto	short	unsigned
continue	if	signed	using

default	inline	sizeof	virtual
delete	int	static	void
do	long	static-cast	volatile
double	main	struct	wchar_t
dynamic_cast	mutable	switch	while

Identifiers

A **C++ identifier** is a name used to identify a variable, function, class, module, or any other user-defined item. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores, and digits (0 to 9).

1. Identifiers can't start with a digit, only underscore(_) and alphabets with no whitespaces.
2. Identifiers are case sensitive, value and Value are two distinct identifiers.
3. You can't use a keyword as an identifier.

Comments in C++

- Single line Comments – // I will only extent to single line
- Double line comments – /* i will span to various …… different lines*/

Comments are used to increase the understandability enhancing readability of the code. Generally, comments are written to help other coders, who are working on same application and want to understand the code written by someone else.

Data Types in C++

Information stored while writing a code can be in different formats like –

- Number/Integers/Decimals
- Characters
- Boolean
- Characteristic Object etc.

What are data types?

Data types are used by variables to tell what kind of data it can store. Example – character/integer/decimal etc.

There are three types of data types in C++ –

- Primary
- Derived
- User-Defined

Data Types Modifiers

These are used in conjunction with primitive(built-in) data types to modify the length of data that a particular data type can hold these are –

- Unsigned
- Signed
- Short
- Long

Primitive Data Types (Built-in)

- Boolean
- Character
- Integer
- Floating point
- Double floating point
- Void
- Wide character

C++ Code for Sizes

The following code will give you the output of the size of each data type –

```
// C++ program to sizes of data types

#include<iostream>

using namespace std;

 int main()

{

   cout << "Size of char : " << sizeof(char) << " byte" << endl;

   cout << "Size of int : " << sizeof(int) << " bytes" << endl;
```

```
    cout << "Size of short int : " << sizeof(short int) << " bytes" << endl;

    cout << "Size of long int : " << sizeof(long int) << " bytes" << endl;

    cout << "Size of signed long int : " << sizeof(signed long int) << " bytes" << endl;

    cout << "Size of unsigned long int : " << sizeof(unsigned long int) << " bytes" << endl;

    cout << "Size of float : " << sizeof(float) << " bytes" << endl;

    cout << "Size of double : " << sizeof(double) << " bytes" << endl;

    cout << "Size of wchar_t : " << sizeof(wchar_t) << " bytes" << endl;

     return 0;        }
```

output

```
Size of char : 1 byte

Size of int : 4 bytes

Size of short int : 2 bytes

Size of long int : 8 bytes

Size of signed long int : 8 bytes

Size of unsigned long int : 8 bytes

Size of float : 4 bytes

Size of double : 8 bytes

Size of wchar_t : 4 bytes
```

Understanding Each Datatype

We will look at all different data types in C++ :

Integer (int)

- Uses to store integer values like : -200, 150, 6812 etc
- Usual Range – it can store values from -2147483648 to 2147483647
- Usual Size – 4 bytes(some older compilers may support 2 bytes)

```
int age = 25;
```

Float and Double

Float and double are used to store floating-point numbers (decimals and exponentials)

- The size of float is 4 bytes
- The size of the double is 8 bytes.
- Double has two times the precision of float.

Datatype	Range	Macro
float	1.17549e-38 to 3.40282e+38	FLT_MIN
float(negative)	-1.17549e-38 to -3.40282e+38	-FLT_MIN
double	2.22507e-308 to 1.79769e+308	DBL_MIN
double(negative)	-2.22507e-308 to -1.79769e+308	-DBL_MIN

An example of initializing variables would be –

```
float val1 = 21.25;
double val2 = 1531.24595;
double val3 = 21.34E14    // 45E12 is equal to 21.34 * 10^14
double val4 = 1.23E-12    // 45E12 is equal to 1.23 * 10^-12
```

Char

- Its size is 1 byte.
- Characters in C++ are enclosed inside single quotes ' '.

For example –

```
char c = 'a';
```

char val = ‘5’; // 5 stored as ascii character rather than int

Bool

- Has a size of 1 byte
- Used to store true/false values
- Also used in conditional operations

Example –

```
bool val = false;
```

wchar_t

- Wide character wchar_t is similar to the char data type
- However, its size is 2 bytes instead of 1
- These are used to represent characters that need more memory than 1 byte due to a large number of unique characters

Example –

```
wchar_t test = L'א' // storing Hebrew character;
```

Void

- Void is used with functions and pointers
- They mean that they do not have any data type of value
- We will learn more about it later

We cannot declare variables of the void type.

Predefined Datatypes in C++

DefinitionIt is a primary datatype that directly interacts with machine instructions

Datatypes describe what type of data is stored in the variables and describe the amount of memory required to be allocated for the data

1. Integer DataType

We will discuss the following –

1. Default int
2. Unsigned int
3. Long int
4. Unsigned long int
5. Short int
6. Unsigned short int

1. Default int

- **Description**: It can accept a non-decimal value in both negative and positive ranges. Thus, the name integers.
- **Size**: 2 or 4 bytes
- **Range**: *-32768* to *+32768 or -2,147,483,648 to +2,147,483,648*
- **Macros** : (INT_MIN, INT_MAX)

Example:

```
int a = 10;
int b = -3512;
```

Note -Some compilers may support int as 2 bytes and some may support int as 4 bytes.

For 2 bytes range is: (-32,768, +32,767)
For 4 bytes range is : (-2,147,483,648, +2,147,483,648)

Most modern compiler support 4 bytes generally.

2. Unsigned int

- **Description**: It can accept only positive values. A good use case for applications where only positive numbers are required example – Roll No., ticket system.
- **Size**: 2 or 4 bytes
- **Range**: *0* to *+65,535 or 0 to +4,294,967,295*
- **Macros** : (USHRT_MAX)

Example:

```
unsigned int a = 10;
unsigned int b = 10987;
```

Note -Some compilers may support int as 2 bytes and some may support int as 4 bytes.

For 2 bytes range is: (0, +65,535)
For 4 bytes range is : (0, +4,294,967,295)

Most modern compiler support 4 bytes generally.

3. long int

- **Description**: Ideal for very large integer values in positive/negative ranges.
- **Size**: 4 or 8 bytes
- **Range**: *-2,147,483,648* to *+2,147,483,647* *or* *-9,223,372,036,854,775,808* *to* *+9,223,372,036,854,775,807*
- **Macros** : (LONG_MIN, LONG_MAX)

Example:

```
long int a = 10172817;
long int b = -9876543210123;
```

Note -Some compilers may support int as 4 bytes and some may support int 8 bytes.

For 2 bytes range is: (-2,147,483,648 to +2,147,483,647)
For 4 bytes range is : (-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807)

Most modern compiler support 8 bytes generally.
However, we can force 8 bytes guarantee if we use long long int instead of just long int

4. Unsigned long

- **Description**: Ideal for very large integer values only in positive ranges.
- **Size**: 4 or 8 bytes
- **Range**: (0 to 4,294,967,295) *or (0 to 18,446,744,073,709,551,615)*
- **Macros** : (ULONG_MAX)

Example:

```
long int a = 3120172817;
long int b = -9876512323443210123;
```

Note -Some compilers may support int as 4 bytes and some may support int 8 bytes.

For 2 bytes range is: (0 to 4,294,967,295)
For 4 bytes range is : (0 to 18,446,744,073,709,551,615)

Most modern compiler support 8 bytes generally.
However, we can force 8 bytes guarantee if we use unsigned long long int instead of just unsigned long int

5. Short

- **Description**: Ideal for very small integer values in positive/negative ranges.
- **Size**: 2 bytes guaranteed
- **Range**: (-32,768 to 32,767)
- **Macros** : (SHRT_MIN, SHRT_MAX)

Example:

```
long int a = 12;
long int b = -1087;
```

6. Unsigned short

- **Description**: Ideal for very small integer values only in positive ranges.
- **Size**: 2 bytes guaranteed
- **Range**: (0 to 65,535)
- **Macros** : (USHRT_MAX)

Example:

```
long int a = 1027;
long int b = 10;
```

2. Char DataType

We will discuss the following in this –

1. Default char
2. Unsigned char

1. Default char

- **Description**: Every key in the keyboard acts as a char with an ASCII value. Characters are declared in the single quotation to distinguish them from variable names and character values
- **Size**: 1 byte (8 bits)
- **Range**: -128 to 127
- **Macros:** CHAR_MAX and CHAR_MIN

Example

```
char a='@';
char b='c';
```

2. Unsigned char

- **Description**: It accepts more number of characters than normal char type
- **Size**: 1 bytes(8 bits)
- **Range** : 0 to 255
- **Macro**: UCHAR_MAX

Example

```
unsigned char a='ß'//beta charecter has ASCII value 223
```

3. float DataType

- **Description**: It accepts fractional numbers up to ***6 decimal place precision***
- **Size**: 4 bytes(32bits)
- **Macros :** FLT_MAX and FLT_MIN

Range :

- In positive range : +1.17549e-38 to +3.40282e+38
- In negative range : -1.17549e-38 to -3.40282e+38

Note: 1.17549e-38 means 1.17549*10^(-38) which is very very small close to zero

Example

```
float a = 56.771;
float b = 6.5;
float c = 1.23e10; // 1.23 * 10^10
float d = 1.98e-12;// 1.98 * 10^(-12)
```

4. Double DataType

- **Description**: It accepts fractional numbers up to 12 decimal places mainly used for scientific computations
- **Size**: 8 bytes(64 bits)
- **Macros:** DBL_MAX and DBL_MIN

Range :

- In positive range : +2.22507e-308 to +1.79769e+308
- In negative range : -2.22507e-308 to -1.79769e+308

Note: 2.22507e-308 means 2.22507*10^(-308) which is very very small (close to zero).

Example

```
// same numbers as float
// but below have higher precision and higher range
double a = 56.771;
double b = 6.5;
double c = 1.23e10; // 1.23 * 10^10
```

```
double d = 1.98e-12;// 1.98 * 10^(-12)
```

5. Long Double DataType

- **Description**: It accepts fractional numbers up to 19 decimal places mainly used for scientific computations
- **Size**: 16 bytes(128 bits)
- **Macros:** LDBL_MAX and LDBL_MIN

Range :

- In positive range : +3.3621e-4932 to +1.18973e+4932
- In negative range : -3.3621e-4932 to -1.18973e+4932

Note: 3.3621e-4932 means 3.3621e *10^(4932) which is very very small (close to zero).

Example

```
// same numbers as float/double
// but below have higher precision and higher range
long double a = 56.771;
long double b = 6.5;
long double c = 1.23e10; // 1.23 * 10^10
long double d = 1.98e-12;// 1.98 * 10^(-12)
```

6. wchar_t DataType

- **Description**: Wide char can take on ***65536 values which correspond to UNICODE values*** which is a recent international standard that allows for the encoding of characters for virtually all languages and commonly used symbols.
- **Size**: 16 bytes(128 bits)
- **Range** :0 to 65,535 or 4294967296

Mostly the wchar_t data type is used when international languages like Chinese, French are used.

Example

```
wchar_t b = L'艾儿';//Chinese alphabet
```

7. bool DataType

- The ISO/ANSI C++ Standard has added certain new data types to the original C++ specifications.
- They are provided to provide better control in certain situations as well as for providing conveniences to C++ programmers.

Syntax:

```
bool b = true; // declaring a boolean variable with true value
```

- In C++, the data type bool has been introduced to **hold a boolean value, true or false**.
- The values true or false have been added as keywords in the C++ language.

Note:

- The default numeric value of true is 1 and false is 0.
- We can use bool type variables or values true and false in mathematical expressions also.

Example

```
int x = false + true + 5;//6
```

- The above expression is valid and the expression on the right will evaluate to 6 as false has value 0 and true will have value 1.
- It is also possible to convert implicitly the data type integers or floating-point values to bool type.

Example:-

```
bool x = 0; // false
bool y = 100; // true
bool z = 18.75; // true
```

C++ program demonstrating predefined datatypes

```
#include<iostream>

using namespace std;



int main()

{

   int a = 567; //valid
```

```
    int b = 2147483800; //invalid : will result in loss of precison max int value 2147483647

    long long int c = 2147483800; // valid because of enough size max long int value 9,223,372,036,854,775,807

    char d ='_'; //underscorecharecter

    double e = 66.91;

    bool f= false;

    bool g= true;



    cout << "a:" << a << endl;

    cout << "b:" << b << endl; // incorrect value will be printed due to overflow

    cout << "c:" << c << endl;

    cout << "d:" << d << endl;

    cout << "e:" << e << endl;

    cout << "f:" << f << endl;

    cout << "g:" << g << endl;



    return 0;

}
```

Output

```
a:567

b:-2147483496
```

c:2147483800

d:_

e:66.91

f:0

g:1

Important Points

1. Rage of datatypes always in between -2^{n-1} to $2^{n-1}-1$ and 0 to 2^n -1 for unsigned types, where ***n*** is the number of bits

Example:

- int: -2^{15} to $2^{15}-1$ i.e -128 to 127
- unsigned int: 0 to $2^{16}-1$ i.e 0 to 65535

2. -ve numbers are always stored in,2's compliment form hence you can allocate with +1 in the maximum negative range

Example:
-32768,-128 etc

Modifiers in C++

Some modifiers are –

- long
- signed
- unsigned
- short

Modifiers Details
These modifiers also modify the range and size of of the data types that they are preceding. The below list gives you an idea on the same – [table id=551 /] The sizes of the variables might be different for various compilers, what we have given are general sizes that are useful for various competitive examinations.

- int – can be preceded with long, signed, unsigned, short
- char – can be preceded with signed and unsigned

The following code will print the sizes of various data types and their modifiers –

```
#include <iostream>

using namespace std;


int main() {

   cout << "Size of char : " << sizeof(char) << endl;

   cout << "Size of int : " << sizeof(int) << endl;

   cout << "Size of short int : " << sizeof(short int) << endl;

   cout << "Size of long int : " << sizeof(long int) << endl;

   cout << "Size of long int : " << sizeof(unsigned long int) << endl;

   cout << "Size of float : " << sizeof(float) << endl;

   cout << "Size of double : " << sizeof(double) << endl;

   cout << "Size of wchar_t : " << sizeof(wchar_t) << endl;


   return 0;

}
```

output –

```
Size of char : 1

Size of int : 4

Size of short int : 2
```

Size of long int : 8

Size of long int : 8

Size of float : 4

Size of double : 8

Size of wchar_t : 4

User Defined Data Types in C++

User defined datatypes in C++ are :

- Structure
- Union
- Enumeration

Structure

- An array can store elements of one data type at a time **Example:** int a[5] can hold only integer elements
- Definition of Structure: A structure is a collection of **multiple variables of different data types grouped** under a single name for convenience handling
- The members of structures can be ordinary variables, pointers, arrays or even another structure

Declaring Structures:

Using struct keyword structures are created

```
struct
{
  member 1;
  member 2;
  ....;
  member n;
};
```

Creating a structure to store student record

```
struct student  // student as structure name

{

  //structure members

  int sid;

  char sname[20];

  float marks, attendance;

};
```

Allocating Memory to structure

Declaration of structure will allocate no memory, memory is allocated by creating structure variable

```
struct student s1;
```

C++ program to demonstrate declaration, initialization, and access to structure types
The member elements of the structure can be accessed using 2 operators

1. Member access operator(.)
2. Pointer to member access operator(->)

```
#include <iostream>

#include <string.h>

using namespace std;



struct student  //student as structure name

{

  int id;
```

```
  char name[20];
  float marks,attendance;
};//no memory allocated as of now

int main()
{

  struct student s1 = {66,"Ronaldo",75.0,50.0};  //memory allocated
  struct student s2;  //memory allocated now
  struct student s3;  //memory allocated now

  //Method 2 of initialisation
  cout << "Enter Student ID for S2:" << endl;
  cin >> s2.id;

  cout << "\nEnter Student Name for S2:" << endl;
  cin >> s2.name;

  cout << "\nEnter Marks for S2:" << endl;
  cin >> s2.marks;
```

```
  cout << "\nEnter attendance for S2:" << endl;
  cin >> s2.attendance;

  //Method 3 of initilisation
  s3.id = 36;
  s3.marks = 79;
  strcpy(s3.name, "Neymar");
  s3.attendance = 200;

  //acessing structure members
  cout << "\n\nDetails of Student 1:\n";

  cout << "Name: " << s1.name << "\nID: "
  << s1.id << "\nMarks: "<< s1.marks << "\nAttendance: " << s1.attendance;

  return 0;
}
```

Output

```
Enter Student ID for S2:
100
Enter Student Name for S2:
```

Atul

Enter Marks for S2:

90

Enter attendance for S2:

100

Details of Student 1:

Name: Ronaldo

ID: 66

Marks: 75

Attendance: 50

Array of Structures

- We can also make an array of structures. In the first example in structures, we stored the data of 3 students. Now suppose we need to store the data of 100 such students, Declaring 100 separate variables of the structure is definitely not a good option. For that, we need to create an array of structures.
- struct student **s1, s2, s3… s100;** can be written **struct s[100];**

An array of structure that stores records of 5 students

```
#include <iostream>

using namespace std;

struct student

{
```

```
  int roll_no;
  string name;
  int phone_number;
};

int main()
{
  struct student stud[5];  //array of structure variables
  int i;

  for(i = 0; i < 5; i++)
  {
    //taking values from user
    cout << "Student " << i + 1 << endl;

    cout << "Enter roll no: " << endl; cin >> stud[i].roll_no;

    cout << "Enter name: " << endl; cin >> stud[i].name;

    cout << "Enter phone number: " << endl; cin >> stud[i].phone_number;
  }
```

```
  for(i = 0; i < 5; i++)
  {
    //printing values
    cout << "\nStudent " << i + 1 << endl;

    //accessing struture members
    cout << "\nRoll no : " << stud[i].roll_no << endl;
    cout << "Name : " << stud[i].name << endl;
    cout << "Phone no : " << stud[i].phone_number << endl;
  }
  return 0;
}
```

Output

Printing the first 2 only

Student 1

Enter roll no: 1

Enter name: John

Enter phone number: 890780109

Student 2

Enter roll no: 2

Enter name: Kallis

Enter phone number: 780182912

Student 1

Roll no : 1

Name : John

Phone no : 890780109

Local and global structure

Defining a structure inside the main can be accessible only inside the main but not outs whereas as structures defined outside can be accessed anywhere in the program

```
#include <iostream>

using namespace std;



int main() {

    struct emp

    {

        int id;
```

```
        char ename[20];
        int sal;
    };

    emp e1, e2;
}

//outside main declared in a function
void abc()
{
    //error:elements are not accesible outside main
    emp e3,e4;
}
```

Structure within Structure:

One Structure variable acting as a member element of another structure is called a structure to structure

```
#include <iostream>
struct X
{
  int a;
};
```

```
struct Y
{
  int b;

  struct X x; //structure variable x as a member of struct Y
};
int main()
{
  struct Y y;

  x.y.b = 10; y.x.a = 20 //indirect initilaisation by through
}
```

Copying Structures:

We can ***copy one structure into another by assignment of structure variables***

```
#include<iostream>
using namespace std;
struct student{
  long long int roll_no;
  string name;
  int phone_number;
};
```

```
int main()

{

  struct student p1 = {160815733066,"trishaank",12345};

  struct student p2;

  p2 = p1;//copying structures



  cout << "roll no : " << p2.roll_no << endl;  //accessing

  cout << "name : " << p2.name << endl;

  cout << "phone number : " << p2.phone_number << endl;

}
```

Note-

- **struct student *ptr;** - We declared 'ptr' as a pointer to the structure student.
- **ptr = &stud;** - We made our pointer ptr to point to the structure variable stud.
- Thus, 'ptr' now stores the address of the structure variable 'stud'.This is the same which we do while defining a pointer to any other variable.

Passing Structure to function

There are two methods by which we can pass structures to functions.

Passing by Value in structures

In this, we pass a structure variable as an argument to a function.

```
#include<iostream>

using namespace std;

struct student

{
```

```
    int roll_no;
    string name;
    int phone_number;
};

void display(struct student st)  //structure variable in function
{
    cout << "Roll no : " << st.roll_no << endl;
    cout << "Name : " << st.name << endl;
    cout << "Phone no : " << st.phone_number << endl;
}

int main()
{
    struct student s;  //memory allocated

    s.roll_no = 66;
    s.name = "Jim Halpert";
    s.phone_number = 756888;

    display(s);  // function call with structure variable
```

```
    return 0;
}
```

Passing by Reference in structures

- In passing by reference, the address of a structure variable is passed to a function.
- In this, **if we change the structure variable which is inside the function, the original structure variable which is used for calling the function changes**. This was not the case in calling by value.

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

struct student
{
    int roll_no;
    string name;
    int phone_number;
};

void display(struct student *st)  //structure pointer in function definition
{
    cout << "Roll no : " << st -> roll_no << endl;
    cout << "Name : " << st -> name << endl;
```

```
    cout << "Phone no : " << st -> phone_number << endl;
}

int main()
{
    struct student s;

    s.roll_no = 66;
    s.name = "Ron";
    s.phone_number = 12345;

    display(&s);  //passing address of struct variable in function call

    return 0;
}
```

Output

```
Roll no : 66
Name : 3sh
Phone no : 12345
```

- This case is similar to the previous one, the only difference is that this time, we are passing the address of the structure variable to the function.

- While declaring the function, we passed the pointer of the copy ‘st’ of the structure variable ‘s’ in its parameter.
- Try to change the value of the variables inside the function in both the cases and see the changes in the actual and formal parameters

Union

- **Union definition**: A union is Collection of different types where ***each element can be accessed one at a time***
- Usage of the union is exactly the same as Structure only difference is the size of the structure is the sum of sizes of types of all members whereas memory allocated for the union is the size of the member which is having a larger size than all other members
- Before Introducing, structures Unions were introduced where is a severe concern for memory but nowadays memory is cheaper and unions are no more used.
- It is based on the common memory mechanism: A single memory space is shared by all members of the union one at a time

```
#include<iostream>

#include <bits/stdc++.h>

using namespace std;

union demo //declaring union

{

  int a;

  float b;

}d;

// 4 bytes allocated
```

```
int main()
{
 //Whole union memory i.e. 4 bytes allocated to a
 d.a = 12;
 cout << "a: " << d.a << endl; //prints 12
 cout << "b: " << d.b << endl; //will print garbage

 //Union memory deallocated from int a & allocated to float b
 d.b = 14.5;
 cout << "a: " << d.a << endl; //will print garbage
 cout << "b: " << d.b << endl; //prints 14.5
 cout << "\nMemory allocated for this union is "<< sizeof(d) << " bytes" << endl;
 return 0;
}
```

Output

```
a: 12
b: 1.68156e-44
a: 1097334784 b: 14.5 Memory allocated for this union is 4 bytes
```

- In the above example size of the union is 4 bytes because among int a and float b, float is larger than int hence .hence size of the float will be the size of the union here
- As there is only ***single space,*** initialization of a member with value makes all members initialize with that value and any changes one member reflect the same change in all the union members
- If d.a=14 then the value of b also become s14

Enumeration

- Definition: consists of various constants that are just grouped under a single name just to make our codes neat and more readable.
- Example: So this becomes an enumeration with name season and Summer, Spring, Winter and Autumn as its elements.
- enum is created by keyword andenum the elements separated by ‘comma’ as follows.

```
enum enum_name
{
  element1,
  element2,
  .....
  elementn
};
```

Creating enum of type seasons

```
enum Season{
  //creating enum
  Summer,
```

```
  Spring,
  Winter,
  Autumn
};
```

Here, we have defined an enum with name ‘Season’ and Summer, Spring, Winter and Autumn’ as its elements.

Values of the Members of Enum

All the elements of an enum have a value.

By default, the value of the first element is 0, that of the second element is 1 and so on.

size of enum is the no of elements present in it

Example

```
enum Season{ Summer, Spring, Winter, Autumn}; //creating an enum
int main()
{
  enum Season s;  // s is a variable with datatype season
  s = Spring;  //initialising enum members
  cout << s << endl;
  cout << "\nsize of enum" << sizeof(s); //4bytes
}
```

Output

1 size of enum 4 bytes

Here, first, we defined an enum named ‘Season’ and declared its variable ‘s’in the main function as we have seen before.

The values of Summer, Spring, Winter, and Autumn are 0, 1, 2 and 3 respectively.

So, by writing s = Spring, we assigned a value ‘1’ to the variable ‘s’ since the value of ‘Spring’ is 1.

Directly print the value of any element of an enum.

```
enum Season{ Summer, Spring, Winter, Autumn};  //creating a type season

int main()

{

  cout << Winter;  //position ordering is printed

}
```

Output

2

Customizing the ordering of enum values

Once we change the default value of an enum element, then the values of all the elements after it will also be changed accordingly. An example will make this point clearer.

```
//customising default values of enum members

enum Days{ sun, mon, tue = 5, wed, thurs, fri, sat};
```

```
int main()

{

  enum Days day; // s is a varible with datatype season


  day = thurs;//initialising enum members


  cout << day <<endl;

  cout << day+2 << endl;//In this example, the value of thurs i.e.


  //Since we are printing 'day+2' i.e. 7 (=5+2), so the output will be 9

return 0;


}
```

Output

```
7

9
```

The default value of sun will be 0, Mon will be 1, tue will be 2 and so on.

In the above example, we defined the value of tue as 5. So the values of wed, thurs, fri andsat will become 6, 7, 8 and 9 respectively.

There will be no effect on the values of sun and mon which will remain 0 and 1 respectively. Thus the value of thurs i.e. 9 will get printed.

Enum class(Exclusively in C++)

On comparison of normal enum values Positions are compared and the output is true

ex: if we compare the first element of the enum (say spring) and the first element of another enum (say blue) .it always returns true because positions are compared .we can overcome this by enum class

unlike enum, enum class compares the values rather than positions

To make an enum class, we simply have to add the keyword class after the keyword enum.

```
enum class Season

{

  Summer, Spring, Winter, Autumn

};
```

We made our enum Season an enum class by adding the keyword class after the keyword enum. accessing any element (enumerator) of enum class.

```
Season::Summer
```

The :: symbol simply means belong to. Here, Summer belongs to the enum class Season and thus cannot be directly accessed anywhere. We can compare the enumerators which belong to the same enum class only

Run

```
#include<iostream>

using namespace std;


class names{
```

```
  trish,
  rishi
};

int main()
{
  names n = names::trish;
  if( n == names::trish )
    cout << "Your name is trish" <<endl;
  else
    cout << "Your name is rishi" << endl;

  return 0;
}
```

Output

```
Your name is trish
```

In this example, we compared the enumerators within the enum class ‘names’.

Note that enumerators are not converted to integers in the case of enum class.

By writing names n = names::trish, we are assigning the colour Blue of enum class names to the variable n. So the condition (n == names::trish) became true and “Your name is trish” got printed.

In this condition, we compared n and names::trish, both of which belong to the enum class names.

Difference between C and C++ structures

Direct Initialization: We cannot directly initialize structure data members in C but we can do it in C++.

```
struct Record

{

  int x = 3; //this type of initialisation not possible in C

};
```

Member functions inside the structure: Structures in C cannot have member functions inside the structure but Structures in C++ can have member functions along with data members.

```
struct demo

{

  int a;


  void display() //function inside structure

  {

    cout<<"My name is Trishaank"<< endl;

  }

}T;
```

```
int main()

{

  T.get_data();


  return 0;

}
```

Using struct keyword:

In C, we need to use the struct to declare a struct variable. In C++, a struct is optional in the variable declaration.

```
struct emp e;

or

emp e  //valid and same as above in C++
```

C structures cannot have static members, constructors, data hiding, constructors but is allowed in C++.

```
struct sample

{

  static int a; //static memer inside structure

  int roll;
```

```
  Student(int x) //constructor inside structure
  {
    roll = x;
  }
};

int main()
{
  struct Student s(3);
  cout << s.x;
}
```

sizeof operator:

This operator will assign 0 for an empty structure in C whereas 1 for an empty structure in C++.

```
// empty structure
struct Record
{
  //empty
}r;

int main()
```

```
{
  cout<<sizeof(r); // 1 in C++ and 0 in C
}
```

Difference between structure and union

Structure

Members do not share memory in a structure.

Any member can be retrieved at any time in a structure.

Several members of a structure can beinitialized

Size of the structure is equal to the sum of the size of each member.

Altering the value of one member will not affect the value of another.

Stores different values for all the members.

Union

Members share the memory space in a union.

Only one member can be accessed at a time in a union.

Only the first member can be initialized

Size of the union is equal to the size of the largest member.

Change in value of one member will affect other member values.

Stores same value for all the members.

Namespaces in C++

Namespaces in C++

To access an identifier in a namespace, you use the scope resolution operator. For example, to call the print_x function in the my_namespace namespace, you would use my_namespace::print_x();

You can also use using namespace directive to make all the identifiers in a namespace available without the need to qualify them with the namespace name.

```
using namespace my_namespace;

print_x();
```

For example

```
namespace my_namespace {
   int x = 5;
   void print_x() {
      cout << x << endl;
   }
}
```

Namespace in C++

```
// A C++ program to demonstrate use of class
// in a namespace
#include <iostream>
using namespace std;

namespace ns
```

```
{
	// A Class in a namespace
	class prepster
	{
	public:
		void display()
		{
			cout<<"ns::prepster::display()"<<endl;;
		}
	};
}

int main()
{
	// Creating Object of prepster Class
	ns::prepster obj;

	obj.display();

	return 0;
}
```

Type Qualifiers in C++

Definition

C++ provides us with the feature to add extra quality to our variables by specifying volatility or constantness to them. These extra qualities are added with the help of tokens known as type qualifiers.

In general words, we can say those type qualifiers are the tokens that add special qualities to the variables. These qualities specify where the variables are to be stored in a memory and how they can be accessed from there.

C++ programming language provides us three types of type qualifiers:-

1. const

2. volatile

3. mutable

What is const type qualifier in C++?

If the type of object is cont then this type of object cannot be modified

If we try to modify the const type of object then there will be a compile-time error.

An object whose type is const type qualifier is a const object.

Syntax:

```
const data_type variable = value;
```

Example

```
#include <iostream>

using namespace std;
```

```
int main()

{

    const int var = 10;

    cout << var;



    var = 20; // error: Const values can't be changed



    return 0;

}
```

What is a volatile type qualifier in C++?

Saying that a variable is volatile would mean for us to tell the compiler to not optimize the code as the value for something may change.

To understand this we will need to take the below scenario –

```
int var = 123;



while(var == 123)

{

   //your code
```

}

Forced Optimization by compiler

When this program gets compiled, the compiler may optimize this code, if it finds that the program never ever makes any attempt to change the value of var So it may be tempted to optimize the while loop by changing it from while(var == 123) to something which is equivalent to while(true) This may be done so that the execution could be fast (since the condition in while loop appears to be true always). (if the compiler doesn't optimize it, then it has to fetch the value of var and compare it with 123, in each iteration which obviously is a little bit slow.)

Sometimes Optimization is bad

However, sometimes, optimization (of some parts of your program) may be undesirable, because it may be that someone else is changing the value of some_int from outside the program which compiler is not aware of, since it can't see it; but it's how you've designed it. In that case, compiler's optimization would not produce the desired result! So, to ensure the desired result, you need to somehow stop the compiler from optimizing the while loop. That is where the volatile keyword plays its role. All you need to do is this,

volatile int var = 123; //note the 'volatile' qualifier now!

We can do below that is just add a volatile keyword and the compiler will not optimize the code and knows the code/variables etc maybe sensitive that is volatile

volatile int var = 123;

while(var == 123)

{

  //your code

}

What is a mutable type qualifier in C++?

Const objects do not allow data member values to be changed. They can only be read after initialization and not updated again. Data members which are mutable are allowed to change their values even if they are of a const object. This may be needed when almost all other data members are required to remain constant but one may change. It’s a good idea to declare that data member as mutable. The following example demonstrates the same –

```
#include<iostream>

using namespace std;


class MyClass {

   public:

      int var1;

      // mutable will allow var2 to be updated

      // even though its called by a const object

      mutable int var2;


   MyClass(int x=0, int y=0) {

      var1 = x;

      var2 = y;

  }

  void display()

  {

      cout << endl << "var1: " << var1 << " var2: "<< var2 << endl;
```

```
    }
};
int main()
{
    // const object, we can only read but not change values
    // values can only be initialized once, here we using a
    // parameterized constructor to initialize them
    const MyClass obj(10,20);

    // reading values is allowed as we are not changing
    // data members of const object
    cout << obj.var1 << " " << obj.var2 << "\n";

    // Below gives error as var1 can't be changed( because object is constant)
    // obj.var1 = 30;

    // var2 can be changed as var2 is mutable data member
    obj.var2 = 100;

    cout << obj.var1 << " " << obj.var2 << "\n";
```

```
    return 0;
}
```

Output

```
10 20
10 100
```

Typedef in C++

C++ typedef Syntax

```
typedef built_in_datatype user_defined_name;
```

Here existing name is the C++ datatype and user_defined_name is the alternate name given to existing_type

C++ program to demonstrate typedef

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

int main()
{
    typedef int trish;

    trish a = 1; //a is int type internally
    trish b = 2; //b is int type internally
```

```
    cout << "a: " << a << "\nb: " << b;


    return 0;
}
```

Output:

a: 1

b: 2

Using typedef for struct types

Here we have used the student as an alias name for the struct name stuAlias, so where ever you create a variable with type student internally structure is created and that variable will have size same structure

```
#include<iostream.h>

using namespace std;


// typedef struct created here

typedef struct stuAlias

{

    string name;

    int id;
```

```
    string klg;
}student;

int main()
{

    student s1,s2;  //2 variables of type student

    s1.id = 66;
    s1.name = "trishaank";
    s1.klg = "MECS";

    s2.id = 27;
    s2.klg = "KMIT";
    s2.name = "Rishi";

    cout << "Trishaank Details:\n";
    cout << s1.name << " "<< s1.id << " "<< s1.klg << endl;

    cout << "\nRishi Details:\n";
    cout << s2.name << " " << s2.id <<" "<< s2.klg;
```

```
    return 0;
}
```

Output

```
Trishaank Details:

trishaank 66 MECS



Rishi Details:

Rishi 27 KMIT
```

Nested typedef

Once you create an alias name for a datatype using a typedef, you can further be given another alias name with the previous alias name but finally, Compiler Replaces the with root alias

Here trish is replaced with int but whenever compiler encounters the name rish it initially replaces with trish then replaces with int

Similarly, when vicky is encountered it is replaced with rish initially, as no datatype, is found with that name it replaces with trish and finally, trish is replaced with int

```
#include<iostream>

using namespace std;


int main()

{
```

```
    typedef int trish;
    typedef trish rish;
    typedef rish vicky;

    trish a = 1;
    rish b = 2;
    vicky c = 3;

    //trish, rish, vicky treated as int internally
    cout << "a:" << a << "\t" << "b:" << b << "\t" << "c:" << c;

    return 0;
}
```

Output

```
a:1
b:2
c:3
```

Note:

You can freely use the original name even if you have an alias name within the program

```
#include <iostream>
{
```

```
    typedef int trish;

    trish a = 1;  // a is int type internally

    int b = 2;  // you can use original datatype name

}
```