15 Exception Handling

REVIEW QUESTIONS

1. The traditional approach handles the error in the code that detects the error while the C++ exception handler separates the detection and handling of errors.
3. While an exception can be handled entirely in one function, it is more common for the exception to be raised in a called function and handled in the calling function.
5. The criteria of choosing an exception handler (*catch* statement) is based on the **type** of data thrown?
7. Polymorphism allows us to use a "generic" handler that provides different logic and error messages for each situation. This allows us to write one catch statement to handle every error in the hierarchy. Furthermore, if error messages are printed in virtual functions, we can write only one catch statement that catches different exceptions by using a reference to the object.
9. All C++ standard exception-handling classes are derived (directly or indirectly) from a class named **std::exception**.
11. A function can specify that (1) only specified exceptions can be thrown, (2) that no exceptions can be thrown, or that (3) any exception may be thrown.
13. The three classes under the *runtime_error* class are (1) range error, (2) overflow error, and (3) underflow error.
15. If a file cannot be opened or closes, some compilers throw an ios failure object (`ios_base::failure`).

EXERCISES

17. There is a *catch* but there is no *try* statement.
19. The *catch* statement must follow the *try*, not be before it.
21. For the *throw* statement to be valid, exception would have to be a global type. Also, while not necessarily an error, there is no error detection logic in fun.
23. The function specifies that only an `x` exception can be *thrown*; therefore, the function cannot throw a `y` exception.

25. There is a logic error in that the first statement in the function throws an exception. None of the other statements could ever be executed.

PROBLEMS

27.

```
/* Allocate memory for an array. If error occurs,
   return null pointer.
      Pre  nothing
      Post memory allocated and address returned
           or null pointer returned
*/
template <class TYPE>
TYPE* alloc (TYPE data, int size)
{
   TYPE* ptr ;
   try
      {
       ptr = new TYPE[size];
       return ptr;
      } // try
   catch (...)
      {
       return 0;
      } // catch
}  // alloc
```

29.

```
/*  This program demonstrates the setting of a function
    to catch unexpected errors.
       Written by:
       Date:
*/
#include <iostream>
#include <exception>
#include <cstdlib>
using namespace std;

void error () throw (int);
void myTerminateHandler ();

int main ()
{
    cout << "*** start of program ***\n\n";
    set_unexpected (myTerminateHandler);

    try
       {
        error();
        cout << "No error detected\n";
       } // try
    catch (int)
       {
        cout << "Can't get here\n";
        exit (200);
       } // catch
    cout << "\n\n*** end of program ***\n";
    return 0;
}   // main

/* Throws error to test unexpected errors.
```

```
      Pre  nothing
      Post error thrown
*/
void error () throw (int)
{
   double x = 0;
   throw (x);
}  // divide

/* Called by exception handler when an unknown error
   occurs.
     Pre  Unknown error has been thrown
     Post Message displayed and program exited
*/
void myTerminateHandler ()
{
   cout << "An unexpected error has occurred."
        << " Terminating\n";
   exit (100);
}  // myTerminateHandles
```