Virtual Function:

```
#include <iostream>

using namespace std;



class Base

{

    public:

    void Display ()

    {

cout<< "Display of Base" <<endl;

    }

};

class Derived:public Base

{

public:void Display ()

    {

cout<< "Display of Derived " <<endl;

    }

};
```

```
intmain()

{

    Base *p = new Derived ();

    p->Display ();

}
```

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

class Base
{
   public:
   virtual void Display ()
   {
cout<< "Display of Base" <<endl;
   }
};

class Derived:public Base
{
   public:
   void Display ()
   {
cout<< "Display of Derived" <<endl;
   }
};

intmain()
{
   Base *p = new Derived();
   p->Display();
}
```

```
// C++ program to illustrate
// concept of Virtual Functions
```

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

class base {
public:
	virtual void print() { cout<< "print base class\n"; }

	void show() { cout<< "show base class\n"; }
};

class derived : public base {
public:
	void print() { cout<< "print derived class\n"; }

	void show() { cout<< "show derived class\n"; }
};

intmain()
{
	base* bptr;
	derived d;
	bptr = &d;

	// Virtual function, binded at runtime
	bptr->print();

	// Non-virtual function, binded at compile time
	bptr->show();

	return 0;
}

// C++ program to illustrate
// working of Virtual Functions
#include <iostream>
using namespace std;

class base {
public:
	void fun_1() { cout<< "base-1\n"; }
	virtual void fun_2() { cout<< "base-2\n"; }
	virtual void fun_3() { cout<< "base-3\n"; }
```

```
        virtual void fun_4() { cout<< "base-4\n"; }
};

class derived : public base {
public:
        void fun_1() { cout<< "derived-1\n"; }
        void fun_2() { cout<< "derived-2\n"; }
        void fun_4(int x) { cout<< "derived-4\n"; }
};

intmain()
{
        base* p;
        derived obj1;
        p = &obj1;

        // Early binding because fun1() is non-virtual
        // in base
        p->fun_1();

        // Late binding (RTP)
        p->fun_2();

        // Late binding (RTP)
        p->fun_3();

        // Late binding (RTP)
        p->fun_4();

        // Early binding but this function call is
        // illegal (produces error) because pointer
        // is of base type and function is of
        // derived class
        // p->fun_4(5);

        return 0;
}
```

Limitations of Virtual Functions

- **Slower:** The function call takes slightly longer due to the virtual mechanism and makes it more difficult for the compiler to optimize because it does not know exactly which function is going to be called at compile time.

- **Difficult to Debug:** In a complex system, virtual functions can make it a little more difficult to figure out where a function is being called from.

```
// C++ Program to demonstrate Virtual
// functions in derived classes
#include <iostream>
using namespace std;

class A {
public:
	virtual void fun() { cout<< "\n A::fun() called "; }
};

class B : public A {
public:
	void fun() { cout<< "\n B::fun() called "; }
};

class C : public B {
public:
	void fun() { cout<< "\n C::fun() called "; }
};

intmain()
{
	// An object of class C
	C c;

	// A pointer of class B pointing
	// to memory location of c
	B* b = &c;

	// this line prints "C::fun() called"
	b->fun();

	getchar(); // to get the next character
	return 0;
}

// C++ Program to Demonstrate
// Operator Overloading
#include <iostream>
using namespace std;
```

```
class Complex {
private:
	int real, imag;

public:
	Complex(int r = 0, inti = 0)
	{
		real = r;
		imag = i;
	}

	// This is automatically called when '+' is used with
	// between two Complex objects
	Complex operator+(Complex const&obj)
	{
		Complex res;
		res.real = real + obj.real;
		res.imag = imag + obj.imag;
		return res;
	}
	void print() { cout<< real << " + i" <<imag<< '\n'; }
};

intmain()
{
	Complex c1(10, 5), c2(2, 4);
	Complex c3 = c1 + c2;
	c3.print();
}
```

What is a C++ Exception?

An exception is an unexpected problem that arises during the execution of a program .our program terminates suddenly with some errors/issues. Exception occurs during the running of the program (runtime).

Types of C++ Exception

There are two types of exceptions in C++

1. **Synchronous:** Exceptions that happen when something goes wrong because of a mistake in the input data or when the program is not equipped to handle the current type of data it's working with, such as dividing a number by zero.
1. **Asynchronous**: Exceptions that are beyond the program's control, such as disc failure, keyboard interrupts, etc.

C++ try and catch

C++ provides an inbuilt feature for Exception Handling. It can be done using the following specialized keywords**: try, catch, and throw** with each having a different purpose.

Syntax of try-catch in C++

```
try {
    // Code that might throw an exception
throw SomeExceptionType("Error message");
 }
catch( ExceptionName e1 )  {
// catch block catches the exception that is thrown from try block
 }
```

1. try in C++

The try keyword represents a block of code that may throw an exception placed inside the try block. It's followed by one or more catch blocks. If an exception occurs, try block throws that exception.

2. catch in C++

The catch statement represents a block of code that is executed when a particular exception is thrown from the try block. The code to handle the exception is written inside the catch block.

3. throw in C++

An exception in C++ can be thrown using the throw keyword. When a program encounters a throw statement, then it immediately terminates the current function and starts finding a matching catch block to handle the thrown exception.

Note: *Multiple catch statements can be used to catch different type of exceptions thrown by try block.*

The try and catch keywords come in pairs: We use the try block to test some code and If the code throws an exception we will handle it in our catch block.

Why do we need Exception Handling in C++?

1. **Separation of Error Handling Code from Normal Code**:
2. **Functions/Methods can handle only the exceptions they choose**:
3. **Grouping of Error Types**:

```
#include<iostream>

using namespace std;

int main()
{
int a=10, b=0, c;
try
{
	//if a is divided by b(which has a value 0);
	if(b==0)
		throw(c);
	else
	c=a/b;

}
catch(char c)	//catch block to handle/catch exception
{
	cout<<"Caught exception : char type ";
}
catch(int i)	//catch block to handle/catch exception
{
	cout<<"Caught exception :int type ";
}
catch(short s)	//catch block to handle/catch exception
{
	cout<<"Caught exception : short type ";
}
cout<<"\n Hello";
```

```
}
```

Let us see another example to declare a catch block which catches all types of exceptions, irrespective of their types. In such catch block, instead of declaring a certain type of exception it can catch, we are going to use three dots(...) within its parenthesis.

```
#include<iostream>


using namespace std;

class A
{
public:
char ch ='c';
inti  = 1;

void function1();
void function2();
};


void A :: function1()
{
if(ch == 'c')
        throw(ch);
}

void A :: function2()
{
if(i == 1)
        throw(i);
}



Intmain()
{
A ob;
try
{
        ob.function1();
}
```

```
catch(...)     //catch block to handle/catch exception
{
        cout<<"A catch block for all types of exceptions has caught an exception \n";
}

try
{
        ob.function2();
}
catch(...)     //catch block to handle/catch exception
{
        cout<<"A catch block for all types of exceptions has caught an exception";
}
}


// C++ program to demonstate the use of try,catch and throw
// in exception handling.

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

intmain()
{

        // try block
        try {
                int numerator = 10;
                int denominator = 0;
                int res;

                // check if denominator is 0 then throw runtime
                // error.
                if (denominator == 0) {
                        throw runtime_error(
                                "Division by zero not allowed!");
                }

                // calculate result if no exception occurs
                res = numerator / denominator;
                //[printing result after division
                cout<< "Result after division: " << res <<endl;
```

```
	}
	// catch block to catch the thrown exception
	catch (const exception& e) {
		// print the exception
		cout<< "Exception " <<e.what() <<endl;
	}

	return 0;
}

// C++ program to demonstate the use of try,catch and throw
// in exception handling.

#include <iostream>
using namespace std;

intmain()
{
	int x = -1;

	// Some code
	cout<< "Before try \n";

	// try block
	try {
		cout<< "Inside try \n";
		if (x < 0) {
			// throwing an exception
			throw x;
			cout<< "After throw (Never executed) \n";
		}
	}

	// catch block
	catch (int x) {
		cout<< "Exception Caught \n";
	}

	cout<< "After catch (Will be executed) \n";
	return 0;
}

// C++ program to illustrate the concept
```

```
// of exception handling using class

#include <bits/stdc++.h>
using namespace std;

// Class declaration
class Number {
private:
	int a, b;

public:
	// Constructors
	Number(int x, int y)
	{
		a = x;
		b = y;
	}

	// Function that find the GCD
	// of two numbers a and b
	intgcd()
	{
		// While a is not equal to b
		while (a != b) {

			// Update a to a - b
			if (a > b)
				a = a - b;

			// Otherwise, update b
			else
				b = b - a;
		}

		// Return the resultant GCD
		return a;
	}

	// Function to check if the
	// given number is prime
	bool isPrime(int n)
	{
		// Base Case
```

```
		if (n <= 1)
			return false;

		// Iterate over the range [2, N]
		for (inti = 2; i< n; i++) {

			// If n has more than 2
			// factors, then return
			// false
			if (n % i == 0)
				return false;
		}

		// Return true
		return true;
	}
};

// Empty class
class MyPrimeException {
};

// Driver Code
intmain()
{
	int x = 13, y = 56;

	Number num1(x, y);

	// Print the GCD of X and Y
	cout<< "GCD is = "
		<<num1.gcd() <<endl;

	// If X is prime
	if (num1.isPrime(x))
		cout<< x
			<< " is a prime number"
			<<endl;

	// If Y is prime
	if (num1.isPrime(y))
		cout<< y
			<< " is a prime number"
```

```
                <<endl;

        // Exception Handling
        if ((num1.isPrime(x))
                || (num1.isPrime(y))) {

                // Try Block
                try {
                        throw MyPrimeException();
                }

                // Catch Block
                catch (MyPrimeException t) {

                        cout<< "Caught exception "
                                << "of MyPrimeException "
                                << "class." <<endl;
                }
        }

        return 0;
}
```