Exception Handling

Chapter 15

Objectives

- **Define the C++ exception mechanism and contrast it with handling errors by function return codes as in C.**
- **Describe “throw”, “try” and “catch” as they are used to implement exception handling in C++.**
- **Implement exception handling in your programs.**
- **Explain the concepts of context and stack unwinding.**
- **Describe what happens to an uncaught exception.**
- **Explain the automatic cleanup process that occurs with C++ exception handling.**
- **Describe how matching of a thrown exception is done in the case of multiple catch handlers.**
- **Gain experience through code walk-throughs and lab exercises.**
  - The example programs are in the **chapter directory**
  - Labs located in Labs/Lab15

Error Handling

- **One way of making a call that may result in an error is to have a "status" return value.**
  - status = some_call(...);
  - if (status != OK) // handle error
- **Not always feasible in C++:**
  - Overloaded operators
  - Constructors

Exception Handling

- **C++ exception mechanism supports "catching" exceptions without having to check a return code.**
    - Exception handling is an important feature of C++, part of the ANSI/ISO standard.
- **Exceptions are "thrown."**
    - In code that detects an exception use a **throw** statement.
    - Enclose code that might cause an exception in a **try** block.
    - Put exception handling code in a **catch** block.
    - There is no **finally**. Why not?

Guidelines

- **Use asserts to check for errors that should never occur.**
- **Use exceptions to check for errors or exceptional cases that might occur.**
- **Throw exceptions by value, catch them by reference. Don't catch what you can't handle.**
- **Don't allow exceptions to escape from destructors or memory-deallocation functions.**
- **Use standard library exception types when they apply. Derive custom exception types from the exception class hierarchy.**

Exception Flow of Control

- **Code which might cause an exception to be thrown should be enclosed in a "guarded" section of code known as a *try block*.**
- **Below the try block is one or more *catch handlers*.**
  - Each catch handler has a parameter specifying the type of exception that it can handle.
  - The exception data type can be any built-in type or a class type.
  - If an exception is thrown, the *first* catch handler that matches the exception data type is executed, and then control passes to the statement just after the catch block(s).
  - If no handler is found, the exception is thrown to the next higher "context" (e.g. the function that called the current one).
  - If no exception is thrown inside the try block, then all of the catch handlers are skipped.

demo Exception Handling

- The program in [Array/Step1]throws an exception on an “out of bounds” error.

```
template<class T>
void Array<T>::SetAt(int i, const T& x) {
        if ( (i < 0) || (i >= m_size) )
                throw ("Out of bounds");
        m_array[i] = x;
}
```

- The code in main() uses a try block to handle the exceptions.

Context and Stack Unwinding

- **As the flow of control of a program passes into nested blocks, local variables are pushed onto the stack and a new "context" is entered.**
  - Likewise a new context is entered on a function call, which also pushes a return address onto the stack.

- **If an exception is not handled in the current context, the exception is passed to successively higher contexts until it is finally handled**
  - Else is "uncaught" and is handled by a default *terminate* function

- **When the higher context is entered, C++ adjusts the stack properly, a process known as *stack unwinding*.**
  - In C++ exception handling, stack unwinding involves both setting the program counter and cleaning up variables.

Handling Exceptions in Best Context

- **One of the benefits of the C++ exception handling mechanism is that it is easier to handle an exception in the appropriate context.**
  - The exception automatically propagates to higher contexts until an appropriate handler is found.
  - In C you must use status return codes, and be carefully to keep passing the right return code at each level of function call.

Context Example

- **The program in folder [Array/Step2] demonstrates exception handling done at the top level**

Benefits of Exception Handling

- **As we have just seen, exceptions can be handled in the context most convenient to the program logic**
- **In many cases a number of operations that might cause an exception can be taken inside a single guarded section of code, without having to check each individual operation**
  - Our array example did not need individual checks on the calls **SetAt** and **GetAt**
- **In contrast to status returns, exceptions *cannot* be ignored**
  - How many programmers check the return code of **printf**?
- **The stack unwinding process automatically cleans up variables, including calls to appropriate destructors**

Unhandled Exceptions

- **If no handler at any levels catches an exception, the exception is said to be "unhandled" or "uncaught".**
  - An uncaught exception can also occur if a new exception is thrown before an existing exception reaches its handler.
- **When an uncaught exception occurs, the special function *terminate* is called.**
  - The default behavior of **terminate** is to print an error message and call **abort**.
  - Although this "uncaught" behavior is not graceful, it is better than having no error message printed and unpredictable results occur.
- **You can customize the treatment of an uncaught exception by calling the *set_terminate* function**.

Clean Up

- **As part of unwinding the stack, C++ takes care of popping local variables, which causes the destructors to be called for class objects.**
  - But objects on the heap are *not* automatically deleted.

Demo Multiple Catch Handlers

- **An exception object in C++ is typed.**
  - If you have several catch handlers, the first one that matches the thrown object will be invoked.
  - Standard automatic type conversions by constructors and cast operators are *not* performed when an exception is thrown.
- **As an example consider the example Multicatch.**
  - Run first with throwing an integer. As expected the **int** catch handler gets called.
  - Now rebuild and run with throwing a **String**. Although the **String** class has a conversion to **const char ***, the only match is with the **String** catch handler.

Standard Library Exceptions

- Standard exception classes derive from the class exception, defined in the header <exception>.
- The two main derived classes are logic_error and runtime_error, which are found in <stdexcept>.
- The class logic_error represents errors in programming logic, such as passing an invalid argument.
- Runtime errors are those that occur as the result of unforeseen forces such as hardware failure or memory exhaustion.
- Both runtime_error and logic_error provide a constructor that takes a std::string argument so that you can store a message in the exception object and extract it later with exception::what( )

Generalizing Exceptions

- **While standard exceptions are usually sufficient in most cases, you may want to provide a custom type that adheres to the standard interface.**
- **The following template provides a boundary guard using invalid_argument as a base for derivation.**

```
template <typename T>
class bounds_error : public invalid_argument
{
private:
   T _value;
public:
   bounds_error(T value, const char *what_arg)
     :   _value(value),  invalid_argument(what_arg) {}
   bounds_error(const std::string &what_arg)
     : invalid_argument(what_arg) {}
   int get_value() const{
      return _value;
   }
};
```

Summary

- **The C++ exception mechanism provides a robust means of dealing with exceptional program behavior without need of tracking function return codes.**
- **You “throw” an exception. An operation which may throw an exception should be performed in a “try” block, and the exception is handled in a “catch” block.**
- **Exceptions not handled at the current context are propagated to higher contexts.**
- **An “uncaught exception” is ultimately handled by the *terminate* function, which will abort the program. Thus exceptions cannot be ignored.**
- **The stack unwinding process automatically cleans up local variables, but not objects on the heap.**
- **If there are several catch handlers, the first one that matches the thrown exception will be called.**