Using #pragma detect_mismatch to help catch ODR violations

devblogs.microsoft.com/oldnewthing/20160803-00

August 3, 2016

Raymond Chen

One of the more insidious problems you may encounter are those traced back to violations of the C++ One Definition Rule. As a general rule, if the conflicting definitions occur in separate translation units, the behavior is undefined, but no diagnostic is required. The lack of a diagnostic means that if two translation units define a type differently (say, because they were compiled with different compile-time configurations), you may not notice a problem until you start dealing with mysterious memory corruption.

These types of bugs are not fun to diagnose.

If you use the Microsoft Visual C++ toolchain, then you can use the `#pragma detect_mismatch("name", "value")` directive to give the linker some help in identifying mismatched definitions. Specifically, the linker verifies that all such declarations with the same name also have the same value.

The idea here is that if you have something that is declared differently based on compilation settings, you can emit a different `#pragma detect_mismatch("name", "value")` for each version, using the same name but a different value. The linker will then verify that everybody used the same version of the header file.

Here's an example:

```
// This is a fake mutex that does no locking.
struct fake_mutex
{
 void lock() {}
 void unlock() {}
};

class Contoso
{
#ifdef SINGLE_THREADED
   // single-threaded doesn't need a mutex.
   typedef fake_mutex mutex_t;
#else
   // multi-threaded needs a true mutex.
   typedef std::mutex mutex_t;
#endif

public:
  Contoso();

  void Activate()
  {
     std::lock_guard<mutex_t> lock(object_mutex);
#ifndef NDEBUG
    isActivated = true;
#endif
    ... business logic to activate the object ...
  }

  void Charge()
  {
     std::lock_guard<mutex_t> lock(object_mutex);
    // You must activate before you can charge.
    assert(!isActivated);
    ... business logic to charge the object ...
  }

private:
  ...
  mutex_t object_mutex;
#ifndef NDEBUG
  bool isActivated = false;
#endif
};
```

If this class is used in a project, but one file in the project is compiled with `SINGLE_THREADED` and another file is compiled without `SINGLE_THREADED`, or if the two files disagree on `NDEBUG`, then you have an ODR violation. In practice, this means that bad things will happen if the two files try to access the same `Contoso` object.

You can use `#pragma detect_mismatch` to encode which definition is being used. This allows the linker to detect whether a single project uses multiple conflicting definitions.

```
// This is a fake mutex that does no locking.
struct fake_mutex
{
 void lock() {}
 void unlock() {}
};

class Contoso
{
#ifdef SINGLE_THREADED
   // single-threaded doesn't need a mutex.
   typedef fake_mutex mutex_t;
   #pragma detect_mismatch("Contoso threading", "Single");
#else
   // multi-threaded needs a true mutex.
   typedef std::mutex mutex_t;
   #pragma detect_mismatch("Contoso threading", "Multi");
#endif

#ifdef NDEBUG
   #pragma detect_mismatch("Contoso debug", "Nondebug");
#else
   #pragma detect_mismatch("Contoso debug", "Debug");
#endif

public:
  Contoso();

  void Activate()
  {
     std::lock_guard<mutex_t> lock(object_mutex);
#ifndef NDEBUG
    isActivated = true;
#endif
    ... business logic to activate the object ...
  }

  void Charge()
  {
     std::lock_guard<mutex_t> lock(object_mutex);
    // You must activate before you can charge.
    assert(!isActivated);
    ... business logic to charge the object ...
  }

private:
  ...
  mutex_t object_mutex;
#ifndef NDEBUG
  bool isActivated = false;
#endif
};
```

You can see the directive in action in this Channel 9 video starring C++ library master Stephan T. Lavavej. The `detect_mismatch` trick appears around timecode 29:30.

Note of course that you can use this technique for things other than catching ODR violations.

Raymond Chen

Follow