Brief Tutorial on C++ Exceptions

References

The material in this handout is collected from the following references:

- Sections 5.6 and 18.1 of the text book C++ Primer.
- Chapter 13 of The C++ Programming Language.

Assuming errors will not occur

Things go wrong, sometimes horribly wrong. Programmers have always been forced to deal with error conditions and exceptional situations. A few types of errors commonly encountered include *user input errors* as when users enters invalid input, such as a file name that doesn't exist; *device errors* such as when a disk drive may be unavailable, a printer is turned off, a network router is offline; *physical limitations* as when disk drives are full, available memory is exhausted; *software component failures* caused by incorrect behavior of functions or classes or incorrect use of functions and classes.

Probably the easiest way to deal with exceptional situations is to simply assume that they will not occur. Beginning programmers often make this mistake, by writing functions that will, for example, operate correctly on only a limited range of input values, and never bothering to check the legitimacy of their arguments. What should the following function `future_balance()` do if either `n` or `p` is less than zero?

```
1 double future_balance(double initial_balance, double p, int n) {
2   return initial_balance * pow(1 + p / 100.0, n);
3 }
```

Illegal inputs can lead to erroneous results that will likely lead to even more bizarre errors later in execution. More often, mistakes arise not from a single function call, but as a consequence of the interaction between different components in a program. Consider programmer $P1$ that implemented the C++ standard library container `std::stack` by assuming that a call on `stack::pop()` will never take place without a previous call on `stack::push()` to place a value on to the stack. Suppose programs built for cars and planes by a careless programmer $P2$ doesn't test for this case. Programmer $P1$ has, after all, provided programmer $P2$ with the ability to check whether the stack is empty through `stack::size()`, so there is no excuse for such an error. Nevertheless, the error is almost certain to occur at some point. Whose fault is the error? Is it the fault of programmer $P2$, for not using the stack correctly, or is it the fault of programmer $P1$, for not anticipating the possibility of the error? Finger pointing aside, it is simply the case that data type `std::stack` would be more reliable and robust if programmer $P1$ had provided a better mechanism for dealing with errors.

Efficiency is usually cited as the reason for creating components that neglect to check their data, such as function `stack::pop()`. The argument is that in those situations where reliability is a concern a good programmer will use `stack::size()` function to check their data before calling `stack::pop()`, that one should not have to pay the execution time cost required by the check in situations where it is not needed, and that if checks are performed there will be a tendency to repeatedly check for the

same error. A similar argument is used to justify the fact that array index expressions in C++ programs are not checked.

As machines have become ever faster, however, the balance has shifted from efficiency to security as a primary concern. Programmers are notorious for not recognizing the possibility of error. In real terms the cost of checks at execution time is small. And this small cost pales in comparison to the time spent hunting down mysterious errors that occur when “impossible” things happen. And even this pales in comparison to the millions of man years that have been spent combating computer viruses, many of which were made possible by the simple decision not to check array bounds in C and C++ code. You should never assume that things will not go wrong - that your arguments will never be invalid, that two member functions will never be invoked in an incorrect sequence.

> ***Program defensively. Assume that if something can go wrong, it will go wrong. Check for these conditions, then take appropriate actions when they occur.***

The three principal ways to deal with unexpected behavior in C++ are *assertions*, *old-style error handling using error flags*, and *exceptions*. Assertions allow a program to monitor its own behavior and detect programming errors during both compile-time and at run-time. Error flags are used to communicate computational errors between caller and callee functions. Exceptions provide a more general and robust way for callees to complain to callers upon occurrence of exceptional situations that prevent proper continuation of the program.

Assertions

An *assertion* specifies that a program satisfies certain conditions at particular points in its execution. There are three types of assertions:

- *Preconditions* that specify conditions at the start of a function.
- *Postconditions* that specify conditions at the end of a function.
- *Invariants* that specify conditions over a defined region of a program.

An assertion violation indicates a bug in the program. Thus, assertions are an effective means of improving the reliability of programs. In other words, they are a systematic debugging tool.

In C++, assertions are implemented with standard C library `assert` macro that is defined in header file `<cassert>`. The argument to `assert` must be true when the macro is executed, otherwise the program prints a diagnostic message and aborts. For example, the assertion

```
1 assert(size <= LIMIT);
```

will print an error message if the value stored in variable `size` is greater than value `LIMIT`

```
1 Assertion violation: file test.cpp, line 20: size <= LIMIT
```

and then abort the program.

Preconditions

Preconditions specify the *input conditions to a function*. They can be best explained using the following analogy. Suppose you [the function] are employed as an apple-packer. One of the conditions of your contract is that the temperature in the warehouse will be no greater than 30°C. If the temperature exceeds 30°C, you are not obliged to do anything: you can keep packing apples if you want to, or you can choose to go to the beach. Your employer [the caller], however, knows that you are not required to pack apples if the temperature exceeds 30°C, so he or she makes sure that the air-conditioning in the warehouse is operating correctly. Here is an example of a function with preconditions:

```
int magic(int size, double *data) {
  assert(size <= LIMIT);
  assert(data != nullptr);
  // other code here ...
}
```

These pre-conditions have two consequences:

1. Function `magic()` is only required to perform its task if the pre- conditions are satisfied. Thus, as the author of this function, you are not required to make `magic()` do anything sensible if `size` or `data` are not as stated in the assertions.
2. The caller is certain of the conditions under which function `magic()` will perform its task correctly. Thus, if your code is calling function `magic()`, you must ensure that the `size` or `data` arguments to the call are as specified by the assertions.

Postconditions

Postconditions specify the output conditions of a function. They are used much less frequently than preconditions, partly because implementing them in C++ can be a little awkward. Here is an example of a postcondition in function `magic()`:

```
int magic(int size, double *data) {
  // other code here ...
  assert( result <= LIMIT );
  return result;
}
```

The postcondition also has two consequences:

1. Function `magic()` guarantees that the stated condition will hold when it completes execution. As the author of function `magic()`, you must make certain that your code never produces a value of `result` greater than `LIMIT`.
2. The caller is certain of the task that function `magic()` will perform [provided its preconditions are satisfied]. If your program is calling function `magic()`, then you know that the result returned by `magic()` can be no greater than `LIMIT`.

Compare these consequences with the apple-picker analogy. Another part of your contract states that you will not bruise the apples. It is therefore your responsibility to ensure that you do not [and if you do, you have failed]. Your employer is thus relieved of the need to check that the apples are not bruised before shipping them.

Recommended practices

Writing preconditions

The simplest and most effective use of assertions is as preconditions - that is, to specify and check input conditions to functions. Two very common uses are to assert that:

1. Pointers are not `nullptr`.
2. Indexes and size values are non-negative and less than a known limit.

Each assertion must be listed in the *Asserts* section of the function description comment in the corresponding header file. For example, the comment describing function `magic()` will include:

```
/*
*  Asserts:
*      'size' is no greater then LIMIT.
*      'data' is not nullptr.
*      The function result is no greater than LIMIT.
*/
```

If there are no assertions, say that by writing *Nothing*:

```
/*
*  Asserts:
*      Nothing
*/
```

Satisfying preconditions

When your code calls a function with preconditions, you must ensure that the function's preconditions are satisfied. This does not mean that you have to include code to check the argument to every function that you call! For example, in the following code, function `resize()` does not need to check that the argument to function `measure()` is `nullptr`, since its own assertion ensures this:

```
void resize(int *value) {
  assert(value != nullptr);
  // some code here ...
  measure(value, 0);
  // other code here ...
}
```

In other words, you need to decide for yourself when and where values must explicitly be checked to avoid violating preconditions.

Assertion violations

If a precondition is violated during program testing and debugging, then there is a bug in the code that called the function containing the precondition. The bug must be found and fixed.

If a postcondition is violated during program testing and debugging, then there is a bug in the function containing the precondition. The bug must be found and fixed.

Assertions and error-checking

It is important to distinguish between program errors and run- time errors:

1. A logic error is a bug, and should never occur.
2. A run-time error can validly occur at any time during program execution.

Assertions are not a mechanism for handling run-time errors. For example, an assertion violation caused by the user inadvertently entering a negative number when a positive number is expected is poor program design. Cases like this must be handled by appropriate error-checking and recovery code [such as requesting another input], not by assertions.

Realistically, of course, programs of any reasonable size do have bugs, which appear at run-time. Exactly what conditions are to be checked by assertions and what by run-time error-checking code is a design issue. Assertions are very effective in reusable libraries, for example, since

- the library is small enough for it to be possible to guarantee bug-free operation, and
- the library routines cannot perform error-handling because they do not know in what environment they will be used.

At higher levels of a program, where operations are more complex, run-time error-checking must be designed into the code.

Turning assertions off

By default, C++ compilers generate code to check assertions at run-time. Assertion-checking can be turned off by defining the `NDEBUG` flag to your compiler, either by adding the following directive to a source file

```
#define NDEBUG
```

or by calling your compiler with the `-D` option:

```
g++ -D NDEBUG ...
```

This should be done only you are confident that your program is operating correctly, and only if the program's run-time efficiency is a pressing concern.

Exceptions

In the preceding section, we looked at how assertions help us to detect programming errors. However, there are many critical situations that we cannot prevent even with the smartest programming, like files that we need to read but which are deleted. Or our program needs more memory than is available on the actual machine. Other problems are preventable in theory but the practical effort is disproportionally high, e.g., to check whether a matrix is regular is feasible but might be as much or more work than the actual task. In such cases, it is usually more efficient to try to accomplish the task and check for exceptions along the way.

Like most programming languages, C++ provides a mechanism to help deal with errors: exceptions. The fundamental idea is to separate detection of an error [which should be done in a called function] from the handling of an error [which should be done in the calling function] while ensuring that a detected error cannot be ignored. Nothing makes error handling easy, but exceptions make it easier.

Let's study exceptions through the mechanism of finding the roots of a quadratic equation of the form:

$$ax^2 + bx + c = 0$$

We can solve the equation for its roots with this formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Run-time errors

For simplicity, we'll only compute a single root [the $+$ part of the $\pm\sqrt{b^2 - 4ac}$]. Function `qroot()` for [*q*uadratic *r*oot] looks like this:

```
1  double qroot(double a, double b, double c) {
2    double discriminant = b*b - 4.0*a*c;
3    return (-b + sqrt(discriminant)) / (2.0*a);
4  }
```

If your program has no compile-time errors and no link errors, it'll run. This is where the fun really starts. When you write the program you're able to detect errors, but it is not always easy to know what to do with an error once you catch it at run time. Consider:

```
1  #include <iostream>
2  #include <cmath>
3
4  double qroot(double a, double b, double c) {
5    double discriminant = b*b - 4.0*a*c;
6    return (-b + sqrt(discriminant)) / (2.0*a);
7  }
8
9  int main() {
10   double a = 1.0, b = 5.0, c = 2.0;
11   std::cout << "qroot: " << qroot(a, b, c) << '\n';
```

```
12    a = 1.0; b = 2.0; c = 5.0;
13    std::cout << "qroot: " << qroot(a, b, c) << '\n';
14    a = 0.0; b = 5.0; c = 2.0;
15    std::cout << "qroot: " << qroot(a, b, c) << '\n';
16 }
```

The output from this program looks like this:

```
1 qroot a=1, b=5, c=2: -0.438447
2 qroot a=1, b=2, c=5: -nan
3 qroot a=0, b=2, c=5: -nan
```

The call on line 13 of the program computes the root of a quadratic equation with $a = 1, b = 2, c = 5$. The discriminant evaluates to $b^2 - 4ac = 4 - 20 = -16$. Evaluating the square root of a negative value with standard library function `sqrt()` declared in `<cmath>` will result in an invalid value.

Did you notice something wrong with line 15? If not, look again: this call computes the root of a quadratic equation with $a = 0, b = 5, c = 2$. The denominator will be 0.0, so that `(-b + sqrt(discriminant)) / (2.0*a)` divides by zero. This leads to a hardware-detected error that terminates the program with some cryptic message relating to hardware. This is the kind of error that you - or your users will have to deal with if you don't detect and deal sensibly with run-time errors. Most people have low tolerance for such "hardware violations" because to anyone not intimately familiar with the program all the information provided is "Something went wrong somewhere!" That's insufficient for any constructive action, so we feel angry and would like to yell at whoever supplied the program.

So, let's tackle the problem of argument errors with function `qroot()`. We've two obvious alternatives:

1. Let the caller of `qroot()` deal with bad arguments.
2. Let `qroot()` [the called function] deal with bad parameters.

The caller deals with errors

Let's try the first alternative first. That's the one we'd have to choose if `qroot()` was a function in a library where we couldn't modify it. For better or worse, this is the most common approach. Protecting the call of `qroot()` in `main()` is relatively easy:

```
1 if (b*b-4.0*a*c < 0.0) error("discriminant is negative!!!");
2 if (a == 0.0) error("division by zero!!!");
3 std::cout << "qroot: " << qroot(a, b, c) << '\n';
```

Really, the only question is what to do if we find an error. Here, we've called a function `error()` which we assume will do something sensible. For trivial programs, a sensible approach is to print the error message passed as an argument to `error()` and then terminate the program with standard library function `std::terminate()` [which is declared in `<exception>` and is the usual way for C++ to abort a program]:

```
#include <iostream>
#include <cmath>
#include <exception>

void error(std::string const& msg) {
  std::cout << msg << '\n';
  std::terminate();
}

double qroot(double a, double b, double c) {
  double discriminant = b*b - 4.0*a*c;
  return (-b + sqrt(discriminant)) / (2.0*a);
}

int main() {
  double a = 1.0, b = 5.0, c = 2.0;
  if (b*b-4.0*a*c < 0.0) error("discriminant is negative!!!");
  if (a == 0.0) error("division by zero!!!");
  std::cout << "qroot: " << qroot(a, b, c) << '\n';

  a = 1.0; b = 2.0; c = 5.0;
  if (b*b-4.0*a*c < 0.0) error("discriminant is negative!!!");
  if (a == 0.0) error("division by zero!!!");
  std::cout << "qroot: " << qroot(a, b, c) << '\n';

  a = 0.0; b = 5.0; c = 2.0;
  if (b*b-4.0*a*c < 0.0) error("discriminant is negative!!!");
  if (a == 0.0) error("division by zero!!!");
  std::cout << "qroot: " << qroot(a, b, c) << '\n';
}
```

The output from the above program looks like this:

```
qroot: -0.438447
discriminant is negative!!!
terminate called without an active exception
Aborted
```

Look at that code! Are you sure it is correct? Do you find it pretty? Is it easy to read? Actually, we find it ugly and therefore error-prone. We've more than trebled the size of the code and must expose an implementation detail [computing the discriminant]. There has to be a better way! Look at the original code:

```
a = 1.0; b = 2.0; c = 5.0;
std::cout << "qroot: " << qroot(a, b, c) << '\n';
```

It may be wrong, but at least we can see what it is supposed to do. We can keep this code if we put the check inside `qroot()`.

The callee deals with errors

Checking for valid parameters within `qroot()` is easy, and `error()` can still be used to report a problem:

```
double qroot(double a, double b, double c) {
  double discriminant = b*b - 4.0*a*c;
  if (discriminant < 0.0) error("discriminant is negative!!!");
  if (a == 0.0) error("division by zero!!!");
  return (-b + sqrt(discriminant)) / (2.0*a);
}
```

This is rather nice, and we no longer have to write a test for each call of `qroot()`. For a useful function that we call 500 times in a large program, that can be a huge advantage. Furthermore, if anything to do with the error handling changes, we only have to modify the code in one place. We don't have to search the whole program for calls to `qroot()`. Furthermore, that one place is exactly where the parameters are to be used, so all the information we need is easily available for us to do the check.

Checking parameters in the function seems so simple, so why don't people do that always? Inattention to error handling is one answer, sloppiness is another, but there are also respectable answers:

- *We can't modify the function definition:* The function is in a library that for some reason can't be changed. Maybe it's used by others who don't share your notions of what constitutes good error handling. Maybe it's owned by someone else and you don't have the source code. Maybe it's in a library where new versions come regularly so that if you made a change, you'd have to change it again for each new release of the library.
- *The called function doesn't know what to do in case of error:* This is typically the case for library functions. The library writer can detect the error, but only you know what is to be done when an error occurs.
- *The called function doesn't know where it was called from:* When you get an error message, it tells you that something is wrong, but not how the executing program got to that point. Sometimes, you want an error message to be more specific.
- *Performance:* For a small function the cost of a check can be more than the cost of calculating the result. For example, that's the case with `qroot()` where the check also more than doubles the size of the function [that is, the number of machine instructions that need to be executed, not just the length of the source code]. For some programs, that can be critical, especially if the same information is checked repeatedly as functions call each other, passing information along more or less unchanged.

So what should you do? Check your parameters in a function unless you've a good reason not to.

Error reporting

Let's consider a slightly different question from the perspective of the called function: Once the called function has checked a set of parameters and found an error, what should the function do? Sometimes the function can return an *error value*. For example:

```
#include <iostream>
#include <cmath>

bool qroot(double a, double b, double c, double& root) {
  double discriminant = b*b - 4.0*a*c;
  if (discriminant < 0.0 || a == 0.0) {
    return false; // indicates there is a problem
  }
  root = (-b + sqrt(discriminant)) / (2.0*a);
  return true; // indicates all is well
}

int main() {
  double a = 1.0, b = 5.0, c = 2.0, result;
  if (qroot(a, b, c, result)) {
    std::cout << "qroot: " << result << '\n';
  } else {
    std::cout << "qroot failed because of bad arguments\n";
  }

  a = 1.0; b = 2.0; c = 5.0;
  if (qroot(a, b, c, result)) {
    std::cout << "qroot: " << result << '\n';
  } else {
    std::cout << "qroot failed because of bad arguments\n";
  }

  a = 0.0; b = 5.0; c = 2.0;
  if (qroot(a, b, c, result)) {
    std::cout << "qroot: " << result << '\n';
  } else {
    std::cout << "qroot failed because of bad arguments\n";
  }
}
```

The output now looks like this:

```
qroot: -0.438447
qroot failed because of bad arguments
qroot failed because of bad arguments
```

This way, we can have the called function do the detailed checking, while letting each caller handle the error as desired. This approach seems like it could work, but it has a couple of problems that make it unusable in many cases:

- Now both the called function and all callers must perform tests. The callers have only a simple test to do but must still write that test and decide what to do if it fails.
- A caller can forget to test. That can lead to unpredictable behavior further along in the program.
- Many functions do not have an "extra" return value that they can use to indicate an error. For example, a function that reads an integer from input [such as `std::cin`'s operator `>>`] can obviously return any `int` value, so there is no `int` that it could return to indicate failure.

The second case above - a caller forgetting to test - can easily lead to surprises. For example:

```
int main() {
  double a = 1.0, b = 5.0, c = 2.0, result;
  qroot(a, b, c, result);
  std::cout << "qroot: " << result << '\n';

  a = 1.0; b = 2.0; c = 5.0;
  qroot(a, b, c, result);
  std::cout << "qroot: " << result << '\n';

  a = 0.0; b = 5.0; c = 2.0;
  qroot(a, b, c, result);
  std::cout << "qroot: " << result << '\n';
}
```

The output now looks like this:

```
qroot: -0.438447
qroot: -0.438447
qroot: -0.438447
```

Do you see the errors? This kind of error is hard to find because there is no obvious "wrong code" to look at: the error is the absence of a test. There is another solution that deals with these problems: using exceptions.

Better way with exceptions

Exceptions provide a mechanism that allows us to combine the best of the various approaches to error handling explored so far. The basic idea is that if a function finds an error that it cannot handle, it does not `return` normally; instead, it `throw`s an exception indicating what went wrong. Any direct or indirect caller can `catch` the exception, that is, specify what to do if the called code used `throw`. A function expresses interest in exceptions by using a `try` block listing the kinds of exceptions it wants to handle in the `catch` parts of the `try`-block. If no caller catches an exception, the program terminates. So, exception handling in C++ involves:

- `throw` expressions, which the detecting part uses to indicate that it encountered something it can't handle. We say that a `throw` raises an exception.

- `try` blocks, specifies the region of code designated as area where run-time error can occur. A `try` block starts with keyword `try` and ends with one or more `catch` clauses. Exceptions thrown from code executed inside a `try` block are usually handled by one of the `catch` clauses. Because they handle the exception, `catch` clauses are also known as *exception handlers*.
- A set of `exception` classes that are used to pass information about what happened between a `throw` and an associated `catch`.

An outline of `try` block and `catch` clause looks like this:

```
int main() {
  // code here ...
  try {
    // code that might throw an exception and must be protected
  } catch (???) {  // which kind(s) of exceptions to catch?
    // code that will handle the exception from try block above
  }
  // more code here ...
}
```

The most important new idea from the above code is the `try` block. It tries to execute statements in the `{ }` block that follow keyword `try`. If an exception occurs anywhere in these statements, then it stops executing them and continues with the other set of `{ }`-enclosed statements that begin with keyword `catch`, and the type of exception it is catching. Exceptions that are not caught end up calling standard library function `std::terminate()` that will terminate the program.

If statements between `try` and `catch` execute without throwing an exception, then the program skips the `catch` clause entirely and continues with the next statements [labeled `// more code here ...` in above code fragment].

You can catch multiple exceptions from a `try` block:

```
int main() {
  // code here ...
  try {
    // code that might throw an exception and must be protected
  } catch (char const *p) { // catch value of type const char pointer
    // code to handle char pointer exception from try block above
  } catch (int i) { // catch value of type int
    // code to handle int exception from try block above
  } catch (exception e) { // catch an "exception" object
    // code to handle exception object from try block above
  } catch (...) {
    // deal with all other exceptions
  }
  // more code here ...
}
```

A `catch`-block with an ellipsis, i.e., three dots literally, catches all exceptions. Obviously, the `catch`-all handler should be the last one.

`qroot()` with `throw` expressions

Refactoring `qroot()` with `throw` expressions will look like this:

```
double qroot(double a, double b, double c) {
  double discriminant = (b * b) - (4.0 * a * c);

  // protected against sqrt(-x) and division by 0
  if (discriminant < 0.0) {
    throw(discriminant);       // throw double
  } else if (a == 0.0) {
    throw("Division by 0.");  // throw const char *
  }
  // We only reach this point if no exception was thrown
  return (-b + sqrt(discriminant)) / (2.0 * a);
}

int main() {
  try { // protect code
    std::cout << "qroot a=3, b=2, c=1: " << qroot(3, 2, 1) << '\n';
  } catch (char const *message) { // catch const char pointer exception
    std::cout << message << '\n';
  } catch (double value) {  // catch a double exception
    std::cout << value << '\n';
  }
}
```

The output from the program is:

```
qroot a=3, b=2, c=1: -8
```

Unwinding of stack after exception

Consider the following code:

```
double qroot(double a, double b, double c) {
  double discriminant = (b * b) - (4.0 * a * c);

  // protected against sqrt(-x) and division by 0
  if (discriminant < 0.0) {
    throw(discriminant);       // throw double
  } else if (a == 0.0) {
    throw("Division by 0.");  // throw const char *
  }
  // We only reach this point if no exception was thrown
  return (-b + sqrt(discriminant)) / (2.0 * a);
}

void f1() {
  std::cout << "Starting f1...\n";
```

```
16    qroot(???);  // program flow depends on this call
17    std::cout << "Ending f1...\n";
18  }
19
20  void f2() {
21    std::cout << "Starting f2...\n";
22    f1();
23    std::cout << "Ending f2...\n";
24  }
25
26  int main() {
27    try { // protect code
28      std::cout << "Starting main...\n";
29      f2();
30      std::cout << "Ending main...\n";
31    } catch (const char *msg) { // catch a const char pointer exception
32      std::cout << msg << '\n';
33    } catch (double value) { // catch a double exception
34      std::cout << value << '\n';
35    }
36  }
```

When line 16 is replaced with the call:

```
1  qroot(1.0, 5.0, 3.0);
```

`qroot()` does not throw an exception and the program's output is:

```
1  Starting main...
2  Starting f2...
3  Starting f1...
4  Ending f1...
5  Ending f2...
6  Ending main...
```

When line 16 is replaced with the call:

```
1  qroot(0.0, 5.0, 3.0);
```

`qroot()` will execute expression `throw("Division by 0.")` and the program's output is:

```
1  Starting main...
2  Starting f2...
3  Starting f1...
4  Division by 0.
```

In complicated systems the execution path of a program may pass through multiple `try` blocks before encountering code that throws an exception. For example, a `try` block might call a function that contains a `try`, which calls another function with its own `try`, and so on. Our program is simpler. The `try` block in `main()` has a single statement `f2();` and two `catch` clauses. Function `f2()` has a call to function `f1()` in between two print statements. In turn, `f1()` has a call to `qroot()` in between two print statements.

Since $a = 0$ in call `qroot(0.0, 5.0, 3.0)`, function `qroot()` will throw an exception with expression `throw("Division by 0.")`. This will cause `qroot()` to terminate the current function and transfer control to a handler that will know how to handle this error. The search for a handler reverses the call chain. When an exception is thrown, the function that threw the exception is searched first. Since `qroot()` doesn't have a `catch` clause, it terminates execution. The function that called `qroot()` which happens to be function `f1()` is searched next. Since function `f1()` also doesn't have a `catch` clause, it also terminates. The caller of function `f1()` which happens to be function `f2()` is searched next. Again function `f2()` doesn't have a `catch` clause, it also terminates and function `main()` that called `f2()` is searched next. Since function `main()` implements a `catch` clause of type `char const*`, the execution of function `main()` is interrupted and jumps to the `catch(char const *msg)` block.

If no appropriate `catch` is found, execution is transferred to library function `std::terminate()` which terminates and aborts further execution of the program. Exceptions that occur in programs that don't define any `try` blocks are handled in the same manner. After all, if there are no `try` blocks, there can be no handlers. If a program has no `try` blocks and an exception occurs, then `std::terminate()` is called and the program is exited.

Rethrowing exceptions

Sometimes a single `catch` cannot completely handle an exception. After some corrective actions, a `catch` may decide that the exception must be handled by a function further up the call chain. A `catch` passes its exception out to another `catch` by rethrowing the exception. A rethrow is a `throw` that is not followed by an expression:

```
throw;
```

In the following program, the `catch` clause of function `f1()` catches the exception from `qroot()` and then rethrows the exception:

```
void f1() {
  try {
    std::cout << "Starting f1...\n";
    qroot(0.0, 5.0, 3.0);  // program flow depends on this call
    std::cout << "Ending f1...\n";
  } catch (const char *msg) {
    std::cout << "Caught exception f1...\n";
    throw;
  }
}

```

```
void f2() {
  std::cout << "Starting f2...\n";
  f1();
  std::cout << "Ending f2...\n";
}

int main() {
  try { // protect code
    std::cout << "Starting main...\n";
    f2();
    std::cout << "Ending main...\n";
  } catch (const char *msg) { // catch a const char pointer exception
    std::cout << msg << '\n';
  } catch (double value) { // catch a double exception
    std::cout << value << '\n';
  }
}
```

The program's output is:

```
Starting main...
Starting f2...
Starting f1...
Caught exception f1...
Division by 0.
```

Exception classes

You can throw any type of value, primitive or object. For example, you can throw an `int` value:

```
throw 3;
```

And later catch it with a clause that uses the same type:

```
try {
  // call function that may throw exception ...
} catch (int a) {
  // deal with exception here ...
}
```

But while this is legal, it is usually not a good idea. What is the meaning of the value `3` as an error? Why not `4`? There just isn't enough information in a primitive value to make sense of the error. Throwing enumerated constants or strings makes slightly more sense, but you must be careful. Implicit conversions, such as from `int` to `double` or from `char const*` to `std::string`, are not performed when a value is thrown. The following will probably not operate as the programmer intended:

```
try {
  // call function that may throw exception ...
  throw "vector: out of bounds access\n";
} catch (std::string const& err) {
  std::cerr << err << "\n";
}
```

The reason is the literal string is type `char const*`. While this is often converted into a `std::string`, it is not the same type. Because thrown values are not converted, the `catch` clause will not be invoked.

In order to avoid these problems, it is much more common for programs to throw and catch *exception objects*. Since these objects are of user-defined types, they can contain more information and are therefore more flexible. The compiler deals with creating and destroying the exception objects automatically. For example:

```
class MyApplicationError {
  std::string reason;
public:
  MyApplicationError(std::string const& r) : reason(r) {}
  std::string const& what() const { return reason; }
};
```

Errors are now indicated by throwing an instance of this class:

```
try {
  // do stuff ...
  throw MyApplicationError("illegal value");
  // do more stuff ...
} catch (MyApplicationError const& e) {
  std::cerr << "Caught exception " << e.what() << "\n";
}
```

Note that an object is normally caught as a reference parameter. There are two reasons for this. First, it is more efficient, because it avoids the object being duplicated by means of a copy constructor. Second, it avoids the object-slicing problem that can occur if inheritance is used to define the class. Why would inheritance be involved with exceptions? While the programmer is free to select any type of value to be used in a `throw` statement, it is often a good idea to reuse classes from the standard library.

The standard library provides a hierarchy of standard exception classes that are declared in `<stdexcept>`. The classes can be used in two ways. One way is to simply create instances of these standard objects. This technique is illustrated in the following `throw` statement:

```
if (p < 0 || n < 0) {
  throw std::logic_error("illegal parameter");
}
```

Another possibility is to use inheritance to define your own exception types as more specialized categories of the standard classes:

```
class MyError : public std::logic_error {
public:
  MyError(std::string const& reason) : std::logic_error(reason) {}
};
```

Because a `MyError` *is-a* `std::logic_error`, you can still catch it with a `catch (std::logic_error const& e)` clause - that is the reason for using inheritance. Alternatively, you can supply a `catch (MyError const& e)` clause that only catches `MyError` objects and not other logic errors. You can even do both:

```
try {
  // code
} catch (MyError const& e) {
  // handler1 code
} catch (std::logic_error const& e) {
  // handler2 code
} catch (std::bad_alloc const& e) {
  // handler3 code
}
```

In this situation, the first handler catches all errors of type `MyError`, the second handler catches the logic errors that are not `MyValue` errors, and the third handler catches the `std::bad_alloc` exception that is thrown when `new` operator runs out of memory. Within the `catch` clause the error string can be accessed using member function `what()`.

> ***The order of `catch` clauses is important. When an exception occurs, the exception handling mechanism proceeds top to bottom to look for a matching handler and executes the first one found. You should match a derived class before matching its base class.***

Modifying our `Str` class

Recall class `Str` developed in lectures:

```
class Str {
public:
  // ctors, dtor, etc. ...
  char& operator[](size_t index);
  char const& operator[](size_t index) const;
private:
  size_t len;
  char*  data;
};
```

Overloaded functions `operator[]()` were defined without any error handling:

```
char& Str::operator[](size_t index) {
  return data[index];
}

char const& Str::operator[](size_t index) const {
  return data[index];
}
```

We'll add an exception class to provide a flexible way to deal with out-of-bounds access errors in the two `operator[]()` functions:

1. We'll create a class that will be used to *announce* subscript exceptions:

```
class SubscriptError {
public:
  SubscriptError(int Subscript) : subscript(Subscript) {};
  int GetSubscript() const { return subscript; }
private:
  int subscript;
};
```

2. Write code to `throw` the exception, if necessary:

```
char const& String::operator[](size_t index) const {
  if (index >= len) { // make sure index is valid
    throw SubscriptError(index); // throw exception if invalid
  }
  return data[index];         // return the char at index
}
```

3. In client code, wrap the potentially "unsafe" code in a `try` block and include a `catch` block in the client to handle the exception:

```
int main() {
  Str s("Hello"); // Create string "Hello"
  try {
    std::cout << s[0] << '\n'; // Get the first character and print it
    s[9] = 'C';                // Attempt to change tenth character
    std::cout << s << '\n';
  } catch (SubscriptError const& se) {
    std::cout << "Bad subscript: " << se.GetSubscript() << '\n';
  }
}
```

As explained earlier, it is common to derive all exception objects from class `exception` declared in `<stdexcept>`:

```
class exception {
public:
  exception () noexcept;
  exception (const exception&) noexcept;
  exception& operator= (const exception&) noexcept;
  virtual ~exception();
  virtual const char* what() const noexcept;
};
```

So, we derive class `SubscriptError` from class `exception`:

```
class SubscriptError : public std::exception {
private:
  int subscript;
public:
  SubscriptError(int Subscript) : subscript(Subscript) {};
  int GetSubscript() const { return subscript; }
  virtual const char* what() const noexcept {
    static std::string msg;
    msg = "Subscript error at index: ";
    msg += std::to_string(subscript);
    return msg.c_str();
  }
};
```

Now, the client should call function `what`:

```
int main() {
  Str s{"Hello"};  // create string "Hello"
  try {
    std::cout << s[0] << '\n'; // get 1st character and print it
    s[9] = 'C';                // attempt to change ninth character
    std::cout << s << '\n';
  } catch (SubscriptError const& se) {
    std::cout << se.what() << '\n';
  }
}
```

Keyword `noexcept`

C++11 provides keyword `noexcept` to specify that a function cannot throw, or is not prepared to throw. For example:

```
void foo() noexcept;
```

declares that function `foo` won't throw. The benefit of this qualification is that calling code never needs to check for thrown exceptions after `foo`. If an exception is thrown despite the qualification, the program is terminated.