Lab: Error Flags and Exceptions

Learning Outcomes

- Gain experience in writing robust programs that handle exceptional conditions
- Gain experience in writing exception classes
- Gain experience with function pointers and function templates
- Gain experience in using string streams

Overview

As explained in class lectures, 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 will not be of interest to us in this exercise; instead, we will deal with unexpected behavior using error flags and exceptions.

Task

Suppose I'd like to define a function that divides two `int` values. C++ provides the binary `/` operator to implement division. Being pedantic, I read the C++ standard to know more about the binary `/` operator:

> The binary `/` operator yields the quotient, and the binary `%` operator yields the remainder from the division of the first expression by the second. If the second operand of `/` or `%` is zero the behavior is undefined

What does undefined behavior mean? Reading the standard further, we get this:

> **undefined behavior**
>
> behavior for which this document [that is, the C++ standard] imposes no requirements

That is, division by zero causes the program to generate non-meaningful results. This freaks me out because the division function will be used to pilot planes, cars, trains, and possibly control a few nuclear power plants. Being a diligent programmer [and also a good human being who wishes no harm on other humans and even machines], I know I must inform callers of my function about this "divide-by-zero" problem. I could use one of the following techniques to report to the caller that a "divide-by-zero" situation has occurred:

1. The function returns an error flag and writes the calculated value in an "output-only" reference parameter.

```cpp
// return true if denominator is non-zero; otherwise return false
// client will use result only if function returns true
bool divide1(int numerator, int denominator, int& value) {
  value = denominator ? (numerator / denominator) : 0;
  return denominator;
}
```

2. The function returns an object that aggregates the calculated value and an error flag.

```cpp
// return both error flag and result as a pair
// client will use second member only if first member is true
std::pair<bool, int> divide2(int numerator, int denominator) {
  bool success = denominator;
  int value = denominator ? (numerator / denominator) : 0;
  return {success, value};
}
```

3. The function returns the calculated value and uses global variable `errno` [from the C standard library] to indicate whether the result is valid.

```cpp
// global variable errno is defined in <cerrno>
// client will use return value only if errno is positive value
int divide3(int numerator, int denominator) {
  errno = denominator ? 1 : 0;
  return denominator ? (numerator / denominator) : 0;
}
```

4. The function returns the calculated value but reserves a specific value to indicate whether the result is valid.

```cpp
// error code is smallest int value on machine executing the program
// Recall from early lectures that class template numeric_limits<T>
// is declared in <limits>
int divide4(int numerator, int denominator) {
  if (!denominator) {
    return std::numeric_limits<int>::min();
  }
  return numerator / denominator;
}
```

5. The function returns the calculated value and indicates an error by throwing an exception.

```cpp
// if denominator is zero, throw an exception
int divide5(int numerator, int denominator) {
  if (!denominator) {
    throw hlp2::divide_by_zero{numerator};
  }
  return numerator / denominator;
}
```

Each of these error reporting techniques has its advantages and disadvantages. The particular technique used depends on the business problem being solved:

- Returning an aggregate object is fast and simple regardless of the occurrence of an exception, but requires the caller to handle the error flag in a particular way
- Returning an error flag allows outer functions in a nested sequence of function calls to handle errors, but artificially adds output parameters to functions

- Exceptions allow the separation of responsibilities between a function that identifies and reports an error [*throwing an exception*] and code that handles the error [*catching an exception*], without impacting the definition of any intermediate function in a nested sequence of function calls. However, the price to pay is the steeper learning curve required to understand the C++ exceptions mechanism.

As a software engineer you must know these five techniques; so this is what we practice in this exercise.

Implementation details

You will play the role of my client. In that role, you'll write a total of five function templates called `test?()` [assume `?` means one of `1` through `5`] with each function template calling the corresponding `divide?()` function [that is, your function template `test1()` will test my `divide1()`; your function template `test2()` will test my `divide2()`; and so on].

Why do you need to author `test?()` as function templates when `divide?()` divide two `int` values? Because a function template with template type parameter of type pointer to function provides a simpler syntactic alternative to declaring a function with a parameter of type pointer to function:

```cpp
// function definition with fourth parameter declared as
// "pointer to function taking an int and returning an int"
int* xform(int const *first, int const *last, int *dest,
           int (*transformer)(int)) {
  while (first != last) {
    *dest = transformer(*first);
    ++first; ++dest;
  }
  return dest;
}

// equivalent to above but defined as function template with
// compiler using automatic template argument deduction to deduce
// type of fourth parameter
template <typename F>
int* xform(int const *first, int const *last, int *dest, F transformer) {
  while (first != last) {
    *dest = transformer(*first);
    ++first; ++dest;
  }
  return dest;
}
```

Your `test?()` will be called with two `int` arguments [representing numerator and denominator] and a pointer to the corresponding `divide?()`. `test?()` will first write text to standard output stream identifying which `divide?()` will be called. Next, `test?()` must call `divide?()` and determine from the error reporting technique of `divide?()` whether a "divide-by-zero" has occurred. If not, `test?()` must write the value calculated by `divide?()`. If a "divide-by-zero" has occurred, `test?()` must throw a custom exception of type `hlp2::division_by_zero` that you must define. See here for more information on defining an exception class. Since `test5()` calls `divide5()` which itself throws an exception, `test5()` will need to rethrow the exception [thrown by `divide5()`]. The custom exception type must have a `std::string` data member that must be initialized with an error message. Since the error message must combine text and numeric arguments to the corresponding `divide?()`, you'll use class `std::stringstream` in the custom exception type's constructor. See here for more information on `std::stringstream`.

See the input and corresponding output files to identify the exact text that must be written to standard output stream by `test?()`.

There's one more thing to worry about. Suppose your `test?` function is called with its first two arguments specified by values read from standard input stream:

```
1  int numerator{}, denominator{};
2  std::cin >> numerator >> denominator;
3  hlp2::test1(numerator, denominator, divide1);
```

Also suppose the tester enters the following text in the console:

```
x 50
```

What happens is that both `numerator` and `denominator` retain their initial `0` values and `test1()` will report a "divide-by-zero". However, the tester has provided `50` for `denominator` and will be surprised to receive a "divide-by-zero" message. Let's see what's happening here.

1. On line 1, variables `numerator` and `denominator` are initialized to `0`.
2. On line 2, expression `std::cin >> numerator` causes `std::cin` to peek into the input stream buffer and see `x`. `std::cin` is expecting one or more digit characters [since `numerator` is declared as type `int`] and since `x` is not a digit, `std::cin` enters a fail state.
3. This causes `std::cin` to leave character `x` in the buffer and abort the entire `std::cin >> numerator >> denominator` expression [since `std::cin` is now in a fail state]. Notice that both `numerator` nor `denominator` have been left untouched through this entire process.

Input stream types [`std::istream`, `std::fstream`, and other types inherited from base class `std::basic_ios`] don't report input errors via exceptions. Instead, after calling member function `std::istream::operator>>()`, you can call member function `std::istream::fail()` to determine if `std::istream::operator>>()` was unsuccessful. These input stream types also implement a implicit type conversion operator as a member function [`std::istream::operator bool()`] to expose a shorthand for checking read errors.

What we've is an "invalid-input" error rather than a "divide-by-zero" error. Thus, the second part of your task is to write a class `stream_wrapper` that wraps an object of type `std::istream` such as `std::cin` so that it can be manipulated as a regular input stream [only support for `operator>>(T&)` is required], but on invalid input throws an exception of type `hlp2::invalid_input`. You must define this exception type so that an object of this type returns string literal `"Invalid input!"` as an error message. Defining this class will give you experience in reference data members and writing a class member function as a function template.

To summarize, you need to provide the following in file test.hpp:

1. Begin by including all necessary standard library headers.
2. All definitions must be in namespace `hlp2`.

3. Definition of exception class `division_by_zero` that *inherits* from base class `std::exception` [that is declared in header file `<exception>`]. Read the step-by-step description for implementing an exception class to assist with this task.
4. Definition of wrapper class `stream_wrapper` for `std::cin` that throws an exception of type `invalid_input` [which you must define]. Again class `stream_wrapper` must inherit from base class `std::exception`.
5. Definitions of five function templates named `test1` [to test my function `divide1`], `test2` [to test my function `divide2`], `test3` [to test my function `divide3`], `test4` [to test my function `divide4`], and `test5` [to test my function `divide5`].

test-driver.cpp contains code that will give you insights into defining `test?()`.

Submission Details

Please read the following details carefully and adhere to all requirements to avoid unnecessary deductions.

Submission files

You will be submitting file test.hpp.

Compiling, executing, and testing

Download test-driver.cpp; input files containing input values provided to the program; and correct output generated by the program in response to the input. Follow the steps from the previous exercises to refactor a makefile that can compile, link, and test your program.

File-level and function-level documentation

Every source and header file *must* begin with a *file-level* documentation block. This module will use Doxygen to tag source and header files for generating html-based documentation. In addition, every function that you declare and define and submit for assessment must contain *function-level documentation*. This documentation should consist of a description of the function, the inputs, and return value.

Submission and automatic evaluation

1. In the course web page, click on the appropriate submission page to submit the necessary files.
2. Please read the following rubrics to maximize your grade. Your submission will receive:
   - F grade if your submission doesn't compile with the full suite of g++ options.
   - F grade if your submission doesn't link to create an executable.
   - Your implementation's output must exactly match correct output of the grader (you can see the inputs and outputs of the auto grader's tests). There are only two grades possible: $A+$ grade if your output matches correct output of auto grader; otherwise F.
   - A maximum of D grade if Valgrind detects even a single memory leak or error. A teaching assistance will check you submission for such errors.

- A deduction of one letter grade for each missing documentation block in your submissions. Each source file must have **one** file-level documentation block and a function-level documentation block for each defined function. A teaching assistant will physically read submitted source files to ensure that these documentation blocks are authored correctly. Each missing or incomplete or copy-pasted (with irrelevant information from some previous assessment) block will result in a deduction of a letter grade. For example, if the automatic grader gave your submission an $A+$ grade and one documentation block is missing, your grade will be later reduced from $A+$ to $B+$. Another example: if the automatic grade gave your submission a C grade and the two documentation blocks are missing, your grade will be later reduced from C to F.

Exception classes

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

```cpp
throw 3;
```

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

```cpp
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 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 not operate as the programmer intended:

```cpp
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:

```cpp
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:

```cpp
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:

```cpp
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:

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

Because a `MyError` *is-a* `std::logic_error` [think of is-a like this: a student has all the properties of a person and more; therefore, the student is-a person because the student can be used anywhere a person is used but not vice-versa], 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:

```cpp
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 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& at(size_t index);               // not defined in lectures
  char const& at(size_t index) const; // not defined in lectures
private:
  size_t len;
  char*  data;
};
```

Overloaded member functions `at()` are similar to member functions `operator[]()` except that `at()` are defined to handle out-of-range subscripts. The versions of `at()` without error handling would look like this:

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

char const& Str::at(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(size_t Subscript) : subscript(Subscript) {};
  size_t GetSubscript() const { return subscript; }
private:
  size_t subscript;
};
```

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

```cpp
char const& String::at(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:

```cpp
int main() {
  Str s("Hello"); // Create string "Hello"
  try {
    std::cout << s.at(0) << '\n'; // Get 1st character and print it
    s.at(9) = 'C';                // Attempt to change tenth character
    std::cout << s << '\n';
  } catch (SubscriptError const& se) {
    std::cout << "Bad subscript: " << se.GetSubscript() << '\n';
  }
}
```

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

```cpp
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`:

```cpp
class SubscriptError : public std::exception {
private:
  size_t subscript;
public:
  SubscriptError(size_t Subscript) : subscript(Subscript) {};
  size_t 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();
  }
};
```

5. Now, the client should call member function `what()` of class `SubscriptError`:

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

Parsing and formatting strings using `std::stringstream`

You know how to use `std::ifstream` objects to read characters from a file and to use `std::ofstream` objects to write characters to a file. `std::istringstream` objects can be used to read characters from `std::string` objects, and `std::ostringstream` objects to writes characters to `std::string` objects. That doesn't sound so exciting - we already know how to access and change a string's characters. However, string stream classes have the same public interface as the other stream classes. In particular, you can use the familiar `>>` and `<<` operators to read and write numbers that are contained in strings. For that reason, `std::istringstream` and `std::ostringstream` classes are called *adapters* because they adapt strings to the stream interface.

> ***Like an adapter that converts your power plugs to international outlets, string stream adapters allow you to access values in strings using the interface provided by streams.***

Include header `<sstream>` when you use string streams. Here is an example. Suppose string `date` contains a date such as `"December 6, 2001"`, and we want to separate it into month, day, and year. First, construct an object of type `std::istringstream` from string `date`:

```
std::string date{"December 6, 2001"};
std::istringstream iss{date};
```

Or, you could default construct `iss` and later use member function `str()` to set the stream to the string that you want to read:

```
std::string date{"December 6, 2001"};
std::istringstream iss;
iss.str(data);
```

Next, use operator `>>` overload to read the month name, the day, the comma separator, and the year:

```
std::string month, comma;
int day, year;
iss >> month >> day >> comma >> year;
```

Now `month` is `"December"`, `day` is `6`, and `year` is `2001`. Note that this input statement yields `day` and `year` as `int`s. Had we taken `date` apart with member function `substr()`, we'd have obtained only strings, not integers. If you only need to convert a single string to its `int` or `double` value, you can instead use C++ functions `std::stoi` and `std::stod`:

```
std::string year{"2001"};
int y = std::stoi(year); // sets y to integral value 2001
```

With pre-C++ versions, use this helper function:

```
int string_to_int(std::string const& s) {
  std::istringstream iss{s};
  int n;
  iss >> n;
  return n;
}
```

> ***Use a `std::istringstream` object to convert arithmetic values inside a `std::string` object to integers or floating-point values.***

By writing to a string stream, you can convert integers or floating-point value to strings. First construct an `std::ostringstream` object:

```
std::ostringstream oss;
```

Next, use the `<<` operator to add a number to the stream. The number is converted into a sequence of characters:

```
oss << std::fixed << std::setprecision(5) << (10.0/3);
```

Now the stream contains string `"3.33333"`. To obtain that string from the stream, call member function `str`:

```
std::string output{oss.str()};
```

You can build up more complex strings in the same way. Here we build a date string consisting of month, day, and year:

```
std::string month{"December"};
int day{6}, year{2001};
std::ostringstream oss;
oss << month << " " << std::setw(2) << std::setfill('0') << day << ", " <<
year;
std::string output{oss.str()};
```

Now output is the string `"December 06, 2001"`. Note that we converted `int` variables `day` and `year` into a string. If you don't need formatting, you can use C++11 function `std::to_string` to convert numbers to strings. For example, `std::to_string(2001)` is the string `"2001"`.

You can produce a formatted date, without a leading zero for days less than ten, like this:

```cpp
std::string output{month + " " + std::to_string(day) + ", " +
std::to_string(year)};
```

A pre-C++11 version would look like this:

```cpp
std::string int_to_string(int n) {
  std::ostringstream oss;
  oss << n;
  return oss.str();
}
```

> ***Use object of type std::ostringstream to convert numeric values to std::string objects.***