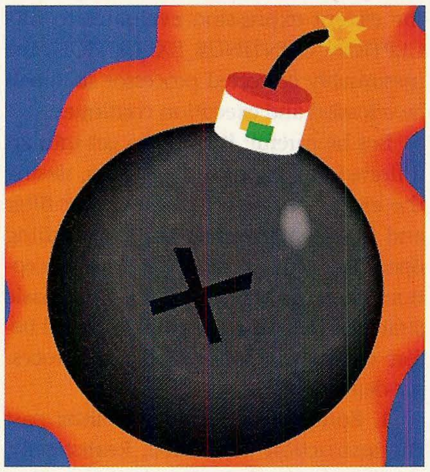
By Douglas A. Hamilton

Things That Make You Go Boom!

SOONER OR LATER, your code will dereference a null pointer, divide by zero, take the square root of -1 or do something else just a little stupid. What happens next is usually whatever the operating system normally does to errant programs, which may or may not be something useful or sensible. For example, even under OS/2 1.*x*, an otherwise fairly robust operating system, there is simply no choice in how a protection violation is handled. You get a big full-screen pop up dumping the registers in any color you like—so long as it's black.

There is an argument (one given to me when I once asked why I couldn't intercept my own protection violations under OS/2 1.*x*) that, when a program encounters a gross error condition, who knows what may have gone wrong. Who knows how it might have compromised its internal structures; if it were allowed to continue to run, who knows what havoc it might have unleashed.



SCOTT BALDWIN

The problem with that argument? It's wrong. Even at its worst, the damage a trashed application might cause is fairly limited by all the other protection mechanisms. If allowed to continue, the application might indeed scramble its own data space more thoroughly, but at least it couldn't scribble outside of its own space.

More important, this is not an all-or-nothing situation. Just because an error has occurred does not mean everything is in wreckage. Programs don't typically scribble randomly all over themselves for a while and then encounter a protection violation. Common sense and a little experience tell us that hardware traps are usually the easiest to debug, precisely because the bug is usually right there where the trap happened. So it may be useful to call a routine that can snapshot useful information, important data structures and so on for debugging.

But also, there are legitimate situations where it's simply better to assume an operation will work and deal with the consequences of any failures on an exception basis than it is to try to test beforehand whether a failure will occur. For example, suppose we have a numeric operation involving thousands of divisions for which performance is everything. Suppose we believe it nearly impossible that a divisor could ever be zero but want to guard against that possibility anyway. What should we do?

What Windows NT offers is a mechanism to intercept these errors on an exception basis. The basic syntax is:

```
try
{
... some operation that might fail ...
}
```

```
except (... test for certain exceptions ...)
  {
  ... handle the exception ...
  }
```

The facility is built with help from both the operating system and the compiler. The NT kernel provides the mechanism for intercepting the exceptions, and the compiler is providing a simple syntax for using it. It's frame-based, meaning that when an exception is raised the system begins looking back through the call stack to find a try/except block that is prepared to deal with that failure. Around all the code in each thread is an implicit try/except block that terminates the whole process; if you don't deal with an exception, the system will.

The except(...) code is an expression, although it can be arbitrarily complex. Typically, it would examine the exception code and indicate if an exception was one it was prepared to deal with. If the expression evaluates to EXCEPTION_CONTINUE_SEARCH, the system continues walking back up through the stack.

The value EXCEPTION_EXECUTE_HANDLER means this is an exception that is handled here. The operation inside the try{...} block is aborted, and the code inside the block following the try(...) runs instead. For example, the following demonstrates how a protection violation caused by dereferencing a null pointer might be intercepted.

```
#include <windows.h>
#include <stdio.h>
#include <excpt.h>
void main ( void )
  {
  char *c = NULL;
  try
    {
    *c = 'x';
    printf("never executed\n");
    }
  except (printf("Got exception = 0x%lx\n",
      GetExceptionCode()),
      EXCEPTION_EXECUTE_HANDLER)
    {
    printf("... inside the handler...\n");
    }
  printf("...back out again...\n");
  }
```

Running this program, we'd see the following:

```
Got exception = 0xc0000005
... inside the handler...
...back out again...
```

It's also possible to handle an exception but allow the execution to continue, although it's more difficult. NT provides a snapshot of the current processor state to the except filter which it can, if it chooses and knows how, fix up so the failing operation can be reattempted. If the except expression evaluates to EXCEPTION_CONTINUE_EXECUTION, the presumably corrected processor state will be reloaded and execution continues.

But be careful: It's easy to get into an infinite loop, trying an operation, raising the exception, entering the except filter and then reattempting the same failing operation only to raise the same exception. To avoid that, there are three basic strategies for fixing the context record before continuing, all of which are processor-dependent:

1. Bump the instruction pointer past the instruction that failed. Realistically, that means dereferencing the instruction pointer (IP) and looking at the opcode to determine the instruction length.

2. Correct the input data. If it was a divide by zero exception, give it another value. However, this may be quite problematic, given compiled code. Even if we know that a/b failed, it's unlikely that simply giving b a new value before continuing will work, since the compiler may have been holding b in a register. So we're back to dereferencing IP, this time disassembling the whole instruction.

3. Mask the possibility of an exception. We can do that with floating point, where the IEEE standard provides that division by zero results in a special "infinity" value.

Here's an example of that third strategy:

```
#include <windows.h>
#include <stdio.h>
#include <excpt.h>
#include <math.h>
#include <float.h>

void cdecl main( void )
  {
  double a = 2.0, b = 0.0;
  CONTEXT *c;

  _controlfp(0, _MCW_EM);
  try
    {
    a /= b;
    printf("a = %f\n", a);
    }

  except (printf("exception = 0x%lx\n",
      GetExceptionCode()),
      c = (GetExceptionInformation())
        ->ContextRecord,
#     ifdef MIPS
      c->Fsr &= ~0xffc,
#     else
      c->FloatSave.ControlWord |=
          0x003f,
#     endif
      EXCEPTION_CONTINUE_EXECUTION)
    {
    }
  }
```

Here's what it prints:

```
about to divide by zero...
exception = 0xc000008e
a = 1.#INF00
```

In this example, we've initially turned on the floating-point exceptions by calling the _controlfp() C runtime library routine. But notice that we could not use that same routine to turn on the mask in the except() filter. The reason is simple: When the processor state is reloaded from the context record, it would overwrite what the _controlfp() routine had done. Therefore, we must make any changes in the context record itself. And, because the MIPS and Intel processors have different hardware architectures, the code is different in each case. Once you are out of the exception handler and the code that caused the exception, remember that the floating-point exceptions are still masked. If we want to catch new exceptions, the mask will have to be cleared again. ■

Douglas Hamilton is president of Hamilton Laboratories (Wayland, Mass.) and author of the Hamilton C Shell, an advanced interactive command processor and tools package for OS/2 and Windows NT. Reach Douglas on WIX as hamilton or care of Editor at the address on page 14.