Support for Template Functions

This topic explains how C++test performs coding standards analysis and unit testing on template functions.

In this section:

- Collecting Code Coverage from C++ Templates
- Understanding C++test's Instance Support
- Using "#pragma instantiate" to Enforce Template Instantiation
- Potential Limitations
- Tips

Collecting Code Coverage from C++ Templates

C++test can collect code coverage information for C++ templates. Code coverage information is collected and reported cumulatively for all template instances. To enable collecting code coverage from C++ templates:

1. Open the test configuration and choose the **Execution> Generation** tabs
2. In the Execution details section, click **Edit** at the Instrumentation mode field
3. Enabled the **Enable coverage for C++ templates** option in the Advanced options section of the Test's Configuration's Execution Tab "Instrumentation features" panel.

Understanding C++test's Instance Support

C++test can perform static analysis and unit testing of instantiated function templates and instantiated members of class templates. In this context, "instantiated" means:

- Instantiated (used) in the tested source.
- Instantiated with `"#pragma instantiate"` in the tested source; see Using "#pragma instantiate" to Enforce Template Instantiation for details. Such functions will be further addressed as `"instances"`.

C++test can:

- Perform static analysis of instances' bodies.
- Automatically generate test cases for instances.

The instances' bodies instrumentation is limited. As a result, there are no stubbed calls from instances, and possibly inaccurate stack traces when reporting exceptions (these restrictions do not apply to explicit template specializations).

Using "#pragma instantiate" to Enforce Template Instantiation

If a given template instance is not used in the tested source, you can enforce template instantiation by inserting `#pragma instantiate` into the tested code; for example:

```
#pragma instantiate atype<int>
```

In *C++ Front End Internal Documentation*, the Edison Design Group, Inc. explains:

"The argument to the instantiation pragma may be:

- a template class name A<int>
- a template class declaration class A<int>
- a member function name A<int>::f
- a static data member name A<int>::i
- a static data declaration int A<int>::i
- a member function declaration void A<int>::f(int, char)
- a template function declaration char* f(int, float)

A pragma in which the argument is a template class name (e.g., A<int> or class A<int>) is equivalent to repeating the pragma for each member function and static data member declared in the class. When instantiating an entire class a given member function or static data member may be excluded using the `do_not_instantiate` pragma. For example:

```
#pragma instantiate A<int>
#pragma do_not_instantiate A<int>::f
```

The template definition of a template entity must be present in the compilation for an instantiation to occur. If an instantiation is explicitly requested by use of the instantiate pragma and no template definition is available or a specific definition is provided, an error is issued.

```
template <class T> void f1(T); // No body provided
template <class T> void g1(T); // No body provided
void f1(int) {} // Specific definition
void main()  {
  int i;
  double d;
   f1(i);
   f1(d);
   g1(i);
   g1(d);
}
#pragma instantiate void f1(int) // error - specific definition
#pragma instantiate void g1(int) // error - no body provided
```

f1(double) and g1(double) will not be instantiated (because no bodies were supplied) but no errors will be produced during the compilation (if no bodies are supplied at link time, a linker error will be produced).

A member function name (e.g., A<int>::f) can only be used as a pragma argument if it refers to a single user defined member function (i.e., not an overloaded function). Compiler-generated functions are not considered, so a name may refer to a user defined constructor even if a compiler-generated copy constructor of the same name exists. Overloaded member functions can be instantiated by providing the complete member function declaration, as in `#pragma instantiate char* A<int>::f(int, char*)`

The argument to an instantiation pragma may not be a compiler-generated function, an inline function, or a pure virtual function."

Potential Limitations

- "`#pragma instantiate`" (inline functions, pure virtual functions etc.) has some limitations; see Using "#pragma instantiate" to Enforce Template Instantiation for details.
- C++test cannot automatically-generate test cases for template classes for which a constructor/destructor is not instantiated; instead, either use `#pragma instantiate` to instantiate the constructor/destructor, or create an instance of a given class in the tested code (as a global variable etc.).

Tips

If you do not have any source code that does template instantiations, you could use

```
#ifdef PARASOFT_CPPTEST
#pragma instantiate ...
...
#endif
```

Alternatively, you could use

```
#ifdef PARASOFT_CPPTEST
#include "pragma_instantiate.h"
#endif
```

where the included header contains the instantiations with necessary types.

The benefit of the second option is that if you add to the type set, your headers do not get changed without PARASOFT_CPPTEST defined (no timestamp change) and you save on recompilation.

This way, the pragma does not interfere with your regular code. Typically, this should be done in header files that declare templates. With some compilers, templates can only be in headers—they do not have source files.