95-0157/N0757

Slides from the Template Compilation Model Technical Session

John Spicer
Edison Design Group
jhs@edg.com

Template Instantiation

- automatically instantiate entities that have not been explicitly instantiated
- provide a single definition of
  - » template functions
  - » member functions of template classes
  - » template static data members
- does not apply to classes

Source vs. Instantiation model

- source model specifies how a source program must be structured for automatic instantiation to work
- instantiation model describes how a particular implementation implements instantiation

Why the standard needs a compilation model

- a source model must be specified so that users can write portable programs
- the source model should permit as many different underlying instantiation models as possible

Some Existing Models

- Borland
- Sun
- cfront
- EDG

Example of Present Usage

File: a.h:
```
struct A {};
void g(A);
void h(A, int);
```
File f.c:
```
template <class T> void f(T t)
{
  A a;
  g(a);
  h(a, t);
}
```
File: f.h:
```
#include "a.h"
template <class T> void f(T);
  #ifdef INCLUDE_TEMPLATE_DEFINITIONS
#include "f.c"
#endif
```
File: t.c:
```
#include "a.h"
#include "f.h"

void g(A){}
void h(A, int){}

int main()
{
  f(1);
}
```


Existing models - "Borland"

- source model: include all template definitions
  - » may or may not be in a separate header file
- instantiation model: generate all referenced instantiations, let linker eliminate duplicates

John Spicer
Edison Design Group
jhs@edg.com

Existing Models - Sun

- source model: template definitions in include file, automatically included by implementation
  - » implementation specified means of finding template definition file when needed
  - » definitions may also be explicitly included
- instantiation model: repository of template definition object files generated by normal compilations

cfront model

- source model: template definitions in file that is automatically included by implementation
- instantiation model: instantiations done at link time in synthesized source file that includes the template definition include file

EDG Model

- source model: template definitions in include file, automatically included by implementation
  - » implementation specified means of finding template definition file when needed
  - » definitions may also be explicitly included
- instantiation model: instantiations generated by normal compilations
  - » prelinker decides where instantiations are done

What would users like?

- template declarations in header files
- template definitions in any source file
- reference those templates from anywhere
- compile all files as usual
- everything works out by magic, including templates in libraries

Why haven't implementors provided this?

- it isn't because it hasn't been thought of
- it isn't (just) because of implementation complexity
- for the same reason that you can't buy a car that seats 10, can do 0-60 (mph) in 6 seconds, and gets 100 miles/gallon.

Current Compilation Model

(as described in N0582/94-0195)

- Template definitions are in separately compiled files
- Instantiations are done in a synthesized context at link time

Slides from the Template Compilation Model Technical Session

John Spicer
Edison Design Group
jhs@edg.com

Example of Current Model

File: a.h:
```
struct A {};
```

File: f.c:
```
#include "a.h"  // added to declare A
#include "t.h"   // added to declare g(A)

 template <class T> void f(T t)
 {
   A a;
   g(a);
   h(a, t);
 }
```

File: f2.c:
```
// Alternate version of template f
template <class T> void f(T t) {}
```

File: f.h:
```
template <class T> void f(T);
// No longer includes f.c
```

File: t.c:
```
#include "f.h"
#include "a.h"

void g(A){}
void h(A, int){}

int main()
{
  f(1);
}
```

File: t.h:
```
void g(A);
void h(A, int);
```


Separate Compilation and the current compilation model

Reflector example from Tony Hansen:

File: a.h:
```
// declare the template function
template <class T> int f(T);
```

File: b.c:
```
#include "a.h"
// define the template function
template <class T> T f(T a) { return a * a * a; }
```

File: c.c:
```
#include "a.h"

void foo()
{
  int x = f(3);  // invoke the template
}
```

Tony says:

I would fully expect this program to be compilable by typing in:

```
xcc b.c c.c
```

I would also expect to be able to do the following:

```
xcc -c b.c     # compile the template definition
ar r b.a b.o   # put it in a library
xcc c.c b.a    # link the library with c.c
```


Problems with the Template Compilation Model

- cannot be implemented efficiently enough to be usable
- synthesized contexts are difficult to debug and context synthesis is itself a new source of errors
- my perspective -- as an implementor
  - » not looking at problems for the implementor
  - » looking at problems for users as a consequence of what an implementation is required to do

Who should be concerned about this?

- Everyone -- profound effect on compilation of any program that uses templates
- the standard library is heavily templatized -- virtually every program will make extensive use of templates
- even if you don't use the current model, a library you use might

What are the problems?

- context merging -- expensive to use
- instantiations forced to take place at link time -- severely constrains the kind of instantiation mechanisms that can be provided
- synthesized context -- difficult for users
- poorly specified, novel and untried technology

Context Merging

- information must be saved from the template definition point
- information must be saved from the template reference point
- merged in a synthesized instantiation context
- large amount of information from both contexts is required

John Spicer
Edison Design Group
jhs@edg.com

Implications of the current model (just how bad is it?)

- nothing can be known about a template body at compile time
- instantiation is forced to occur at link time
- lack of knowledge of the template body makes it impossible to know which information from the referencing context will be required by the instantiation

Implications of the current model (continued)

- fully general separate compilation requires that the context information be saved for ***every*** translation unit
  - » can't be optimized because you don't know how object files will be combined
  - » optimization only possible if the complete set of source files, objects etc. is known in advance
  - » but that would eliminate the desired separate compilation characteristics

Context Merging How expensive is it?

- expense when a referencing translation unit is compiled
- expense when an instantiation is generated

Information from the referencing context

- all types used as template arguments
- all functions that could conceivably be called as "dependent" functions
- all types, members, base classes, functions, variables, templates, etc. that could be transitively accessed by the above

Why so much information?

- you know nothing about the body of the template definition when a reference is compiled
- all information that could possibly be accessed by the template body must be supplied

Almost everything must be saved

- all declarative information must be saved (i.e., everything but the bodies of noninline functions)
- it may (or may not) be possible to exclude certain information
  - » but it would take extensive analysis to be sure that something could really be excluded

John Spicer
Edison Design Group
jhs@edg.com

Example of Information that must be saved

```
struct A {
  int i;
  void f() { /* ... */ }
};
struct B {
  A a;
  void g();
};
struct C {
  C(int);
};
template <class T> void f(T);
int main()
{
  B b;
  f(b);
}
```

1. Could f(B) use A? Yes::

```
template <class T> void f(T t) { t.a.i = 1; }
```

2. Could f(B) use C and/or g(int, C)? Yes:

```
template <class T> void f(T t) { g(t.a.i, 1); }
```


Information from the definition context

- representation of the template
- all types, variables, etc. referenced by the template
- all nondependent functions referenced by the template
- all functions that could conceivably be called as dependent functions, either directly or by a template called by this template

Estimating the space required for context information

- no implementation exists for measurement
- similar to information required for precompiled header files
  - » sample of 3 different compilers, precompiled header information is 4-8 times the size of the preprocessed source

Size of typical contexts

- even simple files are likely to generate at least .5 MB
- typical applications: 1 - 4MB for each translation unit
  - » size is a function of the preprocessed declarative information (classes, templates, inline functions)
  - » small source files with lots of headers would still generate large context files

Multiple contexts in a single translation unit

- information is more complicated than a snapshot at a given point
  - » each instantiation has a different name binding point
  - » saved context needs to specify which names are visible, which types are complete/incomplete, using directives in effect, etc. for each instantiation or template definition

Optimizing information to be saved

- only possible if “project” system is used
  - » complete list of sources known up front
  - » template definitions processed before references
  - » mutual dependencies may make this impossible
  - » eliminates desired benefits of separate compilation (i.e., can’t arbitrarily combine object files)
  - » would not be standard conforming

John Spicer
Edison Design Group
jhs@edg.com

Optimization (continued)

- if a database is being used, you still need to make sure that all required information is in the database
- at best, optimization could reduce the number of places that generate duplicate contexts, not the amount of context information required

Using the context information

- read referencing context information
- read definition context information
- merge the two sets of information
- unique context for each instantiation
  - » each instantiation has a different referencing context
  - » each template has a different definition context

Instantiations caused by other instantiations

- the “referencing” context of the new instantiation is the merged context
- this could require saving synthesized contexts in addition to the user defined contexts

User problems with context merging

- instantiations take place in a synthesized context
- no single place a user can see the full context of an instantiation
- even worse for instantiations caused by other instantiations

More user problems with context merging

- errors dependent on which referencing context is chosen
- merging conflicts are a source of additional errors
  - » context merging is unspecified so it is difficult to know how severe this problem is
- errors delayed to link time, users would like them at compile time

Comparison with cfront instantiation model

- both generate instantiations at link time
- both do the instantiation in a context not under the control of the user
  - » cfront gets this wrong in some cases despite doing a *much* simpler context synthesis
- both require an expensive context synthesis for instantiations
- both defer errors until link time

John Spicer
Edison Design Group
jhs@edg.com

Expected cost of context merging

- how much time does it take to merge two .5 MB contexts?
- who knows? but...
  - » wc runs at about 2.5 MB / second
  - » compiling a file containing only comments runs at about 1 MB / second
- context merging is certainly more complicated than these operations

Expected cost of context merging (continued)

- several seconds for small contexts seems likely
  - » 2 seconds / instantiation = 10 minutes for 300 instantiations
- how does this compare with existing implementations?
  - » many can generate instantiations in .01 to .03 seconds (3 - 9 seconds for 300)
  - » a difference of two orders of magnitude

Effects on implementations

- forces instantiation at link time
- context merging makes this expensive
- template instantiation was already a very difficult problem
  - » need the freedom to provide the best solution for a given user community
  - » one instantiation model will not work for everyone

ABI issues

- context information is part of the information used to link one object file with another
  - » this makes it part of the ABI
  - » format of context information must be well specified for multiple compilers to interoperate on the same platform

ABI issues (continued)

- an issue even if you don't care about compatibility between compilers:
  - » needs to be a well specified form for release to release binary compatibility
  - » unlike PCH which can be specific to a compiler release
  - » increases overhead in creating and using the information
  - » most compact and stable form is probably just putting out the preprocessed source

Vendors are providing solutions that work for their users

- all existing compilers (that I'm aware of) include the template definitions at some point to generate instantiations
- the instantiation models used by existing compilers would not be possible with the current compilation model

John Spicer
Edison Design Group
jhs@edg.com

Proposed Alternatives

- simple - include template definitions wherever they are used
- more complex - separate compilation, but without context merging

Simple Alternative - typical objections

- too expensive
  - » template definitions must be compiled
  - » additional files needed by template definitions must also be included
- subjects template definitions to macros defined in the referencing program
- requires template source to be provided with libraries

Too expensive...

- scanning template definitions is inexpensive in most implementations
- very inexpensive compared to saving large volume of context information
- C++ is already header intensive -- there are well known techniques to optimize this (e.g., precompiled header files)

Subjects template definitions to macros

- already true of class templates and inline functions
- already true of existing implementations

Providing template source with libraries

- library vendors don't want to provide source to their template definitions
- really a separate issue:
  - » an implementation could choose to store template textually in the current model
  - » techniques exist to encrypt template source for existing implementations

Does **not** cause instantiations in every file

- difference between source model and instantiation model
- provides implementations with maximum freedom

John Spicer
Edison Design Group
jhs@edg.com

Existing practice

- existing compilers textually include the template definitions at some point
- most do so at compile time
  - » cfront does so at link time, but still uses textual inclusion of the template definitions

Definitional problems with the current model

- current model is unspecified in the WP
  - » motion from Valley Forge simply says:
    - – "A function template has external linkage"
    - – "A static member of a class template has external linkage"
  - » Chapter 3 already said that templates have external linkage
    - – this had been added simply to indicate that templates are subject to the ODR

Definitional problems with the current model (continued)

- the context merging process is unspecified
- template instantiation is not included in the description of the phases of translation (as would be necessary for link time instantiation)

What needs to be done

- decide whether to replace the current model
- if so, decide what to replace it with
- if not, we need a description of the current model