We can chain multiple handlers (catch expressions), each one with a different parameter type. Only the handler that matches its type with the argument specified in the throw statement is executed.

```
void Xhandler(int test) {
try {
    if (test) throw test;
    else throw "Value is zero"; }
catch(int i) {
    cout << "Caught one! Ex. #: " << i << "\n"; }
catch(char *str) {
    cout << "Caught a string: " << str << "\n"; }
}

int main( ) {
    cout << "start";
    Xhandler(1);
    Xhandler(2);
    Xhandler(0);
    Xhandler(3);
    cout << "end";
    return 0; }
```

```
start
Caught one! Ex. #: 1
Caught one! Ex. #: 2
Caught a string: Value is zero
Caught one! Ex. #: 3
end
```

If we use an ellipsis (...) as the parameter of catch, that handler will catch any exception no matter what the type of the throw exception is.

This can be used as a default handler that catches all exceptions not caught by other handlers if it is specified at last:

```
try {
  // code here
}
catch (int param) { cout << "int exception"; }
catch (char param) { cout << "char exception"; }
catch (...) { cout << "default exception"; }
```

After an exception has been handled the program execution resumes after the try-catch block, not after the throw statement!.

One very good use for **catch(...)** is as last **catch** of a cluster of catches.

```
#include <iostream>
using namespace std;
void Xhandler(int test) {
try {
    if (test==0) throw test; // throw int
    if (test==1) throw 'a'; // throw char
    if (test==2) throw 123.23;// throw double }
catch(int i) { // catch an int exception
    cout << "Caught " << i << "\n"; }
catch(...) { // catch all other exceptions
    cout << "Caught one!\n"; }
}

int main( ) {
    cout << "start\n";
    Xhandler(0);
    Xhandler(1);
    Xhandler(2);
    cout << "end";
    return 0;}
```

```
This program displays:
start
Caught 0
Caught one!
Caught one!
end
```

Rethrowing Exceptions

If a catch block cannot handle the particular exception it has caught, you can rethrow the exception. The rethrow expression (throw without *assignment_expression*) causes the originally thrown object to be rethrown.

```
void g() {
  throw "Exception";
}

 void f() {
  int* i = new int(0);
  g();
  delete i;
}

 int main() {
  f();
 return 0;
}
```

Problem?

If g throws an exception, the variable i is never deleted and we have a memory leak.

To prevent the memory leak, f() must catch the exception, and delete i. But f() can't handle the exception, it doesn't know how!

Solution?

f() shall catch the exception, and then rethrow it:

It is possible to nest try-catch blocks within more external try blocks.

We have the possibility that an internal catch block forwards the exception to its external level.

```
try {
    try {
     // code here
       throw 10; }
    }
    catch (int n) {
        throw;
    }
}
catch (...) {  cout << "Exception occurred";
}
```

```
void g() {
  throw 60; }

 void f() {
  int* i = new int(0);
  try{
    try {
         g();}
    catch (int n) {
         ……

         throw; // This empty throw rethrows the exception we caught
         // An empty throw can only exist in a catch block

    }
    catch (...)   {
        …
     delete i;
 }

int main() {
      f();
      return 0;}
```

Rethrowing an exception. An exception can only be rethrown from within a **catch** block. When you rethrow an exception, it will not be recaught by the same **catch** statement. It will propagate to an outer **catch** statement.

```
#include <iostream>
using namespace std;
void Xhandler( ) {
  try { throw "hello"; // throw char *
  catch(char *) { // catch a char *
      cout << "Caught char * inside Xhandler\n";
      throw ; // rethrow char * out of function
  }
}

int main( ) {
    cout << "start\n";
try {
    Xhandler( ); }
catch(char *) {
    cout << "Caught char * inside main\n"; }
    cout << "end";
    return 0;}
```

```
This program displays:
start
Caught char * inside
Xhandler
Caught char * inside main
end
```

Unhandled Exceptions

- An unhandled exception is propagated to the caller of the function in which it is raised
- This propagation continues to the main function
- If no handler is found, the program is terminated

Exception specifications

- You can control what type of exceptions a function can throw outside itself. In fact, you can also prevent a function from throwing any exceptions whatsoever. To apply these restrictions, you must add a **throw** clause to the function definition.

ret-type-func-name(*arg-list*) **throw**(*type-list*)
{
// ....
}

Here only those data types contained in the comma-separated list *type-list* may be thrown by the function. Throwing any other type of expression will cause the program termination.

```
float myfunction (char param) throw (int);
```

- If this throw specifier is left empty with no type, this means the function is not allowed to throw exceptions.

```
int myfunction (int param) throw(); // no exceptions allowed
```

- Functions with no throw specifier (regular functions) are allowed to throw exceptions with any type:

```
int myfunction (int param);        // all exceptions allowed
```

```
class A { };
class B : public A { };
class C { };

void f(int i) throw (A) {
  switch (i) {
    case 0: throw A();
    case 1: throw B();
    default: throw C();
  }}
```

Function f() can throw objects of types A or B. If the function tries to throw an object of type C, this is a compilation error because type C has not been specified in the function's exception specification, nor does it derive publicly from A.

The following program shows how to restrict the types of exceptions that can be thrown from a function:

```
#include <iostream>
using namespace std;
void Xhandler(int test) throw(int, char, double) {
    if (test==0) throw test; // throw int
    if (test==1) throw 'a'; // throw char
    if (test==2) throw 123.23;// throw double
}

int main( ) {
    cout << "start\n";
try {
    Xhandler(0); // also try passing 1 and 2 to Xhandler( )
}
catch(int i) {
    cout << "Caught int\n";
}
catch(char c) {
    cout << "Caught char\n";
}
catch(double c) {
    cout << "Caught double\n";
}
    cout << "end";
  return 0; }
```

- A function that overrides a virtual function can only throw exceptions specified by the virtual function.

```
class A {
  public:
    virtual void f() throw (int, char);
};

class B : public A{
  public: void f() throw (int) { }
};

/* The following is not allowed. */
/*
  class C : public A {
    public: void f() { }
  };

  class D : public A {
    public: void f() throw (int, char, double) { }
  };
*/
```

The compiler allows B::f() because the member function may throw only exceptions of type int.

The compiler would not allow C::f() because the member function may throw any kind of exception.

The compiler would not allow D::f() because the member function can throw more types of exceptions (int, char, and double) than A::f().