

# Exceptions

Exceptions provide a way to react to exceptional circumstances (like runtime errors) in programs by transferring control to special functions called *handlers*.

To catch exceptions, a portion of code is placed under exception inspection. This is done by enclosing that portion of code in a *try-block*. When an exceptional circumstance arises within that block, an exception is thrown that transfers the control to the exception handler. If no exception is thrown, the code continues normally and all handlers are ignored.

An exception is thrown by using the `throw` keyword from inside the `try` block. Exception handlers are declared with the keyword `catch`, which must be placed immediately after the `try` block:

```
1 // exceptions
2 #include <iostream>
3 using namespace std;
4
5 int main () {
6     try
7     {
8         throw 20;
9     }
10    catch (int e)
11    {
12        cout << "An exception occurred. Exception Nr. "
13 << e << '\n';
14    }
15    return 0;
16 }
```

```
An exception occurred.
Exception Nr. 20
```

The code under exception handling is enclosed in a `try` block. In this example this code simply throws an exception:

```
throw 20;
```

A `throw` expression accepts one parameter (in this case the integer value 20), which is passed as an argument to the exception handler.

The exception handler is declared with the `catch` keyword immediately after the closing brace of the `try` block. The syntax for `catch` is similar to a regular function with one parameter. The type of this parameter is very important, since the type of the argument passed by the `throw` expression is checked against it, and only in the case they match, the exception is caught by that handler.

Multiple handlers (i.e., `catch` expressions) can be chained; each one with a different parameter type. Only the handler whose argument type matches the type of the exception specified in

the `throw` statement is executed.

If an ellipsis (`...`) is used as the parameter of `catch`, that handler will catch any exception no matter what the type of the exception thrown. This can be used as a default handler that catches all exceptions not caught by other handlers:

```
1 try {
2     // code here
3 }
4 catch (int param) { cout << "int exception"; }
5 catch (char param) { cout << "char exception"; }
6 catch (...) { cout << "default exception"; }
```

In this case, the last handler would catch any exception thrown of a type that is neither `int` nor `char`.

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

It is also possible to nest *try-catch* blocks within more external *try* blocks. In these cases, we have the possibility that an internal *catch* block forwards the exception to its external level. This is done with the expression `throw;` with no arguments. For example:

```
1 try {
2     try {
3         // code here
4     }
5     catch (int n) {
6         throw;
7     }
8 }
9 catch (...) {
10     cout << "Exception occurred";
11 }
```

## Exception specification

---

Older code may contain *dynamic exception specifications*. They are now deprecated in C++, but still supported. A *dynamic exception specification* follows the declaration of a function, appending a `throw` specifier to it. For example:

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

This declares a function called `myfunction`, which takes one argument of type `char` and returns

a value of type `double`. If this function throws an exception of some type other than `int`, the function calls `std::unexpected` instead of looking for a handler or calling `std::terminate`.

If this `throw` specifier is left empty with no type, this means that `std::unexpected` is called for any exception. Functions with no `throw` specifier (regular functions) never call `std::unexpected`, but follow the normal path of looking for their exception handler.

```
1 int myfunction (int param) throw(); // all exceptions call unexpected
2 int myfunction (int param);         // normal exception handling
```

## Standard exceptions

---

The C++ Standard library provides a base class specifically designed to declare objects to be thrown as exceptions. It is called `std::exception` and is defined in the `<exception>` header. This class has a virtual member function called `what` that returns a null-terminated character sequence (of type `char *`) and that can be overwritten in derived classes to contain some sort of description of the exception.

```
1 // using standard exceptions
2 #include <iostream>
3 #include <exception>
4 using namespace std;
5
6 class myexception: public exception
7 {
8     virtual const char* what() const throw()
9     {
10         return "My exception happened";
11     }
12 } myex;
13
14 int main () {
15     try
16     {
17         throw myex;
18     }
19     catch (exception& e)
20     {
21         cout << e.what() << '\n';
22     }
23     return 0;
24 }
```

My exception happened.

We have placed a handler that catches exception objects by reference (notice the ampersand `&` after the type), therefore this catches also classes derived from `exception`, like our `myex` object of type `myexception`.

All exceptions thrown by components of the C++ Standard library throw exceptions derived from this `exception` class. These are:

exception	description
<code>bad_alloc</code>	thrown by <code>new</code> on allocation failure
<code>bad_cast</code>	thrown by <code>dynamic_cast</code> when it fails in a dynamic cast
<code>bad_exception</code>	thrown by certain dynamic exception specifiers
<code>bad_typeid</code>	thrown by <code>typeid</code>
<code>bad_function_call</code>	thrown by empty <code>function</code> objects
<code>bad_weak_ptr</code>	thrown by <code>shared_ptr</code> when passed a bad <code>weak_ptr</code>

Also deriving from `exception`, header `<exception>` defines two generic exception types that can be inherited by custom exceptions to report errors:

exception	description
<code>logic_error</code>	error related to the internal logic of the program
<code>runtime_error</code>	error detected during runtime

A typical example where standard exceptions need to be checked for is on memory allocation:

```
1 // bad_alloc standard exception
2 #include <iostream>
3 #include <exception>
4 using namespace std;
5
6 int main () {
7     try
8     {
9         int* myarray= new int[1000];
10    }
11    catch (exception& e)
12    {
13        cout << "Standard exception: " <<
14 e.what() << endl;
15    }
16    return 0;
17 }
```

The exception that may be caught by the exception handler in this example is a `bad_alloc`. Because `bad_alloc` is derived from the standard base class `exception`, it can be caught (capturing by reference, captures all related classes).