

# Simulating Errors

by Dave Pomerantz

As recently as two years ago, I was never sure if my software could gracefully recover from a problem. That's because I had no systematic way to simulate errors. Let's say I had a code sequence buried deep in my program, like this:

```
try
{
    DBTable *pPersTable = new DBTable("Personnel");
    if (pPersTable == 0)
    {
        throw(E_OPENING_DB_TABLE);
    }
    DoUsefulWork(pPersTable);
}
catch(HRESULT hr)
{
    DBErrorRecovery(hr);
}
```

Eleven of the above thirteen lines of code are only present to detect and propagate a failure in new. Did I make a mistake in those eleven lines? If so, how can I find out before shipping to a customer?

Good programs have hundreds of these error-handlers. Most of them respond to conditions that rarely occur and are difficult to force in normal use. If I run my program in low memory, I can force one or two errors. Repeated testing will usually trigger the same errors, especially when testing memory allocation failures. It's nearly impossible to force all the different errors with an external test.

In this article, I'll present a simple but powerful tool to simulate all the errors a program handles. You can use this tool to verify that your error-handlers are working. After all, why bother writing an error-handler if you can't be sure it works?

## A Simple Example

Listing 1 shows a small program with four failure points, *new*, *malloc*, *AdjustValve*, and *UpdateDatabase*. Each of these functions might return an error or throw an exception, or both. After detecting a failure, the sample program responds by logging a message, skipping remaining functions, and returning an error code.

### Instrumenting the example

Listing 2 shows the same program with functions inserted to simulate errors. The functions are conditionally compiled to appear only in a debug version.

### Simulating errors

In the example below, taken from the sample program, the *SimErrMalloc* function simulates an allocation error immediately after a call to *malloc*:

```
pArray = (char *) malloc(100);
pArray = SimErrMalloc(SIMLINE, pArray);
if (pArray==0)
{
    printf("C allocation error\n");
    bOK = false;
}
else
{
    delete pArray;
}
```

*SimErrMalloc* is a conditionally compiled C++ template function. The release version compiles to a null function, eliminating all traces of the error simulator from production software. The debug version compiles the following template, which frees allocated memory and returns zero, to simulate a failed allocation.

```
#define SIMLINE __FILE__, __LINE__

// Simulate failed C allocation
template <class T>
T *SimErrMalloc(const char *strFile,
               int nLine, T *pData,
               long nCount = 1,
               double nProb = 1.0)
{
    if ((pData != 0) &&
        SimErr(strFile, nLine,
              SIMERR_MEMORY_ERROR,
              nCount, nProb))
    {
        free(pData);
        pData = 0;
    }
    return pData;
}
```

*SIMLINE* is a simple macro that passes the first two parameters to *SimErrMalloc*, the source file and line number. These identify the location of the simulation function. The third parameter, *pData*, points to the allocated memory.

The last two parameters, *nCount* and *nProb*, apply only to a random test. They control the number of times to simulate an error at this location and the probability of an error each time. The default is to simulate the error once (*nCount* = 1), the first time through (*nProb* = 1.0). For now, I'll ignore these two parameters and save the discussion of random error simulation for later.

*SimErrMalloc* calls *SimErr* to find out if it should simulate an error. *SimErr* looks up the location, finds out how many times it has already simulated an error in that location, and returns *true* to simulate an error or *false* to peacefully continue on.

If *SimErr* returns *true*, *SimErrMalloc* frees the allocated memory and returns zero, simulating a failed allocation.

## Simulating exceptions

In addition to handling failure codes, most programs face the greater complexity of handling exceptions. In the sample, *UpdateDatabase* can return an error, but it can also throw an exception. The sample rigorously simulates both:

```
try
{
    ...
    if (bOK)
    {
        nResult = UpdateDatabase();
        SimErrException(SIMLINE,
            SIMERR_DATABASE_EXCEPTION);
        nResult = SimErrResult(SIMLINE,
            nResult,
            DATABASE_ERROR);
        if (nResult == DATABASE_ERROR)
        {
            printf("Update failed.\n");
            bOK = false;
        }
    }
}
...
catch(CDatabaseException *pDBException)
{
    printf("Unrecoverable database error.\n");
    delete pDBException;
    bOK = false;
}
```

*SimErrException* throws an exception. *SimErrResult* sets *nResult* to *DATABASE\_ERROR*

Table 1 summarizes the error simulation functions and their parameters.

## Systematic testing

The simulation functions are simple. Inserting the function calls, though tedious, is also simple. The only tricky part is deciding when to trigger the errors. If the first function always triggers an error, the program always exits before exercising other error handlers.

An error simulation function needs to turn itself off after its associated error handler has succeeded. To test the six error handlers in Listing 2, I should run the program six times. On each successive run, the previous error should be inactive and the next one should fire, until all errors have been simulated.

Each simulation function calls *SimErr* to find out if it should trigger an error. *SimErr* maintains a log, by source file and line number, that tracks the number of the times the simulation has fired. Each time it's called, *SimErr* updates an internal copy of the log.

At initialization, *SimErrInit* reads the simulation log and writes a backup copy. Before the program terminates, *SimErrExit* writes the simulation log.

After recovering from an error and exiting properly, the next run of the program will skip the previous failure and advance to the next error simulation. This permits successive program runs to proceed stepwise until each failure point has been tested.

If an error handler fails and the program aborts before calling *SimErrExit*, the updated log won't be written. This is important, because it allows you to repeat the same failure until you fix the error handler that's responding incorrectly.

It's possible for the program to exit successfully even if the error handler fails. For example, the error handler may have caught the error but displayed the wrong message. In this case, you need to manually copy the backup log file over the updated log file, to repeat the same test.

### Oh, the chore of testing!

It's certainly tedious testing several hundred error handlers, each of which causes the program to exit. It's worse if your program takes a long time to restart. Unfortunately, I don't think there's a simple answer. This kind of

thorough testing is at the heart of quality assurance. Using an automated test framework to script the tests will make it easier, but error handlers probably account for roughly twenty percent<sup>1</sup> of your code. If your product is at all complex, testing and fixing twenty percent of it will take time.

For me, the best way to do the testing is piecemeal, at the same time that I unit-test each module. If I test periodically, it's a lot more tolerable than testing all at once, and I catch mistakes closer to when I made them. Besides, it's enormously satisfying to watch my software handle all kinds of system and hardware failures without blinking.

Because of the propagation of errors, however, it's necessary at some point, to test the complete integrated system, even if each module has been tested separately.

## Random testing

Once you've systematically tested all the error handlers, you'll have gone farther than most programmers toward building bulletproof software, but the real world has some nasty ordnance. The bullets don't always come from where you're looking.

A systematic test triggers a failure at the first opportunity. A random test waits for the second, or perhaps the tenth or thousandth opportunity to trigger a failure. The state of the software may be different at that point and error-handling that worked in a systematic test may fail in a random test.

The simulation template functions, like *SimErrMalloc*, set the default count and probability to 1, which guarantees an error will be simulated once, the first time through. By filling in these parameters, you can simulate the error more than once, each time with a given likelihood. If you set the count to `SIMERR_FOREVER`, the error will be simulated indefinitely.

In the sample, the *AdjustValve* function is followed by a simulation function that will fail roughly one-third of the time it's executed until it has failed four times.

```
bOK = AdjustValve();
SimErrException(SIMLINE, SIMERR_DEVICE_EXCEPTION, 4, 0.33);
```

If you change this function to:

```
SimErrException(SIMLINE, SIMERR_DEVICE_EXCEPTION, SIMERR_FOREVER, 0.0001);
```

the valve will indefinitely return a failure at the rate of one in ten-thousand. This is useful for simulating intermittent failures during an extended stress-test.

I typically test all the error handlers once, for basic operation, then test again randomly. As a convenience, *SimErrInit* takes a parameter which turns random testing off. This forces all errors to be simulated exactly once, regardless of their parameters, making it easy to switch back and forth between systematic and random testing.

## The error trace

Most debug environments have some form of trace output. The simulation functions direct all their output through a callback function, which you tailor to your environment. Listing 3 shows the trace output from the first program run, which simulates a single memory error.

The first line of the trace:

```
<SIMERR> Starting error simulator with a systematic test.
```

shows whether the test is random or systematic. If it were a random test, the trace would also show the seed used for the random numbers.

Before forcing an error, the simulator writes the source file and line number of the function along with the type of error.

---

<sup>1</sup> I admit, that's a wild guess. I haven't seen hard numbers on this, so I counted lines in my own software. Excluding comments, I found that testing, propagating, and responding to errors made up between 7% and 34% of the code, averaging about 20%. It's lowest in modules that have few external interfaces. It's highest in I/O modules and other routines that interact heavily with the system.

```
<SIMERR> at sample.cpp(42) *** SIMULATING MEMORY ERROR ***  
C++ allocation error  
Sample program exiting with return code = -1
```

In this case, the simulated failure caused the sample program to report the error and exit. Just before exiting, the program called *SimErrExit*, which wrote a detailed report divided into three sections:

- Errors that were simulated in this session and prior sessions. Errors in this session are marked NEW.
- Simulation functions that were encountered but not yet triggered. This can only happen during random testing.
- A summary count of simulated errors, organized by error type.

Listing 4 shows the output after six systematic runs. At this point, all errors have been simulated.

## How the simulator works

The simulator tracks functions that call it. Each entry in this list contains the source file name, line number, error type, and the number of times a function at that location has simulated an error.

At initialization, the simulator reads the list from the log file and makes a backup copy of the file.

Each time a simulation function is executed, the simulator looks up the corresponding entry and decides whether to simulate a failure. To simulate an error, it returns *true*. To simulate an exception, it invokes a callback routine that the target program provided to throw environment- and application-specific exceptions.

At termination, it writes the simulation log file and the summary report.

The list entries themselves are C++ objects and the list is an STL vector. The simulator is a singleton C++ object that, in the Windows implementation, resides in its own DLL.

## Customizing the simulator

The simulator source code is freeware. It's written in ANSI C++ and is available from <ftp://ftp.mfi.com/pub/cuj> or you can download the latest version from [www.targetsoft.com](http://www.targetsoft.com).

The simulator was built with Visual C++ but could easily be adapted to any ANSI C++ compiler.

Each program that uses the simulator needs a small source file to customize the error types and trace output. Alternatively, you could build the customization right into the simulator library, but I prefer the flexibility of allowing different error types in the target program.

The custom header and source files are shown in Listings 5 and 6. They provide:

- application-defined error and exception types.
- a callback function to write the simulator messages to the debug log file.
- a callback function to throw exceptions.

If you're not using C++, you have more work ahead of you. Fortunately, the simulator is only 1200 lines of code and would not be hard to rewrite in another language. I feel the payoff would be worthwhile for all but the smallest projects. See the FAQ section below on what would be involved to re-implement in C.

## Adding environment-specific functions

Since the simulator DLL provides the error logging and triggering, the simulation functions need only five or six lines of code to simulate specific errors. This makes it easy to add functions for error conditions that are particular to an environment. For example, to simulate an error in a function that creates a Microsoft Windows timer, you might use this code sequence:

```
m_nTimer = SetTimer(1000, // ID
                   100, // interval
                   0); // WM_TIMER
m_nTimer = SimErrZero(SIMLINE,
                    m_nTimer);
```

When the error is simulated, *SimErrZero* will return zero. This causes a problem, however, since a Windows timer really did get created and is now dangling, unused and inaccessible.

For these kinds of system calls, it may be helpful to write a few functions like *SimErrTimer* to destroy the system resource that was created:

```
inline UINT
SimErrTimer(const char *strFile,
            int nLine, UINT nTimer,
            long nCount = 1,
            double nProb = 1.0)
{
    if ((nTimer != 0) &&
        SimErr(strFile, nLine,
              SIMERR_RESULT_ERROR,
              nCount, nProb))
    {
        KillTimer(0, nTimer);
        nTimer = 0;
    }
    return nTimer;
}
```

My own experience is that a few of these functions, for timers, windows, files, and graphic objects, will cover what you need, and will not amount to much code.

## Limitations

### Inaccurate report

The simulator only reports the simulation functions that have called it. It's strictly a run-time analyzer, so it doesn't know about all the simulations that remain unexecuted in the target program. To get the count of simulation functions, I search all source files for all occurrences of SIMLINE, which is clumsy, but works.

I've thought about writing a pre-processor to build the simulation log at compile time. That would, however, dramatically increase the size and complexity of the simulator and complicate administering it, so I've held off.

### Editing source files

The simulation log persists between runs of the target program, tracking the line numbers of simulation functions. If you change a source file, inserting or deleting lines, an existing simulation function will have a different location in the file and will appear to the simulator to be a new function. Typically the simulator will trigger it again, unnecessarily. Worse, it may never trigger a simulation function, thinking it had already been triggered.

To be safe, you should either:

1. Edit the log file to update the line numbers that changed. This is the only viable option if you're in the middle of a long, complex test. The file is ASCII and is easy to edit.
2. Delete the simulator log file and start the tests over. If you've only just started testing, this is the best option.

This problem would be solved, of course, if the simulator had a pre-processor.

## Frequently asked questions

### What is the performance cost?

When you're debugging the error handlers, the performance cost is probably irrelevant, since the program will shortly enter an error handler.

In the release version, there is no performance cost. I've inspected the disassembled output of the Visual C++ optimizing compiler and it generates no code for the null templates.

### Isn't it enough to test failures at the low-level functions?

Error handlers are all over a program, but only a few low-level functions originate most errors. Isn't it enough to simulate errors in these functions?

No. In addition to testing error detection, you need to test error propagation and recovery. That means testing all paths for all errors, not just all errors. The safest way is to simulate an error in every place that you test for an error.

### You used templates and classes. How can I implement this in C?

I used templates to make the simulation functions type-safe and to allow default parameters. I could have used macros instead, like this:

```
// Simulate failed C allocation
#define SimErrMalloc(pVoid) \
{ \
    if ((pVoid) && \
        SimError(__FILE__, \
                 __LINE__, \
                 MEMORY_ERROR, \
                 1, 1.0)) \
    { \
        free(pVoid); \
        pVoid = 0; \
    } \
}
#define SimFailMallocRandom(pVoid, \
                            nCount, \
                            nProb) \
{ \
    if ((pVoid) && \
        SimError(__FILE__, \
                 __LINE__, \
                 MEMORY_ERROR, \
                 nCount, \
                 nProb)) \
    { \
        free(pVoid); \
        pVoid = 0; \
    } \
}
```

Instead of the one template, I had to define two macros, one for systematic errors and one for random errors. Macros have an advantage, however, of being able to hide the `__FILE__` and `__LINE__` parameters directly in the macro.

I used object-oriented programming because I find it makes the code easier to develop and debug. In C, the same result could be accomplished with a module, static data, and static functions.

### Aren't there any commercial tools that simulate errors?

The only tool I could find that claimed to support error simulation was Panorama by International Software Automation ([www.softwareautomation.com](http://www.softwareautomation.com)). They discussed automatic error simulation at their website, but when I contacted the company, they recommended manual error simulation much like I'm discussing here.

I also contacted NuMega Technologies, developers of BoundsChecker™, and they do not have an error simulation tool for C++.

I think this is an excellent area for companies like ISA and NuMega to pursue. More sophisticated tools that use pre-processors can invisibly insert the simulation code during compilation and solve many of the problems associated with my very simple approach.

## Conclusions

The error simulator is effective because it's simple and it solves a problem little else seems to touch. It takes time, however, to insert the simulation functions and more time to run the tests and fix the bugs.

It's hard to justify the time to test such unlikely errors. Certain projects, like embedded software and mission-critical applications, have no choice. For other projects, you might consider the cost of finding and fixing one intermittent data-corruption bug that occurs when an out-of-memory condition triggers an obscure error handler.

In that case, it may be cheaper to fix all the error handlers in controlled tests during development, than to fix a single intermittent bug at a customer site.

## Biography

*Dave Pomerantz is the owner of On Target Software, a consulting firm in Marshfield, Massachusetts, focusing on C++ development for Windows. The error simulator source code, sample program, Win32 DLL, and complete technical documentation are available at <http://www.targetsoft.com>.*

# Listings

## Listing 1. Sample program without instrumentation.

```
int main(int argc, char **argv)
{
    bool          bOK = true; // continue as long as this is true
    int           nResult;
    CSampleObject *pObject;
    char          *pArray;

    // Instantiate object with C++ allocation
    pObject = new CSampleObject;
    if (pObject==0)
    {
        printf("C++ allocation error.\n");
        bOK = false;
    }
    else
    {
        delete pObject;
    }
    // Allocate char array with malloc
    if (bOK)
    {
        pArray = (char *) malloc(100);
        if (pArray==0)
        {
            printf("C allocation error.\n");
            bOK = false;
        }
        else
        {
            delete pArray;
        }
    }
}

try
{
    if (bOK)
    { // Adjust real-time device
      // note: AdjustValve can throw a CDeviceException
      bOK = AdjustValve();
      if (!bOK)
      {
          printf("The valve stuck.\n");
      }
    }
    if (bOK)
    { // Update remote-server database
      // note: UpdateDatabase can throw a CDatabaseException
      nResult = UpdateDatabase();
      if (nResult == DATABASE_ERROR)
      {
          printf("Database update failed.\n");
          bOK = false;
      }
    }
}
catch(CDeviceException *pDeviceException)
{
    printf("Unrecoverable device error.\n");
    delete pDeviceException;
    bOK = false;
}
catch(CDatabaseException *pDBException)
{
    printf("Unrecoverable database error.\n");
    delete pDBException;
    bOK = false;
}
```

```
catch(...)  
{  
    printf("Unknown exception.\n");  
    bOK = false;  
}  
  
nResult = (bOK) ? 0 : -1;  
printf("Sample program exiting with return code = %d.\n", nResult);  
return nResult;  
}
```

## Listing 2. Sample program instrumented for testing.

```
int main(int argc, char **argv)
{
    bool          bOK = true; // continue as long as this is true
    int           nResult;
    CSampleObject *pObject;
    char          *pArray;

    SimErrCustomInit(SIMERR_TIME_SEED, SIMERR_SYSTEMATIC);

    // Instantiate object with C++ allocation
    pObject = new CSampleObject;
pObject = SimErrNew(SIMLINE, pObject, SIMERR_FOREVER, .4);
    if (pObject==0)
    {
        printf("C++ allocation error.\n");
        bOK = false;
    }
    else
    {
        delete pObject;
    }

    // Allocate char array with malloc
    if (bOK)
    {
        pArray = (char *) malloc(100);
pArray = SimErrMalloc(SIMLINE, pArray);
        if (pArray==0)
        {
            printf("C allocation error.\n");
            bOK = false;
        }
        else
        {
            delete pArray;
        }
    }
}

try
{
    if (bOK)
    { // Adjust real-time device
      // note: AdjustValve can throw a CDeviceException
      bOK = AdjustValve();
SimErrException(SIMLINE, SIMERR_DEVICE_EXCEPTION, 4, 0.33);
bOK = SimErrZero(SIMLINE, bOK, SIMERR_FOREVER, 0.2);
      if (!bOK)
      {
          printf("The valve stuck.\n");
      }
    }
    if (bOK)
    { // Update remote-server database
      // note: UpdateDatabase can throw a CDatabaseException
      nResult = UpdateDatabase();
SimErrException(SIMLINE, SIMERR_DATABASE_EXCEPTION);
nResult = SimErrResult(SIMLINE, nResult, DATABASE_ERROR);
      if (nResult == DATABASE_ERROR)
      {
          printf("Database update failed.\n");
          bOK = false;
      }
    }
}
catch(CDeviceException *pDeviceException)
{
    printf("Unrecoverable device error.\n");
    delete pDeviceException;
    bOK = false;
}
catch(CDatabaseException *pDBException)
{
    printf("Unrecoverable database error.\n");
    delete pDBException;
}
```

```
        bOK = false;
    }
    catch(...)
    {
        printf("Unknown exception.\n");
        bOK = false;
    }

    nResult = (bOK) ? 0 : -1;
    printf("Sample program exiting with return code = %d.\n", nResult);

    SimErrExit(true);
    return nResult;
}
```

### Listing 3. Trace output from first run.

```
<SIMERR> Starting error simulator with a systematic test.
<SIMERR> log file "errlog.txt" not found, test begins with first error
<SIMERR> atsample.cpp(42) *** SIMULATING MEMORY ERROR ***
C++ allocation error.
Sample program exiting with return code = -1.
<SIMERR>
<SIMERR> *** ERRORS SIMULATED ***
<SIMERR>
<SIMERR>      File                               Line Error-Type Requested Total New
<SIMERR> -----
<SIMERR> NEW sample.cpp                          42     MEMORY   FOREVER      1    1
<SIMERR>
<SIMERR> *** ERROR SIMULATION TOTALS ***
<SIMERR>
<SIMERR> Type           Triggered Remaining Total
<SIMERR> -----
<SIMERR> RESULT                0           0      0
<SIMERR> MEMORY                1           0      1
<SIMERR> DEVICE                0           0      0
<SIMERR> DATABASE              0           0      0
<SIMERR> -----
<SIMERR> ALL TYPES             1           0      1
<SIMERR>
<SIMERR> Triggered 1 error(s) in this session.
```

### Listing 4. Trace output from sixth run.

```
<SIMERR> Starting error simulator with a systematic test.
<SIMERR> backup log file written to "bkuplog.txt"
<SIMERR> atsample.cpp(87) *** SIMULATING RESULT ERROR ***
Database update failed.
Sample program exiting with return code = -1.
<SIMERR>
<SIMERR> *** ERRORS SIMULATED ***
<SIMERR>
<SIMERR>      File                               Line Error-Type Requested Total New
<SIMERR> -----
<SIMERR> sample.cpp                          42     MEMORY   FOREVER      1    0
<SIMERR> sample.cpp                          57     MEMORY      1      1    0
<SIMERR> sample.cpp                          75     DEVICE     4      1    0
<SIMERR> sample.cpp                          76     RESULT   FOREVER      1    0
<SIMERR> sample.cpp                          86     DATABASE   1      1    0
<SIMERR> NEW sample.cpp                       87     RESULT     1      1    1
<SIMERR>
<SIMERR> *** ERROR SIMULATION TOTALS ***
<SIMERR>
<SIMERR> Type           Triggered Remaining Total
<SIMERR> -----
<SIMERR> RESULT                2           0      2
<SIMERR> MEMORY                2           0      2
<SIMERR> DEVICE                1           0      1
<SIMERR> DATABASE              1           0      1
<SIMERR> -----
<SIMERR> ALL TYPES             6           0      6
<SIMERR>
<SIMERR> Triggered 1 error(s) in this session.
```

## Listing 5. Custom header file for target application

```
// Custom header file for application-specific error simulation
enum ERROR_TYPE
{
    SIMERR_RESULT_ERROR,          // failing function returns an error code
    SIMERR_MEMORY_ERROR,         // failed allocation returns zero
    SIMERR_DEVICE_EXCEPTION,     // application-specific exception
    SIMERR_DATABASE_EXCEPTION,   // application-specific exception
    NUM_SIMERR_TYPES
};

#define SIMERR_MSGLEN    256

//===== DEFINE IF SIMULATING ERRORS =====
#ifdef SIMERR

bool SimErrCustomInit(int nSeed, SIMERR_TEST_TYPE testType);

//===== DEFINE IF NOT SIMULATING ERRORS =====
#else

inline bool SimErrCustomInit(int , SIMERR_TEST_TYPE )
{ return true; }

#endif // SIMERR
```

## Listing 6. Custom source file for target application

```
// Custom source file for application-specific error simulation
//===== DEFINE IF SIMULATING ERRORS =====
#ifdef SIMERR

const char *SAMPLE_ERROR_LOG = "simerr.txt";

/*-----
g_errorInfo    defines each error type and whether it throws an exception
-----*/
static errorInfo_t g_errorInfo[] =
{
  { SIMERR_RESULT_ERROR,      "RESULT",      SIMERR_TYPE_ERROR      },
  { SIMERR_MEMORY_ERROR,     "MEMORY",     SIMERR_TYPE_ERROR      },
  { SIMERR_DEVICE_EXCEPTION, "DEVICE",     SIMERR_TYPE_EXCEPTION  },
  { SIMERR_DATABASE_EXCEPTION, "DATABASE",  SIMERR_TYPE_EXCEPTION  }
};

/*-----
PrintSimulatorMessage  Redirect messages from error simulator
-----*/
static void PrintSimulatorMessage(const char *pMsg)
{
  printf("<SIMERR> %s\n", pMsg);
}

/*-----
ThrowSimulatorException  Throw an exception corresponding to an error type.

The simulator calls this function to simulate an application-specific exception.
-----*/
static void ThrowSimulatorException(int nErrorType,
                                   const char *pSourceFilename,
                                   int nSourceLine)
{
  switch(nErrorType)
  {
    case SIMERR_DEVICE_EXCEPTION:
    {
      CDeviceException *pException = new CDeviceException();
      throw(pException);
      break;
    }
    case SIMERR_DATABASE_EXCEPTION:
    {
      CDatabaseException *pException = new CDatabaseException();
      throw(pException);
      break;
    }
    default:
      PrintSimulatorMessage("Invalid exception type");
  }
}

/*-----
ValidateErrorTypes  Make sure error info array matches ERROR_TYPE enum
-----*/
static bool ValidateErrorTypes(void)
{
  bool bOK = true;
  int nErrorTypes = sizeof(g_errorInfo) / sizeof(errorInfo_t);
  if (nErrorTypes == NUM_SIMERR_TYPERES)
  {
    for (int i=0; bOK && (i < NUM_SIMERR_TYPERES); ++i)
    {
      if (g_errorInfo[i].m_nErrorType != i)
      {
        bOK = false;
      }
    }
  }
  else
  {
    bOK = false;
  }
}
#endif
```

```

    if (!bOK)
    {
        PrintSimulatorMessage("error type enum does not match g_errorInfo.");
    }
    return bOK;
}

/*-----
SimErrSampleInit Allocate and initialize the sample error simulator.
-----*/
bool SimErrCustomInit(int nSeed, SIMERR_TEST_TYPE testType)
{
    bool bOK = false;

    // Make sure we didn't add an error type without modifying g_errorInfo
    bOK = ValidateErrorTypes();

    if (bOK)
    {
        // Fill in information that initializes the simulator
        simerrInit_t init;
        init.m_nVersion      = SIMERR_VERSION;
        init.m_pFilename     = "errlog.txt";
        init.m_nSeed        = nSeed;
        init.m_testType     = testType;
        init.m_pfMsg        = PrintSimulatorMessage;
        init.m_pfThrow      = ThrowSimulatorException;
        init.m_pErrInfo     = g_errorInfo;
        init.m_nErrTypes    = NUM_SIMERR_TYPES;

        // Init simulator
        bOK = SimErrInit(&init);
    }
    if (!bOK)
    {
        PrintSimulatorMessage("error simulator could not be initialized.");
    }
    return bOK;
}

#endif //SIMERR

```

## Tables

| Function         | Purpose  | Parameters   |
|------------------|--|--|
| SimErrCustomInit | Customized initialization for a single target application. Calls SimErrInit. | type of test: random or systematic                                   |
| SimErrInit       | Create error simulation object and read records from prior testing.          | structure containing log file name, random error seed, and callbacks |
| SimErrExit       | Write records from this test and clean up.                                   | whether to print a full report                                       |
| SimErrEnable     | Turn on/off error simulation   | whether to turn on or off  |
| SimErrNew        | simulate failure in C++ allocation   | ptr to C++ object  |
| SimErrMalloc     | simulate failure in C allocation   | ptr to allocated memory  |
| SimErrResult     | simulate failure of a called function that returns zero on success           | returned value, error value  |
| SimErrZero       | simulate failure of a called function that returns false on failure          | returned value   |
| SimErrException  | throw a system- or application-defined exception                             | exception type   |

**Table 1. Error simulation functions**