Chapter

14

Exception Handling

One of the most influential authors in the C++ world is Scott Meyers. Effective C++ 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. The C++ **exception handling** feature allows programs to deal with real life circumstances. The C++ exception handling system is broken into three parts:

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

Let's start by looking at **throw** statements, so you can finish up your functions.

Throwing Exceptions

Click on the running man to open the sample program we've been working on, 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.

```
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?

What should **stoi()** throw when an error occurs? The library documentation says it 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.

The **invalid_argument** exception is ideal because it

- 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.");
```

What Can You throw?

The **invalid_argument** exception is an **exception class**, part of a hierarchy of exception classes, defined in the C++ standard library. It is similar to the **Exception** classes in the Java Class libraries.

Unlike Java, however, in C++, it is legal to **throw any kind of object** (in the C++ sense). So, for instance, all of these are legal:

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

Using *try* and *catch*

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:

```
try {
    // code to run
}
```

If no error occurs, the code inside the **try** block executes as normal. If an exception is thrown inside of the **try** block, it can be caught by a special error-handler block, known as a **catch** block, specialized to handle that particular error.

catch Blocks

A **catch** block is defined like this:

```
catch (T& e)
{
    // Process exception e here
}
```

T represents the **type** of variable this **catch** clause handles. Each **catch** block must directly follow a **try** block. It's illegal to have one without the other. Here's an example:

```
try {
    cout << "stoi(\"42\")->" . . .
    . . .
}
catch (invalid_argument& e)
{
    cerr << e.what() << endl;
}
```

There are three things to note here.

1. When **stoi()** throws an exception, control immediately breaks out of the **try** block and jumps to the **catch** clause. 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 **by reference (`invalid_argument& e`)**. By passing by reference avoids unnecessary copies and **enables polymorphism**.

Other Catch Blocks

If your function may throw **more than one** exception, add **cascading `catch` clauses**, each designed to pick up a different type, like this:

```
try  { /* do something */ }
catch (int ie) { /* jumps here if code throws int */ }
catch (const string& se) { /* same for string */ }
catch (...) { /* catch-all handler */ }
```

Unlike Java, C++ exceptions don't have to throw an object of a standard exception class. You can **throw anything**, including **`int`**, **`string`** and so forth.

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**. 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. Normally, you'd use **`cerr`**, but the **CPP Shell** environment won't print that. Instead, use **`cout`**, print the word **`"Error: "`** and then call **`e.what()`** like this:

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

Now your program should work the same whether compiled with C++14 or C++98 (even if the error messages differ between versions.)

Introducing Templates

Starting in C++ 11 the standard library has functions which convert numbers to strings. Instead of functions named `itos()` or `dtos()`, (similar to `stoi()`), these **overloaded functions** are **all** named `to_string()`. Click the "running man" icon on the left for an example that uses `to_string()`.

These functions also don't appear in C++98, so 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 our version of `to_string()` to work with other types.

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

Here's a version for the type `double`, another for `long`, and yet another for `unsigned int`, which is necessary to avoid printing large unsigned values as negative numbers.

The body of each function will have the identical code which you've already seen. Wrap all of these up inside an `#if __cplusplus <= 199711L` preprocessor directive, and your code would compile and run both in C++ 14 and in C++ 98.

Function Templates

I hope the preceding code 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 you define one version of the function that takes any kind of argument? Surprise! You can!

C++ functions with **generic types** are called function templates. (In Java these are called generic functions, but the term 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** enclosed in angle-brackets `<>`.

```
template <template-parameters> function-declaration
```

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. The keywords **are synonyms** in template declarations

An Example Function Template

For instance, here is a generic `print()` function that works for any type.

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

Instead of passing by value, pass by `const` reference.

When using separate compilation:

- Function templates are **placed inside a header file**, unlike normal functions, which are placed in a `.cpp` file.
- **Fully-qualify all library types**, such as `string` since you don't put `using namespace std;` in a header file.

Template Instantiation

Instantiating a template means **creating a function** using particular types or values for its template parameters. There are two ways to do this, **implicitly** and **explicitly**. The generated functions are called **template functions** (which is, unfortunately, easily confused with function template)

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);
print(s);
```

The first call **implicitly** instantiates (and calls) `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(s)** in the preceding section is an example of this.
- If the function parameter is a **pointer**, you **may** pass an **array**. For instance:

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

Multiple Arguments of the Same Type

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 `T` and `int` for `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 function, you may **explicitly instantiate** the function by specifying the type of template parameter inside angle brackets, 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.

To fix your `addem()` problem, 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 prior to C++11.

Here is the more convenient syntax available in C++11 that doesn't require the caller to explicitly specify the natural return type:

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

In C++11 template argument deduction was extended to allow you to use the **types of the parameters** to determine the correct return type. 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`.

Templates & Overloading

You can overload template functions as well. When you do this:

- Any call to a template function, where the template argument deduction succeeds, is a **viable member** of the **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.

Non-Template Overloads

Consider the `print()` template function shown earlier. What if you want floating-point numbers and Booleans to print differently than other kinds of values? Add **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.

Template Overloads

What if you want to print **pointers** differently than non-pointers? Add an **overloaded template function** for that:

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

Finish Up

- Complete the **reading exercises (REX)** for this chapter.
- Complete the homework using the **CS50 IDE**. The link is on Canvas.
  a. Make sure you submit the assignment using **`make submit`**.
  b. Make sure you check the CS150 Homework Console to see that your scores got reported, before the beginning of the next lecture.
- Take the **pre-class reading quiz** on Canvas. You have two attempts.

See you in class or on the Canvas discussion board.