Class template

A class template defines a family of classes.

Syntax

template < *parameter-list* **>** *class-declaration*	(1)	
export template < *parameter-list* **>** *class-declaration*	(2)	(until C++11)

Explanation

class-declaration - a class declaration. The class name declared becomes a template name.

parameter-list - a non-empty comma-separated list of the template parameters, each of which is either a non-type parameter, a type parameter, a template parameter, or a parameter pack of any of those.

`export` was an optional modifier which declared the template as *exported* (when used with a class template, it declared all of its members exported as well). Files that instantiated exported templates did not need to include their definitions: the declaration was sufficient. Implementations of `export` were rare and disagreed with each other on details. (until C++11)

Class template instantiation

A class template by itself is not a type, or an object, or any other entity. No code is generated from a source file that contains only template definitions. In order for any code to appear, a template must be instantiated: the template arguments must be provided so that the compiler can generate an actual class (or function, from a function template).

Explicit instantiation

template class|struct *template-name* **<** *argument-list* **> ;**	(1)	
extern template class|struct *template-name* **<** *argument-list* **> ;**	(2)	(since C++11)

1) Explicit instantiation definition

2) Explicit instantiation declaration

An explicit instantiation definition forces instantiation of the class, struct, or union they refer to. It may appear in the program anywhere after the template definition, and for a given argument-list, is only allowed to appear once in the entire program.

An explicit instantiation declaration (an extern template) skips implicit instantiation step: the code that would otherwise cause an implicit instantiation instead uses the explicit instantiation definition provided elsewhere (resulting in link errors if no such instantiation exists). This can be used to reduce compilation times by explicitly declaring a template instantiation in all but one of the source files using it, and explicitly defining it in the remaining file. (since C++11)

Classes, functions, variables, and member template specializations can be explicitly instantiated from their templates. Member functions, member classes, and static data members of class templates can be explicitly instantiated from their member definitions.

Explicit instantiation can only appear in the enclosing namespace of the template, unless it uses qualified-id:

```
namespace N {
  template<class T> class Y { void mf() { } }; // template definition
}
// template class Y<int>; // error: class template Y not visible in the global namespace
using N::Y;
// template class Y<int>; // error: explicit instantiation outside
                          // of the namespace of the template
template class N::Y<char*>;      // OK: explicit instantiation
template void N::Y<double>::mf(); // OK: explicit instantiation
```

Explicit instantiation has no effect if an explicit specialization appeared before for the same set of template arguments.

Only the declaration is required to be visible when explicitly instantiating a function template, a variable template, a member function or static data member of a class template, or a member function template. The complete definition must appear before the explicit instantiation of a class template, a member class of a class template, or a member class template, unless an explicit specialization with the same template arguments appeared before.

If a function template, variable template, member function template, or member function or static data member of a class template is explicitly instantiated with an explicit instantiation definition, the template definition must be present in the same translation unit.

When an explicit instantiation names a class template specialization, it serves as an explicit instantiation of the same kind (declaration or definition) of each of its non-inherited non-template members that has not been previously explicitly specialized in the translation unit. If this explicit instantiation is a definition, it is also an explicit instantiation definition only for the members that have been defined at this point.

Explicit instantiation definitions ignore member access specifiers: parameter types and return types may be private.

Implicit instantiation

When code refers to a template in context that requires a completely defined type, or when the completeness of the type affects the code, and this particular type has not been explicitly instantiated, implicit instantiation occurs. For example, when an object of this type is constructed, but not when a pointer to this type is constructed.

This applies to the members of the class template: unless the member is used in the program, it is not instantiated, and does not require a definition.

```
template<class T> struct Z {
    void f() {}
    void g(); // never defined
}; // template definition
template struct Z<double>; // explicit instantiation of Z<double>
Z<int> a; // implicit instantiation of Z<int>
Z<char>* p; // nothing is instantiated here
p->f(); // implicit instantiation of Z<char> and Z<char>::f() occurs here.
// Z<char>::g() is never needed and never instantiated: it does not have to be defined
```

If a class template has been declared, but not defined, at the point of instantiation, the instantiation yields an incomplete class type:

```
template<class T> class X; // declaration, not definition
X<char> ch;                // error: incomplete type X<char>
```

Local classes and any templates used in their members are instantiated as part of the instantiation of the entity within which the local class or enumeration is declared. (since C++17)

See also

- template parameters and arguments allow templates to be parametrized
- function template declaration declares a function template
- template specialization defines an existing template for a specific type
- parameter packs allows the use of lists of types in templates (since C++11)
