Testing Complex C++ Systems

Some systems are so complex that they require non-traditional approaches to thoroughly test large code bases.

January 29, 2013
URL:http://www.drdobbs.com/cpp/testing-complex-c-systems/240147275

In part one of this three-part series on deep testing complex systems, I covered the practical aspects of deep testing and demonstrated how to test difficult areas of code, such as the user interface, networking, and asynchronous code.

In this article, I discuss some techniques and utilities I have used to successfully test complex systems in C++. In the final article next week, I'll discuss similar complex testing for Python (with Swig C++/Python bindings) and high-volume, highly available C#/.NET Web services.

Obstacles to Testing Complex C++ Code

C++ arguably poses the most difficult challenges for testing among the major industrial-strength programming languages and environments. There are several challenges to deal with: Build speed is slow; the language uses weak types; and C++ programmers love twisting code or template-based designs to extract every last bit of performance. Let's look at how these factors affect testing complex code.

Slow Build Speeds

C++ projects (especially cross-platform projects) are often built as one big monolith. They can comprise large numbers of static project libraries along with many more third-party libraries that are all linked together to generate the final executable (often, several executables).

The result of this monolithic construction is less than stellar dependency management. I've seen several large C++ codebases where every merge was a project in itself involving a dedicated build engineer to convene multiple developers to help resolve conflicts and build the software, sometimes requiring multiple building passes to combat cyclic dependencies. Add the tools used to build many cross-platform C++ projects (such as make) and you get a picture of slow, very complex builds.

The impact of this on testing is profound. When simply building the software is such a chore, the tendency is to minimize the number of artifacts you generate. This leads to one of several unpleasant choices:

1. No tests — just build the software and play with it; trust your QA team to find all the bugs.
2. Add some test support to the main software with compile-time or runtime switches.
3. One big test executable (or test framework) that links against all the dependencies and contains all the tests.
4. Many small test executables, each linked against its dependencies.

Option #1 is an obvious no-go from serious testing point of view (but very common in practice).

Option #2 is cumbersome. It doesn't let you to perform isolated testing and complicates the final product with testing code and dependencies.

Option #3 requires a slow build of the huge test executable whenever you make a small change (either to the code under test or to the test itself).

Option #4 requires a *lot* of tinkering around. It's pretty good when you work on an isolated bug or feature and has to build just one small test executable, but when you want to run a suite of regression tests, you will have to wait a long time for the build because each and every small executable will have to link against all its dependencies, which will duplicate a lot of effort (especially for common libraries used by every test executable).

The Quest for Performance

C++ was designed for raw performance. You pay only for what you use. It is compatible with C, which started as a glorified assembly language. In later years, C++ became a multi-paradigm language supporting procedural, object-oriented, and generic programming styles, while still maintaining its C compatibility and performance. However, this led to a programming language that is an order of magnitude more difficult to learn and practice than other mainstream languages. In its glory days (early '90s), C++ was used for everything, especially on Windows. You wrote your number-crunching code and your user interface and your networking code in C++, and that was that. The rise of the Web and Java changed everything. Suddenly, the programming language of your project wasn't a given and you could even mix and match programming languages in the same project. Later, hardware got faster, other languages got better (C#, Python), and Web applications where developed in every possible language except C++ (anyone remember ATL server?). That relegated C++ to the engine to do what it does best — fast processing. C++ programmers (unless they are polyglot and dabble in other languages) justifiably focus more than anything on generating fast and tight code. Many great C++ developers formed their habits before automated unit testing and the agile movement made it to the big stage. Also, because performance is very often a system-wide property and not just the sum of the performance of the various components, its pursuit leads to a dependence on the usefulness of unit testing.

Very Weak Type System

C++ is a statically typed language, but its type system is very weak. You can freely cast any pointer to any other pointer using `reinterpret_cast`. It can perform implicit casts/coercions/promotions on your behalf in certain situations. In addition, it has the legacy C preprocessor that allows inventing private sub-languages in a non-typesafe way. Add to that powerful templates with no constraints. Because all these features are heavily used in industrial strength programs, you end up with a combustible concoction that is pretty difficult to reason through and to feel secure that you have covered in your tests all the ways the code can misbehave.

Solutions

My favorite technique for fast C++ tests is to move from the traditional monolith with lots of static libraries to a component-based approach, where the application is composed of DLLs or shared libraries. (To this end, my series of articles on C++ plugins explains how to do this using a plugin framework.) Then the tests themselves are DLLs, which can be built and executed as needed by the test framework. This requires a lot of foundational work and is suitable either to green-field projects or those where upper-management supports re-engineering the system.

As to the weak type system, if you are able to build (or refactor) the system, you are on solid ground here. All the techniques I demonstrated in the first article are applicable to to C++, too. Interfaces in C++ are just classes/structs with only pure virtual functions. Be careful about the data types you use in the signature of your interface methods and be mindful about the public interfaces you expose from your components.

Testing Non-Public Code

Testing non-public code in C++ is a controversial topic. One camp says you should never do it and only test the public interface. I agree with this sentiment and try to accommodate it by creating small components with clear interfaces that are easy to test from the outside. However, every now and then, you will have a component with complex internal machinery that is not easy to test via the external interface. One way to get to the private parts of classes is via the dreaded "friend" keyword: a C++ class that may grant access to its private parts to other classes. Another approach is to add public introspection methods (perhaps protected by a TEST symbol that requires a special test build and are not available in production).

Be careful when using these techniques. Friendliness can be dangerous, and requires your production code to know of the code that tests it. If you decide to add another test, you need to modify your production code. Also, if developers see "friend" is used, they might decide to use it to provide access to other production classes instead of applying best practices. It can become a slippery slope. Providing TEST-only methods to introspect internal state is not ideal either. Again, some "clever" developer may decide to define the TEST symbol in his production build and call your TEST-only methods in production. Also, the TEST-only code may have unintended side effects. Since you don't test your code without it, you don't know how your code will behave in production.

One other really nefarious technique is to use macros to redefine the "private" and "protected" keywords to "public." This is very hackish and requires a lot of attention, but at least you can do it from within your test code. Here, `class A` is defined in a header file called "A.h":

```
class A
{
public:
    A() { x = 5; }
private:
    int x;
};
```

The following test code is able to access and even modify its private `x` variable:

```
#define private public
#include "A.h"
#undef private

void accessPrivateState()
{
    A a;
    a.x = 3;
}
```

This technique is, of course, super fragile and should be used with caution. It can break in several ways, but the good news is that the compiler will tell you about it. One way it can break is if the private state uses the default "privateness" of C++ classes:

```
class A
{
    int x;
public:
    A() { x = 5; }
};
```

If the private state is defined at the top of the class without any access modifier, it is private by default, but there is no "private" statement in the class that you can redefine to "public." If you have access to the code, you can just add the "private" keyword. Another way it can break is if another class `#includes` "A.h" and it is protected with an `#include` guard. In this case, the first definition of `A` will have `x` as private and your trickery will not work.

File A.h

```
#ifndef A_H
#define A_H

class A
{
    int x;
public:
    A() { x = 5; }
};
```

```
#endif
```

File B.h

```
#include "A.h"
class B
{
        A a;
};
```

You can redefine "private" to "public" as the first statement in your test's `main` function, but if you link with prebuilt static libraries or DLLs, you'll have to ensure they are built with redefined "private," too. As I said, it's a very brittle technique.

Real-Life Scenario: Testing Rockmelt

Rockmelt is a social browser built on top of Chromium — the open source browser behind Google Chrome — which, itself, is built on top of WebKit and other open source projects. WebKit has plugins, Chromium has extensions, and Rockmelt has "Extended" extensions, which enable extensions to access Rockmelt-specific functionality. Rockmelt also has a big back-end component that the browser communicates with. If you think it's a bear to manage, you're right. Chromium releases a new version every six weeks and Rockmelt has to follow closely. If Rockmelt falls behind, then the browser is vulnerable to public security issues that were fixed on Chromium, and extensions that rely on Chrome might stop working. A significant part of each iteration is dedicated to upgrading Rockmelt to the next version of Chromium. Rockmelt is not just Chromium + a few extensions. It is a highly customized version that reaches deep into Chromium (all back-end communication, synch framework, tons of custom UI, Rockmelt JavaScript bindings, etc). It is developed in C++ and supports Windows and Mac OS X.

For various technical reasons, a lot of Rockmelt's Chromium customization is done at the `#ifdef` level and by adding new files to existing Chromium static libraries. This leads to an unusual situation for testing. You don't have nice, isolated components to test, where you can just mock dependencies. The bottom line is that the native client side of Rockmelt has virtually no automated testing. There is an off-shore QA team that manually tests the product (or rather, various parts of it since it is too big to test entirely) every night and reported defects are subsequently assigned. Due to the complexity of Chromium itself and the deep integration of Rockmelt, it is often difficult to figure out what's wrong from the black-box description of the QA team ("When I drag a URL to a friend icon, sometimes it doesn't show the share dialog"). A lot of time is spent trying to reproduce bugs and find the root causes. The Rockmelt technical leadership identified this situation as a serious quality and productivity issue.

They brought in Google's testing guru Miško Hevery to talk about testing. Shortly thereafter, I started a pilot project to make Rockmelt's client code testable. I discovered early on that refactoring Rockmelt's code outside of Chromium's libraries was the key. With a lot of work in which I had to dig my way backwards and define interfaces and wrappers for transitive dependencies, I was able to completely decouple one Rockmelt service, called AppEdge, and make it fully testable with no concrete dependencies. I'll give you an idea of what this service does so you can appreciate the complexity: It manages a list of custom apps and RSS feeds that are displayed on the edge of the browser. It supports downloading over the network and installing/upgrading Chromium and Rockmelt extensions. It responds to user clicks and menu selection, it launches apps, and it sends notifications to other components. Overall, it had about 15 dependencies and served as the source of about 10 events/notifications. At the end, all this technology was completely testable.

Testing via Plugins

Numenta does unusual work, such as reverse engineering the brain and building intelligent machines based on the neo-cortex. Numenta created a platform for intelligent computing called NuPIC that had a C++ engine with Python bindings, several Python frameworks, and a set of Python tools. At Numenta, testing was paramount and there was a battery of tests starting from C++ unit tests for the core engine ranging up to Python-integration tests and performance tests. As a machine-learning platform, NuPIC presented many testing challenges. In the early days, there was a lot of trial-and-error experimentation, but with time, better tools and understanding of the internals allowed more introspective tests and metrics.

My first project when I joined Numenta was to design a plugin framework to allow third-party developers to plug their custom C++ algorithms into NuPIC (NuPIC has always supported custom Python algorithms, but sometimes users needed C++'s raw speed). It was a cross-platform C++ framework and it presented some unique testing challenges. One of the major goals, which is very difficult to do in C++, is to allow the plugins to be built using different compilers than NuPIC used. This way, third-party developers were not locked into Numenta's choice of a compiler and both sides were free to upgrade their tool chain without breaking compatibility. How do you test such property?

An important weapon in the arsenal was incremental integration while running the plugin framework side-by-side with the hard-coded algorithms in the existing runtime engine. This approach was beneficial because it allowed black-box testing of the plugin framework, as well as running experiments using the hard-coded algorithms wrapped as plugins. I created plugins that reused the logic inside the fixed algorithm nodes and then constructed experiments that used the same networks with the same algorithm types and connectivity except that on one network the nodes used were the concrete (old) sub-classes of the runtime engine, and on another network all the nodes were `PluginNode` hosting the same algorithmic code. This allowed apples-to-apples comparison both in terms of accuracy and also in terms of performance (maybe the plugin framework does something funky and wastes a lot of time or memory at some point).

C++ Test Frameworks

With C++ (just like in any other language), you have the choice of writing your own test framework vs. using some off the shelf test framework. Normally, I opt to use existing tools, but test frameworks may be different. One reason you would want to build your own is to get tight integration with the build system. Another reason is that most test frameworks are either geared toward developer-level unit tests or QA-level running *all* the tests. Finally, writing a test framework is easy and fun and gives you full control. If you are working within some application framework that provides a test framework, try to use it first. For example, at Rockmelt we used the Google/Chromium code base, which comes with its own test framework, so it was a no brainer.

I use mock objects a lot in my tests, but I have never tried any mock framework. My only excuse is that I never felt the need. When I write code, I think of each dependency as a potential mocking target that I need to provide via an interface. At this point, writing a mock takes literally seconds and, if I need to, I can even write a little tool to generate a mock object automatically from an interface. This generally works well for me and enables testing using the test framework of my choice.

Deep testing is guaranteed to reduce the number of undiscovered issues down the line. In C++, in particular, it can present difficult problems that require original solutions. In rare cases, it might require rebuilding the code under test. But in other situations, developing plugins to the main program that duplicate its activities will provide a viable testing sandbox. And in other circumstances, it might be necessary to write your own testing framework. But with the right mindset and approach, deep testing is very cost-effective. Non-trivial code will contain bugs. And it's up to you to use deep testing to remove them before they show up in the field.

Gigi Sayfan specializes in cross-platform object-oriented programming in C/C++, C#, Python, and Java with emphasis on large-scale distributed systems. He is a long-time contributor to Dr. Dobb's.

Related Articles

Testing Complex Systems

Testing Python and C# Code