Portably handle exceptional errors in C++

Asked 16 years, 3 months ago Modified 15 years, 5 months ago

Viewed 3k times

7

I'm working on porting a Visual C++ application to GCC (should build on MingW and Linux).

The existing code uses `__try { ... } __except(1) { ... }` blocks in a few places so that almost nothing (short of maybe out of memory type errors?) would make the program exit without doing some minimal logging.

What are the options for doing something similar with GCC?

Edit: Thanks for the pointer to /EH options in Visual Studio, what I need now is some examples on how to handle signals on Linux. I've found this message from 2002.

What other signals besides `SIGFPE` and `SIGSEVG` should I watch out for? (Mostly care about ones that might be raised from *me* doing something wrong)

Bounty Information: I want my application to be able to self-log as many error conditions as possible before it exits.

What signals might I get and which would generally be impossible to log an error message after? (Out of memory, what else?)

How can I handle exceptions and (most importantly) signals in a portable way that the code at least works the same on Linux and MingW. #ifdef is OK.

The reason I don't just have a wrapper process that logs the failure is that for performance reasons I save writing some data to disk till the last minute, so if something goes wrong I want to make all possible attempts to write the data out before exiting.

exception visual-c++ gcc cross-platform portability

Share

Improve this question

Follow

edited Jul 22, 2009 at 20:01

asked Sep 2, 2008 at 10:57

thelsdj

9,134 ● 10 ● 46 ● 59

5 Answers

Sorted by: Highest score (default)

12

try { xxx } catch(...) { xxx } would be more portable but might not catch as much. It depends on compiler settings and environments.

+75

Using the default VC++ settings, asynchronous (SEH) errors are not delivered to the C++ EH infrastructure; to catch them you need to use SEH handlers (__try/__except) instead. VC++ allows you to route SEH errors through C++ error-handling, which allows a catch(...) to trap SEH errors; this includes memory errors such as null pointer dereferences. Details.

On Linux, however, many of the errors that Windows uses SEH for are indicated through signals. These are not ever caught by try/catch; to handle them you need a signal handler.

Share Improve this answer Follow

answered Sep 2, 2008 at 11:11

DrPizza
18.3k ● 7 ● 42 ● 53

0

Why not use the C++ standard exceptions instead of MSFT's proprietary extension? C++ has an exception handling concept.

```
struct my_exception_type : public logic_error {
    my_exception_type(char const* msg) : logic_error(m
};

try {
    throw my_exception_type("An error occurred");
} catch (my_exception_type& ex) {
    cerr << ex.what << endl;
}
```

C++ also has a “catchall” clause so if you want to log exceptions you can use the following wrapper:

```
try {
    // …
}
catch (...) {
}
```

However, this is not very efficient in C++ because creating such a general wrapper means that handling code has to be inserted in *every* subsequent stack frame by the compiler (unlike in managed systems like .NET where exception handling comes at no additional cost as long as no exception is actually thrown).

Share Improve this answer Follow

answered Sep 2, 2008 at 11:07

Konrad Rudolph
545k ● 139 ● 956 ● 1.2k

The OP is looking for something to address lower-level exceptional conditions than C++ language-level exceptions. In a POSIX world, this means installing signal handlers for (most) everything. In MSVC, the __try/__catch mechanism handles similar faults (including the Win32 equivalent of SEGFAULT) – Tom Barta Nov 7, 2008 at 2:07

Right about the traps. But you can configure MSVC to catch SEH using the standard `try` / `catch`, as elaborated by DrPizza. – Konrad Rudolph Nov 7, 2008 at 8:14

For portability, one thing to try is using try-catch blocks for most vanilla exceptions and then set a terminate handler

0

(set_terminate_handler) to have a minimal hook available for catastrophic exit conditions. You can also try adding something like an atexit or on_exit handler. Your execution environment may be bizarre or corrupt when you enter these functions, of course, so be careful of how much you presume a sane environment.

Finally, when using regular try-catch pairs you can consider using function try blocks as opposed to opening a try block in the body of a function:

```
int foo(int x) try {
  // body of foo
} catch (...) {
   // be careful what's done here!
}
```

they're a relatively unknown chunk of C++ and may in some cases offer recovery even in the event of partial (small scale) stack corruption.

Finally, yes, you'll probably want to investigate which signals you can continuably handle on your own or on which you might abort, and if you want less handling mechanisms in place, you might consider call the none-throwing version of the new operator, and compiling to not generate floating point exceptions if needed (you can always check isnan(.), isfinite(.), on FP results to protect yourself).

On that last note, be careful: I've notice that the floating point result classification functions can be in different headers under linux and windows... so you may have to conditionalize those includes.

If you're feeling puckish, write it all using setjmp and longjmp (that's a joke...).

Share Improve this answer Follow

answered Sep 5, 2008 at 22:01

Thomas Kammeyer
4,507 ● 23 ● 31

0

Catching C++ exceptions with `catch(...)` already puts you in a twilight zone.

Trying to catch errors not caught by `catch(...)` puts you squarely inside undefined behaviour. No C++ code is guaranteed to work. Your minimal logging code may cause the missile to launch instead.

My recommendation is to not even try to `catch(...)`. Only catch exceptions that you can meaningfully and safely log and let the OS handle the rest, if any.

Postmortem debugging gets ugly if you have error handling code failures on top of the root cause.

Share Improve this answer Follow

edited Jul 24, 2009 at 18:54

answered Jul 24, 2009 at 18:34

Hans Malherbe
3,018 ● 25 ● 19

-1

One way that is easy to use, portable, and barely use any resources would be to catch empty classes. I know this may sound odd at first, but it can be very useful.

Here is an example I made for another question that applies for your question too: link

Also, you can have more than 1 catch:

```
try
{
   /* code that may throw exceptions */
}
catch (Error1 e1)
{
   /* code if Error1 is thrown */
}
catch (Error2 e2)
{
   /* code if Error2 is thrown */
}
catch (...)
{
   /* any exception that was not expected will be caug
}
```

Share Improve this answer Follow

edited May 23, 2017 at 11:48
Community Bot
1 ● 1

answered Jul 23, 2009 at 17:58
Partial
9,969 ● 12 ● 45 ● 57