Date: 2016-03-17
Document number: P0309R0
Audience: Evolution Working Group
Authors: Daniele Bordes, Markus Hosch
Reply to: daniele.bordes@bmw.de , daniele.bordes@gmail.com , markus.hosch@bmw.de

Partial class

Abstract

The users of a class must be recompiled whenever the private part of the used class is changed, even if not accessible. These compile dependencies are useless if the users do not create instances of the used class, but the only way to remove them while keeping a clean object-oriented style of programming is to use solutions that require additional memory costs and therefore do not fit embedded software.
This "partial class" proposal, a kind of “augmented” forward declaration, keeps the clean customary object-oriented style of programming and removes the recompilation problem with no additional memory need.
In the following paragraphs, the proposal is described in details and an example is provided; compile-time improvements, memory need improvements and interactions with other new proposals are considered. An eventual syntax simplification is also suggested in the last paragraph.

Motivation

The main purpose of the public and the private parts of a C++ class is to separate the interface of that class from its internal representation (a part of its implementation); unfortunately, the private part of a class must be declared in the same header file where the public part is also declared and therefore included by all the users of that class: this means that whenever the private part of a class is changed, as a consequence all the users of that class need to be recompiled.
The point is that, for all the users that just use the class through a reference and its public interface but do not instantiate (or copy) the class, these recompilations are actually unnecessary, since the compiler does not need to know the size of the class when normal public members other than the constructors are invoked.
This is perfectly clarified by Bjarne Stroustrup himself in his “The C++ Programming Language, 3d. Edition” (§2.5.4) when describing the class construct:

“(…) the representation is not decoupled from the user interface; rather, it is a part of what would be included in a program fragment using Stacks. The representation is private, and therefore accessible only through the member functions, but it is present. If it changes in any significant way, a user must recompile. This is the price to pay for having concrete types behave exactly like built-in types (…)”

This problem is particularly tedious when the private part of the class has some members which are instances of other classes: in this case the headers where those classes are declared must be included, and the same applies indirectly for other headers needed by those classes, possibly resulting in a long chain of inclusions and therefore compile dependencies.
To avoid these annoying and unnecessary recompilations while keeping at the same time a clean object-oriented style of programming, basically two solutions are at present possible:

1) Using polymorphic types (the solution proposed by Stroustrup himself in the paragraph of his book referred above)
2) Using idioms like Pimpl or Handle/Body (a “pattern-like” solution)

Unfortunately both solutions (1) and (2) **require unneeded additional costs in time and space, due to an additional pointer for each object and its necessary additional dereferencing: the pointer needed for each object by solution (1) is the “hidden” pointer to the virtual table, the pointer needed for each object by solution (2) is the pointer to the “Body” class. These costs could be unaffordable for embedded systems with tight cpu and memory resources**, in particular the space cost, that could be significant if the class is small but has many instances.
Basically it means that C++ embedded software developers, just because C++ does not offer a real separation of interface from implementation, have often to choose one among these two drawbacks:

- Accept the additional cpu and memory costs to reduce the compile time
- Accept unnecessary longer compile times to avoid the additional cpu and memory costs

In both cases, the C++ developers must “pay for something they do *not* need”.

Solution:

A new "partial class" construct would offer a possibility of a real separation of interface from implementation. With such a construct, **the declaration of a class could be split in two header files**: one header file ("interface header") would contain the declaration of the interface (public methods) while another header file ("internal details header") would contain the declaration of the implementation (private members, private helper methods, constructors, destructor).
All the users instantiating instances of the class would still need to include both; instead, all the users that do not instantiate objects of that class but just use the public interface of the class through references or pointers would need to include only the "interface header", and would not need to be recompiled when the implementation of the class is changed.
No additional cpu or memory costs at runtime are required; besides, this solution would not lead to a "header-files" pollution, since the solutions 1) and 2) mentioned above would require an additional header file as well.
Let's give an example.

Example:

Let's suppose we want to realize a class "Stack", a stack of at most 20 integers. With the current C++ standard, we can have only one header file. The declaration of the header file and the include dependencies would look like this:

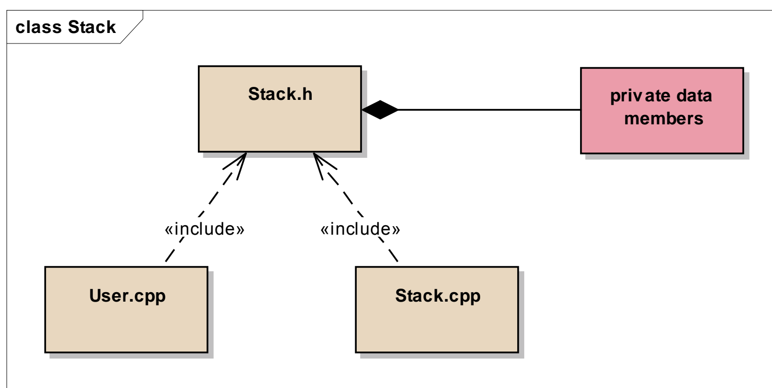
```
// Stack.h
class Stack
{
private:

    // private members
   static const int MAX_SIZE = 20;
   int stack[MAX_SIZE];
   int top;

   // private helper methods
   int remainingSpace();
public:

    // constructors/destructor
   Stack();
    ˜Stack();

   // ordinary public methods:
   void push(int c);
   int pop();
   bool isFull();
   bool isEmpty();
};
```



Traditional include dependencies

As previously stated, each change into the private part of this class (private members or private helper method declarations) would require to recompile all the users of this class.
To avoid this, let's use the new construct "partial class" and split the declaration of Stack, previously located only in the "traditional" header file "Stack.h", in two different parts located into two new header files: "StackInterface.h" and "StackInternals.h".
From now on we will refer the part declared in "StackInterface.h" as "partial class" and the part declared in "StackInternals.h" as "internal class". In details:

- The "StackInterface.h" file will host the "partial class" (new keyword "**partial class**"), that is, the declaration of the public methods exported by class "Stack". This can be thought as something similar to a forward declaration of the class "Stack", with the difference that also some public methods are additionally declared. The "partial class" will have also to specify a name for itself (keyword "**export**" followed by the name): the reason for this will be clear in a moment.
- The "StackInternals.h" file will host the "internal class" (keyword "**class**"), that is, the declaration of the data members and eventually the private methods of the class "Stack". The "internal class" must specify that it will implement the methods declared in the "partial class": the syntax for this resembles a public derivation, but makes use of the new keyword "**partial**" instead of "public". This works as implicit declaration of the methods already declared in the "partial class", therefore those methods do not need to be declared here in the "internal class" again. Obviously the "StackInterface.h" file must be included.

The purpose for the name “StackInterface” is only to establish a “contract” between the two parts and must be referenced by both of them. This is done for the following reasons:

- Saving declarations: the “internal class” specifies that it has “moved” the declaration of its public methods in the named “partial class” and therefore **does not need to declare those public methods again**.
- Preventing hijacking: as the “partial class” also must have a name and at the same time must specify that it is exporting methods belonging to the named “internal class”, the “internal class” has the ability to control which contracts it wants to implement; this **prevents “hijacking”** by third parties who otherwise would be able to declare another, unwanted “partial class” without having to name the “internal class” and thus gaining access to the private members without touching the declaration of the “internal class” itself.

The identifier “StackInterface” cannot be used for other purposes, in particular, it is not a new type: the type is only “Stack”, which in particular will be used:

- by the designer of class Stack as identifier for the method definitions
- by the user of class Stack as type for pointer(s) or reference(s)
- by the compiler for the name mangling of the translated methods.

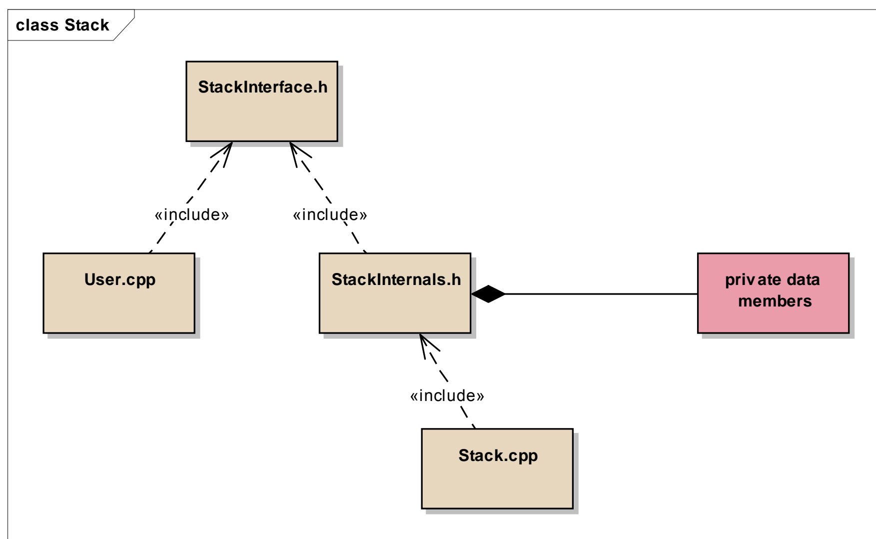
Below an example of the new header files, with the new include dependencies:

```
// StackInterface.h
partial class Stack export StackInterface
{
public:
    // ordinary public methods:
   void push(int c);
   int pop();
   bool isFull();
   bool isEmpty();
};


// StackInternals.h
#include “StackInterface.h”

class Stack : partial StackInterface
{
private:
    // private members
   static const int MAX_SIZE = 20;
   int stack[MAX_SIZE];
   int top;

   // private helper methods
   int remainingSpace();
public:
   // constructors/destructor
   Stack();
   ~Stack();
};
```



Include dependencies with the new “partial class” construct

All the users of class Stack that do not have to create instances of it would now need to include only the “StackInterface.h” file: therefore all changes into StackInternals.h would be transparent for such users.
For example:

```
// User.cpp
#include “StackInterface.h”

void useStack(Stack& stck)
{
    if(!stck.isFull())
    {
        stck.push(3);
    }
}
```

Obviously users of class Stack that have to create instances of it would need to include also “StackInternals.h”.

Interaction with other proposals

Currently no other proposals interfere with this one or give the same benefits. The newly introduced "***Modules***" address also compile improvements, but require to recompile users of a module whenever a private part of a used class is changed, as perfectly stated in the following comment written in the thread where at first this proposal was sent:

*"(...) modules will (...) mitigate the recompilation costs (...). You will only need to recompile code that imports your module if you change the actual declarations within the class (public and **private**).*

The recompilation of the code after changes to the declaration of the private part of the used class, still needed by “Modules”, is **exactly what this proposal eliminates**.

This proposal is also orthogonal to the “**unified call syntax**” one; with this proposal, programmers can use the method-invocation syntax with no drawbacks, while with the “unified call syntax” they can then eventually choose the functional notation to invoke member functions if they prefer it.

Compile-time and memory improvements

All changes into the private part would be completely transparent for all the users that include only the partial class header, and therefore all such users would not be recompiled at all. The same apply in case of changes in those header files included only by the header file of the "internal" class containing the private part. The compile time improvement can be therefore considerable if there are many users which need to include only the partial class, that is, use a concrete-type class without instantiating objects of it; for instance, in embedded software, where concrete-type classes are heavily used, this is a common case.

As for memory consumption, no additional (explicit or implicit) pointers are needed, as on the contrary by the inheritance and handle/body pattern mentioned above.

Compiler-issues and other considerations

This proposal allows ultimately just a new kind of forward declaration, that is, the forward declaration of the non-virtual methods (except constructors and destructors) of a concrete-type class. As such, it should have a minimal impact on the compiler. The following considerations and constraints shall be taken into account:

1. Given a class, many “partial classes” of it may exist with different names, distributed in many header files, exporting for instance different kind of interfaces for that class:

   ```
   partial class Stack : export StackInterface1 { /* … */ }
   partial class Stack : export StackInterface2 { /* … */ }
   partial class Stack : export StackInterface3 { /* … */ }
   ```

   The “internal class” will just have to include those headers and specify that will implement those methods, for instance:

   ```
   class Stack : partial StackInterface1, partial StackInterface2 , partial StackInterface3 { /* … */ }
   ```

2. To avoid ambiguities, identifier like “StackInterface” used in the example cannot be used for purposes other than the two described throughout the document, that is giving a name to the partial classes and specify that the internal class implements the methods declared in its partial classes. For example:

   ```
   // Stack.cpp
   void Stack::push(int c) { /* … */ }                // ok
   void StackInterface::push(int c) { /* … */ }       // compile error
   // User.cpp
   Stack* ptr;                 // ok
   StackInterface* ptr;        // compile error
   ```

As already stated, the compiler will anyhow use univocally **the identifier "Stack" of the internal class** for the name mangling of the translated methods, independently on which partial class was used to declare the methods. For example, the name mangling for the method "push" by an Intel compiler would be anyhow:

_Z5**Stack**4pushEi

3. A "partial class" must not influence the memory layout of an object of the "internal class", so it:
   - cannot specify any derivations from other classes (only the "internal class" may do it)
   - cannot declare data members
   - cannot have virtual methods
   - cannot declare constructors nor the destructor
   - may contain declaration of classes or enums
   - may declare static const plain data types
   - may declare operators
   - may declare template methods

4. For the definitions of the methods of Stack (Stack.cpp), the inclusion of the partial class where they are declared is not enough: also the correct declaration of the internal class "Stack" must be included, otherwise they must be rejected by the compiler

5. By parsing the "partial" header file (StackInterface.h), the declaration of the partial class shall be considered as a forward declaration of the class Stack plus a forward declaration of some non-virtual methods of class Stack itself

   As such, the following things shall be rejected with a compile error:

   - declarations of constructors or destructors
   - declarations of data members
   - declarations of virtual methods
   - declarations of the partial class as derived class from another class

6. By parsing the "internal" header file (StackInternals.h), it must be checked:

   - if the partial class declaration is also included
   - if the internal class itself and the partial class declared in the included "StackInternals.h" refer each other correctly:

     // partial class
     partial class **Stack** export **StackInterface**

     // internal class
     class **Stack** partial **StackInterface**

7. By compiling the implementation file Stack.cpp, the compiler shall apply the same rules as in case the methods declared in the partial class would have been declared in the single header Stack.h as in the current C++ implementation; by translating those methods, it would still apply the same rules, that is, using "Stack" in the name mangling.

8. By compiling a user file of the partial class, it would translate a method invocation by using "Stack" in the name mangling. An invocation to a constructor would be rejected with a compile error.

Eventual syntax simplification:

Being the keyword “export” already existing, the only impact of this proposal on the language syntax would be the introduction of the new keyword “partial”, which was thought in harmony with the concept expressed. This impact can eventually be removed simplifying the syntax by getting rid of this new keyword and recurring to the already existing “public” keyword by the declaration of the “internal class” as follows:

```
// StackInterface.h
// class Stack export StackInterface
{
(…)
}
```

```
// StackInternals.h
// class Stack : public StackInterface
{
(…)
}
```