Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

Removing Memory Errors from 64-Bit Platforms

Source Code Accompanies This Article. Download It Now.

- mem_err.txt

It's crucial to address potential memory errors before porting to 64-bit platforms.

October 01, 2005
URL:http://www.drdobbs.com/parallel/removing-memory-errors-from-64-bit-platf/184406286

Rich is the lead engineer for Insure++ at Parasoft. He can be contacted at rnewman@parasoft.com.

It is crucial that 64-bit platforms be stable and reliable as development teams create applications for them. Any memory errors or memory corruptions can cause them to fail. The great challenge with memory errors is that they are elusive problems that are extremely difficult and time consuming to find. Memory errors do not expose themselves during typical means of testing. Because of their detrimental crashing potential, it is imperative that you remove memory problems from code before it goes into production.

There are powerful memory-error-detection tools available that identify the cause of threaded memory errors in dual-core applications. Such detection tools let you find and fix elusive, crash-causing memory errors that traditional testing techniques fail to uncover. Memory-error-detection tools help you find and fix C/C++ memory errors prior to release. By fixing these problems before porting, you can improve the quality of applications on new platforms and architectures, streamline the porting process, and make the original application more robust and reliable.

Why Is Porting So Difficult?

Most developers responsible for porting C/C++ code to 64-bit processors—or to any new hardware, for that matter—find that memory problems seem to multiply when they reach the new platform or architecture.

The fundamental problem with the transition to 64-bit architectures is that assumptions about the size in bits of the various integral and pointer types are no longer true. Coding constructs that are potentially problematic are implicit restrictions from *long* to *int* by assignment and explicit casts. The former most likely causes compilers to issue warnings; the latter, on the other hand, will be accepted silently, leading to all sorts of problems that will not surface until runtime. Another issue is that integer constants that are not explicitly sized are assumed to be *int*s. This is of some concern for mixing signed and unsigned constants. Proper use of the relevant suffixes should alleviate such problems.

Other major sources of trouble are the various types of pointer incompatibilities. For example, on most 64-bit architectures, a pointer no longer fits inside an *int* and code that stores pointer values inside *int* variables no longer functions properly.

These and other problems are typically exposed during porting because porting is essentially a type of mutation testing. When you port code, you essentially create what can be called an "equivalent mutant"—a version of the original code with minor changes that should not affect the outcomes of your test cases. You can expose many strange errors by creating and running these equivalent mutants. In C++, this process of creating and running equivalent mutants can uncover:

- Lack of copy constructors or bad copy constructors.
- Missing or incorrect constructors.
- Wrong order of initialization of code.
- Problems with operations of pointers.
- Dependence on undefined behavior, such as order of evaluation.

Porting Preparation

There are several steps involved in preparing your applications for porting.

Step 1. Before you start porting, you should rid original code of problems such as memory corruption and memory leaks that will plague you on 64-bit processors. One of the most effective ways to expose the types of pointer and integer problems that causes trouble on 64-bit processors is to leverage mutation testing for runtime error detection.

Mutation testing was introduced as an attempt to solve the problem of not being able to measure the accuracy of test suites. In mutation testing, you are in some sense trying to solve this problem by inverting the scenario. The thinking goes like this: Assume that you have a perfect test suite, one that covers all possible cases. Also assume that you have a perfect program that passes this test suite. If you change the code of the program (this process is called "mutating") and run the mutated program ("mutant") against the test suite, you will have two possible scenarios:

1. The results of the program were affected by the code change and the test suite detects it. It was assumed that the test suite is perfect, which means that it must detect the change. If this happens, the mutant is called a "killed mutant."
2. The results of the program are not changed and the test suite does not detect the mutation. The mutant is called an "equivalent mutant."

If you take the ratio of killed mutants to all the mutants that were created, you get a number that is smaller than 1. This number measures how sensitive the program is to the code changes. In reality, neither the perfect program nor the perfect test suite exists, which means that one more scenario can exist.

The results of the program are different, but the test suite does not detect it because it does not have the right test case. If you take the ratio of all the killed mutants to all the mutants generated, you get a number smaller than 1 that also contains information about accuracy of the test suite.

In practice, there is no way to separate the effect that is related to test-suite inaccuracy and the effect that is related to equivalent mutants. In the absence of other possibilities, you can accept the ratio of killed mutants to all the mutants as the measure of the test suite accuracy.

Example 1 (test1.c) illustrates the ideas just described (this code, and all subsequent code presented here, compiles and runs under Linux). The test1.c program can be compiled with the command:

```
cc -o test1 test1.c.
```

The program reads its arguments and prints messages accordingly. Now assume that you have this test suite that tests the program:

```
Test Case 1:
input 2 4
output Got less than 3
Test Case 2:
input 4 4
output Got more than 3
Test Case 3:
input 4 6
output Got more than 3
Test Case 4:
input 2 6
output Got less than 3
Test Case 5:
input 4
output Got more than 3
```

This test suite is representative of the test suites in the industry. It tests positive tests, which means it tests if the program reports correct values for the correct inputs. It completely neglects illegal inputs to the program. The test1 program fully passes the test suite; however, it has serious hidden errors.

Now, mutate the program. You can start with these simple changes:

```
Mutant 1: change line 9 to the form
if(atoi(argv[2]) <= 5)
Mutant 2: change line 7 to the form
if(atoi(argv[1]) >= 3)
Mutant 3: change line 5 to the form
int c=3;
```

If you run this modified program against the test suite, for *Mutant*s *1* and *3* the program completely passes the test suite. For *Mutant 2*, the program fails all test cases.

*Mutant*s *1* and *3* do not change the output of the program, and are thus equivalent mutants. The test suite does not detect them. *Mutant 2*, however, is not an equivalent mutant. Test Cases 1-4 will detect it through wrong output from the program. Test Case 5 may have different behavior on different machines. It may show up as bad output from the program, but at the same time, it may be visible as a program crash.

Switching gears for a moment, if you calculate the statistics, you see that you created three mutants and only one was killed. This tells you that the number that measures the quality of the test suite is 1/3. As you can see, the number 1/3 is low. It is low because you generated two equivalent mutants. This number should serve as a warning that you are not testing enough. In fact, the program has two serious errors that should be detected by the test suite.

Returning to *Mutant 2*, run it against Test Case 5. If the program crashes, then the mutation testing that you performed not only measured the quality of the test suite, but also detected a serious error in the code. This is how mutation testing can find errors.

Consider the equivalent mutant (*Mutant 4*) in Example 2. The difference between *Mutant 4* and previous mutants is that *Mutant 4* was created in an attempt to make an equivalent mutant. This means that when it was constructed, an effort was made to build a program that should execute exactly the same as the original program. If you run *Mutant 4* against the test suite, Test Case 5 will probably fail—the program will crash. As you can see, by creating an equivalent mutant, you actually increased the detection power of the test suite. The conclusion that you can draw here is that you can increase the accuracy of the test suite in two ways:

1. Increase the number of test cases in the test suite.
2. Run equivalent mutants against the test suite.

These conclusions are important; the second conclusion is especially important because it reveals that mutants result in more effective tests. In the examples, you created each mutant by manually making a single change to a program. The process of generating mutants is difficult and time consuming, but it is possible to generate equivalent mutants automatically. Example 3 helps illustrate how this is done. This program has no input and only one output. In principle, it only needs one test case:

```
Test Case 1:
input none
output 12
```

The interesting thing about this program is that it can give the answer *13* or *12* depending on the behavior of the compiler. Suppose that you were given the task of creating this program and making sure that it runs on two different platforms. If the platforms have compilers that exhibit different behavior, you will discover the difference when running the program, triggering the question, "What is wrong?" This will probably result in the program's problem being fixed.

Suppose that you create the equivalent mutant in Example 4. The result of this program does not depend on the compiler and is, in fact, exactly predictable—it is 13. If you run the mutant against the test suite, you will discover the error.

The most amazing thing about mutation testing is that it can discover errors that normally are almost impossible to detect. Frequently, when these errors are uncovered, they manifest themselves as the program is crashing. Often, programmers do not understand that. The equivalent mutant is an opportunity to discover errors, not a headache. Typically, programmers expect equivalent mutants to behave the same as the original program. If this were true all the time, mutation testing would be completely useless.

Step 2. After you clean the most critical errors, use a static-analysis tool to identify code that is likely to cause trouble when it is ported to the new platform/architecture. There are two main tasks to focus on while performing static analysis:

1. Identify and fix code that is likely to result in an error on any platform or architecture.
2. Identify and fix code that might not port well.

First, check industry-respected C/C++ coding standards that identify coding constructs, which are likely to lead to problems on any platform or architecture. By ensuring that code complies with these coding standards, you prevent errors. This translates to less debugging on the new platform or architecture and reduces the chance of having bugs that elude testing and make their way into the release.

Some coding standards to check include:

- Never return a reference to a local object or a dereferenced pointer initialized by "new" within the function. Returning a reference to a local object might cause stack corruption. Returning a dereferenced pointer initialized by "new" within the function might cause a memory leak.
- Never convert a *const* to non*const*. This can undermine the data integrity by allowing values to change that are assumed to be constant. This practice also reduces the readability of the code because you cannot assume *const* variables to be constant.
- If a class has any virtual functions, it shall have a virtual destructor. This standard prevents memory leaks in derived classes. A class that has any virtual functions is intended to be used as a base class, so it should have a virtual destructor to guarantee that the destructor is called when the derived object is referenced through a pointer to the base class.

- Public member functions shall return *const* handles to member data.
- When you provide non*const* handles to member data, you undermine encapsulation by allowing callers to modify member data outside of member functions.
- A pointer to a class shall not be converted to a pointer of a second class unless it inherits from the second. This "invalid" down casting can result in wild pointers, data corruption problems, and other errors.
- Do not directly access global data from a constructor.

The order of initialization of static objects defined in different compilation units is not defined in the C++ language definition. Therefore, accessing global data from a constructor might result in reading from uninitialized objects. (For more rules, see the works of Scott Meyers, Martin Klaus, and Herb Sutter.)

After you locate and repair this error-prone code, start looking for code that works fine on your current platform/architecture, but that might not port well. Some rules that are applicable to most 64-bit porting projects include:

- Use standard types whenever applicable. Consider using *size_t* rather than *int*, for example. Use *uint64_t* if you want a 64-bit unsigned integer. Not only will this practice help identify and prevent current bugs in the code, it will also help with the porting effort in the future when the code is ported to 128-bit processors.
- Review all existing uses of *long* data types in the source code. If the values to be held in such variables, fields, and parameters fit in the range of 2*Gig*-1 to -2*Gig* or 4*Gig* to 0, then it is probably best to use *int32_t* or *uint32_t*, respectively.
- Examine all instances of narrowing assignment. Avoid such assignments because the assignment of a *long* value to an *int* results in truncation of the 64-bit value.
- Find narrowing casts. Use narrowing casts on expressions, not operands.
- Find casts from *long** to *int**. In 32-bit environments, these might have been used interchangeably. Examine all instances of incompatible pointer assignments.
- Find casts from *int** to *long**. In 32-bit environments, these might have been used interchangeably. Examine all instances of incompatible pointer assignments.
- Find uses of multiplicative expressions not containing a *long* in either operand. To have integral expressions produce 64-bit results, at least one of the operands must have a data type of *long* or *unsigned long*.
- Find *long* values that are initialized with *int* literals. Avoid such initializations because integral constants might be represented as 32-bit types even when used in expressions with 64-bit types.
- Locate *int* literals in binary operations for which the result is assigned to a *long* value. 64-bit multiplication is desired if the result is a 64-bit value.
- Find *int* constants used in 64-bit expressions. Use 64-bit values in 64-bit expressions.
- Find all pointers cast to *int* values. Code involving conversions of pointers from or to integral values should be reviewed.
- Find and review any in-line assembly This probably will not port well.

Step 3. Repeat runtime error detection to verify that the modifications you made while fixing coding standard violations did not introduce any runtime errors.

Step 4. At this point, you have the option of performing one more step to ensure that your code is as clean as possible before you port it. This additional step is unit testing. Unit testing is traditionally used to find errors as soon as each application unit is completed. It can also be beneficial later in the development process because, at the unit level, it is easier to design inputs that reach all of the functions. In turn, this helps you find errors faster and expose errors that you might not uncover with application-level testing.

Identifying Problems on The 64-Bit Processor

Of course, there may be problems with the 64-bit processor itself. If so, follow these guidelines:

Step 1. Recompile your application on the 64-bit processor. If you have trouble compiling it, work out all of the quirks related to compiler variations. You might also want to create coding standard rules that will automatically identify the code associated with these quirks so that you can prevent these compilation problems from occurring in the future.

Step 2. Once you recompile the code, perform code inspection again to check if the new code complies with all appropriate coding standards. At this point, every change that should have been made but that was not made is an error. Fix these errors immediately! You don't want to look for these errors as the application is running.

Step 3. Link your application and try to build it.

Step 4. At this point, you should try to run your code. If you have a problem getting code running on the 64-bit processor, use a unit-testing framework to run the code function by function; that way, you can flush exactly what in the code is not portable. Test the function, and then fix the problem. Continue this process until the entire application is running.

Step 5. Repeat runtime error detection.

Once the application is running, you'll want to repeat runtime error detection because the porting process is likely to cause some new problems that could not be detected before the port (for example, new memory corruption problems or different behaviors). If the runtime error detection exposes problems, fix every bug related to porting.

Conclusion

Following the guidelines proposed in this article, you will find and fix C/C++ memory errors prior to release to save weeks of debugging time and prevent costly crashes from affecting your customers.

DDJ

```
main(argc, argv)                        /* line 1 */
int argc;                               /* line 2 */
char *argv[];                           /* line 3 */
{                                 /* line 4 */
  int c=0;                        /* line 5 */
                                  /* line 6 */
    if(atoi(argv[1]) < 3){                /* line 7 */
      printf("Got less than 3\n");        /* line 8 */
      if(atoi(argv[2]) > 5)               /* line 9 */
        c = 2;                            /* line 10 */
    }                                     /* line 11 */
    else                                  /* line 12 */
      printf("Got more than 3\n");        /* line 13 */
   exit(0);                               /* line 14 */
}                                         /* line 15 */
```

Example 1: The test1.c program.

```
main(argc, argv)                        /* line 1 */
int argc;                               /* line 2 */
char *argv[];                           /* line 3 */
{                                       /* line 4 */
  int c=0;                              /* line 5 */
  int a, b;                             /* line 6 */
                                        /* line 7 */
  a = atoi(argv[1]);                    /* line 8 */
  b = atoi(argv[2]);                    /* line 9 */
  if(a < 3){                            /* line 10 */
    printf("Got less than 3\n");        /* line 12 */
    if(b > 5)                           /* line 13 */
      c = 2;                            /* line 14 */
```

```
  }                                    /* line 15 */
  else                                 /* line 16 */
    printf("Got more than 3\n");       /* line 17 */
 exit(0);                              /* line 18 */
}                                      /* line 19 */
```

Example 2: Equivalent mutant.

```
int doublew(x)
int x;
{ return x*2; }

int triple( y)
int y;
{ return y*3; }

main() {
  int i = 2;
  printf("Got %d \n", doublew(i++)+ triple(i++));
}
```

Example 3: Automatically generating mutants.

```
int doublew(x)
int x;
{ return x*2; }

int triple( y)
int y;
{ return y*3; }

main() {
  int i = 2;
  int a, b;

  a = doublew(i++);
  b = triple(i++);
  printf("Got %d \n", a+b);
}
```

Example 4: A mutant.