Explicit instantiation (C++ only)

You can explicitly tell the compiler when it should generate a definition from a template. This is called explicit instantiation. Explicit instantiation includes two forms: explicit instantiation declaration and explicit instantiation definition.

C++11

Note: IBM supports selected features of C++11, known as C++0x before its ratification. IBM will continue to develop and implement the features of this standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C++11 features is complete, including the support of a new C++11 standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++11 features.

Explicit instantiation declaration

The explicit instantiation declarations feature is introduced in the C++11 standard. With this feature, you can suppress the implicit instantiation of a template specialization or its members. The `extern` keyword is used to indicate explicit instantiation declaration. The usage of `extern` here is different from that of a storage class specifier.

```
Explicit instantiation declaration syntax


>>-extern--template--template_declaration----------------------><
```

You can provide an explicit instantiation declaration for a template specialization if an explicit instantiation definition of the template exists in other translation units or later in the same file. If one translation unit contains the explicit instantiation definition, other translation units can use the specialization without having the specialization instantiated multiple times. The following example demonstrates this concept:

```
//sample1.h:

template<typename T, T val>

union A{

  T func();

};

extern template union A<int, 55>;


template<class T, T val>

T A<T,val>::func(void){

  return val;

}


//sampleA.C"

#include "sample1.h"


template union A<int,55>;
```

```
//sampleB.C:
#include "sample1.h"

int main(void){
  return A<int, 55>().func();
}
```

`sampleB.C` uses the explicit instantiation definition of `A<int, 55>().func()`in `sampleA.C`.

If an explicit instantiation declaration of a function or class is declared, but there is no corresponding explicit instantiation definition anywhere in the program, the compiler issues an error message. See the following example:

```
// sample2.C
template <typename T, T val>
struct A{
  virtual T func();
  virtual T bar();
}

extern template int A<int,55>::func();

template <class T, T val>
T A<T,val>::func(void){
  return val;
}

template <class T, T val>
T A<T,val>::bar(void){
  return val;
}

int main(void){
  return A<int,55>().bar();
}
```

When you use explicit instantiation declaration, pay attention to the following restrictions:

- You can name a static class member in an explicit instantiation declaration, but you cannot name a static function because a static function cannot be accessed by name in other translation units.
- The explicit instantiation declaration of a class is not equivalent to the explicit instantiation declaration of each of its members.

C++11

Explicit instantiation definition

An explicit instantiation definition is an instantiation of a template specialization or its members.

Explicit instantiation definition syntax

```
>>-template--template_declaration-----------------------------><
```

Here is an example of explicit instantiation definition:

```
template<class T> class Array { void mf(); };
template class Array<char>;       /* explicit instantiation definition */
template void Array<int>::mf();   /* explicit instantiation definition */

template<class T> void sort(Array<T>& v) { }
template void sort(Array<char>&); /* explicit instantiation definition */

namespace N {
  template<class T> void f(T&) { }
}

template void N::f<int>(int&);
// The explicit instantiation definition is in namespace N.

int* p = 0;
template<class T> T g(T = &p);
template char g(char);           /* explicit instantiation definition */

template <class T> class X {
  private:
     T v(T arg) { return arg; };
};

template int X<int>::v(int);    /* explicit instantiation definition */

template<class T> T g(T val) { return val;}
template<class T> void Array<T>::mf() { }
```

An explicit instantiation definition of a template is in the same namespace where you define the template.
Access checking rules do not apply to the arguments in the explicit instantiation definitions. Template arguments in an explicit instantiation definition can be private types or objects. In this example, you can use the explicit instantiation definition `template int X<int>::v(int)` even though the member function is declared to be private.
The compiler does not use default arguments when you explicitly instantiate a template. In this example, you can use the explicit instantiation definition `template char g(char)` even though the default argument is an address of the type `int`.
Note: You cannot use the `inline` or C++11 `constexpr` C++11 specifier in an explicit instantiation of a function template or a member function of a class template.

C++11

Explicit instantiation and inline namespace definitions

Inline namespace definitions are namespace definitions with an initial `inline` keyword. Members of an inline namespace can be explicitly instantiated or specialized as if they were also members of the enclosing namespace. For more information, see Inline namespace definitions (C++11).

Parent topic: Template instantiation (C++ only)

Related reference:
C++11 compatibility