

CPP Basics

Thursday, March 31, 2022 6:29 PM

- C++ is a middle-level language, as it encapsulates both high and low level language features.

Object-Oriented Programming (OOPs)

C++ supports the object-oriented programming, the four major pillar of object-oriented programming (OOPs) used in C++ are:

1. Inheritance
2. Polymorphism
3. Encapsulation
4. Abstraction

C++ Program

In this tutorial, all C++ programs are given with C++ compiler so that you can easily change the C++ program code.

File: main.cpp

```
#include <iostream>
using namespace std;
int main() {
    cout << "Hello C++ Programming";
    return 0;
}
```

#include<iostream.h> includes the **standard input output** library functions. It provides **cin** and **cout** methods for reading from input and writing to output respectively.

#include <conio.h> includes the **console input output** library functions. The **getch()** function is defined in **conio.h** file.

void main() The **main()** function is the entry point of every program in C++ language. The **void** keyword specifies that it returns no value.

cout << "Welcome to C++ Programming." is used to print the data **"Welcome to C++ Programming."** on the console.

- Stream is the sequence of bytes or flow of data. It makes the performance fast.

Header File	Function and Description
<iostream>	It is used to define the cout , cin and cerr objects, which correspond to standard output stream, standard input stream and standard error stream, respectively.
<iomanip>	It is used to declare services useful for performing formatted I/O, such as setprecision and setw .
<fstream>	It is used to declare services for user-controlled file processing.

```
#include <iostream>
using namespace std;
int main() {
    int age;
    cout << "Enter your age: ";
    cin >> age;
    cout << "Your age is: " << age << endl;
}
```

Output:

```
Enter your age: 22
Your age is: 22
```

- The endl is a predefined object of ostream class. It is used to insert a new line characters and flushes the stream.

```
#include <iostream>
using namespace std;
int main() {
    cout << "C++ Tutorial";
    cout << " Javatpoint" << endl;
    cout << "End of line" << endl;
}
```

Output:

```
C++ Tutorial Javatpoint
End of line
```

- 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.

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

Rules for defining variables

A variable can have alphabets, digits and underscore.

A variable name can start with alphabet and underscore only. It can't start with digit.

No white space is allowed within variable name.

A variable name must not be any reserved word or keyword e.g. char, float etc.

Valid variable names:

```
int a;
int _ab;
int a30;
```

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 Datatypes:

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

A keyword is a reserved word. You cannot use it as a variable name, constant name etc. **A list of 32 Keywords in C++ Language which are also available in C language are given below.**

auto	break	case	char	const	continue	default	do
double	else	enum	extern	float	for	goto	if
int	long	register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void	volatile	while

A list of 30 Keywords in C++ Language which are not available in C language are given below.

asm	dynamic_cast	namespace	reinterpret_cast	bool
explicit	new	static_cast	false	catch
operator	template	friend	private	class
this	inline	public	throw	const_cast
delete	mutable	protected	true	try
typeid	typename	using	virtual	wchar_t

	Operator	Type
Binary Operator	+, -, *, /, %	Arithmetic Operators
	<, <=, >, >=, ==, !=	Relational Operators
	&&, , !	Logical Operators
	&, , <<, >>, ~, ^	Bitwise Operators
	=, +=, -=, *=, /=, %=	Assignment Operators
Unary Operator	→ ++, --	Unary Operator
Ternary Operator	→ ?:	Ternary or Conditional Operator

Category	Operator	Associativity
Postfix	0 [] -> . ++ --	Left to right
Unary	+ - ! ~ ++ -- (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Right to left
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Right to left
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR		Right to left
Logical AND	&&	Left to right
Logical OR		Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= =	Right to left
Comma	,	Left to right

- C++ identifiers in a program are used to refer to the name of the variables, functions, arrays, or other user-defined data types created by the programmer.

In short, we can say that the C++ identifiers represent the essential elements in a program which are given below:

In short, we can say that the C++ identifiers represent the essential elements in a program which are given below:

- **Constants**
- **Variables**
- **Functions**
- **Labels**
- **Defined data types**

Some naming rules are common in both C and C++. They are as follows:

- Only alphabetic characters, digits, and underscores are allowed.
- The identifier name cannot start with a digit, i.e., the first letter should be alphabetical. After the first letter, we can use letters, digits, or underscores.
- In C++, uppercase and lowercase letters are distinct. Therefore, we can say that C++ identifiers are case-sensitive.
- A declared keyword cannot be used as a variable name.
- The major difference between C and C++ is the limit on the length of the name of the variable. ANSI C considers only the first 32 characters in a name while ANSI C++ imposes no limit on the length of the name.
- Keywords are the reserved words that have a special meaning to the compiler. They are reserved for a special purpose, which cannot be used as the identifiers. For example, 'for', 'break', 'while', 'if', 'else', etc. are the predefined words where predefined words are those words whose meaning is already known by the compiler. Whereas, the identifiers are the names which are defined by the programmer to the program elements such as variables, functions, arrays, objects, classes.

Identifiers	Keywords
Identifiers are the names defined by the programmer to the basic elements of a program.	Keywords are the reserved words whose meaning is known by the compiler.
It is used to identify the name of the variable.	It is used to specify the type of entity.
It can consist of letters, digits, and underscore.	It contains only letters.
It can use both lowercase and uppercase letters.	It uses only lowercase letters.
No special character can be used except the underscore.	It cannot contain any special character.
The starting letter of identifiers can be lowercase, uppercase or underscore.	It can be started only with the lowercase letter.
It can be classified as internal and external identifiers.	It cannot be further classified.
Examples are test, result, sum, power, etc.	Examples are 'for', 'if', 'else', 'break', etc.

If Else if and else example

```

#include <iostream>
using namespace std;
int main () {
    int num;
    cout<<"Enter a number to check grade:";
    cin>>num;
    if (num <0 || num >100)
    {
        cout<<"wrong number";
    }
    else if(num >= 0 && num < 50){
        cout<<"Fail";
    }
    else if (num >= 50 && num < 60)
    {
        cout<<"D Grade";
    }
    else if (num >= 60 && num < 70)
    {
        cout<<"C Grade";
    }
    else if (num >= 70 && num < 80)
    {
        cout<<"B Grade";
    }
    else if (num >= 80 && num < 90)
    {
        cout<<"A Grade";
    }
    else if (num >= 90 && num <= 100)
    {
        cout<<"A+ Grade";
    }
}

```

C++ Switch Example

```

#include <iostream>
using namespace std;
int main () {
    int num;
    cout<<"Enter a number to check grade:";
    cin>>num;
    switch (num)
    {
        case 10: cout<<"It is 10"; break;
        case 20: cout<<"It is 20"; break;
        case 30: cout<<"It is 30"; break;
        default: cout<<"Not 10, 20 or 30"; break;
    }
}

```

Output:

```

Enter a number:
10
It is 10

```

- Default case executes only if none of the cases matched.

For Loop

- If the number of iterations is fixed, it is recommended to use for loop than while or do-while loops.

C++ For Loop Example

C++ For Loop Example

```
#include <iostream>
using namespace std;
int main() {
    for(int i=1;i<=10;i++){
        cout<<i <<"\n";
    }
}
```

Nested For loop

C++ Nested For Loop Example

Let's see a simple example of nested for loop in C++.

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

int main () {
    for(int i=1;i<=3;i++){
        for(int j=1;j<=3;j++){
            cout<<i<<" " <<j<<"\n";
        }
    }
}
```

C++ Infinite For Loop

If we use double semicolon in for loop, it will be executed infinite times. Let's see a simple example of infinite for loop in C++.

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

int main () {
    for (;)
    {
        cout<<"Infinitive For Loop";
    }
}
```

Output:

```
Infinitive For Loop
Infinitive For Loop
Infinitive For Loop
Infinitive For Loop
Infinitive For Loop
ctrl+c
```

C++ While Loop Example

Let's see a simple example of while loop to print table of 1.

```
#include <iostream>
using namespace std;
int main() {
    int i=1;
    while(i<=10)
    {
        cout<<i<<"\n";
        i++;
    }
}
```

C++ Infinitive While Loop Example:

We can also create infinite while loop by passing true as the test condition.

```
#include <iostream>
using namespace std;
int main () {
    while(true)
    {
        cout<<"Infinitive While Loop";
    }
}
```

Output:

```
Infinitive While Loop
Infinitive While Loop
Infinitive While Loop
Infinitive While Loop
Infinitive While Loop
ctrl+c
```

C++ do-while Loop Example

```
#include <iostream>
using namespace std;
int main() {
    int i = 1;
    do{
        cout<<i<<"\n";
        i++;
    } while (i <= 10);
}
```


C++ Infinitive do-while Loop Example

```
#include <iostream>
using namespace std;
int main() {
    do{
        cout<<"Infinitive do-while Loop";
    } while(true);
}
```

Output:

```
Infinitive do-while Loop
Infinitive do-while Loop
Infinitive do-while Loop
Infinitive do-while Loop
Infinitive do-while Loop
ctrl+c
```

C++ Break Statement Example

Let's see a simple example of C++ break statement which is used inside the loop.

```
#include <iostream>
using namespace std;
int main() {
    for (int i = 1; i <= 10; i++)
    {
        if (i == 5)
        {
            break;
        }
        cout<<i<<"\n";
    }
}
```

C++ Continue Statement Example

```
#include <iostream>
using namespace std;
int main()
{
    for(int i=1;i<=10;i++){
        if(i==5){
            continue;
        }
        cout<<i<<"\n";
    }
}
```

C++ Continue Statement with Inner Loop

C++ Continue Statement continues inner loop only if you use continue statement inside the inner loop.

```
#include <iostream>
using namespace std;
int main()
{
    for(int i=1;i<=3;i++){
        for(int j=1;j<=3;j++){
            if(i==2&&j==2){
                continue;
            }
            cout<<i<<" "<<j<<"\n";
        }
    }
}
```

C++ Single Line Comment

The single line comment starts with // (double slash). Let's see an example of single line comment in C++.

C++ Multi Line Comment

The C++ multi line comment is used to comment multiple lines of code. It is surrounded by slash and asterisk (/ * */). Let's see an example of multi line comment in C++.

C++ Function Example

Let's see the simple example of C++ function.

```
#include <iostream>
using namespace std;
void func() {
    static int i=0; //static variable
    int j=0; //local variable
    i++;
    j++;
    cout<<"i=" << i<<" and j=" <<j<<endl;
}
int main()
{
    func();
    func();
    func();
}
```

Output:

```
i= 1 and j= 1
i= 2 and j= 1
i= 3 and j= 1
```

Call by value in C++

In call by value, **original value is not modified**.

In call by value, value being passed to the function is locally stored by the function parameter in stack memory location. If you change the value of function parameter, it is changed for the current function only. It will not change the value of variable inside the caller method such as main().

```
#include <iostream>
using namespace std;
void change(int data);
int main()
{
    int data = 3;
    change(data);
    cout << "Value of the data is: " << data<< endl;
    return 0;
}
void change(int data)
{
    data = 5;
}
```

Output:

```
Value of the data is: 3
```

Call by reference in C++

In call by reference, original value is modified because we pass reference (address).

Here, address of the value is passed in the function, so actual and formal arguments share the same address space. Hence, value changed inside the function, is reflected inside as well as outside the function.

```

#include<iostream>
using namespace std;
void swap(int *x, int *y)
{
    int swap;
    swap=*x;
    *x=*y;
    *y=swap;
}
int main()
{
    int x=500, y=100;
    swap(&x, &y); // passing value to function
    cout<<"Value of x is: "<<x<<endl;
    cout<<"Value of y is: "<<y<<endl;
    return 0;
}

```

Output:

```

Value of x is: 100
Value of y is: 500

```

Difference between call by value and call by reference in C++

No.	Call by value	Call by reference
1	A copy of value is passed to the function	An address of value is passed to the function
2	Changes made inside the function is not reflected on other functions	Changes made inside the function is reflected outside the function also
3	Actual and formal arguments will be created in different memory location	Actual and formal arguments will be created in same memory location

C++ Recursion

When function is called within the same function, it is known as recursion in C++. The function which calls the same function, is known as recursive function.

```

#include<iostream>
using namespace std;
int main()
{
    int factorial(int);
    int fact,value;
    cout<<"Enter any number: ";
    cin>>value;
    fact=factorial(value);
    cout<<"Factorial of a number is: "<<fact<<endl;
    return 0;
}
int factorial(int n)
{
    if(n<0)
        return(-1); /*Wrong value*/
    if(n==0)
        return(1); /*Terminating condition*/
    else
    {
        return(n*factorial(n-1));
    }
}

```

Output:

```

Enter any number: 5
Factorial of a number is: 120

```

C++ Storage Classes

Storage Class	Keyword	Lifetime	Visibility	Initial Value
Automatic	auto	Function Block	Local	Garbage
Register	register	Function Block	Local	Garbage
Mutable	mutable	Class	Local	Garbage
External	extern	Whole Program	Global	Zero
Static	static	Whole Program	Local	Zero

- Register variables must be used only when we need quick accessing of the memory. Because it stores the data on RAM , which is very limited.
- Static variable is initialized only once and exists till the end of a program. It retains it's value between multiple function calls.
- Extern variable is visible to all the programs , it is used if two or more files are sharing the same variables or functions.

C++ Single Dimensional Array

```
#include <iostream>
using namespace std;
int main()
{
    int arr[5]={10, 0, 20, 0, 30}; //creating and initializing array
    //traversing array
    for (int i = 0; i < 5; i++)
    {
        cout<<arr[i]<<"\n";
    }
}
```

Output:/p>

```
10
0
20
0
30
```

Iterating through foreach loop

```
#include <iostream>
using namespace std;
int main()
{
    int arr[5]={10, 0, 20, 0, 30}; //creating and initializing array
    //traversing array
    for (int i: arr)
    {
        cout<<i<<"\n";
    }
}
```

Output:

```
10
20
30
40
50
```

C++ Passing Array to Function Example: print array elements

Let's see an example of C++ function which prints the array elements.

```
#include <iostream>
using namespace std;
void printArray(int arr[5]);
int main()
{
    int arr1[5] = { 10, 20, 30, 40, 50 };
    int arr2[5] = { 5, 15, 25, 35, 45 };
    printArray(arr1); //passing array to function
    printArray(arr2);
}
void printArray(int arr[5])
{
    cout << "Printing array elements:" << endl;
    for (int i = 0; i < 5; i++)
    {
        cout << arr[i] << "\n";
    }
}
```

Output:

```
Printing array elements:
10
20
30
40
50
Printing array elements:
5
15
25
35
45
```

C++ Multidimensional Array Example

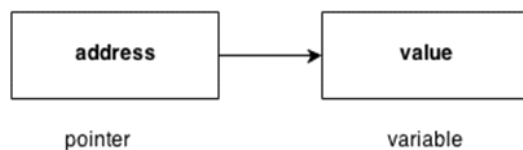
Let's see a simple example of multidimensional array in C++ which declares, initializes and traverse two dimensional arrays.

```
#include <iostream>
using namespace std;
int main()
{
    int test[3][3]; //declaration of 2D array
    test[0][0]=5; //initialization
    test[0][1]=10;
    test[1][1]=15;
    test[1][2]=20;
    test[2][0]=30;
    test[2][2]=10;
    //traversal
    for(int i = 0; i < 3; ++i)
    {
        for(int j = 0; j < 3; ++j)
        {
            cout<< test[i][j]<<" ";
        }
        cout<<"\n"; //new line at each row
    }
    return 0;
}
```

```
int test[3][3] =
{
    {2, 5, 5},
    {4, 0, 3},
    {9, 1, 8} }; //declaration and initialization
```

C++ Pointers

The pointer in C++ language is a variable, it is also known as locator or indicator that points to an address of a value.



Usage of pointer

There are many usage of pointers in C++ language.

1) Dynamic memory allocation

In c language, we can dynamically allocate memory using malloc() and calloc() functions where pointer is used.

2) Arrays, Functions and Structures

Pointers in c language are widely used in arrays, functions and structures. It reduces the code and improves the performance.

Symbols used in pointer

Symbol	Name	Description
& (ampersand sign)	Address operator	Determine the address of a variable.
* (asterisk sign)	Indirection operator	Access the value of an address.

Pointer Example

Let's see the simple example of using pointers printing the address and value.

```
#include <iostream>
using namespace std;
int main()
{
    int number=30;
    int * p;
    p=&number;//stores the address of number variable
    cout<<"Address of number variable is:"<<&number<<endl;
    cout<<"Address of p variable is:"<<p<<endl;
    cout<<"Value of p variable is:"<<*p<<endl;
    return 0;
}
```

Output:

```
Address of number variable is:0x7ffccc8724c4
Address of p variable is:0x7ffccc8724c4
Value of p variable is:30
```

Pointer Program to swap 2 numbers without using 3rd variable

```
#include <iostream>
using namespace std;
int main()
{
    int a=20,b=10,*p1=&a,*p2=&b;
    cout<<"Before swap: *p1="<<*p1<<" *p2="<<*p2<<endl;
    *p1=*p1+*p2;
    *p2=*p1-*p2;
    *p1=*p1-*p2;
    cout<<"After swap: *p1="<<*p1<<" *p2="<<*p2<<endl;
    return 0;
}
```

Output:

```
Before swap: *p1=20 *p2=10
After swap: *p1=10 *p2=20
```

The `sizeof()` is an operator that evaluates the size of data type, constants, variable. It is a compile-time operator as it returns the size of any variable or a constant at the compilation time.

The size, which is calculated by the `sizeof()` operator, is the amount of RAM occupied in the computer.

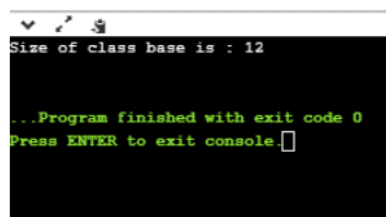
```
#include <iostream>

using namespace std;

class Base
{
    int a;
    int d;
    char ch;
};

int main()
{
    Base b;
    std::cout << "Size of class base is : "<<sizeof(b) << std::endl;
    return 0;
}
```

In the above code, the class has two integer variables, and one char variable. According to our calculation, the size of the class would be equal to 9 bytes (int+int+char), but this is wrong due to the concept of structure padding.



```
Size of class base is : 12

...Program finished with exit code 0
Press ENTER to exit console.
```

Pointer Memory Interpretation

```
#include <iostream>
using namespace std;
int main()
{
    int arr[]={10,20,30,40,50};
    std::cout << "Size of the array 'arr' is : " << sizeof(arr) << std::endl;
    return 0;
}
```

In the above program, we have declared an array of integer type which contains five elements. We have evaluated the size of the array by using **sizeof()** operator. According to our calculation, the size of the array should be 20 bytes as int data type occupies 4 bytes, and array contains 5 elements, so total memory space occupied by this array is $5 \times 4 = 20$ bytes. The same result has been shown by the **sizeof()** operator as we can observe in the following output.

Output

```
Size of the array 'arr' is : 20
...Program finished with exit code 0
Press ENTER to exit console.
```

```
#include <iostream>
using namespace std;
void fun(int arr[])
{
    std::cout << "Size of array is : " << sizeof(arr) << std::endl;
}
int main()
{
    int arr[]={10,20,30,40,50};
    fun(arr);
    return 0;
}
```

In the above program, we have tried to print the size of the array using the function. In this case, we have created an array of type integer, and we pass the 'arr' to the function **fun()**. The **fun()** would return the size of the integer pointer, i.e., **int***, and the size of the **int*** is 8 bytes in the 64-bit operating system.

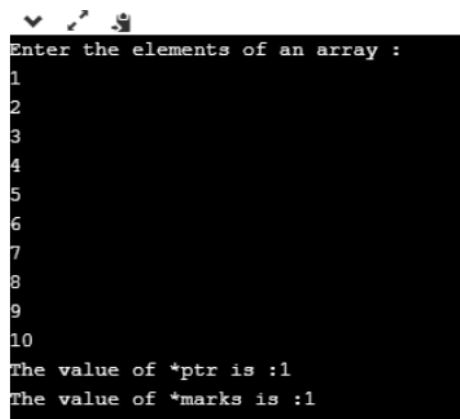
```
main.cpp:15:52: warning: 'sizeof' on array function parameter 'arr' will return size of 'int*' [-Wsizeof-array-argument]
main.cpp:13:18: note: declared here
Size of array is : 8
...Program finished with exit code 0
Press ENTER to exit console.
```

- If the computer has 32bit operating system, then the size of the pointer would be 4 bytes. If the computer has 64-bit operating system, then the size of the pointer would be 8 bytes.
- We have declared two variables num1 and num2 of type int and double, respectively. The size of the int is 4 bytes, while the size of double is 8 bytes. The result would be the variable, which is of double type occupying 8 bytes.

- The first element of the array contains the address of the whole elements of the array. Using the first element address we find all the consecutive elements.

```
#include <iostream>
using namespace std;
int main()
{
    int *ptr; // integer pointer declaration
    int marks[10]; // marks array declaration
    std::cout << "Enter the elements of an array : " << std::endl;
    for(int i=0;i<10;i++)
    {
        cin>>marks[i];
    }
    ptr=marks; // both marks and ptr pointing to the same element..
    std::cout << "The value of *ptr is : " << *ptr << std::endl;
    std::cout << "The value of *marks is : " << *marks << std::endl;
}
```

In the above code, we declare an integer pointer and an array of integer type. We assign the address of marks to the ptr by using the statement ptr=marks; it means that both the variables 'marks' and 'ptr' point to the same element, i.e., marks[0]. When we try to print the values of *ptr and *marks, then it comes out to be same. Hence, it is proved that the array name stores the address of the first element of an array.



```
Enter the elements of an array :
1
2
3
4
5
6
7
8
9
10
The value of *ptr is :1
The value of *marks is :1
```

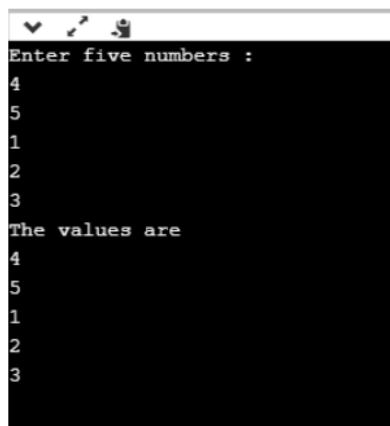
Array Of Pointers to Integers

```

#include <iostream>
using namespace std;
int main()
{
    int ptr1[5]; // integer array declaration
    int *ptr2[5]; // integer array of pointer declaration
    std::cout << "Enter five numbers :." << std::endl;
    for(int i=0;i<5;i++)
    {
        std::cin >> ptr1[i];
    }
    for(int i=0;i<5;i++)
    {
        ptr2[i]=&ptr1[i];
    }
    // printing the values of ptr1 array
    std::cout << "The values are" << std::endl;
    for(int i=0;i<5;i++)
    {
        std::cout << *ptr2[i] << std::endl;
    }
}

```

- In the above code, we declare an array of integer type and an array of integer pointers. We have defined the 'for' loop, which iterates through the elements of an array 'ptr1', and on each iteration, the address of element of ptr1 at index 'i' gets stored in the ptr2 at index 'i'.



```

Enter five numbers :
4
5
1
2
3
The values are
4
5
1
2
3

```

Array of Pointer to Strings

An array of pointer to strings is an array of character pointers that holds the address of the first character of a string or we can say the base address of a string.

The following are the differences between an array of pointers to string and two-dimensional array of characters:

- An array of pointers to string is more efficient than the two-dimensional array of characters in case of memory consumption because an array of pointer to strings consumes less memory than the two-dimensional array of characters to store the strings.
- In an array of pointers, the manipulation of strings is comparatively easier than in the case of 2d array. We can also easily change the position of the strings by using the pointers.

```
char *names[5] = {"john",
                 "Peter",
                 "Marco",
                 "Devin",
                 "Ronan"};
```

In the above code, we declared an array of pointer names as 'names' of size 5. In the above case, we have done the initialization at the time of declaration, so we do not need to mention the size of the array of a pointer. The above code can be re-written as:

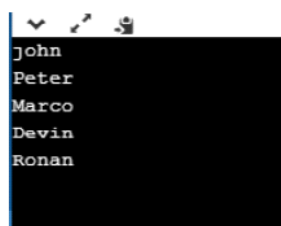
```
char *names[ ] = {"john",
                 "Peter",
                 "Marco",
                 "Devin",
                 "Ronan"};
```

In the above case, each element of the 'names' array is a string literal, and each string literal would hold the base address of the first character of a string. For example, names[0] contains the base address of "john", names[1] contains the base address of "Peter", and so on. It is not guaranteed that all the string literals will be stored in the contiguous memory location, but the characters of a string literal are stored in a contiguous memory location.

```
#include <iostream>
using namespace std;
int main()
{
    char *names[5] = {"john",
                     "Peter",
                     "Marco",
                     "Devin",
                     "Ronan"};
    for(int i=0;i<5;i++)
    {
        std::cout << names[i] << std::endl;
    }
    return 0;
}
```

In the above code, we have declared an array of char pointer holding 5 string literals, and the first character of each string is holding the base address of the string.

Output



```
john
Peter
Marco
Devin
Ronan
```

- A void pointer is a general-purpose pointer that can hold the address of any data type, but it is not associated with any data type.

Syntax of void pointer

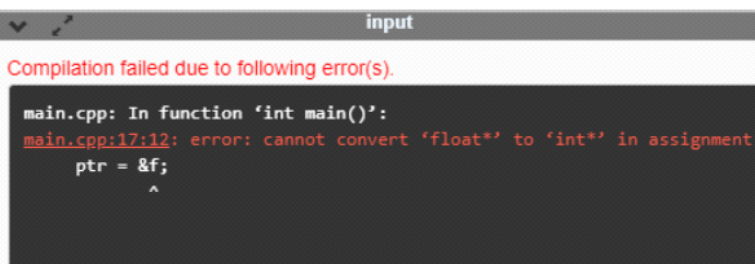
```
void *ptr;
```

```
int *ptr; // integer pointer declaration
float a=10.2; // floating variable initialization
ptr= &a; // This statement throws an error.
```

```
#include <iostream.h>
using namespace std;
int main()
{
    int *ptr;
    float f=10.3;
    ptr = &f; // error
    std::cout << "The value of *ptr is : " <<*ptr<< std::endl;
    return 0;
}
```

In the above program, we declare a pointer of integer type and variable of float type. An integer pointer variable cannot point to the float variable, but it can point to an only integer variable.

Output



```
input
Compilation failed due to following error(s).
main.cpp: In function 'int main()':
main.cpp:17:12: error: cannot convert 'float*' to 'int*' in assignment
    ptr = &f;
           ^
```

- To solve such errors C++ introduces void pointer, where we can declare a pointer of void type and can store address of pointer of any datatype.
- In C, we no need to typecast a void pointer to any type of pointer while assigning. But such type of type casting is necessary in C++.

In C++,

```
#include <iostream>
using namespace std;
int main()
{
    void *ptr; // void pointer declaration
    int *ptr1; // integer pointer declaration
    int data=10; // integer variable initialization
    ptr=&data; // storing the address of data variable in void pointer variable
    ptr1=(int *)ptr; // assigning void pointer to integer pointer
    std::cout << "The value of *ptr1 is : " <<*ptr1<< std::endl;
    return 0;
}
```

Output

```
▼ ↗ 🐞
The value of *ptr1 is : 10

...Program finished with exit code 0
Press ENTER to exit console.□
```

POINTER VERSUS REFERENCE

Pointer holds the memory address of a variable.	Reference is an alias for another variable.
An indirection operator * is used to dereference a pointer.	Reference is a constant pointer which doesn't need a dereferencing operator.
It's an independent variable which can be reassigned to refer to different objects.	It must be assigned at initialization and once created, the address values cannot be reassigned.
NULL value can be assigned to a pointer variable directly.	NULL value cannot be assigned directly.
It lacks automatic indirection.	Automatic indirection is convenient.

Difference Between .net

```
#include <iostream>
using namespace std;
int main()
{
    int a=10;
    int &value=a;
    std::cout << value << std::endl;
    return 0;
}
```

Output

```
10
```



```

#include <iostream>
using namespace std;
int main()
{
    int a=9; // variable initialization
    int b=10; // variable initialization
    swap(a, b); // function calling
    std::cout << "value of a is : " <<a<< std::endl;
    std::cout << "value of b is : " <<b<< std::endl;
    return 0;
}

void swap(int &p, int &q) // function definition
{
    int temp; // variable declaration
    temp=p;
    p=q;
    q=temp;
}

```

Output

```

value of a is :10
value of b is :9

```

- In the case of References, reference to reference is not possible. If we try to do c++ program will throw a compile-time error

Pointer Arithmetic Operations

```

#include <iostream>
using namespace std;
int main()
{
    int a[]={1,2,3,4,5}; // array initialization
    int *ptr; // pointer declaration
    ptr=a; assigning base address to pointer ptr.
    cout<<"The value of *ptr is : "<<*ptr;
    ptr=ptr+1; // incrementing the value of ptr by 1.
    std::cout << "\nThe value of *ptr is: " <<*ptr<< std::endl;
    return 0;
}

```

Output:

```

The value of *ptr is :1
The value of *ptr is: 2

```

- In references we are not having any reference arithmetic operations.

Address of a function

We can get the address of a function very easily. We just need to mention the name of the function, we do not need to call the function.

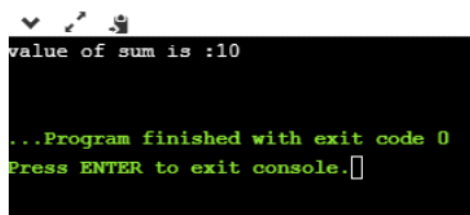
Let's illustrate through an example.

```
#include <iostream>
using namespace std;
int main()
{
    std::cout << "Address of a main() function is : " << &main << std::endl;
    return 0;
}
```

Calling a function indirectly

```
#include <iostream>
using namespace std;
int add(int a , int b)
{
    return a+b;
}
int main()
{
    int (*funcptr)(int,int); // function pointer declaration
    funcptr=add; // funcptr is pointing to the add function
    int sum=funcptr(5,5);
    std::cout << "value of sum is : " << sum << std::endl;
    return 0;
}
```

Output:



```
value of sum is :10
...Program finished with exit code 0
Press ENTER to exit console.
```

Passing a function pointer as a parameter

```

#include <iostream>
using namespace std;
void func1()
{
    cout<<"func1 is called";
}
void func2(void (*funcptr)())
{
    funcptr();
}
int main()
{
    func2(func1);
    return 0;
}

```

What is Memory Management?

Memory management is a process of managing computer memory, assigning the memory space to the programs to improve overall system performance.

In C language, we use the malloc() or calloc() functions to allocate the memory dynamically at run time, and free() function is used to deallocate the dynamically allocated memory. C++ also supports these functions, but C++ also defines unary operators such as new and delete to perform the same tasks, i.e., allocating and freeing the memory. New operator creates an object and delete operator deletes an object. The object that we created using new operator removes only when we use delete operator or else it will be for life time of the program.

New operator

```

int *p = new int;
float *q = new float;

```

```

int *p = new int(45);
float *p = new float(9.8);

```

create a single dimensional array

```

int *a1 = new int[8];

```

Delete operator

```

delete p;
delete q;

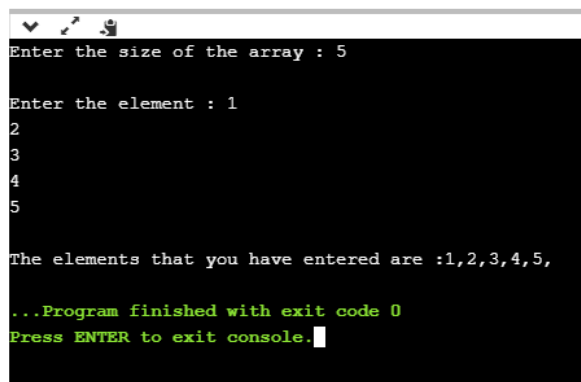
```

```

#include <iostream>
using namespace std
int main()
{
    int size; // variable declaration
    int *arr = new int[size]; // creating an array
    cout<<"Enter the size of the array : ";
    std::cin >> size; //
    cout<<"\nEnter the element : ";
    for(int i=0;i<size;i++) // for loop
    {
        cin>>arr[i];
    }
    cout<<"\nThe elements that you have entered are :";
    for(int i=0;i<size;i++) // for loop
    {
        cout<<arr[i]<<" ";
    }
    delete arr; // deleting an existing array.
    return 0;
}

```

Output



```

Enter the size of the array : 5

Enter the element : 1
2
3
4
5

The elements that you have entered are :1,2,3,4,5,

...Program finished with exit code 0
Press ENTER to exit console.

```

The new is a memory allocation operator, which is used to allocate the memory at the runtime. The memory initialized by the new operator is allocated in a heap. It returns the starting address of the memory, which gets assigned to the variable.

The new operator does not use the sizeof() operator to allocate the memory. It also does not use the resize as the new operator allocates sufficient memory for an object. It is a construct that calls the constructor at the time of declaration to initialize an object.

As we know that the new operator allocates the memory in a heap; if the memory is not available in a heap and the new operator tries to allocate the memory, then the exception is thrown. If our code is not able to handle the exception, then the program will be terminated abnormally.

```
#include <iostream>
using namespace std;
int main()
{
    int *ptr; // integer pointer variable declaration
    ptr=new int; // allocating memory to the pointer variable ptr.
    std::cout << "Enter the number : " << std::endl;
    std::cin >> *ptr;
    std::cout << "Entered number is " << *ptr << std::endl;
    return 0;
}
```

Output:

```
Enter the number :
15
Entered number is 15

...Program finished with exit code 0
Press ENTER to exit console
```

```
#include <iostream>
#include <stdlib.h>
using namespace std;

int main()
{
    int len; // variable declaration
    std::cout << "Enter the count of numbers : " << std::endl;
    std::cin >> len;
    int *ptr; // pointer variable declaration
    ptr=(int*) malloc(sizeof(int)*len); // allocating memory to the pointer variable
    for(int i=0;i<len;i++)
    {
        std::cout << "Enter a number : " << std::endl;
        std::cin >> *(ptr+i);
    }
    std::cout << "Entered elements are : " << std::endl;
    for(int i=0;i<len;i++)
    {
        std::cout << *(ptr+i) << std::endl;
    }
    free(ptr);
    return 0;
}
```

We can do realloc operation using following process

```
newbuf = new Type[newsize];
std::copy_n(oldbuf, std::min(oldsized, newsize), newbuf);
delete[] oldbuf;
return newbuf;
```

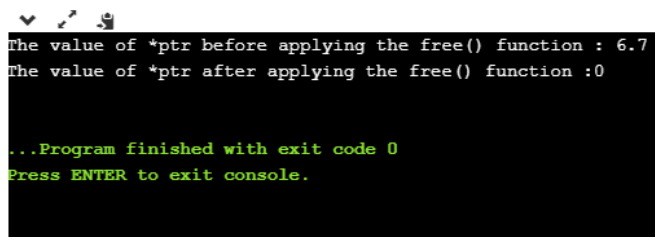
But this method is not suggestable, we can use deque because deque stores its contents non contiguously.

Memory allocated using new operator cannot be relocated.

If the memory is freed which is allocated by malloc then if we try to access that freed memory we

would get some garbage value. Where as if we try to free memory which is allocated by calloc and try to access the freed memory we would get 0 which is default value.

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
    float *ptr; // float pointer declaration
    ptr=(float*)calloc(1,sizeof(float));
    *ptr=6.7;
    std::cout << "The value of *ptr before applying the free() function : " <<*ptr<< std::endl;
    free(ptr);
    std::cout << "The value of *ptr after applying the free() function : " <<*ptr<< std::endl;
    return 0;
}
```



```

The value of *ptr before applying the free() function : 6.7
The value of *ptr after applying the free() function : 0

...Program finished with exit code 0
Press ENTER to exit console.
```

Some important points related to delete operator are:

- It is either used to delete the array or non-array objects which are allocated by using the new keyword.
- To delete the array or non-array object, we use delete[] and delete operator, respectively.
- The new keyword allocated the memory in a heap; therefore, we can say that the delete operator always de-allocates the memory from the heap
- It does not destroy the pointer, but the value or the memory block, which is pointed by the pointer is destroyed.

Differences between delete and free()

The following are the differences between delete and free() in C++ are:

- The delete is an operator that de-allocates the memory dynamically while the free() is a function that destroys the memory at the runtime.
- The delete operator is used to delete the pointer, which is either allocated using new operator or a NULL pointer, whereas the free() function is used to delete the pointer that is either allocated using malloc(), calloc() or realloc() function or NULL pointer.
- When the delete operator destroys the allocated memory, then it calls the destructor of the class in C++, whereas the free() function does not call the destructor; it only frees the memory from the heap.
- The delete() operator is faster than the free() function.

C++ Vector

Saturday, April 2, 2022 7:55 PM

Container	Description	Header file	iterator
vector	vector is a class that creates a dynamic array allowing insertions and deletions at the back.	<vector>	Random access
list	list is the sequence containers that allow the insertions and deletions from anywhere.	<list>	Bidirectional
deque	deque is the double ended queue that allows the insertion and deletion from both the ends.	<deque>	Random access
set	set is an associate container for storing unique sets.	<set>	Bidirectional
multiset	Multiset is an associate container for storing non- unique sets.	<set>	Bidirectional
map	Map is an associate container for storing unique key-value pairs, i.e. each key is associated with only one value(one to one mapping).	<map>	Bidirectional
multimap	multimap is an associate container for storing key- value pair, and each key can be associated with more than one value.	<map>	Bidirectional
stack	It follows last in first out(LIFO).	<stack>	No iterator
queue	It follows first in first out(FIFO).	<queue>	No iterator
Priority-queue	First element out is always the highest priority element.	<queue>	No iterator

C++ Vector

- A vector is a sequence container class that implements dynamic array, means size automatically changes when appending elements. A vector stores the elements in contiguous memory locations and allocates the memory as needed at run time.

```
#include<iostream>
#include<vector>
using namespace std;
int main()
{
    vector<string> v1;
    v1.push_back("javaTpoint ");
    v1.push_back("tutorial");
    for(vector<string>::iterator itr=v1.begin();itr!=v1.end();++itr)
        cout<<*itr;
    return 0;
}
```

Output:

```
javaTpoint tutorial
```

```
#include<iostream>
#include<vector>
using namespace std;
int main()
{
vector<int> v1{1,2,3,4};
for(int i=0;i<v1.size();i++)
cout<<v1.at(i);
return 0;
}
```

Output:

```
1234
```

```
#include<iostream>
#include<vector>
using namespace std;
int main()
{
vector<string> fruit{"mango","apple","banana"};
cout<<fruit.back();
return 0;
}
```

Output:

```
banana
```

```
#include<iostream>
#include<vector>
using namespace std;
int main()
{
vector<string> language{"java","C","C++"};
cout<<language.front();
return 0;
}
```

Output:

```
java
```



```

#include<iostream>
#include<vector>
using namespace std;
int main()
{
vector<int> v1={1,2,3,4,5};
vector<int> v2={6,7,8,9,10};
cout<<"Before swapping,elements of v1 are :";
for (int i=0;i<v1.size();i++)
cout<<v1[i]<<" ";
cout<<"\n";
cout<<"Before swapping,elements of v2 are :";
for(int i=0;i<v2.size();i++)
cout<<v2[i]<<" ";
cout<<"\n";
v1.swap(v2);
cout<<"After swapping,elements of v1 are :";
for(int i=0;i<v1.size();i++)
cout<<v1[i]<<" ";
cout<<"\n";
cout<<"After swapping,elements of v2 are:";
for(int i=0;i<v2.size();i++)
cout<<v2[i]<<" ";
return 0;
}

```

Output:

```

Before swapping,elements of v1 are :1 2 3 4 5
Before swapping,elements of v2 are :6 7 8 9 10
After swapping,elements of v1 are :6 7 8 9 10
After swapping,elements of v2 are :1 2 3 4 5

```

```

#include<iostream>
#include<vector>
using namespace std;
int main()
{
vector<char> v;
v.push_back('j');
v.push_back('a');
v.push_back('v');
v.push_back('a');
for(int i=0;i<v.size();i++)
cout<<v[i];
return 0;
}

```

Output:

```

java

```

```

#include<iostream>
#include<vector>
using namespace std;
int main()
{
    vector<string> v{"welcome","to","javaTpoint","tutorial"};
    cout<<"Initial string is :";
    for(int i=0;i<v.size();i++)
    cout<<v[i]<<" ";
    cout<<"\n";
    cout<<"After deleting last string, string is :";
    v.pop_back();
    for(int i=0;i<v.size();i++)
    cout<<v[i]<<" ";
    return 0;
}

```

Output:

```

Initial string is :welcome to javaTpoint tutorial
After deleting last string, string is :welcome to javaTpoint

```

```

#include<iostream>
#include<vector>
using namespace std;
int main()
{
    vector<char> v1{'j','a','v','a'};
    if(v1.empty())
    cout<<"Vector v1 is empty";
    else
    cout<<"Vector v1 is not empty";
    return 0;
}

```

Output:

```

Vector v1 is not empty.

```

```

#include<iostream>
#include<vector>
using namespace std;
int main()
{
vector<string> v{"java"};
stringstr="programs";
v.insert(v.begin()+1,str);
for(int i=0;i<v.size();i++)
cout<<v[i]<<" ";
return 0;
}

```

Output:

```
java programs
```

```

#include<iostream>
#include<vector>
using namespace std;
int main()
{
vector<string> v{"C" ,"Tutorials"};
v.insert(v.begin()+1,2,"C");
for(int i=0;i<v.size();i++)
cout<<v[i]<<" ";
return 0;
}

```

Output:

```
C CC Tutorials
```

```

#include<iostream>
#include<vector>
using namespace std;
int main()
{
vector<int> v{1,2,3,4,5};
vector<int> v1{6,7,8,9,10};
v.insert(v.end(),v1.begin(),v1.begin()+5);
for(int i=0;i<v.size();i++)
cout<<v[i]<<" ";
return 0;
}

```

Output:

```
1 2 3 4 5 6 7 8 9 10
```

```

#include<iostream>
#include<vector>
using namespace std;
int main()
{
vector<string> fruit{"mango","apple","strawberry","kiwi","banana"};
cout<<"fruit names are :";
for(int i=0;i<fruit.size();i++)
cout<<fruit[i]<<" ";
cout<<"\n";
fruit.erase(fruit.begin()+1,fruit.begin()+3);
cout<<"After removing apple and strawberry fruits,"<<"\n";
for(int i=0;i<fruit.size();i++)
cout<<fruit[i]<<" ";
return 0;
}

```

Output:

```

fruit names are :mango, apple,strawberry, kiwi, banana
After removing apple and strawberry fruits,
Mango, kiwi, banana

```

```

#include<iostream>
#include<vector>
using namespace std;
int main()
{
vector<int> v;
for(int i=1;i<=10;i++)
{
v.push_back(i);
}
cout<<"Initial elements are :";
for(int i=0;i<v.size();i++)
cout<<v[i]<<" ";
v.resize(5);
cout<<"\n";
cout<<"After resizing its size to 5,elements are :";
for(int i=0;i<v.size();i++)
cout<<v[i]<<" ";
return 0;
}

```

Output:

```

Initial elements are: 1 2 3 4 5 6 7 8 9 10
After resizing its size to 5, elementsare: 1 2 3 4 5

```

```

#include<iostream>
#include<vector>
using namespace std;
int main()
{
    vector<string> v1{"java","C","C++"};
    cout<<"Elements of v1 are :";
    for(int i=0;i<v1.size();i++)
        cout<<v1[i]<<" ";
    v1.resize(5,".Net");
    for(int i=0;i<v1.size();i++)
        cout<<v1[i]<<" ";
    return 0;
}

```

Output:

```

Elements of v1 are :java C C++ java C C++ .Net .Net

```

```

#include<iostream>
#include<vector>
using namespace std;
int main()
{
    vector<int> v{1,2,3,4,5};
    cout<<"Elements of v vector are :";
    for(int i=0;i<v.size();i++)
        cout<<v[i]<<" ";
    v.clear();
    for(int i=0;i<v.size();i++)
        cout<<v[i];
    return 0;
}

```

Output:

```

Elements of v vector are :1 2 3 4 5

```

```
#include<iostream>
#include<vector>
using namespace std;
int main()
{
    vector<string> v{"Welcome to javaTpoint","c"};
    int n=v.size();
    cout<<"Size of the string is : "<<n;
    return 0;
}
```

Output:

```
Size of the string is:2
```

```
#include<iostream>
#include<vector>
using namespace std;
int main()
{
    vector<int> v{1,2,3,4,5};
    vector<int> v1;
    v1.assign(v.begin()+1,v.end()-1);
    for(int i=0;i<v1.size();i++)
        std::cout<<v1[i] <<std::endl;
    return 0;
}
```

Output:

```
2
3
4
```

```
#include<iostream>
#include<vector>
using namespace std;
int main()
{
    vector<char> v;
    v.assign(5,'C');
    for(int i=0;i<v.size();i++)
        std::cout<<v[i] <<" ";
    return 0;
}
```

Output:

```
C CCCC
```

```

#include<iostream>
#include<vector>
using namespace std;
int main()
{
vector<string> v{"java"};
vector<string> v1{".NET"};
cout<<"initially,value of v1 is :";
for(int i=0;i<v1.size();i++)
std::cout<<v1[i];

cout<<"\n";
cout<<"Now, the value of vector v1 is :";
v1.operator=(v);
for(int i=0;i<v1.size();i++)
std::cout<<v1[i];
return 0;
}

```

Output:

```
java
```

```

#include <iostream>
#include<vector>
using namespace std;
int main()
{
vector<string> v{"Computer science","electronics","electrical","mechanical"};
vector<string>::reverse_iterator ritr;
vector<string>::iterator itr;
std::cout<<"Strings are :";
for(itr=v.begin();itr!=v.end();itr++)
cout<<*itr<<" ";
cout<<"\n";
cout<<"Reverse strings are :";
for(ritr=v.rbegin();ritr!=v.rend();ritr++)
cout<<*ritr<<" ";
return 0;
}

```

Output:

```
Strings are :Computer science, electronics, mechanical
Reverse strings are :mechanical, electrical, electronics, Computer science
```

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> v{1,2,3,4,5};
    std::cout<<v.max_size() <<std::endl;
    return 0;
}
```

Output:

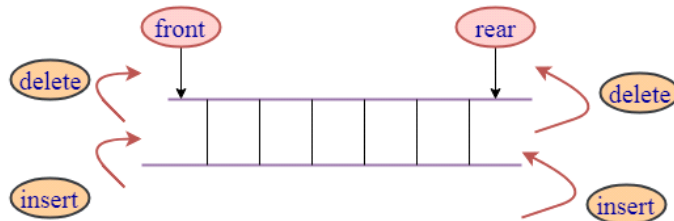
```
4611686018427387903
```

- With the parent 'Collection class,' vector is sent in the form of a template class.
- Vector is not indexed based. They are iterator based.
- Arrays are memory saving and vectors are memory costly.

CPP Deque

Sunday, April 3, 2022 12:08 PM

- Deque stands for double ended queue. It generalizes the queue data structure i.e insertion and deletion can be performed from both the ends either front or back.



```
#include <iostream>
#include <deque>
using namespace std;
int main()
{
    deque<int> deq={7,8,4,5};
    deque<int>::iterator itr;
    deq.emplace(deq.begin(),1);
    for(itr=deq.begin();itr!=deq.end();++itr)
        std::cout << *itr << " ";
    return 0;
}
```

Output:

```
1 7 8 4 5
```

Deque functions:

Method	Description
<code>assign()</code>	It assigns new content and replacing the old one.
<code>emplace()</code>	It adds a new element at a specified position.
<code>emplace_back()</code>	It adds a new element at the end.
<code>emplace_front()</code>	It adds a new element in the beginning of a deque.
<code>insert()</code>	It adds a new element just before the specified position.
<code>push_back()</code>	It adds a new element at the end of the container.
<code>push_front()</code>	It adds a new element at the beginning of the container.
<code>pop_back()</code>	It deletes the last element from the deque.
<code>pop_front()</code>	It deletes the first element from the deque.
<code>swap()</code>	It exchanges the contents of two deques.
<code>clear()</code>	It removes all the contents of the deque.
<code>empty()</code>	It checks whether the container is empty or not.
<code>erase()</code>	It removes the elements.
<code>max_size()</code>	It determines the maximum size of the deque.
<code>resize()</code>	It changes the size of the deque.
<code>shrink_to_fit()</code>	It reduces the memory to fit the size of the deque.
<code>size()</code>	It returns the number of elements.
<code>at()</code>	It access the element at position pos.
<code>operator[]()</code>	It access the element at position pos.
<code>operator=()</code>	It assigns new contents to the container.
<code>back()</code>	It access the last element.
<code>begin()</code>	It returns an iterator to the beginning of the deque.
<code>cbegin()</code>	It returns a constant iterator to the beginning of the deque.
<code>end()</code>	It returns an iterator to the end.
<code>cend()</code>	It returns a constant iterator to the end.
<code>rbegin()</code>	It returns a reverse iterator to the beginning.
<code>crbegin()</code>	It returns a constant reverse iterator to the beginning.
<code>rend()</code>	It returns a reverse iterator to the end.
<code>crend()</code>	It returns a constant reverse iterator to the end.
<code>front()</code>	It access the last element.

CPP List

Sunday, April 3, 2022 12:19 PM

- List is a contiguous container while vector is a non-contiguous container i.e list stores the elements on a contiguous memory and vector stores on a non-contiguous memory.
- Insertion and deletion in the middle of the vector is very costly as it takes lot of time in shifting all the elements. Linklist overcome this problem and it is implemented using list container.
- List supports a bidirectional and provides an efficient way for insertion and deletion operations.
- Traversal is slow in list as list elements are accessed sequentially while vector supports a random access.

List can be initialised in two ways.

```
list<int> new_list{1,2,3,4};  
or  
list<int> new_list = {1,2,3,4};
```

Method	Description
<code>insert()</code>	It inserts the new element before the position pointed by the iterator.
<code>push_back()</code>	It adds a new element at the end of the vector.
<code>push_front()</code>	It adds a new element to the front.
<code>pop_back()</code>	It deletes the last element.
<code>pop_front()</code>	It deletes the first element.
<code>empty()</code>	It checks whether the list is empty or not.
<code>size()</code>	It finds the number of elements present in the list.
<code>max_size()</code>	It finds the maximum size of the list.
<code>front()</code>	It returns the first element of the list.
<code>back()</code>	It returns the last element of the list.
<code>swap()</code>	It swaps two list when the type of both the list are same.
<code>reverse()</code>	It reverses the elements of the list.
<code>sort()</code>	It sorts the elements of the list in an increasing order.
<code>merge()</code>	It merges the two sorted list.
<code>splice()</code>	It inserts a new list into the invoking list.
<code>unique()</code>	It removes all the duplicate elements from the list.
<code>resize()</code>	It changes the size of the list container.
<code>assign()</code>	It assigns a new element to the list container.
<code>emplace()</code>	It inserts a new element at a specified position.
<code>emplace_back()</code>	It inserts a new element at the end of the vector.
<code>emplace_front()</code>	It inserts a new element at the beginning of the list.

```

#include <iostream>
#include<list>
using namespace std;
int main()
{
    std::list<int> li={1,2,3,4,5,6};
    cout<<"content of list li is :";
    for(list<int> :: iterator itr=li.begin();itr!=li.end();++itr)
        cout<<*itr;
    li.reverse();
    cout<<"\n";
    cout<<"After reversing, content of list li is :";
    for(list<int> :: iterator itr=li.begin();itr!=li.end();++itr)
        cout<<*itr;

    cout<<"\n";
    return 0;
}

```

Output:

```

content of list li is : 123456
After reversing, content of list li is : 654321

```

```

#include <iostream>
#include<list>
using namespace std;
int main()
{
    list<int> li={6,4,10,2,4,1};
    list<int>:: iterator itr;
    cout<<"Elements of list are :?";
    for(itr=li.begin();itr!=li.end();++itr)
        std::cout << *itr<<" ";
    li.sort();
    cout<<"\n";
    cout<<"Sorted elements are :?";
    for(itr=li.begin();itr!=li.end();++itr)
        std::cout << *itr <<" ";
    return 0;
}

```

Output:

```

Elements of list are : 6,4,10,2,4,1,
Sorted elements are : 1,2,4,4,6,10

```

```

#include <iostream>
#include<list>
using namespace std;
bool comparison(int first, int second)
{
    bool a;
    a=first<second;
    return (a);
}
int main()
{
    list<int> li={9,10,11};
    list<int> li1={5,6,7,15};
    li.merge(li1,comparison);
    for(list<int>::iterator itr=li.begin();itr!=li.end();++itr)
        std::cout << *itr << " " << std::endl;
    return 0;
}

```

```
5 6 7 9 10 11 15
```

```

#include <iostream>
#include<list>
using namespace std;
bool pred( float x,float y)
{
    return(int(x)==int(y));
}
int main()
{
    list<float> l1={12,12.5,12.4,13.1,13.5,14.7,15.5};
    list<float> ::iterator itr;
    l1.unique(pred);
    for(itr=l1.begin();itr!=l1.end();++itr)
        std::cout << *itr << " , ";
    return 0;
}

```

Output:

```
12 ,13.1,14.7,15.5
```

Method	Description
<code>insert()</code>	It inserts the new element before the position pointed by the iterator.
<code>push_back()</code>	It adds a new element at the end of the vector.
<code>push_front()</code>	It adds a new element to the front.
<code>pop_back()</code>	It deletes the last element.
<code>pop_front()</code>	It deletes the first element.
<code>empty()</code>	It checks whether the list is empty or not.
<code>size()</code>	It finds the number of elements present in the list.
<code>max_size()</code>	It finds the maximum size of the list.
<code>front()</code>	It returns the first element of the list.
<code>back()</code>	It returns the last element of the list.
<code>swap()</code>	It swaps two list when the type of both the list are same.
<code>reverse()</code>	It reverses the elements of the list.
<code>sort()</code>	It sorts the elements of the list in an increasing order.
<code>merge()</code>	It merges the two sorted list.
<code>splice()</code>	It inserts a new list into the invoking list.
<code>unique()</code>	It removes all the duplicate elements from the list.
<code>unique()</code>	It removes all the duplicate elements from the list.
<code>resize()</code>	It changes the size of the list container.
<code>assign()</code>	It assigns a new element to the list container.
<code>emplace()</code>	It inserts a new element at a specified position.
<code>emplace_back()</code>	It inserts a new element at the end of the vector.
<code>emplace_front()</code>	It inserts a new element at the beginning of the list.

Love Babber

Sunday, April 3, 2022 1:53 PM

```
main.cpp
1 #include <iostream>
2 #include <array>
3
4 using namespace std;
5 int main() {
6
7     int basic[3] = {1,2,3};
8
9     array<int,4> a = {1,2,3,4};
10
11     int size = a.size();
12
13     for(int i=0; i<size; i++){
14         cout<<a[i]<<endl;
15     }
16
17     cout<<"Element at 2nd Index-> " <<a.at(2)<<endl;
18
19     cout<<"Empty or not-> " <<a.empty()<<endl;
20
21     cout<<"First Element-> " <<a.front()<<endl;
22     cout<<"last Element-> " <<a.back()<<endl;
23 }
24 }
```

```
Console
Shell
> clang++-7 -pthread -std=c++17 -o main main.cpp
> ./main
1
2
3
4
Element at 2nd Index-> 3
Empty or not-> 0
First Element-> 1
last Element-> 4
> 
```

We could not insert more elements into an array than declared size but when coming to the vector the size will be double if we need extra space and if we need to reduce the memory we can use `shrink_to_fit` function on the vector to reduce the memory consumption.

```
main.cpp
4 int main() {
5
6     vector<int> v;
7     cout<<"Capacity-> " <<v.capacity()<<endl;
8
9     v.push_back(1);
10    cout<<"Capacity-> " <<v.capacity()<<endl;
11
12    v.push_back(2);
13    cout<<"Capacity-> " <<v.capacity()<<endl;
14
15    v.push_back(3);
16    cout<<"Capacity-> " <<v.capacity()<<endl;
17    cout<<"Size-> " <<v.size()<<endl;
18
19    cout<<"Element at 2nd Index" <<v.at(2)<<endl;
20
21    cout<<"front " <<v.front()<<endl;
22    cout<<"back " <<v.back()<<endl;
23
24    cout<<"before pop"<<endl;
25    for(int i:v) {
26        cout<<i<<" ";
27    }cout<<endl;
28
29    v.pop_back();
30
31    cout<<"after pop"<<endl;
32    for(int i:v) {
33        cout<<i<<" ";
34    }
35
36    cout<<"before clear size " <<v.size()<<endl;
37    v.clear();
38    cout<<"after clear size " <<v.size()<<endl;
39 }
```

```
Console
Shell
> clang++-7 -pthread -std=c++17 -o main main.cpp
> ./main
Capacity-> 0
Capacity-> 1
Capacity-> 2
Capacity-> 4
Size-> 3
Element at 2nd Index3
front 1
back 3
before pop
1 2 3
after pop
1 2 before clear size 2
after clear size 0
> 
```

- Vector takes memory size as even number of locations , you can see above the memory size is determined by the capacity function and the memory occupied by our inputs is given by the size function. Check the below example.

```
main.cpp
4 int main() {
5
6     vector<int> v;
7     cout<<"Capacity-> " <<v.capacity()<<endl;
8
9     v.push_back(1);
10    cout<<"Capacity-> " <<v.capacity()<<endl;
11
12    v.push_back(2);
13    cout<<"Capacity-> " <<v.capacity()<<endl;
14
15    v.push_back(3);
16    cout<<"Capacity-> " <<v.capacity()<<endl;
17    cout<<"Size-> " <<v.size()<<endl;
18
19    cout<<"Element at 2nd Index" <<v.at(2)<<endl;
20
21    cout<<"front " <<v.front()<<endl;
22    cout<<"back " <<v.back()<<endl;
23
24    cout<<"before pop"<<endl;
25    for(int i:v) {
26        cout<<i<<" ";
27    }cout<<endl;
28
29    v.pop_back();
30
31    cout<<"after pop"<<endl;
32    for(int i:v) {
33        cout<<i<<" ";
34    }
35
36    cout<<"before clear size " <<v.size()<<endl;
37    v.clear();
38    cout<<"after clear size " <<v.size()<<endl;
39 }
```

```
Console
Shell
> clang++-7 -pthread -std=c++17 -o main main.cpp
> ./main
Capacity-> 0
Capacity-> 1
Capacity-> 2
Capacity-> 4
Size-> 3
Element at 2nd Index3
front 1
back 3
before pop
1 2 3
after pop
1 2 before clear size 2
after clear size 0
> 
```

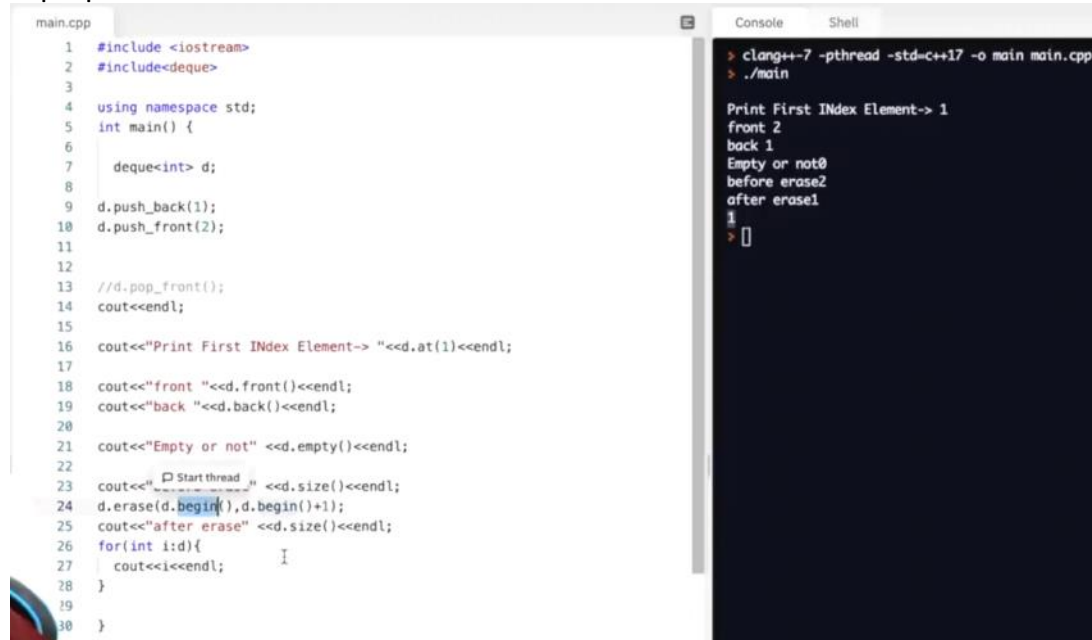
In the following example , to declare a array of defined size and declaring a initial value to store in that.

```
vector<int> a(5,1);
```

Copying elements of one vector to the other vector

```
vector<int> last(a);
cout<<"print last"<<endl;
for(int i:last) {
    cout<<i<<" ";
}cout<<endl;
```

Deque operations

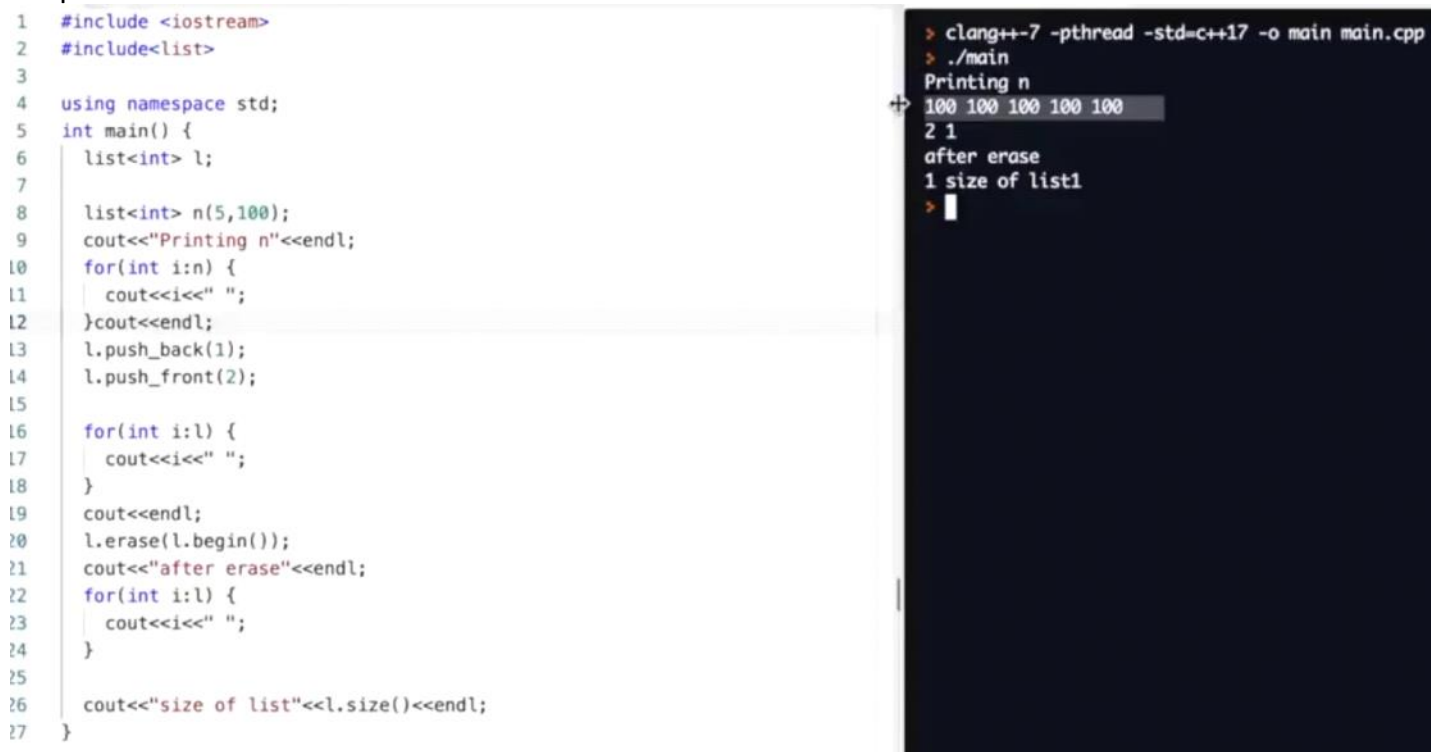


```
main.cpp
1 #include <iostream>
2 #include<deque>
3
4 using namespace std;
5 int main() {
6
7     deque<int> d;
8
9     d.push_back(1);
10    d.push_front(2);
11
12
13    //d.pop_front();
14    cout<<endl;
15
16    cout<<"Print First Index Element-> "<<d.at(1)<<endl;
17
18    cout<<"front "<<d.front()<<endl;
19    cout<<"back "<<d.back()<<endl;
20
21    cout<<"Empty or not" <<d.empty()<<endl;
22
23    cout<<" Start thread " <<d.size()<<endl;
24    d.erase(d.begin(),d.begin()+1);
25    cout<<"after erase" <<d.size()<<endl;
26    for(int i:d){
27        cout<<i<<endl;
28    }
29
30 }
```

```
Console
> clang++-7 -pthread -std=c++17 -o main main.cpp
> ./main

Print First Index Element-> 1
front 2
back 1
Empty or not0
before erase2
after erase1
1
> []
```

List Operations:



```
1 #include <iostream>
2 #include<list>
3
4 using namespace std;
5 int main() {
6     list<int> l;
7
8     list<int> n(5,100);
9     cout<<"Printing n"<<endl;
10    for(int i:n) {
11        cout<<i<<" ";
12    }cout<<endl;
13    l.push_back(1);
14    l.push_front(2);
15
16    for(int i:l) {
17        cout<<i<<" ";
18    }
19    cout<<endl;
20    l.erase(l.begin());
21    cout<<"after erase"<<endl;
22    for(int i:l) {
23        cout<<i<<" ";
24    }
25
26    cout<<"size of list"<<l.size()<<endl;
27 }
```

```
> clang++-7 -pthread -std=c++17 -o main main.cpp
> ./main
Printing n
100 100 100 100 100
2 1
after erase
1 size of list1
> []
```

Stack Operations (Last In First Out)


```

1  #include <iostream>
2  #include<stack>
3
4  using namespace std;
5  int main() {
6      stack<string> s;
7
8      s.push("love");
9      s.push("babbar");
10     s.push("Kumar");
11
12     //cout<<"Top Element-> "<<s.top()<<endl;
13     // Start thread
14     s.pop();
15     cout<<"Top Element-> "<<s.top()<<endl;
16
17     cout<<"size of stack"<<s.size()<<endl;
18
19     cout<<"Empty or not "<<s.empty()<<endl;
20
21 }
22

```

```

> clang++-7 -pthread -std=c++17 -o main main.cpp
> ./main
Top Element-> Kumar
Top Element-> babbar
size of stack2
Empty or not 0
>

```

Queue Operations (First in First Out)

```

1  #include <iostream>
2  #include<queue>
3
4  using namespace std;
5  int main() {
6
7      queue<string> q;
8
9      q.push("love");
10     q.push("Babbar");
11     q.push("Kumar");
12
13     // Start thread
14     cout<<"Size before pop" <<q.size()<<endl;
15
16     cout<<"First Element " <<q.front()<<endl;
17     q.pop();
18     cout<<"First Element " <<q.front()<<endl;
19
20     cout<<"Size after pop" <<q.size()<<endl;
21 }
22

```

```

> clang++-7 -pthread -std=c++17 -o main main.cpp
> ./main
Size before pop3
First Element love
First Element Babbar
Size after pop2
>

```

Priority Queue (Follows Max Heap)

- It returns maximum element of the queue every time .

```

1  #include <iostream>
2  #include<queue>
3
4  using namespace std;
5  int main() {
6      //max heap
7      priority_queue<int> maxi;
8
9      //min - heap
10     priority_queue<int,vector<int> , greater<int> > mini;
11
12     maxi.push(1);
13     maxi.push(3);
14     maxi.push(2);
15     maxi.push(0);
16     cout<<"size-> "<<maxi.size()<<endl;
17     int n = maxi.size();
18     for(int i=0;i<n;i++) {
19         cout<<maxi.top()<<" ";
20         maxi.pop();
21     }cout<<endl;
22
23     mini.push(5);
24     mini.push(1);
25     mini.push(0);
26     mini.push(4);
27     mini.push(3);
28
29     int m = mini.size();
30     for(int i=0;i<m;i++) {
31         cout<<mini.top()<<" ";
32         mini.pop();
33     }cout<<endl;
34 }
35

```

```

> clang++-7 -pthread -std=c++17 -o main main.cpp
> ./main
size-> 4
3 2 1 0
0 1 3 4 5
>

```

Set Operations

- Set returns the elements by removing duplicates and in sorted manner.

```

1  #include <iostream>
2  #include<set>
3
4  using namespace std;
5  int main() {
6      set<int> s;
7
8      s.insert(5);
9      s.insert(5);
10     s.insert(5);
11     s.insert(1);
12     s.insert(6);
13     s.insert(6);
14     s.insert(0);
15     s.insert(0);
16     s.insert(0);
17
18     for(auto i : s) {
19         cout<<i<<endl;
20     }cout<<endl;
21
22     set<int>::iterator it = s.begin();
23     it++;
24
25     s.erase(it);
26
27     for(auto i : s) {
28         cout<<i<<endl;
29     }
30     cout<<endl;
31     cout<<"-5 is present or not -> "<<s.count(-5)<<endl;
32
33     set<int>::iterator itr = s.find(5);
34
35     cout<<"value present at itr-> "<<itr<<endl;

```

```
set<int>::iterator itr = s.find(5);
```

```

for(auto it=itr;it!=s.end();it++) {
    cout<<it<<" ";
}cout<<endl;

```

```

> clang++-7 -pthread -std=c++17 -o main main.cpp
> ./main
0
1
5
6

0
5
6

-5 is present or not -> 0
value present at itr-> 1
>

```

Map Operation

- Values stored in form of key value pair.

```

1  #include <iostream>
2  #include<map>
3
4  using namespace std;
5  int main() {
6      map<int,string> m;
7
8      m[1] = "babbar";
9      m[13] = "kumar";
10     // m.insert({5,"bheem"});
11
12     m.insert({5,"bheem"});
13
14     cout<<"before erase"<<endl;
15     for(auto i:m) {
16         cout<<i.first<<" "<<i.second<<endl;
17     }
18
19     cout<<"finding -13 -> " <<m.count(-13)<<endl;
20
21     // m.erase(13);
22     cout<<"after erase"<<endl;
23     for(auto i:m) {
24         cout<<i.first<<" "<<i.second<<endl;
25     }cout<<endl<<endl;
26
27     auto it = m.find(5);
28
29     for(auto i=it;i!=m.end();i++) {
30         cout<<(*i).first<<endl;
31     }
32
33
34 }

```

```

> clang++-7 -pthread -std=c++17 -o main main.cpp
> ./main
before erase
1 babbar
2 love
5 bheem
13 kumar
finding -13 -> 0
after erase
1 babbar
2 love
5 bheem
13 kumar

5
13
>

```

Map operation uses red black tree. So its time complexity is $O(\log n)$

Un ordered set and un ordered map both are same.

Binary Search

```

1 #include <iostream>
2 #include <algorithm>
3 #include <vector>
4
5 using namespace std;
6 int main() {
7
8     vector<int> v;
9
10    v.push_back(1);
11    v.push_back(3);
12    v.push_back(6);
13    v.push_back(7);
14
15    cout<<"Finding 6-> "<<binary_search(v.begin(),v.end(),6)<<endl;
16
17
18    cout<<"lower bound-> "<<lower_bound(v.begin(),v.end(),6)-v.begin()<<endl;
19    cout<<"Uppper bound-> "<<upper_bound(v.begin(),v.end(),4)-v.begin()<<endl;
20
21    int a =3;
22    int b =5;
23
24    cout<<"max -> "<<max(a,b);
25
26    cout<<"min -> "<<min(a,b);
27
28    swap(a,b);
29    cout<<endl<<"a-> "<<a<<endl;
30
31    string abcd = "abcd";
32    reverse(abcd.begin(),abcd.end());
33    cout<<"string-> "<<abcd<<endl;
34
35 }

```

```

> clang++-7 -pthread -std=c++17 -o main main.cpp
> ./main
Finding 6-> 1
lower bound-> 2
Uppper bound-> 2
max -> 5min -> 3
a-> 5
string-> dcba
>

```

```

rotate(v.begin(),v.begin()+1,v.end());
cout<<"after rotate"<<endl;
for(int i:v){
    cout<<i<<" ";
}

```

CPP Set

Sunday, April 3, 2022

9:20 PM

Iterators

Functions	Description
<code>Begin</code>	Returns an iterator pointing to the first element in the set.
<code>cbegin</code>	Returns a const iterator pointing to the first element in the set.
<code>End</code>	Returns an iterator pointing to the past-end.
<code>Cend</code>	Returns a constant iterator pointing to the past-end.
<code>rbegin</code>	Returns a reverse iterator pointing to the end.
<code>Rend</code>	Returns a reverse iterator pointing to the beginning.
<code>crbegin</code>	Returns a constant reverse iterator pointing to the end.
<code>Crend</code>	Returns a constant reverse iterator pointing to the beginning.

Capacity

Functions	Description
<code>empty</code>	Returns true if set is empty.
<code>Size</code>	Returns the number of elements in the set.
<code>max_size</code>	Returns the maximum size of the set.

Modifiers

Functions	Description
<code>insert</code>	Insert element in the set.
<code>Erase</code>	Erase elements from the set.
<code>Swap</code>	Exchange the content of the set.
<code>Clear</code>	Delete all the elements of the set.
<code>emplace</code>	Construct and insert the new elements into the set.
<code>emplace_hint</code>	Construct and insert new elements into the set by hint.

Operations

Functions	Description
<code>Find</code>	Search for an element with given key.
<code>count</code>	Gets the number of elements matching with given key.
<code>lower_bound</code>	Returns an iterator to lower bound.
<code>upper_bound</code>	Returns an iterator to upper bound.
<code>equal_range</code>	Returns the range of elements matches with given key.

```
#include <iostream>
#include <set>

using namespace std;

int main ()
{
    set<int> myset;
    set<int>::iterator itlow,itup;

    for (int i=1; i<10; i++) myset.insert(i*10); // 10 20 30 40 50 60 70 80 90

    itlow=myset.lower_bound (30);           //      ^
    itup=myset.upper_bound (60);           //      ^

    myset.erase(itlow,itup);               // 10 20 70 80 90

    std::cout << "myset contains: ";
    for (set<int>::iterator it=myset.begin(); it!=myset.end(); ++it)
        cout << ' ' << *it;
    cout << '\n';

    return 0;
}
```

Output:

```
myset contains: 10 20 70 80 90
```

CPP Queue

Monday, April 4, 2022 11:29 AM

First In First Out

Front => First position

Read => Last position

Function	Description
(constructor)	The function is used for the construction of a queue container.
empty	The function is used to test for the emptiness of a queue. If the queue is empty the function returns true else false.
size	The function returns the size of the queue container, which is a measure of the number of elements stored in the queue.
front	The function is used to access the front element of the queue. The element plays a very important role as all the deletion operations are performed at the front element.
back	The function is used to access the rear element of the queue. The element plays a very important role as all the insertion operations are performed at the rear element.
push	The function is used for the insertion of a new element at the rear end of the queue.
pop	The function is used for the deletion of element; the element in the queue is deleted from the front end.
emplace	The function is used for insertion of new elements in the queue above the current rear element.
swap	The function is used for interchanging the contents of two containers in reference.
relational operators	The non member function specifies the relational operators that are needed for the queues.
uses allocator<queue>	As the name suggests the non member function uses the allocator for the queues.

```
#include <iostream>
#include <queue>
using namespace std;
void showsg(queue <int> sg)
{
    queue <int> ss = sg;
    while (!ss.empty())
    {
        cout << '\t' << ss.front();
        ss.pop();
    }
    cout << '\n';
}
```

```

int main()
{
    queue <int> fquiz;
    fquiz.push(10);
    fquiz.push(20);
    fquiz.push(30);

    cout << "The queue fquiz is : ";
    showsg(fquiz);

    cout << "\nfquiz.size() : " << fquiz.size();
    cout << "\nfquiz.front() : " << fquiz.front();
    cout << "\nfquiz.back() : " << fquiz.back();

    cout << "\nfquiz.pop() : ";
    fquiz.pop();
    showsg(fquiz);

    return 0;
}

```

Output:

```

The queue fquiz is :    10    20    30

fquiz.size() : 3
fquiz.front() : 10
fquiz.back() : 30
fquiz.pop() :    20    30

```

C++ Priority Queue

Monday, April 4, 2022 11:35 AM

It is similar to the ordinary queue in certain aspects but differs in the following ways:

- o In a priority queue, every element in the queue is associated with some priority, but priority does not exist in a queue data structure.
- o The element with the highest priority in a priority queue will be removed first while queue follows the **FIFO(First-In-First-Out)** policy means the element which is inserted first will be deleted first.
- o If more than one element exists with the same priority, then the order of the element in a queue will be taken into consideration.

Operation	Priority Queue	Return Value
Push(1)	1	
Push(4)	1 4	
Push(2)	1 4 2	
Pop()	1 2	4
Push(3)	1 2 3	

Function	Description
push()	It inserts a new element in a priority queue.
pop()	It removes the top element from the queue, which has the highest priority.
top()	This function is used to address the topmost element of a priority queue.
size()	It determines the size of a priority queue.
empty()	It verifies whether the queue is empty or not. Based on the verification, it returns the status.
swap()	It swaps the elements of a priority queue with another queue having the same type and size.
emplace()	It inserts a new element at the top of the priority queue.

```
#include <iostream>
#include <queue>
using namespace std;
int main()
{
    priority_queue<int> p; // variable declaration.
    p.push(10); // inserting 10 in a queue, top=10
    p.push(30); // inserting 30 in a queue, top=30
    p.push(20); // inserting 20 in a queue, top=20
    cout << "Number of elements available in 'p' : " << p.size() << endl;
    while(!p.empty())
    {
        std::cout << p.top() << std::endl;
        p.pop();
    }
    return 0;
}
```



```
Number of elements available in 'p' :3
```

```
30
```

```
20
```

```
10 zzzzz/
```

CPP Map

Monday, April 4, 2022

11:50 AM

```
#include <string.h>
#include <iostream>
#include <map>
#include <utility>
using namespace std;
int main()
{
    map<int, string> Employees;
    // 1) Assignment using array index notation
    Employees[101] = "Nikita";
    Employees[105] = "John";
    Employees[103] = "Dolly";
    Employees[104] = "Deep";
    Employees[102] = "Aman";
    cout << "Employees[104]=" << Employees[104] << endl << endl;
    cout << "Map size: " << Employees.size() << endl;
    cout << endl << "Natural Order:" << endl;
    for( map<int,string>::iterator ii=Employees.begin(); ii!=Employees.end(); ++ii)
    {
        cout << (*ii).first << ": " << (*ii).second << endl;
    }
    cout << endl << "Reverse Order:" << endl;
    for( map<int,string>::reverse_iterator ii=Employees.rbegin(); ii!=Employees.rend(); ++ii)
    {
        cout << (*ii).first << ": " << (*ii).second << endl;
    }
}
```

Employees[104]=Deep

Map size: 5

Natural Order:

101: Nikita

102: Aman

103: Dolly

104: Deep

105: John

Reverse Order:

105: John

104: Deep

103: Dolly

102: Aman

101: Nikita

Iterators

Functions	Description
<code>begin</code>	Returns an iterator pointing to the first element in the map.
<code>cbegin</code>	Returns a const iterator pointing to the first element in the map.
<code>end</code>	Returns an iterator pointing to the past-end.
<code>cend</code>	Returns a constant iterator pointing to the past-end.
<code>rbegin</code>	Returns a reverse iterator pointing to the end.
<code>rend</code>	Returns a reverse iterator pointing to the beginning.
<code>crbegin</code>	Returns a constant reverse iterator pointing to the end.
<code>crend</code>	Returns a constant reverse iterator pointing to the beginning.

Capacity

Functions	Description
<code>empty</code>	Returns true if map is empty.
<code>size</code>	Returns the number of elements in the map.
<code>max_size</code>	Returns the maximum size of the map.

```
#include <iostream>
#include <map>

using namespace std;

int main() {
    map<char, int> m = {
        {'a', 1},
        {'b', 2},
        {'c', 3},
    };

    // inserting new element
    m.insert(pair<char, int>('d', 4));
    m.insert(pair<char, int>('e', 5));

    cout << "Map contains following elements" << endl;

    for (auto it = m.begin(); it != m.end(); ++it)
        cout << it->first << " = " << it->second << endl;

    return 0;
}
```

```

#include <iostream>
#include <map>

using namespace std;

int main(void) {
    map<char, int> m = {
        {'b', 2},
        {'c', 3},
        {'d', 4},
    };

    //inserting element with the given position
    m.insert(m.begin(), pair<char, int>('a', 1));
    m.insert(m.end(), pair<char, int>('e', 5));

    cout << "Map contains following elements" << endl;

    for (auto it = m.begin(); it != m.end(); ++it)
        cout << it->first << " = " << it->second << endl;

    return 0;
}

```

```

#include <iostream>
#include <map>

using namespace std;

int main() {

    map<char, int> m1 = {
        {'a', 1},
        {'b', 2},
        {'c', 3},
        {'d', 4},
        {'e', 5},
    };

    map<char, int> m2; // creating new map m2
    m2.insert(m1.begin(), m1.end()); //inserting the elements of m1 to m2 from begin to end

    cout << "Map contains following elements" << endl;

    for (auto it = m2.begin(); it != m2.end(); ++it)
        cout << it->first << " = " << it->second << endl;

    return 0;
}

```

```

#include <iostream>
#include <map>

using namespace std;

int main(void) {
    map<int, string> m = {
        {1, "Java"},
        {2, "C++"},
        {3, "SQL"},
    };

    m.insert({{4, "VB"}, {5, "Oracle"}});

    cout << "Map contains following elements" << endl;

    for (auto it = m.begin(); it != m.end(); ++it)
        cout << it->first << " : " << it->second << endl;

    return 0;
}

```

Modifiers

Functions	Description
insert	Insert element in the map.
erase	Erase elements from the map.
swap	Exchange the content of the map.
clear	Delete all the elements of the map.
emplace	Construct and insert the new elements into the map.
emplace_hint	Construct and insert new elements into the map by hint.

CPP Multimap

Monday, April 4, 2022 12:03 PM

Multimaps are part of the **C++ STL (Standard Template Library)**. Multimaps are the associative containers like map that stores sorted key-value pair, but unlike maps which store only unique keys, **multimap can have duplicate keys**. By default it uses < operator to compare the keys.

```
#include <iostream>
#include <map>
#include <string>

using namespace std;

int main()
{
    multimap<string, string> m = {
        {"India", "New Delhi"},
        {"India", "Hyderabad"},
        {"United Kingdom", "London"},
        {"United States", "Washington D.C"}
    };

    cout << "Size of map m: " << m.size() << endl;
    cout << "Elements in m: " << endl;

    for (multimap<string, string>::iterator it = m.begin(); it != m.end(); ++it)
    {
        cout << " [" << (*it).first << ", " << (*it).second << "]" << endl;
    }

    return 0;
}
```

Output:

```
Size of map m: 4
Elements in m:
[India, New Delhi]
[India, Hyderabad]
[United Kingdom, London]
[United States, Washington D.C]
```

Iterators

Functions	Description
<code>begin</code>	Returns an iterator pointing to the first element in the multimap.
<code>cbegin</code>	Returns a const_iterator pointing to the first element in the multimap.
<code>end</code>	Returns an iterator pointing to the past-end.
<code>cend</code>	Returns a constant iterator pointing to the past-end.
<code>rbegin</code>	Returns a reverse iterator pointing to the end.
<code>rend</code>	Returns a reverse iterator pointing to the beginning.
<code>crbegin</code>	Returns a constant reverse iterator pointing to the end.
<code>crend</code>	Returns a constant reverse iterator pointing to the beginning.

Capacity

Functions	Description
<code>empty</code>	Return true if multimap is empty.
<code>size</code>	Returns the number of elements in the multimap.
<code>max_size</code>	Returns the maximum size of the multimap.

Modifiers

Functions	Description
<code>insert</code>	Insert element in the multimap.
<code>erase</code>	Erase elements from the multimap.
<code>swap</code>	Exchange the content of the multimap.
<code>clear</code>	Delete all the elements of the multimap.
<code>emplace</code>	Construct and insert the new elements into the multimap.
<code>emplace_hint</code>	Construct and insert new elements into the multimap by hint.

Operations

Functions	Description
<code>find</code>	Search for an element with given key.
<code>count</code>	Gets the number of elements matching with given key.
<code>lower_bound</code>	Returns an iterator to lower bound.
<code>upper_bound</code>	Returns an iterator to upper bound.
<code>equal_range()</code>	Returns the range of elements matches with given key.

CPP Upper bound and lower bound

Monday, April 4, 2022 12:21 PM

Upper bound and lower bound => [Upper Bound and Lower Bound in C++ STL | CP Course | EP 35](#)

CPP Bit Set functions

Monday, April 4, 2022

12:31 PM

Function
<code>all()</code>
<code>any()</code>
<code>count()</code>
<code>flip()</code>
<code>none()</code>
<code>operator[]</code>
<code>reset()</code>
<code>set()</code>
<code>size()</code>
<code>test()</code>
<code>to_string()</code>
<code>to_ullong()</code>
<code>to_ulong()</code>

CPP Algorithmic Operations

Monday, April 4, 2022 12:36 PM

Non-modifying sequence operations:

Function	Description
<code>all_of</code>	The following function tests a condition to all the elements of the range.
<code>any_of</code>	The <code>following</code> function tests a condition to some or any of the elements of the range
<code>none_of</code>	The following function checks if none of the elements follow the condition or not.
<code>for_each</code>	The function applies an operation to all the elements of the range.
<code>find</code>	The function finds a value in the range.
<code>find_if</code>	The function finds for an element in the range.
<code>find_if_not</code>	The function finds an element in the range but in the opposite way as the above one.
<code>find_end</code>	The function is used to return the last element of the range.
<code>find_first_of</code>	The function finds for the element that satisfies a condition and occurs at the first.
<code>adjacent_find</code>	The function makes a search for finding the equal and adjacent elements in a range.
<code>count</code>	The function returns the count of a value in the range.
<code>count_if</code>	The function returns the count of values that satisfies a condition.
<code>mismatch</code>	The function returns the value in sequence which is the first mismatch.
<code>equal</code>	The function is used to check if the two ranges have all elements equal.
<code>is_permutation</code>	The function checks whether the range in reference is a permutation of some other range.
<code>search</code>	The function searches for the subsequence in a range.
<code>search_n</code>	The function searches the range for the occurrence of an element.

```
#include<iostream>
#include<algorithm>
#include<array>
int main()
{
    std::array<int, 6> arr= {25,27,29,31,33,35};
    if ( std::all_of(arr.begin(), arr.end(), [](int k) {return k%2;} ) )
        std::cout << "All the array elements are odd.";
    return 0;
}
```

Output:

```
All the array elements are odd.
```

```

#include <iostream>
#include <algorithm>
#include <array>
using namespace std;
int main()
{
    int arr[7] = {2,4,6,5,10,3,14};
    any_of(arr,arr+6, [](int k){return k%2;})?
    cout << "There are elements which exist in the table of 2":
    cout << "No elements in the table of 2 exists";
    return 0;
}

```

Output:

```
There are elements which exist in the table of 2.
```

```

#include <iostream>
#include <algorithm>
#include <array>
int main()
{
    std::array<int, 6> arr= {25,27,29,31,33,35};
    if ( std::none_of(arr.begin(), arr.end(), [](int k) {return k%2==0;} ) )
    std::cout << "None of the elements is divisible by 2";
    return 0;
}

```

Output:

```
None of the elements is divisible by 2
```

```

#include <iostream>
#include <algorithm>
#include <vector>
void newfunction (int k)
{
    std::cout << " " <<k;
}
struct newclass
{
    void operator () (int k)
    {
        std::cout << " " <<k;
    }
}
newobject;
int main()
{
    std::vector<int> newvector;
    newvector.push_back(50);
    newvector.push_back(100);
    newvector.push_back(150);
    std::cout << "newvector contains:\n";
    for_each (newvector.begin () , newvector.end (), newfunction);
    std::cout<< "\n newvector contains:\n";
    for_each (newvector.begin (), newvector.end(), newfunction);
    std::cout<< "\n";
    return 0;
}

```

Output:

```

newvector contains: 50 100 150
newvector contains: 50 100 150

```

```

#include <iostream>
#include <algorithm>
#include <vector>
int main ()
{
    int newints[] = { 50, 60, 70, 80 };
    int *q;
    q = std::find (newints, newints+4, 60);
    if (q != newints+4)
        std::cout << "Element found in newints: " << *q << '\n';
    else
        std::cout << "Element not found in newints\n";
    std::vector<int> newvector (newints,newints+4);
    std::vector<int>::iterator ti;
    ti = find (newvector.begin(), newvector.end(), 60);
    if (ti != newvector.end())
        std::cout << "Element found in newvector: " << *ti << '\n';
    else
        std::cout << "Element not found in newvector\n";
    return 0;
}

```

Output:

```

Elements that are found in newints: 60
Elements that are found in newvector: 60

```

```

#include<iostream>
#include<algorithm>
#include<vector>
using namespace std;
bool isAnOdd(int i)
{
    return((i%2) == 1);
}
int main()
{
    std::vector<int> newvector;
    newvector.push_back(20);
    newvector.push_back(35);
    newvector.push_back(50);
    newvector.push_back(65);
    std::vector<int>::iterator ti = std::find_if(newvector.begin(), newvector.end(),isAnOdd);
    std::cout<<"Out of the given elements, first odd element is "<<*ti<<"\n";
    return 0;
}

```

Output:

```

Out of the given elements, first odd element is 35

```

Modifying sequence operations

Function	Description
<code>copy</code>	The function copies the range of elements.
<code>copy_n</code>	The function copies n elements of the range
<code>copy_if</code>	The function copies the elements of the range if a certain condition is fulfilled.
<code>copy_backward</code>	The function copies the elements in a backward order
<code>move</code>	The function moves the ranges of elements.
<code>move_backward</code>	The function moves the range of elements in the backward order
<code>swap</code>	The function swaps the value of two objects.
<code>swap_ranges</code>	The function swaps the value of two ranges.
<code>iter_swap</code>	The function swaps the values of two iterators under reference.
<code>transform</code>	The function transforms all the values in a range.
<code>replace</code>	The function replaces the values in the range with a specific value.
<code>replace_if</code>	The function replaces the value of the range if a certain condition is fulfilled.
<code>replace_copy</code>	The function copies the range of values by replacing with an element.
<code>replace_copy_if</code>	The function copies the range of values by replacing with an element if a certain condition is fulfilled.
<code>fill</code>	The function fills the values in the range with a value.
<code>fill_n</code>	The function fills the values in the sequence.
<code>generate</code>	The function is used for the generation of values of the range.
<code>generate_n</code>	The function is used for the generation of values of the sequence.
<code>remove</code>	The function removes the values from the range.
<code>remove_if</code>	The function removes the values of the range if a condition is fulfilled.
<code>remove_copy</code>	The function copies the values of the range by removing them.
<code>remove_copy_if</code>	The function copies the values of the range by removing them if a condition is fulfilled.
<code>unique</code>	The function identifies the unique element of the range.
<code>unique_copy</code>	The function copies the unique elements of the range.
<code>reverse</code>	The function reverses the range.
<code>reverse_copy</code>	The function copies the range by reversing values.
<code>rotate</code>	The function rotates the elements of the range in left direction.
<code>rotate_copy</code>	The function copies the elements of the range which is rotated left.
<code>random_shuffle</code>	The function shuffles the range randomly.
<code>shuffle</code>	The function shuffles the range randomly with the help of a generator.

Partitions

Function	Description
<code>is_partitioned</code>	The function is used to deduce whether the range is partitioned or not.
<code>partition</code>	The function is used to partition the range.
<code>stable_partition</code>	The function partitions the range in two stable halves.
<code>partition_copy</code>	The function copies the range after partition.
<code>partition_point</code>	The function returns the partition point for a range.

Sorting

Function	Description
<code>sort</code>	The function sorts all the elements in a range.
<code>stable_sort</code>	The function sorts the elements in the range maintaining the relative equivalent order.
<code>partial_sort</code>	The function partially sorts the elements of the range.
<code>partial_sort_copy</code>	The function copies the elements of the range after sorting it.
<code>is_sorted</code>	The function checks whether the range is sorted or not.
<code>is_sorted_until</code>	The function checks till which element a range is sorted.
<code>nth_element</code>	The functions sorts the elements in the range.

Binary search

Function	Description
<code>lower_bound</code>	Returns the lower bound element of the range.
<code>upper_bound</code>	Returns the upper bound element of the range.
<code>equal_range</code>	The function returns the subrange for the equal elements.
<code>binary_search</code>	The function tests if the values in the range exists in a sorted sequence or not.

Merge

Function	Description
<code>merge</code>	The function merges two ranges that are in a sorted order.
<code>inplace_merge</code>	The function merges two consecutive ranges that are sorted.
<code>includes</code>	The function searches whether the sorted range includes another range or not.
<code>set_union</code>	The function returns the union of two ranges that is sorted.
<code>set_intersection</code>	The function returns the intersection of two ranges that is sorted.
<code>set_difference</code>	The function returns the difference of two ranges that is sorted.
<code>set_symmetric_difference</code>	The function returns the symmetric difference of two ranges that is sorted.

Heap

Function	Description
push_heap	The function pushes new elements in the heap.
pop_heap	The function pops new elements in the heap.
make_heap	The function is used for the creation of a heap.
sort_heap	The function sorts the heap.
is_heap	The function checks whether the range is a heap.
is_heap_until	The function checks till which position a range is a heap.

Min/Max

Function	Description
min	Returns the smallest element of the range.
max	Returns the largest element of the range.
minmax	Returns the smallest and largest element of the range.
min_element	Returns the smallest element of the range.
max_element	Returns the largest element of the range.
minmax_element	Returns the smallest and largest element of the range.

Other functions

Function	Description
lexicographical_compare	The function performs the lexicographical less-than comparison.
next_permutation	The function is used for the transformation of range into the next permutation.
prev_permutation	The function is used for the transformation of range into the previous permutation.

CPP Object Class

Wednesday, April 6, 2022 6:56 AM

- Main objective is to introduce OOPS to the C programming language.
- Small talk is first programming language.

OOPs (Object Oriented Programming System)

Object means a real world entity such as pen, chair, table etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Object

Any entity that has state and behavior is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.

Class

Collection of objects is called class. It is a logical entity.

Inheritance

When one object acquires all the properties and behaviours of parent object i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

Polymorphism

When **one task is performed by different ways** i.e. known as polymorphism. For example: to convince the customer differently, to draw something e.g. shape or rectangle etc.

In C++, we use Function overloading and Function overriding to achieve polymorphism.

Abstraction

Hiding internal details and showing functionality is known as abstraction. For example: phone call, we don't know the internal processing.

In C++, we use abstract class and interface to achieve abstraction.

Encapsulation

Binding (or wrapping) code and data together into a single unit is known as encapsulation. For example: capsule, it is wrapped with different medicines.

Advantage of OOPs over Procedure-oriented programming language

1. OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grows.
2. OOPs provide data hiding whereas in Procedure-oriented programming language a global data can be accessed from anywhere.
3. OOPs provide ability to simulate real-world event much more effectively. We can provide the solution of real world problem if we are using the Object-Oriented Programming language.

Example 1:

```
#include <iostream>
using namespace std;
class Student {
public:
    int id;//data member (also instance variable)
    string name;//data member(also instance variable)
};
int main() {
    Student s1; //creating an object of Student
    s1.id = 201;
    s1.name = "Sonoo Jaiswal";
    cout<<s1.id<<endl;
    cout<<s1.name<<endl;
    return 0;
}
```

```
#include <iostream>
using namespace std;
class Student {
public:
    int id;//data member (also instance variable)
    string name;//data member(also instance variable)
    void insert(int i, string n)
    {
        id = i;
        name = n;
    }
    void display()
    {
        cout<<id<<" "<<name<<endl;
    }
};
int main(void) {
    Student s1; //creating an object of Student
    Student s2; //creating an object of Student
    s1.insert(201, "Sonoo");
    s2.insert(202, "Nakul");
    s1.display();
    s2.display();
    return 0;
}
```

Output:

```
201 Sonoo
202 Nakul
```

C++ Constructor

In C++, constructor is a special method which is invoked automatically at the time of object creation. It is used to initialize the data members of new object generally. The constructor in C++ has the same name as class or structure.

There can be two types of constructors in C++.

- Default constructor
- Parameterized constructor

Default Constructor

```
#include <iostream>
```

```
#include <iostream>
using namespace std;
class Employee
{
public:
    Employee()
    {
        cout<<"Default Constructor Invoked"<<endl;
    }
};
int main(void)
{
    Employee e1; //creating an object of Employee
    Employee e2;
    return 0;
}
```

Output:

```
Default Constructor Invoked
Default Constructor Invoked
```

Parameterized Constructor

- used to provide different values to distinct objects.

```
#include <iostream>
using namespace std;
class Employee {
public:
    int id;//data member (also instance variable)
    string name;//data member(also instance variable)
    float salary;
    Employee(int i, string n, float s)
    {
        id = i;
        name = n;
        salary = s;
    }
    void display()
    {
        cout<<id<<" "<<name<<" "<<salary<<endl;
    }
};
int main(void) {
    Employee e1 =Employee(101, "Sonoo", 890000); //creating an object of Employee
    Employee e2=Employee(102, "Nakul", 59000);
    e1.display();
    e2.display();
    return 0;
}
```

Output:

```
101 Sonoo 890000
102 Nakul 59000
```

CPP Destructor (must starts with ~)(automatically called)

```

#include <iostream>
using namespace std;
class Employee
{
    public:
        Employee()
        {
            cout<<"Constructor Invoked"<<endl;
        }
        ~Employee()
        {
            cout<<"Destructor Invoked"<<endl;
        }
};
int main(void)
{
    Employee e1; //creating an object of Employee
    Employee e2; //creating an object of Employee
    return 0;
}

```

```

Constructor Invoked
Constructor Invoked
Destructor Invoked
Destructor Invoked

```

C++ this Pointer

In C++ programming, **this** is a keyword that refers to the current instance of the class. There can be 3 main usage of this keyword

- It can be used **to pass current object as a parameter to another method.**
- It can be used **to refer current class instance variable.**
- It can be used **to declare indexers.**

```

#include <iostream>
using namespace std;
class Employee {
public:
    int id; //data member (also instance variable)
    string name; //data member(also instance variable)
    float salary;
    Employee(int id, string name, float salary)
    {
        this->id = id;
        this->name = name;
        this->salary = salary;
    }
    void display()
    {
        cout<<id<<" "<<name<<" "<<salary<<endl;
    }
};

int main(void) {
    Employee e1 =Employee(101, "Sonoo", 890000); //creating an object of Employee
    Employee e2=Employee(102, "Nakul", 59000); //creating an object of Employee
    e1.display();
    e2.display();
    return 0;
}

```

```

101 Sonoo 890000
102 Nakul 59000

```

C++ static

In C++, static is a keyword or modifier that belongs to the type not instance. So instance is not required to access the static member can be field, method, constructor, class, properties, operator and event.

A field which is declared as static is called static field. Unlike instance field which gets memory each time whenever you create object, there is only one copy of static field created in the memory. It is shared to all the objects.

```

#include <iostream>
using namespace std;
class Account {
public:
    int accno; //data member (also instance variable)
    string name; //data member(also instance variable)
    static float rateOfInterest;
    Account(int accno, string name)
    {
        this->accno = accno;
        this->name = name;
    }
    void display()
    {
        cout<<accno<< " "<<name<< " "<<rateOfInterest<<endl;
    }
};
float Account::rateOfInterest=6.5;
int main(void) {
    Account a1 =Account(201, "Sanjay"); //creating an object of Employee
    Account a2=Account(202, "Nakul"); //creating an object of Employee
    a1.display();
    a2.display();
    return 0;
}

```

Output:

```
201 Sanjay 6.5
```

```
202 Nakul 6.5
```

CPP STRUCTS

In C++, classes and structs are blueprints that are used to create the instance of a class. Structs are used for lightweight objects such as Rectangle, color, Point, etc.

Unlike class, structs in C++ are value type than reference type. It is useful if you have data that is not intended to be modified after creation of struct.

```

#include <iostream>
using namespace std;
struct Rectangle
{
    int width, height;
};
int main(void) {
    struct Rectangle rec;
    rec.width=8;
    rec.height=5;
    cout<<"Area of Rectangle is: "<<(rec.width * rec.height)<<endl;
    return 0;
}

```

Output:

```
Area of Rectangle is: 40
```

C++ Struct Example: Using Constructor and Method

Let's see another example of struct where we are using the constructor to initialize data and method to calculate the area of rectangle.

```
#include <iostream>
using namespace std;
struct Rectangle {
    int width, height;
    Rectangle(int w, int h)
    {
        width = w;
        height = h;
    }
    void areaOfRectangle() {
        cout << "Area of Rectangle is: " << (width*height);
    };
};
int main(void) {
    struct Rectangle rec=Rectangle(4,6);
    rec.areaOfRectangle();
    return 0;
}
```

Output:

```
Area of Rectangle is: 24
```

Class

If access specifier is not declared explicitly, then by default access specifier will be private.

C++ Enumeration

Enum in C++ is a data type that contains fixed set of constants.

It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY and SATURDAY) , directions EAST and WEST) etc. The C++ enum constants are static and final implicitly.

C++ Enums can be thought of as classes that have fixed set of constants.

```
#include <iostream>
using namespace std;
enum week { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday };
int main()
{
    week day;
    day = Friday;
    cout << "Day: " << day+1 << endl;
    return 0;
}
```

Output:

```
Day: 5
```

C++ Friend function

If a function is defined as a friend function in C++, then the protected and private data of a class can be accessed using the function.

By using the keyword friend compiler knows the given function is a friend function.

For accessing the data, the declaration of a friend function should be done inside the body of a class starting with the keyword friend.

Characteristics of a Friend function:

- The function is not in the scope of the class to which it has been declared as a friend.
- It cannot be called using the object as it is not in the scope of that class.
- It can be invoked like a normal function without using the object.
- It cannot access the member names directly and has to use an object name and dot membership operator with the member name.
- It can be declared either in the private or the public part.

C++ friend function Example

```
#include <iostream>
using namespace std;
class B;      // forward declaration.
class A
{
    int x;
public:
    void setdata(int i)
    {
        x=i;
    }
    friend void min(A,B);    // friend function.
};
class B
{
    int y;
public:
    void setdata(int i)
    {
        y=i;
    }
    friend void min(A,B);    // friend function
};
void min(A a,B b)
{
    if(a.x<=b.y)
        std::cout << a.x << std::endl;
    else
        std::cout << b.y << std::endl;
}
int main()
{
    A a;
    B b;
    a.setdata(10);
    b.setdata(20);
    min(a,b);
    return 0;
}
```

Output:

```
10
```


C++ Friend class

A friend class can access both private and protected members of the class in which it has been declared as friend.

```
#include <iostream>

using namespace std;

class A
{
    int x =5;
    friend class B;    // friend class.
};

class B
{
public:
    void display(A &a)
    {
        cout<<"value of x is : "<<a.x;
    }
};

int main()
{
    A a;
    B b;
    b.display(a);
    return 0;
}
```

Output:

```
value of x is : 5
```

In the above example, class B is declared as a friend inside the class A. Therefore, B is a friend of class A. Class B can access the private members of class A.

```
//The most common use of polymorphism is when a
//parent class reference is used to refer to a child class object
```

C++ OOPS

Wednesday, April 6, 2022 1:12 PM

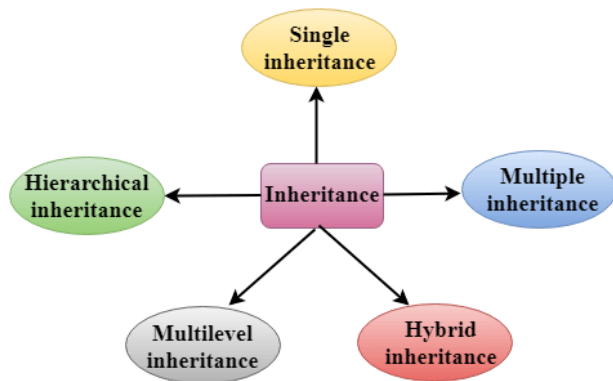
C++ Inheritance

In C++, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviors which are defined in other class.

In C++, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

Advantage of C++ Inheritance

Code reusability: Now you can reuse the members of your parent class. So, there is no need to define the member again. So less code is required in the class.



When the base class is privately inherited by the derived class, public members of the base class becomes the private members of the derived class. Therefore, the public members of the base class are not accessible by the objects of the derived class only by the member functions of the derived class.

When the base class is publicly inherited by the derived class, public members of the base class also become the public members of the derived class. Therefore, the public members of the base class are accessible by the objects of the derived class as well as by the member functions of the base class.

C++ Single Level inheritance

```
#include <iostream>
using namespace std;
class Account {
public:
    float salary = 60000;
};
class Programmer: public Account {
public:
    float bonus = 5000;
};
int main(void) {
    Programmer p1;
    cout<<"Salary: "<<p1.salary<<endl;
    cout<<"Bonus: "<<p1.bonus<<endl;
    return 0;
}
```

Output:

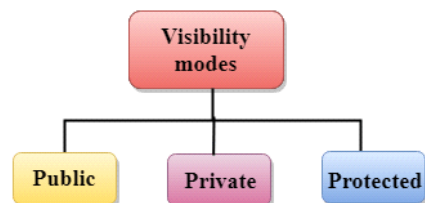
```
Salary: 60000
Bonus: 5000
```

```
#include <iostream>
using namespace std;
class Animal {
public:
void eat() {
    cout<<"Eating..."<<endl;
}
};
class Dog: public Animal
{
public:
void bark(){
    cout<<"Barking...";
}
};
int main(void) {
    Dog d1;
    d1.eat();
    d1.bark();
    return 0;
}
```

```
Eating...
Barking...
```

C++ introduces a third visibility modifier, i.e., **protected**. The member which is declared as protected will be accessible to all the member functions within the class as well as the class immediately derived from it.

Visibility modes can be classified into three categories:



- **Public:** When the member is declared as public, it is accessible to all the functions of the program.
- **Private:** When the member is declared as private, it is accessible within the class only.
- **Protected:** When the member is declared as protected, it is accessible within its own class as well as the class immediately derived from it.

Visibility of Inherited Members

Base class visibility	Derived class visibility		
	Public	Private	Protected
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

CPP Multi Level Inheritance

```

#include <iostream>
using namespace std;
class Animal {
public:
void eat() {
cout<<"Eating..."<<endl;
}
};
class Dog: public Animal
{
public:
void bark(){
cout<<"Barking..."<<endl;
}
};
class BabyDog: public Dog
{
public:
void weep() {
cout<<"Weeping...";
}
};
int main(void) {
BabyDog d1;
d1.eat();
d1.bark();
d1.weep();
return 0;
}

```

Output:

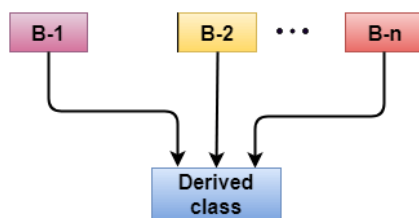
```

Eating...
Barking...
Weeping...

```

CPP Multiple Inheritance

Multiple inheritance is the process of deriving a new class that inherits the attributes from two or more classes.



```

#include <iostream>
using namespace std;
class A
{
    protected:
        int a;
    public:
        void get_a(int n)
        {
            a = n;
        }
};

class B
{
    protected:
        int b;
    public:
        void get_b(int n)
        {
            b = n;
        }
};

class C : public A, public B
{
    public:
        void display()
        {
            std::cout << "The value of a is : " << a << std::endl;
            std::cout << "The value of b is : " << b << std::endl;
            cout << "Addition of a and b is : " << a + b;
        }
};

int main()
{
    C c;
    c.get_a(10);
    c.get_b(20);
    c.display();

    return 0;
}

```

Output:

```

The value of a is : 10
The value of b is : 20
Addition of a and b is : 30

```

Be careful while performing multiple inheritance, because we get an error if we inherit two classes which having same function signature and if try to access it we would get an ambiguity error. And if we face that issue , we can solve it using class resolution operator "::" as the following.

- o The above issue can be resolved by using the class resolution operator with the function. In the above example, the derived class code can be rewritten as:

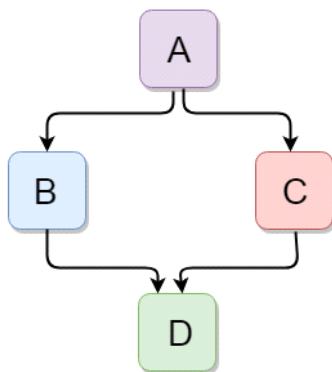
```
class C : public A, public B
{
    void view()
    {
        A :: display();    // Calling the display() function of class A.
        B :: display();    // Calling the display() function of class B.
    }
};
```

- We can face such ambiguity even in single inheritance.

```
int main()
{
    B b;
    b.display();    // Calling the display() function of B class.
    b.B :: display(); // Calling the display() function defined in B class.
}
```

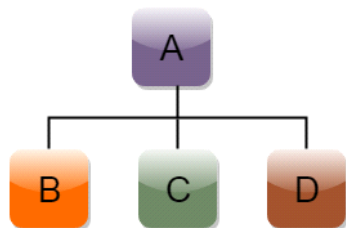
C++ Hybrid Inheritance

Hybrid inheritance is a combination of more than one type of inheritance.



C++ Hierarchical Inheritance

Hierarchical inheritance is defined as the process of deriving more than one class from a base class.



C++ Aggregation (HAS-A Relationship)

In C++, aggregation is a process in which one class defines another class as any entity reference. It is another way to reuse the class. It is a form of association that represents HAS-A relationship.

Let's see an example of aggregation where Employee class has the reference of Address class as data member. In such way, it can reuse the members of Address class.

```
#include <iostream>
using namespace std;
class Address {
public:
    string addressLine, city, state;
    Address(string addressLine, string city, string state)
    {
        this->addressLine = addressLine;
        this->city = city;
        this->state = state;
    }
};

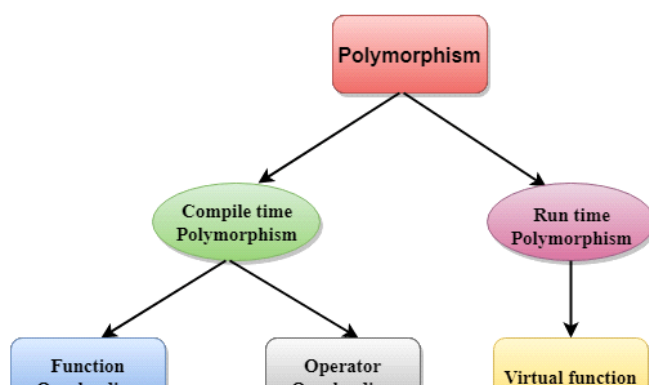
class Employee
{
private:
    Address* address; //Employee HAS-A Address
public:
    int id;
    string name;
    Employee(int id, string name, Address* address)
    {
        this->id = id;
        this->name = name;
        this->address = address;
    }
    void display()
    {
        cout<<id <<" "<<name<<" "<<
        address->addressLine<<" "<< address->city<<" "<<address->state<<endl;
    }
};

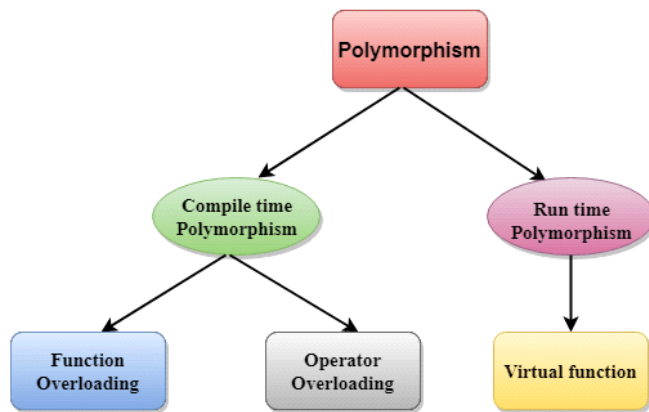
int main(void) {
    Address a1= Address("C-146, Sec-15","Noida","UP");
    Employee e1 = Employee(101,"Nakul",&a1);
    e1.display();
    return 0;
}
```

Output:

```
101 Nakul C-146, Sec-15 Noida UP
```

CPP Polymorphism





Compile time polymorphism: The overloaded functions are invoked by matching the type and number of arguments. This information is available at the compile time and, therefore, compiler selects the appropriate function at the compile time. It is achieved by function overloading and operator overloading which is also known as static binding or early binding. Now, let's consider the case where function name and prototype is same.

Function overloading: same function name with different number or types of parameters. **Operator overloading:** A + can add two int and two float and two string as a single operator perform different functions it is operator overloading.

Virtual Functions: This is in general called function overriding or dynamic binding or late binding. This generally occurs when we are having a function with same name and same number of parameters but perform different activities and exists in different level of inherited classes.

Differences b/w compile time and run time polymorphism.

Compile time polymorphism	Run time polymorphism
The function to be invoked is known at the compile time.	The function to be invoked is known at the run time.
It is also known as overloading, early binding and static binding.	It is also known as overriding, Dynamic binding and late binding.
Overloading is a compile time polymorphism where more than one method is having the same name but with the different number of parameters or the type of the parameters.	Overriding is a run time polymorphism where more than one method is having the same name, number of parameters and the type of the parameters.
It is achieved by function overloading and operator overloading.	It is achieved by virtual functions and pointers.
It provides fast execution as it is known at the compile time.	It provides slow execution as it is known at the run time.
It is less flexible as mainly all the things execute at the compile time.	It is more flexible as all the things execute at the run time.


```

#include <iostream>
using namespace std;
class Animal {
public:
void eat(){
cout<<"Eating...";
}
};
class Dog: public Animal
{
public:
void eat()
{
    cout<<"Eating bread...";
}
};
int main(void) {
    Dog d = Dog();
    d.eat();
    return 0;
}

```

Output:

```
Eating bread...
```

```

#include <iostream>
using namespace std;
class Shape { // base class
public:
virtual void draw(){ // virtual function
cout<<"drawing..."<<endl;
}
};
class Rectangle: public Shape // inheriting Shape class.
{
public:
void draw()
{
    cout<<"drawing rectangle..."<<endl;
}
};
class Circle: public Shape // inheriting Shape class.
{
public:
void draw()
{
    cout<<"drawing circle..."<<endl;
}
};

```

```

int main(void) {
    Shape *s;           // base class pointer.
    Shape sh;           // base class object.
    Rectangle rec;
    Circle cir;
    s=&sh;
    s->draw();
    s=&rec;
    s->draw();
    s=?
    s->draw();
}

```

Output:

```

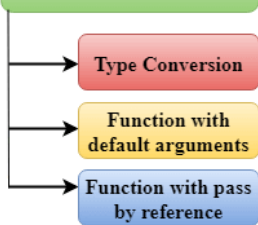
drawing...
drawing rectangle...
drawing circle...

```

C++ Function Overloading

Function Overloading is defined as the process of having two or more function with the same name, but different in parameters is known as function overloading in C++. In function overloading, the function is redefined by using either different types of arguments or a different number of arguments. It is only through these differences compiler can differentiate between the functions.

Causes Of Ambiquity



```

#include<iostream>
using namespace std;
int mul(int,int);
float mul(float,int);

int mul(int a,int b)
{
    return a*b;
}
float mul(double x, int y)
{
    return x*y;
}
int main()
{
    int r1 = mul(6,7);
    float r2 = mul(0.2,3);
    std::cout << "r1 is : " << r1 << std::endl;
    std::cout << "r2 is : " << r2 << std::endl;
    return 0;
}

```

Output:

```

r1 is : 42
r2 is : 0.6

```

Possible Error 1: Type Conversion

- Type Conversion:

Let's see a simple example.

```

#include<iostream>
using namespace std;
void fun(int);
void fun(float);
void fun(int i)
{
    std::cout << "Value of i is : " << i << std::endl;
}
void fun(float j)
{
    std::cout << "Value of j is : " << j << std::endl;
}
int main()
{
    fun(12);
    fun(1.2);
    return 0;
}

```

The above example shows an error "**call of overloaded 'fun(double)' is ambiguous**". The fun(10) will call the first function. The fun(1.2) calls the second function according to our prediction. But, this does not refer to any function as in C++, all the floating point constants are treated as double not as a float. If we replace float to double, the program works. Therefore, this is a type conversion from float to double.

Possible Error 2: Function with Default arguments

```

#include<iostream>
using namespace std;
void fun(int);
void fun(int,int);
void fun(int i)
{
    std::cout << "Value of i is : " <<i<< std::endl;
}
void fun(int a,int b=9)
{
    std::cout << "Value of a is : " <<a<< std::endl;
    std::cout << "Value of b is : " <<b<< std::endl;
}
int main()
{
    fun(12);

    return 0;
}

```

The above example shows an error "call of overloaded 'fun(int)' is ambiguous". The fun(int a, int b=9) can be called in two ways: first is by calling the function with one argument, i.e., fun(12) and another way is calling the function with two arguments, i.e., fun(4,5). The fun(int i) function is invoked with one argument. Therefore, the compiler could not be able to select among fun(int i) and fun(int a,int b=9).

Possible Error 3: Function with pass by reference

- o Function with pass by reference

Let's see a simple example.

```

#include <iostream>
using namespace std;
void fun(int);
void fun(int &);
int main()
{
    int a=10;
    fun(a); // error, which f()?
    return 0;
}
void fun(int x)
{
    std::cout << "Value of x is : " <<x<< std::endl;
}
void fun(int &b)
{
    std::cout << "Value of b is : " <<b<< std::endl;
}

```

The above example shows an error "call of overloaded 'fun(int&)' is ambiguous". The first function takes one integer argument and the second function takes a reference parameter as an argument. In this case, the compiler does not know which function is needed by the user as there is no syntactical difference between the fun(int) and fun(int &).

Operator Overloading

The advantage of Operators overloading is to perform different operations on the same operand.

Operator that cannot be overloaded are as follows:

- Scope operator (::)
- Sizeof
- member selector(.)
- member pointer selector(*)
- ternary operator(?:)

Rules for Operator Overloading

- Existing operators can only be overloaded, but the new operators cannot be overloaded.
- The overloaded operator contains atleast one operand of the user-defined data type.
- We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.
- When unary operators are overloaded through a member function take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.
- When binary operators are overloaded through a member function takes one explicit argument, and if they are overloaded through a friend function takes two explicit arguments.

```
#include <iostream>
using namespace std;
class Test
{
private:
    int num;
public:
    Test(): num(8){}
    void operator ++() {
        num = num+2;
    }
    void Print() {
        cout<<"The Count is: "<<num;
    }
};
int main()
{
    Test tt;
    ++tt; // calling of a function "void operator ++()"
    tt.Print();
    return 0;
}
```

Output:

```
The Count is: 10
```

```

#include <iostream>
using namespace std;
class A
{
    int x;
    public:
    A(){}
    A(int i)
    {
        x=i;
    }
    void operator+(A);
    void display();
};

void A :: operator+(A a)
{
    int m = x+a.x;
    cout<<"The result of the addition of two objects is : "<<m;

}

int main()
{
    A a1(5);
    A a2(4);
    a1+a2;
    return 0;
}

```

Output:

```
The result of the addition of two objects is : 9
```

Function Overriding

```

#include <iostream>
using namespace std;
class Animal {
    public:
    void eat(){
        cout<<"Eating...";
    }
};
class Dog: public Animal
{
    public:
    void eat()
    {
        cout<<"Eating bread...";
    }
};
int main(void) {
    Dog d = Dog();
    d.eat();
    return 0;
}

```

Output:

```
Eating bread...
```

CPP Virtual Function

- A C++ virtual function is a member function in the base class that you redefine in a derived class. It is declared using the virtual keyword.
- It is used to tell the compiler to perform dynamic linkage or late binding on the function.
- There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the 'virtual' function.
- A 'virtual' is a keyword preceding the normal declaration of a function.
- When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

C++ virtual function Example

Let's see the simple example of C++ virtual function used to invoke the derived class in a program.

```
#include <iostream>
{
    public:
    virtual void display()
    {
        cout << "Base class is invoked" << endl;
    }
};
class B:public A
{
    public:
    void display()
    {
        cout << "Derived Class is invoked" << endl;
    }
};
int main()
{
    A* a; //pointer of base class
    B b; //object of derived class
    a = &b;
    a->display(); //Late Binding occurs
}
```

Output:

```
Derived Class is invoked
```


Pure virtual function can be defined as:

```
virtual void display() = 0;
```

Let's see a simple example:

```
#include <iostream>
using namespace std;
class Base
{
public:
    virtual void show() = 0;
};
class Derived : public Base
{
public:
    void show()
    {
        std::cout << "Derived class is derived from the base class." << std::endl;
    }
};
int main()
{
    Base *bptr;
    //Base b;
    Derived d;
    bptr = &d;
    bptr->show();
    return 0;
}
```

Output:

```
Derived class is derived from the base class.
```

Interfaces in CPP

To achieve abstraction we have two ways

- Abstract class
- Interface

Abstract Class

```

#include <iostream>
using namespace std;
class Shape
{
public:
virtual void draw()=0;
};
class Rectangle : Shape
{
public:
void draw()
{
cout << "drawing rectangle..." << endl;
}
};
class Circle : Shape
{
public:
void draw()
{
cout << "drawing circle..." << endl;
}
};
int main() {
Rectangle rec;
Circle cir;
rec.draw();
cir.draw();
return 0;
}

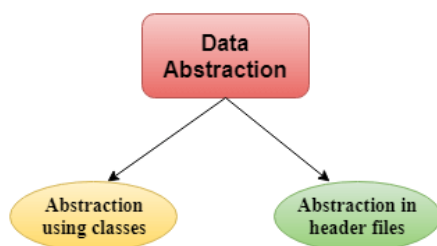
```

Output:

```

drawing rectangle...
drawing circle...

```



Abstraction using classes: An abstraction can be achieved using classes. A class is used to group all the data members and member functions into a single unit by using the access specifiers. A class has the responsibility to determine which data member is to be visible outside and which is not.

Abstraction in header files: Another type of abstraction is header file. For example, `pow()` function available is used to calculate the power of a number without actually knowing which algorithm function uses to calculate the power. Thus, we can say that header files hide all the implementation details from the user.

Access Specifiers Implement Abstraction:

- **Public specifier:** When the members are declared as public, members can be accessed anywhere from the program.
- **Private specifier:** When the members are declared as private, members can only be accessed only by the member functions of the class.

Let's see a simple example of abstraction in header files.

// program to calculate the power of a number.

```
#include <iostream>
#include <math.h>
using namespace std;
int main()
{
    int n = 4;
    int power = 3;
    int result = pow(n,power);    // pow(n,power) is the power function
    std::cout << "Cube of n is : " << result << std::endl;
    return 0;
}
```

Output:

```
Cube of n is : 64
```

Abstraction using classes

```
#include <iostream>
using namespace std;
class Sum
{
    private: int x, y, z; // private variables
    public:
    void add()
    {
        cout << "Enter two numbers: ";
        cin >> x >> y;
        z = x + y;
        cout << "Sum of two number is: " << z << endl;
    }
};
int main()
{
    Sum sm;
    sm.add();
    return 0;
}
```

Output:

```
Enter two numbers:
3
6
Sum of two number is: 9
```

Advantages Of Abstraction:

- Implementation details of the class are protected from the inadvertent user level errors.
- A programmer does not need to write the low level code.
- Data Abstraction avoids the code duplication, i.e., programmer does not have to undergo the same tasks every time to perform the similar operation.
- The main aim of the data abstraction is to reuse the code and the proper partitioning of the code across the classes.
- Internal implementation can be changed without affecting the user level code.

CPP Exception Handling

Wednesday, April 6, 2022 7:55 PM

- Exception Handling in C++ is a process to handle runtime errors. We perform exception handling so the normal flow of the application can be maintained even after runtime errors
- In C++, exception is an event or object which is thrown at runtime. All exceptions are derived from `std::exception` class. It is a runtime error which can be handled. If we don't handle the exception, it prints exception message and terminates the program.

C++ Exception Handling Keywords

In C++, we use 3 keywords to perform exception handling:

- `try`
- `catch`, and
- `throw`

Moreover, we can create user-defined exception which we will learn in next chapters.

C++ try/catch example

```
#include <iostream>
using namespace std;
float division(int x, int y) {
    if (y == 0) {
        throw "Attempted to divide by zero!";
    }
    return (x/y);
}
int main () {
    int i = 25;
    int j = 0;
    float k = 0;
    try {
        k = division(i, j);
        cout << k << endl;
    } catch (const char* e) {
        cerr << e << endl;
    }
    return 0;
}
```

Output:

```
Attempted to divide by zero!
```

User Defined Exception

```
#include <iostream>
#include <exception>
using namespace std;
class MyException : public exception{
public:
    const char* what() const throw()
    {
        return "Attempted to divide by zero!\n";
    }
};
```

```

int main()
{
    try
    {
        int x, y;
        cout << "Enter the two numbers : \n";
        cin >> x >> y;
        if (y == 0)
        {
            MyException z;
            throw z;
        }
        else
        {
            cout << "x / y = " << x/y << endl;
        }
    }
    catch(exception& e)
    {
        cout << e.what();
    }
}

```

Output:

```

Enter the two numbers :
10
2
x / y = 5

```

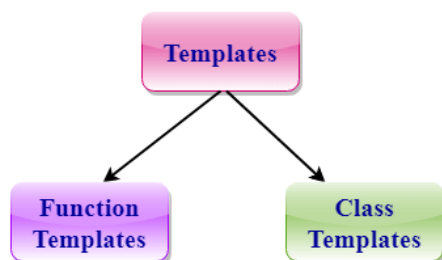
```

Enter the two numbers :
10
0
Attempted to divide by zero!

```

CPP Templates

A C++ template is a powerful feature added to C++. It allows you to define the generic classes and generic functions and thus provides support for generic programming. Generic programming is a technique where generic types are used as parameters in algorithms so that they can work for a variety of data types.



Function Template

- Generic functions use the concept of a function template. Generic functions define a set of operations that can be applied to the various types of data.
- The type of the data that the function will operate on depends on the type of the data passed as a parameter.
- For example, Quick sorting algorithm is implemented using a generic function, it can be implemented to an array of integers or array of floats.
- A Generic function is created by using the keyword template. The template defines what function will do.

CPP Templates

Wednesday, April 6, 2022 9:09 PM

Function Templates

Where **Ttype**: It is a placeholder name for a data type used by the function. It is used within the function definition. It is only a placeholder that the compiler will automatically replace this placeholder with the actual data type.

class: A class keyword is used to specify a generic type in a template declaration.

```
#include <iostream>
using namespace std;
template<class T> T add(T &a,T &b)
{
    T result = a+b;
    return result;
}
int main()
{
    int i =2;
    int j =3;
    float m = 2.3;
    float n = 1.2;
    cout<<"Addition of i and j is :"<<add(i,j);
    cout<<"\n";
    cout<<"Addition of m and n is :"<<add(m,n);
    return 0;
}
```

Output:

```
Addition of i and j is :5
Addition of m and n is :3.5
```

```
#include <iostream>
using namespace std;
template<class X,class Y> void fun(X a,Y b)
{
    std::cout << "Value of a is : " <<a<< std::endl;
    std::cout << "Value of b is : " <<b<< std::endl;
}
int main()
{
    fun(15,12.3);

    return 0;
}
```

Output:

```
Value of a is : 15
Value of b is : 12.3
```

In the above example, we use two generic types in the template function, i.e., X and Y.

```

#include <iostream>
using namespace std;
template<class X> void fun(X a)
{
    std::cout << "Value of a is : " << a << std::endl;
}
template<class X, class Y> void fun(X b, Y c)
{
    std::cout << "Value of b is : " << b << std::endl;
    std::cout << "Value of c is : " << c << std::endl;
}
int main()
{
    fun(10);
    fun(20, 30.5);
    return 0;
}

```

Output:

```

Value of a is : 10
Value of b is : 20
Value of c is : 30.5

```

In the above example, template of fun() function is overloaded.

We can use templates only if we want to do same operation even if we have different data types. But when the functionality changes from datatype to datatype the templates are useless.

Class Templates

```

using namespace std;
template<class T>
class A
{
public:
    T num1 = 5;
    T num2 = 6;
    void add()
    {
        std::cout << "Addition of num1 and num2 : " << num1 + num2 << std::endl;
    }
};

int main()
{
    A<int> d;
    d.add();
    return 0;
}

```

Output:

```

Addition of num1 and num2 : 11

```

Class Templates with multiple parameters


```

#include <iostream>
using namespace std;
template<class T1, class T2>
class A
{
    T1 a;
    T2 b;
public:
    A(T1 x,T2 y)
    {
        a = x;
        b = y;
    }
    void display()
    {
        std::cout << "Values of a and b are : " << a<<" , "<<b<<std::endl;
    }
};

int main()
{
    A<int,float> d(5,6.5);
    d.display();
    return 0;
}

```

Output:

```

Values of a and b are : 5,6.5

```

Non Type Template arguments

```

#include <iostream>
using namespace std;
template<class T, int size>
class A
{
public:
    T arr[size];
    void insert()
    {
        int i = 1;
        for (int j=0;j<size;j++)
        {
            arr[j] = i;
            i++;
        }
    }

    void display()
    {
        for(int i=0;i<size;i++)
        {
            std::cout << arr[i] << " ";
        }
    }
};

int main()
{
    A<int,10> t1;
    t1.insert();
    t1.display();
    return 0;
}

```

Output:

```
1 2 3 4 5 6 7 8 9 10
```