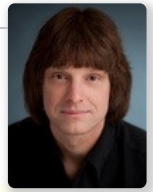
Exception Handling



One of the most influential authors in the C++ world is Scott Meyers. His **Effective C++** series is the gold standard for learning how to use C++ correctly. He wrote:

> *"Forty years ago, goto-laden code was considered perfectly good practice. Now we strive to write structured control flows. Twenty years ago, globally accessible data was considered perfectly good practice. Now we strive to encapsulate data. Ten years ago, writing functions without thinking about the impact of exceptions was considered good practice. Now we strive to write exception-safe code.*
>
> *Time goes on. We live. We learn."*
>
> *– Scott Meyers, author of Effective C++ and one of the leading experts on C++. [Mey05]*

In a perfect world, users would never mistype a URL, programs would have no bugs, disks would never fill up, and your Wi-Fi would never go down. We, however, don't live in a perfect world. You may wake up, planning to head off for school just like any other day, but sometimes, **something unexpected appears at your front door**.

Fragile code ignores that possibility, but **robust** code plans for any eventuality.

Throwing Exceptions

The C++ exception handling feature allows programs to deal with real life circumstances. The system is broken into three parts:

- `try` blocks
- `catch` blocks
- `throw` statements

Let's start by looking at `throw` statements, so we can finish up our functions. Click on the running man to open the sample program we've been working on in **Replit**, **Fork it**, and we'll continue by learning how to apply exception handling.

When a function encounters a situation from which it cannot recover–for example, a call to `stoi("twelve")`—you can **report the error** by using the `throw` keyword to notify the nearest appropriate error-handling code that something has gone wrong.

```
istringstream in(str);
int result = 0;
if (in >> result) return result;
throw … // signal error at this point using throw
```

Inside `stoi()`, if the read succeeds, return the result. If the read **fails**, **jump out of the function**, looking for the closest error handler. You do that with `throw`.

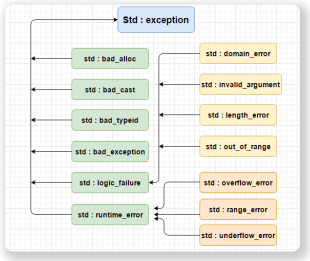
Like `return`, `throw` accepts a single parameter, an object which provides information about the error which occurred.

What Should You Throw?

Unlike Java, in C++, it is legal to throw any kind of object, not just members of an exception class hierarchy. So, in C++, all of these are legal:

```
if (len < 3) throw "Too short"s; // throw a string
if (a > b) throw 42;     // throw an integer error code
if (b < c) throw 3.5;    // throw a double
```

The question is, though, what **should** `stoi()` throw when an error occurs? The library documentation says that the function throws an `invalid_argument` exception.



The header file, `<stdexcept>` defines this and several other classes that let us specify what specific error triggered the exception, similar to the `Exception` class hierarchy from the Java Class Libraries. (Click the image to enlarge it.)

The `invalid_argument` exception is ideal because

- its constructor takes a `string` argument, useful for error messages.
- it has a member function `what()` that returns what the error was

Include `<stdexcept>`, and rewrite the `throw` statement like this:

```
throw invalid_argument(str + " not an int.");
```

The *try* and *catch* Blocks

To handle and recover from errors, you need a combination of `try` and `catch` blocks. A `try` block is simply a block of code where runtime errors might occur; write the keyword `try`, and then surround the appropriate code in a pair of curly braces, like this:

```
string str;
cin >> str;
try {
    int a = stoi(str);
    cout << "a = " << a << endl;  // skipped if exception thrown
}
catch (const invalid_argument& e) {
    cerr << e.what() << endl;     // if exception thrown
}
```

There are three things to note here.

1. If the user enters `"one"` then `stoi()` will throw an exception, and control **immediately** breaks out of the `try` block and **jumps** to the `catch` block, skipping the line that prints `a`. We're guaranteed that **the rest of the code in the `try` block will not execute**, preventing error cascades.
2. If an exception is thrown and caught, control **does not return** to the `try` block. Instead, control resumes directly **following** the `try`/`catch` pair.
3. Catch exception classes from the standard library **by const reference**. This avoids making copies and **enables polymorphism**. You will get a compiler warning if you don't do this.

If no error occurs, then all of the code inside the `try` block executes as normal, and the subsequent `catch` block is ignored. The function `what()` will return the `string` used to construct the exception object. Here, its used to print the error message in the `catch` block.

Other *catch* Blocks

If your function may throw more than one exception, add cascading `catch` blocks following the **`try`** block, each designed to handle a different type of exception, like this:

```
try {
    int n = stoi(str);    // may throw an invalid_argument exception
    int x = str.at(5);    // may throw an out_of_range exception
    // ... other statements that may throw exceptions
}
catch (const invalid_argument& e) {
    // handle errors from stoi
}
case (const out_of_range& e) {
    // handle errors from at()
}
case (...) {
    // handle any exceptions not previously caught
}
```

The last block, with the **`...`** in the argument list is the **catch all** handler. It catches**any exceptions** thrown in the **`try`** block, **not previously caught.** The **catch all** hander **only** catches thrown exceptions, not other errors like segmentation faults or **operating system traps or signals** such as those caused by dividing by zero. Code jumps to **only one** of the **`catch`** blocks shown here. If no exceptions are thrown, then no **`catch`** blocks are entered.

Finish the Sample

After adding **`try-catch`** to **`main()`**, print an error message inside the **`catch`** block. Use **`cerr`**, print the word *"Error: "* and then call **`e.what()`** like this:

```
catch (const invalid_argument& e) {
    cerr << "Error: " << e.what() << endl;
}
cout << "--program done--" << endl;
```

Now your program should work the same whether compiled with C++17 or C++98 (even if the error messages differ between versions.)

The to_string() Functions

In the last few lessons, you've looked at the new C++ 11 functions for converting `string` values to **numbers**, such as `stoi()`, `stod()` and so on. Along with these functions, C++ 11 also introduced a group of which go the other direction, converting **numbers to `string`**. Instead naming the functions `itos()` or `dtos()`, (similar to `stoi()`), these **overloaded functions** are **all** named `to_string()`.

Just like the string-to-number conversion functions, the `to_string()` functions do not appear in C++98. So, as you did with `stoi()` and friends, you must to write your our own, using the `ostringstream` class like this:

```
string to_string(int n)
{
    ostringstream out;
    out << n;
    return out.str();
}
```

You can easily **overload** your version of `to_string()` to work with other types.

```
string to_string(double n) { . . . }
string to_string(long n) { . . . }
string to_string(unsigned n) { . . . }
```

The body of each function will have **the identical code** to the `int` version. Wrap all of these up inside an `#if __cplusplus <= 199711L` **preprocessor directive**, and you should be able to use the `to_string()` function in both C++ 17 and in C++ 98.

Function Templates

I hope the code on the preceding page bothers you as much as it bothers me. It doesn't take much to notice that **the body of each function is identical**. Why can't we define one version of the function that takes any kind of argument?

Surprise! We can!

C++ functions with **generic types** are called **function templates**. (In Java these are called generic functions, but **template** is used more often in C++).

You define a function template with the same syntax as a regular function, **preceded by** the `template` keyword and a series of **template parameters** inside angle-brackets `<>`.

```
1  template <typename T>
2  std::string to_string(const T& n)
3  {
4      std::ostringstream out;
5      out << n;
6      return out.str();
7  }
```

The template parameters are separated by commas, and use **generic template type names**: names preceded by either the `class` or `typename` keyword followed by an identifier. Both keywords **are synonyms** in template declarations

When using separate compilation:

- Function templates ar **placed inside the header file**, unlike normal functions, which are placed inside the implementation file.
- You must **fully qualify all library types**, such as `string` and `ostringstream` in the example shown here, since you are not allowed to add `using namespace std;` to a header file.

Template Instantiation

Instantiating a template means using the template to create a function using specific types for its template parameters. There are two ways to do this, **implicitly** and **explicitly**. The functions generated by a pararticular instantiation are called **template functions** (which is, unfortunately, easily confused with function template).

Let's look at another example:

```
template <typename T>
void print(const T& val)
{
    cout << val;
}
```

When you **call the function**, the compiler will **implicitly deduce** the types of arguments you pass and then generate and call a **version of the function** with those parameters.

```
string s = "hello";
print(23);        // generates print(const int&)
print(str);       // generates print(const string&)
```

The first call **implicitly** instantiates (and calls) a **`print(int)`** function, while the second generates and calls a **`print(string)`** template function.

Template Argument Deduction

The process used to determine the type of each function parameter from the arguments used in the **function call** is called **template argument deduction**.

None of the automatic arithmetic conversions that happen with normal functions take place, **except**:

- You may call a template having a `const` parameter with a non-const argument. Calling `print(str)` in the preceding section is an example of this.
- If the function parameter is a **pointer**, you **may** pass an **array**. For instance, this template will be used if you call it with an array as the first argument:

```
template <typename T>
T sum_array(const T* a, size_t len)
{
    T result{};
    for (size_t i = 0; i < len; ++i)
        result += a[i];
    return result;
}
```

Multiple Template Arguments

Suppose you have a template function like this:

```
template <typename T>
T addem(const T& a, const T& b)
{
    return a + b;
}
```

You can **call** the function in any of these ways:

```
string a{"hello"}, b{" world"};
cout << addem(3, 5) << endl;
cout << addem(4.5, 2.5) << endl;
cout << addem(a, b) << endl;
```

But, you **cannot** call the function like this:

```
cout << addem(3.5, 2) << endl;
```

The compiler does not know **what type to substitute** for `T` in the template. You could, however, write the template with **two template parameters**, like this:

```
template <typename T, typename U>
T addem(const T& a, const U& b)
{
    return a + b;
}
```

Now the call `addem(3.5, 2)` uses `double` for type `T`, `int` for type `U`, and the function returns a `double`, (`5.5`) as you'd expect. However, what about that call `addem(2, 3.5);`? Now the function returns an `int`, (5) which is **not what you'd expect**. You can fix this in two ways.

Explicit Template Arguments

Suppose you wish to pass `3.5` to the `print(int)` version of the template. You may **explicitly instantiate** the function by specifying the type of template parameter inside angle brackets, when calling the function, like this:

```
print<int>(3.5);
```

Even though you pass a **`double`** as the function argument, the function is **instantiated** with the generic parameter **`T`** replaced by type **`int`**. So, this call truncates the fractional part of the argument before it prints the number, rather than generating an overloaded **`print(double)`** function.

To fix your **`addem()`** problem, you can just add an extra **template parameter** for the return type:

```
template <typename RET, typename T, typename U>
RET addem(const T& a, const U& b)
{
    return a + b;
}
```

Call the function by providing an explicit template argument: `addem<double>(2, 3.5);`. Here, the template parameter **`RET`** is replaced with **`double`**. That's a little awkward, but is the only way to handle this problem prior to C++11.

Inferred Return Types

Rather than having the caller explicitly instantiate the template and supply a return type, C++ 11 allows a template to **infer** the return type like this:

```
template <typename T, typename U>
auto addem(const T& a, const U& b)->decltype(a + b)
{
    return a + b;
}
```

Using `auto` as the formal template return type, and moving the **deduced** return type so it **follows** the argument list allows the compiler to replace the return type with the **declared type** of the expression `a+b`. (That's what the `decltype` keyword calculates at compile time.

In C++14 this was further simplified. We can now write the `addem` template like this. We **don't** need the trailing return type or `decltype`.

```
template <typename T, typename U>
auto addem(const T& a, const U& b)
{
    return a + b;
}
```

That doesn't mean, however, that you can write a template where only the return type differs. For instance, the following template will not compile because it can't determine whether to use `int` or `double` for the return type when called like `mybad(3, 4.5)`:

```
template <typename T, typename U>
auto mybad(const T& a, const U& b)
{
    if (a > 0) return a;
    return b;
}
```

Templates & Overloading

Suppose you wanted to print pointers differently than non-pointers, you can add an **overloaded template function** like this:

```
template <typename T>
void print(const T* p)
{
    cout << "Pointer: " << p << " ";
    if (p) cout << *p; else cout << "nullptr";
}
```

Now, what if you want floating-point numbers and Booleans to print differently than other kinds of values? You can add a pair of **explicit, non-template, overloaded functions**, like this:

```
void print(double val, int dec=2)
{
    cout << fixed << setprecsion(dec) << val;
}
void print(bool val)
{
    cout << boolalpha << val;
}
```

Now `print(2.5)` will print `2.50`, while `print(2.5, 4)` will print `2.5000`. Printing a *Boolean* expression will print `true` or `false`, not `0` or `1` like the original template.

When you overload template functions:

- Any call to a template function, where the template argument deduction succeeds, is a **viable member** of the overload **candidate set**.
- If there is a **non-template** function in the viable set, then **it is preferred**.

The **most specialized** template function in the viable set is preferred over the other template functions.