➢ Multiple catch statements

➢ In C++, we can use multiple catch statements for different kinds of exceptions that can result from a single block of code.

```
try {
   // code
}
catch (exception1) {
   // code
}
catch (exception2) {
   // code
}
catch (...) {
   // code
}
```

Here, our program catches exception1 if that exception occurs. If not, it will catch exception2 if it occurs.
If there is an error that is neither exception1 nor exception2, then the code inside of catch (...) {} is executed.

Why multiply blocks are useful?

Multiple catch blocks are used when we have to catch a specific type of exception out of many possible type of exceptions

```
#include<iostream>

using namespace std;

int main()
{
int a=10, b=0;
int 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";
}
```

Example 2:

```
     #include<iostream.h>
     #include<conio.h>

     void test(int x)
     {
        try

                  {
                        if (x > 0)
                           throw x;
                        else
                           throw 'x';

                     }
     catch (int x)
          {       cout << "Catch a integer and that integer is:" << x;    }
      catch (char x)
            {       cout << "Catch a character and that character is:" << x;    }
```

```
}
void main()
{
   clrscr();
   cout << "Testing multiple catches\n:";
   test(10);
   test(0);
   getch();
}
```

C++ Standard Exception-

The exceptions used with the Standard C++ library are also available for your use. Generally it's easier and faster to start with a standard exception class than to try to define your own. If the standard class doesn't do exactly what you need, you can derive from it.

All standard exception classes derive ultimately from the class exception, defined in the header <exception>. The two main derived classes are logic_error and runtime_error, which are found in <stdexcept> (which itself includes <exception>). The class logic_error represents errors in programming logic, such as passing an invalid argument. Runtime errors are those that occur as the result of unforeseen forces such as hardware failure or memory exhaustion. Both runtime_error and logic_error provide a constructor that takes a std::string argument so that you can store a message in the exception object and extract it later with exception::what( ) .

The following tables describe the standard exception classes:

exception	The base class for all the exceptions thrown by the C++ Standard library. You can ask **what( )** and retrieve the optional string with which the exception was initialized.
logic_error	Derived from exception. Reports program logic errors, which could presumably be detected by inspection.
runtime_error	Derived from exception. Reports runtime errors, which can presumably be detected only when the program executes.

C++ has provided us with a number of standard exceptions that we can use in our exception handling. Some of them are shown in the table below.

Exception	Description
std::exception	The parent class of all C++ exceptions.
std::bad_alloc	Thrown when a dynamic memory allocation fails.
std::bad_cast	Thrown by C++ when an attempt is made to perform a dynamic_cast to an invalid type.
std::bad_exception	Typically thrown when an exception is thrown and it cannot be rethrown.