Why can templates only be implemented in the header file?

Asked 15 years, 2 months ago Modified 7 months ago Viewed 794k times

2280

Quote from The C++ standard library: a tutorial and handbook:

> The only portable way of using templates at the moment is to implement them in header files by using inline functions.

Why is this?

(Clarification: header files are not the *only* portable solution. But they are the most convenient portable solution.)

c++ templates undefined-reference c++-faq

Share Improve this question Follow

edited Dec 19, 2020 at 0:09
πάντα ῥεῖ
1

asked Jan 30, 2009 at 10:06
MainID
29.6k ● 19 ● 58 ● 70

30 While it is true that placing all template function definitions into the header file is probably the most convenient way to use them, it is still not clear what's "inline" doing in that quote. There's no need to use inline functions for that. "Inline" has absolutely nothing to do with this. – AnT stands with Russia Sep 18, 2014 at 4:11

17 A template is not like a function which can be compiled into byte code. It is just a pattern to generate such a function. If you put a template on its own into a *.cpp file, there is nothing to compile. Moreover, the explicite instanciation is actually not a template, but the starting point to make a function out of the template which ends up in the *.obj file. – dgrat May 28, 2019 at 9:02

5 @AnT perhaps they meant "inline" not as the keyword but rather as "methods implemented at the place of declaration, inside the class". – Vorac Feb 13, 2022 at 23:12

@AnTstandswithRussia Template functions are implicitly inline – palapapa Apr 22, 2023 at 11:26

1 @palapapa: That sounds like another question (or maybe two, if it's not “When does this matter?” but “What does this mean?”). – Davis Herring Jul 21, 2023 at 14:40

19 Answers

Sorted by: Highest score (default)

1991

Caveat: It is *not* necessary to put the implementation in the header file, see the alternative solution at the end of this answer.

Anyway, the reason your code is failing is that, when instantiating a template, the compiler creates a new class with the given template argument. For example:

```
template<typename T>
struct Foo
{
    T bar;
    void doSomething(T param) {/* do stuff using T */}
};

// somewhere in a .cpp
Foo<int> f;
```

When reading this line, the compiler will create a new class (let's call it `FooInt`), which is equivalent to the following:

```
struct FooInt
{
    int bar;
    void doSomething(int param) {/* do stuff using int */}
};
```

Consequently, the compiler needs to have access to the implementation of the methods, to instantiate them with the template argument (in this case `int`). If these implementations were not in the header, they wouldn't be accessible, and therefore the compiler wouldn't be able to instantiate the template.

A common solution to this is to write the template declaration in a header file, then implement the class in an implementation file (for example .tpp), and include this implementation file at the end of the header.

Foo.h

```
template <typename T>
struct Foo
{
    void doSomething(T param);
};

#include "Foo.tpp"
```

Foo.tpp

```
template <typename T>
void Foo<T>::doSomething(T param)
{
    //implementation
}
```

This way, implementation is still separated from declaration, but is accessible to the compiler.

Alternative solution

Another solution is to keep the implementation separated, and explicitly instantiate all the template instances you'll need:

Foo.h

```
// no implementation
template <typename T> struct Foo { ... };
```

Foo.cpp

```
// implementation of Foo's methods

// explicit instantiations
template class Foo<int>;
template class Foo<float>;
// You will only be able to use Foo with int or float
```

If my explanation isn't clear enough, you can have a look at the C++ Super-FAQ on this subject.

Share Improve this answer Follow

edited Aug 25, 2023 at 21:47
Jan Schultke

answered Jan 30, 2009 at 10:26
Luc Touraille

29.4k ● 8 ● 72 ● 128 81.1k ● 15 ● 96 ● 137

123 Actually the explicit instantiation needs to be in a .cpp file which has access to the definitions for all of Foo's member functions, rather than in the header. – Mankarse May 28, 2011 at 10:21

20 "the compiler needs to have access to the implementation of the methods, to instantiate them with the template argument (in this case int). If these implementations were not in the header, they wouldn't be accessible" But why is an implementation in the .cpp file not accessible to the compiler? A compiler can also access .cpp information, how else would it turn them into .obj files? EDIT: answer to this question is in the link provided in this answer... – xcrypt Jan 14, 2012 at 23:56

46 I don't think this explains the question that clearly, the key thing is obviously related with the compilation UNIT which is not mentioned in this post – zinking Aug 23, 2012 at 9:47

7 @Gabson: structs and classes are equivalent with the exception that the default access modifier for classes is "private", while it is public for structs. There are some other tiny differences that you can learn by looking at this question. – Luc Touraille Feb 21, 2014 at 14:12

5 I've added a sentence at the very start of this answer to clarify that the question is based on a false premise. If somebody asks "Why is X true?" when in fact X is not true, we should quickly reject that assumption. – Aaron McDaid Aug 5, 2015 at 9:39

352

It's because of the requirement for separate compilation and because templates are instantiation-style polymorphism.

Lets get a little closer to concrete for an explanation. Say I've got the following files:

- foo.h
  - declares the interface of `class MyClass<T>`
- foo.cpp
  - defines the implementation of `class MyClass<T>`
- bar.cpp
  - uses `MyClass<int>`

Separate compilation means I should be able to compile **foo.cpp** independently from **bar.cpp**. The compiler does all the hard work of analysis, optimization, and code generation on each compilation unit completely independently; we don't need to do whole-program analysis. It's only the linker that needs to handle the entire program at once, and the linker's job is substantially easier.

bar.cpp doesn't even need to exist when I compile **foo.cpp**, but I should still be able to link the **foo.o** I already had together with the **bar.o** I've only just produced, without needing to recompile **foo.cpp**. **foo.cpp** could even be compiled into a dynamic library, distributed somewhere else without **foo.cpp**, and linked with code they write years after I wrote **foo.cpp**.

"Instantiation-style polymorphism" means that the template `MyClass<T>` isn't really a generic class that can be compiled to code that can work for any value of `T`. That would add overhead such as boxing, needing to pass function pointers to allocators and constructors, etc. The intention of C++ templates is to avoid having to write nearly identical `class MyClass_int`, `class MyClass_float`, etc, but to still be able to end up with compiled code that is mostly as if we *had* written each version separately. So a template is *literally* a template; a class template is *not* a class, it's a recipe for creating a new class for each `T` we encounter. A template cannot be compiled into code, only the result of instantiating the template can be compiled.

So when **foo.cpp** is compiled, the compiler can't see **bar.cpp** to know that `MyClass<int>` is needed. It can see the template `MyClass<T>`, but it can't emit code for that (it's a template, not a class). And when **bar.cpp** is compiled, the compiler can see that it needs to create a `MyClass<int>`, but it can't see the template `MyClass<T>` (only its interface in **foo.h**) so it can't create it.

If **foo.cpp** itself uses `MyClass<int>`, then code for that will be generated while compiling **foo.cpp**, so when **bar.o** is linked to **foo.o** they can be hooked up and will work. We can use that fact to allow a finite set of template instantiations to be implemented in a .cpp file by writing a single template. But there's no way for **bar.cpp** to use the template *as a template* and instantiate it on whatever types it likes; it can only use pre-existing versions of the templated class that the author of **foo.cpp** thought to provide.

You might think that when compiling a template the compiler should "generate all versions", with the ones that are never used being filtered out during linking. Aside from the huge overhead and the extreme difficulties such an approach would face because "type modifier" features like pointers and arrays allow even just the built-in types to give rise to an infinite number of types, what happens when I now extend my program by adding:

- baz.cpp
  - declares and implements `class BazPrivate`, and uses `MyClass<BazPrivate>`

There is no possible way that this could work unless we either

1. Have to recompile **foo.cpp** every time we change *any other file in the program*, in case it added a new novel instantiation of `MyClass<T>`
2. Require that **baz.cpp** contains (possibly via header includes) the full template of `MyClass<T>`, so that the compiler can generate `MyClass<BazPrivate>` during compilation of **baz.cpp**.

Nobody likes (1), because whole-program-analysis compilation systems take *forever* to compile , and because it makes it impossible to distribute compiled libraries without the source code. So we have (2) instead.

Share Improve this answer Follow

edited May 11, 2013 at 11:26

answered May 11, 2013 at 3:54
Ben
69.9k ●21 ●129 ●179

86 emphasized quote **a template is literally a template; a class template is not a class, it's a recipe for creating a new class for each T we encounter** – v.oddou Apr 25, 2016 at 9:57

1 @Birger You should be able to do it from any file that has access to the full template implementation (either because it's in the same file or via header includes). – Ben Mar 9, 2017 at 11:02

22 @ajeh It's not rhetoric. The question is "why do you have to implement templates in a header?", so I explained the technical choices the C++ language makes that lead to this requirement. Before I wrote my answer others already provided workarounds that are not full solutions, because there *can't* be a full solution. I felt those answers would be complemented by a fuller discussion of the "why" angle of the question. – Ben Apr 2, 2018 at 22:02

1 imagine it this way folks... if you weren't using templates (to efficiently code what you needed), you'd only be offering a few versions of that class anyway. so you have 3 options. 1). don't use templates. (like all other classes/functions, nobody cares that others can't alter the types) 2). use templates, and document which types they can use. 3). give them the whole implementation (source) bonus 4). give them the whole source in case they want to make a template from another one of your classes ;) – Puddle Nov 27, 2018 at 0:01

3 @VoB Yes, a `.tpp` file in that sense is just a naming convention for a kind of header file. A "header file" isn't something specific to the C++ compiler, it's just what we call a file that we intend to include into other compilation units by using `#include`. If it helps you work with your code to have template implementations in a separate file from the one that describes the interface of a .cpp file, and to give those template implementation files a specific extension like `.tpp`, then go for it! The compiler doesn't know or care about the difference, but it can help humans. – Ben Jan 23, 2021 at 0:03

276

Plenty correct answers here, but I wanted to add this (for completeness):

If you, at the bottom of the implementation cpp file, do explicit instantiation of all the types the template will be used with, the linker will be able to find them as usual.

Edit: Adding example of explicit template instantiation. Used after the template has been defined, and all member functions has been defined.

```
template class vector<int>;
```

This will instantiate (and thus make available to the linker) the class and all its member functions (only). Similar syntax works for function templates, so if you have non-member operator overloads you may need to do the same for those.

The above example is fairly useless since vector is fully defined in headers, except when a common include file (precompiled header?) uses `extern template class vector<int>` so as to keep it from instantiating it in all the *other* (1000?) files that use vector.

Share Improve this answer Follow

edited Aug 31, 2021 at 10:31
Ted Lyngmo
106k ● 7 ● 71 ● 116

answered Aug 13, 2009 at 13:49
MaHuJa
3,158 ● 1 ● 18 ● 6

81 Ugh. Good answer, but no real clean solution. Listing out all possible types for a template does not seem to go with what a template is supposed to be. – Jiminion Jul 17, 2014 at 17:49

11 This can be good in many cases but generally breaks the purpose of template which is meant to allow you to use the class with any `type` without manually listing them. – Tomáš Zato Dec 9, 2014 at 3:27

12 `vector` is not a good example because a container is inherently targeting "all" types. But it does happen very frequently that you create templates that are only meant for a specific set of types, for instance numeric types: int8_t, int16_t, int32_t, uint8_t, uint16_t, etc. In this case, it still makes sense to use a template, but explicitly instantiating them for the whole set of types is also possible and, in my opinion, recommended. – UncleZeiv Jun 3, 2015 at 15:41

Used after the template has been defined, "and all member functions has been defined". Thanks ! – Vitt Volt Feb 16, 2017 at 6:04

1 I feel like I'm missing something … I put the explicit instantiation for two types into the class's `.cpp` file and the two instantiations are referred to from other `.cpp` files, and I still get the linking error that the members are not found. – oarfish Sep 4, 2019 at 14:38

95

Templates need to be *instantiated* by the compiler before actually compiling them into object code. This instantiation can only be achieved if the template arguments are known. Now imagine a scenario where a template function is declared in `a.h`, defined in `a.cpp` and used in `b.cpp`. When `a.cpp` is compiled, it is not necessarily known that the upcoming compilation `b.cpp` will require an instance of the template, let alone which specific instance would that be. For more header and source files, the situation can quickly get more complicated.

One can argue that compilers can be made smarter to "look ahead" for all uses of the template, but I'm sure that it wouldn't be difficult to create recursive or otherwise complicated scenarios. AFAIK, compilers don't do such look aheads. As Anton pointed out, some compilers support explicit export declarations of template instantiations, but not all compilers support it (yet?).

Share Improve this answer Follow

edited Aug 4, 2016 at 7:41
K DawG
13.5k ● 9 ● 35 ● 66

answered Jan 30, 2009 at 10:23
David Hanak
10.8k ● 3 ● 32 ● 39

1 "export" is standard, but it's just hard to implement so most of the compiler teams just haven't done yet. – vava Jan 30, 2009 at 10:27

6 export doesn't eliminate the need for source disclosure, nor does it reduce compile dependencies, while it requires a massive effort from compiler builders. So Herb Sutter himself asked compiler builders to 'forget about' export. As the time investment needed would be better spend elsewhere... – Pieter Jan 30, 2009 at 15:13

2 So I don't think export isn't implemented 'yet'. It'll probably never get done by anyone else than EDG after the others saw how long it took, and how little was gained – Pieter Jan 30, 2009 at 15:14

3 If that interests you, the paper is called "Why we can't afford export", it's listed on his blog (gotw.ca/publications) but no pdf there (a quick google should turn it up though) – Pieter Jan 30, 2009 at 15:22

1 Ok, thanks for good example and explanation. Here is my question though: why compiler cannot figure out where template is called, and compile those files first before compiling definition file? I can imagine it can be done in a simple case... Is the answer that interdependencies will mess up the order pretty fast? – Vlad Oct 11, 2013 at 17:54

76

Actually, prior to C++11 the standard defined the `export` keyword that *would* make it possible to declare templates in a header file and implement them elsewhere. In a manner of speaking. Not really, as the only ones who ever *implemented* that feature pointed out:

> Phantom advantage #1: Hiding source code. Many users, have said that they expect that by using export they will no longer have to ship definitions for member/nonmember function templates and member functions of class templates. This is not true. With export, library writers still have to ship full template source code or its direct equivalent (e.g., a system-specific parse tree) because the full information is required for instantiation. [...]

> Phantom advantage #2: Fast builds, reduced dependencies. Many users expect that export will allow true separate compilation of templates to object code which they expect would allow faster builds. It doesn't because the compilation of exported templates is indeed separate but not to object code. Instead, export almost always makes builds slower, because at least the same amount of compilation work must still be done at prelink time. Export does not even reduce dependencies between template definitions because the dependencies are intrinsic, independent of file organization.

None of the popular compilers implemented this keyword. The only implementation of the feature was in the frontend written by the Edison Design Group, which is used by the Comeau C++ compiler. All others required you to write templates in header files, because the compiler needs the template definition for proper instantiation (as others pointed out already).

As a result, the ISO C++ standard committee decided to remove the `export` feature of templates with C++11.

Share Improve this answer Follow

edited Jun 30, 2022 at 14:04

answered Jan 30, 2009 at 13:38
DevSolar
68.9k ● 21 ● 135 ● 214

9 ...and a couple of years later, I **finally** understood what `export` would actually have *given* us, and what not... and now I wholeheartedly agree with the EDG people: It would not have brought us what most people (myself in '11 included) *think* it would, and the C++ standard is better off without it. – DevSolar Nov 19, 2015 at 10:27

6 @DevSolar : this paper is political, repetitive and badly written. that's not usual standard level prose there. Uneedingly long and boring, saying basically 3 times the same things accross tens of pages. But I am now informed that export is not export. That's a good intel ! – v.oddou Apr 25, 2016 at 9:51

1 @v.oddou: Good developer and good technical writer are two seperate skillsets. Some can do both, many can't. ;-) – DevSolar Apr 25, 2016 at 9:58

1 @v.oddou The paper isn't just badly written, it's disinformation. Also it's a spin on reality: what are actually extremely strong arguments for exports are mixed in a way to make it sound like they are against export: "discovering numerous ODRrelated holes in the standard in the presence of export. Before export, ODR violations didn't have to be diagnosed by the compiler. Now it's necessary because you need to combine internal data structures from different translation units, and you can't combine them if they're actually representing different things, so you need to do the checking." – curiousguy Dec 14, 2019 at 1:20

1 @DevSolar I still don't see a case made against export in the paper. (I see a case *for* export.) – curiousguy Dec 19, 2019 at 4:11

42

Although standard C++ has no such requirement, some compilers require that all function and class templates need to be made available in every translation unit they are used. In effect, for those compilers, the bodies of template functions must be made available in a header file. To repeat: that means those compilers won't allow them to be defined in non-header files such as .cpp files

There is an *export* keyword which is supposed to mitigate this problem, but it's nowhere close to being portable.

Share Improve this answer Follow

edited Sep 19, 2013 at 17:16
Jonny Henly
4,163 ● 4 ● 27 ● 44

answered Jan 30, 2009 at 10:15
Anton Gogolev
115k ● 39 ● 202 ● 290

Why can't I implement them in .cpp file with the keyword "inline"? – MainID Jan 30, 2009 at 10:20

3 You can, and you don't have to put "inline" even. But you'd be able to use them just in that cpp file and nowhere else. – vava Jan 30, 2009 at 10:28

13 This is almost the most *accurate* answer, except "that means those compilers won't allow them to be defined in non-header files such as .cpp files" is patently false. – Lightness Races in Orbit Aug 14, 2011 at 17:59

37

Templates are often used in headers because the compiler needs to instantiate different versions of the code, depending on the parameters given/deduced for template parameters, and it's easier (as a programmer) to let the compiler recompile the same code multiple times and deduplicate later. Remember that a template doesn't represent code directly, but a template for several versions of that code. When you compile a non-template function in a `.cpp` file, you are compiling a concrete function/class. This is not the case for templates, which can be instantiated with different types, namely, concrete code must be emitted when replacing template parameters with concrete types.

There was a feature with the `export` keyword that was meant to be used for separate compilation. The `export` feature is deprecated in `C++11` and, AFAIK, only one compiler implemented it. You shouldn't make use of `export`. Separate compilation is not possible in `C++` or `C++11` but maybe in `C++17`, if concepts make it in, we could have some way of separate compilation.

For separate compilation to be achieved, separate template body checking must be possible. It seems that a solution is possible with concepts. Take a look at this paper recently presented at the standards committee meeting. I think this is not the only requirement, since you still need to instantiate code for the template code in user code.

The separate compilation problem for templates I guess it's also a problem that is arising with the migration to modules, which is currently being worked.

EDIT: As of August 2020 Modules are already a reality for C++: https://en.cppreference.com/w/cpp/language/modules

Share Improve this answer Follow

edited Apr 21, 2021 at 23:11
Cameron Tacklind
6,527 ● 1 ● 39 ● 47

answered May 12, 2013 at 16:42
Germán Diago
7,513 ● 1 ● 37 ● 63

As of now I know of know implementation in the common compilers for modules. – chadmc May 6, 2021 at 16:05

29

Even though there are plenty of good explanations above, I'm missing a practical way to separate templates into header and body.

My main concern is avoiding recompilation of all template users, when I change its definition.

Having all template instantiations in the template body is not a viable solution for me, since the template author may not know all if its usage and the template user may not have the right to modify it.

I took the following approach, which works also for older compilers (gcc 4.3.4, aCC A.03.13).

For each template usage there's a typedef in its own header file (generated from the UML model). Its body contains the instantiation (which ends up in a library which is linked in at the end).

Each user of the template includes that header file and uses the typedef.

A schematic example:

MyTemplate.h:

```
#ifndef MyTemplate_h
#define MyTemplate_h 1

template <class T>
class MyTemplate
{
public:
  MyTemplate(const T& rt);
  void dump();
  T t;
};

#endif
```

MyTemplate.cpp:

```
#include "MyTemplate.h"
#include <iostream>

template <class T>
MyTemplate<T>::MyTemplate(const T& rt)
: t(rt)
{
}

template <class T>
void MyTemplate<T>::dump()
{
  cerr << t << endl;
}
```

MyInstantiatedTemplate.h:

```
#ifndef MyInstantiatedTemplate_h
#define MyInstantiatedTemplate_h 1
#include "MyTemplate.h"

typedef MyTemplate< int > MyInstantiatedTemplate;

#endif
```

MyInstantiatedTemplate.cpp:

```
#include "MyTemplate.cpp"

template class MyTemplate< int >;
```

main.cpp:

```
#include "MyInstantiatedTemplate.h"

int main()
{
  MyInstantiatedTemplate m(100);
  m.dump();
  return 0;
}
```

This way only the template instantiations will need to be recompiled, not all template users (and dependencies).

Share Improve this answer Follow

edited Sep 8, 2022 at 18:18
Remy Lebeau
571k ● 31 ● 476 ● 804

answered May 12, 2016 at 14:02
lafrecciablu
385 ● 3 ● 8

2 I like this approach with the exception of the `MyInstantiatedTemplate.h` file and added `MyInstantiatedTemplate` type. It's a little cleaner if you don't use that, imho. Checkout my answer on a different question showing this: stackoverflow.com/a/41292751/4612476 – Cameron Tacklind Feb 16, 2019 at 8:55

1 This takes best of two worlds. I wish this answer was rated higher! Also see the link above for a slightly cleaner implementation of the same idea. – Wormer Oct 30, 2019 at 20:10

17

It means that the most portable way to define method implementations of template classes is to define them inside the template class definition.

```
template < typename ... >
class MyClass
{

    int myMethod()
    {
       // Not just declaration. Add method implementation here
    }
};
```

Share Improve this answer Follow

edited Feb 10, 2009 at 21:44
Evan Teran
89.1k ● 32 ● 183 ● 240

answered Jan 30, 2009 at 10:53
Benoît
16.9k ● 8 ● 47 ● 67

12

The compiler will generate code for each template instantiation when you use a template during the compilation step. In the compilation and linking process .cpp files are converted to pure object or machine code which in them contains references or undefined symbols because the .h files that are included in your main.cpp have no implementation YET. These are ready to be linked with another object file that defines an implementation for your template and thus you have a full a.out executable.

However since templates need to be processed in the compilation step in order to generate code for each template instantiation that you define, so simply compiling a template separate from it's header file won't work because they always go hand and hand, for the very reason that each template instantiation is a whole new class literally. In a regular class you can separate .h and .cpp because .h is a blueprint of that class and the .cpp is the raw implementation so any implementation files can be compiled and linked regularly, however using templates .h is a blueprint of how the class should look not how the object should look meaning a template .cpp file isn't a raw regular implementation of a class, it's simply a blueprint for a class, so any implementation of a .h template file can't be compiled because you need something concrete to compile, templates are abstract in that sense.

Therefore templates are never separately compiled and are only compiled wherever you have a concrete instantiation in some other source file. However, the concrete instantiation needs to know the implementation of the template file, because simply modifying the `typename T` using a concrete type in the .h file is not going to do the job because what .cpp is there to link, I can't find it later on because remember templates are abstract and can't be compiled, so I'm forced to give the implementation right now so I know what to compile and link, and now that I have the implementation it gets linked into the enclosing source file. Basically, the moment I instantiate a template I need to create a whole new class, and I can't do that if I don't know how that class should look like when using the type I provide unless I make notice to the compiler of the template implementation, so now the compiler can replace `T` with my type and create a concrete class that's ready to be compiled and linked.

To sum up, templates are blueprints for how classes should look, classes are blueprints for how an object should look. I can't compile templates separate from their concrete instantiation because the compiler only compiles concrete types, in other words, templates at least in C++, is pure language abstraction. We have to de-abstract templates so to speak, and we do so by giving them a concrete type to deal with so that our template abstraction can transform into a regular class file and in turn, it can be compiled normally. Separating the template .h file and the template .cpp file is meaningless. It is nonsensical because the separation of .cpp and .h only is only where the .cpp can be compiled individually and linked individually, with templates since we can't compile them separately, because templates are an abstraction, therefore we are always forced to put the abstraction always together with the concrete instantiation where the concrete instantiation always has to know about the type being used.

Meaning `typename T` get's replaced during the compilation step not the linking step so if I try to compile a template without `T` being replaced as a concrete value type that is completely meaningless to the compiler and as a result object code can't be created because it doesn't know what `T` is.

It is technically possible to create some sort of functionality that will save the template.cpp file and switch out the types when it finds them in other sources, I think that the standard does have a keyword `export` that will allow you to put templates in a separate cpp file but not that many compilers actually implement this.

Just a side note, when making specializations for a template class, you can separate the header from the implementation because a specialization by definition means that I am specializing for a concrete type that can be compiled and linked individually.

Share Improve this answer Follow

edited Jan 19, 2020 at 18:46

answered Jul 19, 2016 at 1:10
abc
1,957 ● 16 ● 31

10

If the concern is the extra compilation time and binary size bloat produced by compiling the .h as part of all the .cpp modules using it, in many cases what you can do is make the template class descend from a non-templatized base class for non type-dependent parts of the interface, and that base class can have its implementation in the .cpp file.

Share Improve this answer Follow

answered Jul 27, 2016 at 5:01
Eric Shaw
141 ● 1 ● 5

2 This response should be modded up quite more. I "*independently*" discovered your same approach and was specifically looking for somebody else to have used it already, since I'm curious if it's an *official pattern* and whether it's got a name. My approach is to implement a `class XBase` wherever I need to implement a `template class X`, putting the type-dependent parts in `X` and all the rest in `XBase`. – Fabio A. Nov 4, 2016 at 8:38

10

Just to add something noteworthy here. One can define methods of a templated class just fine in the implementation file when they are not function templates.

---

myQueue.hpp:

```
template <class T>
class QueueA {
    int size;
    ...
public:
    template <class T> T dequeue() {
       // implementation here
    }

    bool isEmpty();

    ...
}
```

---

myQueue.cpp:

```
// implementation of regular methods goes like this:
template <class T> bool QueueA<T>::isEmpty() {
    return this->size == 0;
}


main()
{
    QueueA<char> Q;

    ...
}
```

Share Improve this answer Follow

answered Jul 19, 2018 at 0:49

KeyC0de
4,977 ● 9 ● 48 ● 71

3 For real man???If that's true then your answer should be checked as correct one.Why does anyone need all those hacky voodo stuff if you can just define non template member methods in .cpp? – Michael IV Feb 18, 2020 at 17:11

Well that doesn't work.At least on MSVC 2019,getting unresolved external symbol for a member function of template class. – Michael IV Feb 20, 2020 at 16:19

I don't have MSVC 2019 to test. This is allowed by the C++ standard. Now, MSVC is notorious for not always adhering to the rules. If you haven't already, try Project Settings -> C/C++ -> Language -> Conformance Mode -> Yes (permissive-). – KeyC0de Feb 20, 2020 at 18:57

4 This exact example works but then you can't call `isEmpty` from any other translation unit besides `myQueue.cpp` ... – M.M Apr 11, 2020 at 0:39

so this can be a good strategy for moving bulky functions to .cpp files and declaring them private, while the public functions stay in header file and call them. – Abhinav Gauniyal Nov 8, 2020 at 5:55

9

A way to have separate implementation is as follows.

inner_foo.h

```
template <typename T>
struct Foo
{
    void doSomething(T param);
};
```

foo.tpp

```
#include "inner_foo.h"

template <typename T>
void Foo<T>::doSomething(T param)
{
    //implementation
}
```

foo.h

```
#include <foo.tpp>
```

main.cpp

```
#include <foo.h>
```

`inner_foo.h` has the forward declarations. `foo.tpp` has the implementation and includes `inner_foo.h`; and `foo.h` will have just one line, to include `foo.tpp`.

On compile time, contents of `foo.h` are copied to `foo.tpp` and then the whole file is copied to `foo.h` after which it compiles. This way, there is no limitations, and the naming is consistent, in exchange for one extra file.

I do this because static analyzers for the code break when it does not see the forward declarations of class in `*.tpp`. This is annoying when writing code in any IDE or using YouCompleteMe or others.

Share Improve this answer Follow

edited Sep 8, 2022 at 18:17
Remy Lebeau
571k ●31 ●476 ●804

answered May 13, 2017 at 1:42
Pranay
403 ●5 ●14

5 s/inner_foo/foo/g and include foo.tpp at the end of foo.h. One less file. – user246672 Jan 23, 2019 at 5:19

user246672 had it slightly wrong -- just include the `.tpp` files (I use `.ft`) in the `.cpp` files that need them. – Spencer Mar 29, 2022 at 16:46

7

That is exactly correct because the compiler has to know what type it is for allocation. So template classes, functions, enums,etc.. must be implemented as well in the header file if it is to be made public or part of a library (static or dynamic) because header files are NOT compiled unlike the c/cpp files which are. If the compiler doesn't know the type is can't compile it. In .Net it can because all objects derive from the Object class. This is not .Net.

Share Improve this answer Follow

edited Sep 17, 2011 at 12:27
Flexo ♦
88k ●22 ●194 ●277

answered Sep 17, 2011 at 3:40
Robert
89 ●1 ●1

7 "header files are NOT compiled" - that's a really odd way of describing it. Header files can be part of a translation unit, just like a "c/cpp" file. – Flexo ♦ Sep 17, 2011 at 12:26

4 In fact, it's almost the opposite of the truth, which is that header files are very frequently compiled many times, whereas a source file is usually compiled once. – xaxxon Dec 21, 2015 at 22:46

5

I suggest looking at this gcc page which discusses the tradeoffs between the "cfront" and "borland" model for template instantiations.

https://gcc.gnu.org/onlinedocs/gcc-4.6.4/gcc/Template-Instantiation.html

The "borland" model corresponds to what the author suggests, providing the full template definition, and having things compiled multiple times.

It contains explicit recommendations concerning using manual and automatic template instantiation. For example, the "-repo" option can be used to collect templates which need to be instantiated. Or another option is to disable automatic template instantiations using "-fno-implicit-templates" to force manual template instantiation.

In my experience, I rely on the C++ Standard Library and Boost templates being instantiated for each compilation unit (using a template library). For my large template classes, I do manual template instantiation, once, for the types I need.

This is my approach because I am providing a working program, not a template library for use in other programs. The author of the book, Josuttis, works a lot on template libraries.

If I was really worried about speed, I suppose I would explore using Precompiled Headers https://gcc.gnu.org/onlinedocs/gcc/Precompiled-Headers.html

which is gaining support in many compilers. However, I think precompiled headers would be difficult with template header files.

Share Improve this answer Follow

edited Feb 17, 2020 at 20:36

answered Feb 17, 2020 at 20:30
Juan
3,707 ● 3 ● 30 ● 32

3

(copying here from a closed duplicate)

I prefer to have all of my functions in the `.cpp` file, regardless of whether they are template functions or regular functions. And there is a way to do that with some basic `#ifndef` magic. Here's what you can do:

main.cpp

```
#include "myclass.hpp"

int main()
{
  // ...
}
```

myclass.hpp

```
#ifndef MYCLASS
#define MYCLASS

template<class T>
class MyClass
{
  T val;
public:
  MyClass(T val_);
}

#define MYCLASS_FUNCTIONS
#include "myclass.cpp"

#endif
```

myclass.cpp

```
#ifndef MYCLASS_FUNCTIONS
#include "myclass.hpp"

// regular functions:
// ...

#else

// template functions:
template<class T>
MyClass<T>::MyClass(T val_)
    :val(val_)
{}

// ...
#endif
```

Here's how the precompiler sees it. We have two `.cpp` files.

1. When we compile main.cpp we:
2. include `myclass.hpp`
3. check that `MYCLASS` is undefined, and it is
4. define it
5. give compiler the definitions of the generated class (from template class)
6. include `myclass.cpp`
7. define `MYCLASS_FUNCTIONS`
8. check if `MYCLASS_FUNCTIONS` is defined, it is
9. give compiler the definitions of the generated functions (from template functions)
10. When we compile myclass.cpp
11. check if `MYCLASS_FUNCTIONS` is defined, it isn't
12. include `myclass.hpp`
13. check that `MYCLASS` is undefined, and it is
14. define it
15. give compiler the definitions of the class
16. include `myclass.cpp`
17. include `myclass.hpp` again

18. this time `MYCLASS` is defined so do nothing inside, return to `myclass.cpp`
19. check if `MYCLASS_FUNCTIONS` is defined, it is
20. give compiler the definition of the generated functions (from template functions)
21. exit include twice
22. pass to the compiler all the regular functions

Share Improve this answer Follow

answered Jul 21, 2023 at 16:13
v010dya
5,506 ●7 ●29 ●49

1

With motivation from Moshe's answer from: https://stackoverflow.com/a/38448106/6459849

A small contribution from my side with an extended example. Let's say there is an overall OperationSuccess and it contains a ResponseSuccess which has a generic type in it.

ResponseSuccess.h

```
template <class T>
class ResponseSuccess {
public:
    ResponseSuccess(const ResponseStatus responseStatus, const T& data) :
        m_responseStatus(responseStatus),
        m_data(data) {}

    ~ResponseSuccess() = default;

    // Basis requirement, have Copy/Move constructor/delete assignment operator

    ResponseStatus getResponseStatus() const {
        return m_responseStatus;
    }

    T getData() const {
        return m_data;
    };

private:
    ResponseStatus m_responseStatus;

    T m_data;
};
```

OperationSuccess.h

```
template <class T>
class OperationResponse {
public:
    explicit OperationResponse(ResponseSuccess<T> responseSuccess) :
        m_responseSuccess(std::move(responseSuccess)) {}

    ~OperationResponse() = default;
    // Basis requirement, have Copy/Move constructor/delete assignment operator

    ResponseSuccess<T> getResponseSuccess() const {
        return m_responseSuccess;
    }

private:
    ResponseSuccess<T> m_responseSuccess;
    // have a failure, in case required
};
```

Usage:

```
MyObject myObj(<ctor_args>);
    ResponseSuccess<MyObject> responseSuccess(ResponseStatus::SUCCESS, myObj);
    OperationResponse<MyObject> successOperationResponse(responseSuccess);
..
// Fetches the response -> successOperationResponse.getResponseSuccess();
```

Share Improve this answer Follow

edited May 1, 2023 at 8:44

answered Apr 25, 2023 at 4:56
Arnab Das
133 ●1 ●5

-5

Another reason that it's a good idea to write both declarations and definitions in header files is for readability. Suppose there's such a template function in Utility.h:

```
template <class T>
T min(T const& one, T const& theOther);
```

And in the Utility.cpp:

```
#include "Utility.h"
template <class T>
T min(T const& one, T const& other)
{
    return one < other ? one : other;
}
```

This requires every T class here to implement the less than operator (<). It will throw a compiler error when you compare two class instances that haven't implemented the "<".

Therefore if you separate the template declaration and definition, you won't be able to only read the header file to see the ins and outs of this template in order to use this API on your own classes, though the compiler will tell you in this case about which operator needs to be overridden.

Share Improve this answer Follow

answered Mar 19, 2019 at 19:33
ClarHandsome
74 ● 4

-12

I had to write a template class an d this example worked for me

Here is an example of this for a dynamic array class.

```
#ifndef dynarray_h
#define dynarray_h

#include <iostream>

template <class T>
class DynArray{
    int capacity_;
    int size_;
    T* data;
public:
    explicit DynArray(int size = 0, int capacity=2);
    DynArray(const DynArray& d1);
    ~DynArray();
    T& operator[]( const int index);
    void operator=(const DynArray<T>& d1);
    int size();

    int capacity();
    void clear();

    void push_back(int n);

    void pop_back();
    T& at(const int n);
    T& back();
    T& front();
};

#include "dynarray.template" // this is how you get the header file

#endif
```

Now inside you .template file you define your functions just how you normally would.

```
template <class T>
DynArray<T>::DynArray(int size, int capacity){
    if (capacity >= size){
        this->size_ = size;
        this->capacity_ = capacity;
        data = new T[capacity];
    }
    //    for (int i = 0; i < size; ++i) {
    //        data[i] = 0;
    //    }
}

template <class T>
DynArray<T>::DynArray(const DynArray& d1){
    //clear();
    //delete [] data;
    std::cout << "copy" << std::endl;
    this->size_ = d1.size_;
    this->capacity_ = d1.capacity_;
    data = new T[capacity()];
    for(int i = 0; i < size(); ++i){
        data[i] = d1.data[i];
    }
}

template <class T>
DynArray<T>::~DynArray(){
    delete [] data;
}

template <class T>
T& DynArray<T>::operator[]( const int index){
    return at(index);
}

template <class T>
void DynArray<T>::operator=(const DynArray<T>& d1){
    if (this->size() > 0) {
        clear();
    }
    std::cout << "assign" << std::endl;
    this->size_ = d1.size_;
    this->capacity_ = d1.capacity_;
    data = new T[capacity()];
    for(int i = 0; i < size(); ++i){
        data[i] = d1.data[i];
    }

    //delete [] d1.data;
}
```

```
template <class T>
int DynArray<T>::size(){
    return size_;
}

template <class T>
int DynArray<T>::capacity(){
    return capacity_;
}

template <class T>
void DynArray<T>::clear(){
    for( int i = 0; i < size(); ++i){
        data[i] = 0;
    }
    size_ = 0;
    capacity_ = 2;
}

template <class T>
void DynArray<T>::push_back(int n){
    if (size() >= capacity()) {
        std::cout << "grow" << std::endl;
        //redo the array
        T* copy = new T[capacity_ + 40];
        for (int i = 0; i < size(); ++i) {
            copy[i] = data[i];
        }

        delete [] data;
        data = new T[ capacity_ * 2];
        for (int i = 0; i < capacity() * 2; ++i) {
            data[i] = copy[i];
        }
        delete [] copy;
        capacity_ *= 2;
    }
    data[size()] = n;
    ++size_;
}

template <class T>
void DynArray<T>::pop_back(){
    data[size()-1] = 0;
    --size_;
}

template <class T>
T& DynArray<T>::at(const int n){
    if (n >= size()) {
        throw std::runtime_error("invalid index");
    }
    return data[n];
}

template <class T>
T& DynArray<T>::back(){
    if (size() == 0) {
        throw std::runtime_error("vector is empty");
    }
    return data[size()-1];
}

template <class T>
T& DynArray<T>::front(){
    if (size() == 0) {
        throw std::runtime_error("vector is empty");
    }
    return data[0];
    }
```

Share Improve this answer Follow

edited Dec 7, 2022 at 20:49

answered Mar 11, 2019 at 18:38
Kode
592 ● 1 ● 6 ● 16

13 Most people would define a header file to be anything that propagates definitions to source files. So you may have decided to use the file extension ".template" but you've written a header file. – Tommy Aug 28, 2019 at 15:42

Highly active question. Earn 10 reputation (not counting the association bonus) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.