Visual Studio Tutorial: https://www.youtube.com/watch?v=3FkWddODLno
What's new? https://www.youtube.com/watch?v=TYEoJorFn54
6 Visual Studio Tips to Increase Your Productivity: https://www.youtube.com/watch?v=JhxC-K-Eehg

What goes in a header file C++?

functions do not, but inline functions do (or macros), because they produce code only where called. templates are not code, they are only a recipe for creating code. so they also go in h **files**. In general, you put declarations in the **header file** and definitions in the implementation (.cpp) **file**.

Header files (`.h`) are designed to provide the information that will be needed in multiple files. Things like class declarations, function prototypes, and enumerations typically go in header files. In a word, "definitions".

Code files (`.cpp`) are designed to provide the implementation information that only needs to be known in one file. In general, function bodies, and internal variables that should/will never be accessed by other modules, are what belong in `.cpp` files. In a word, "implementations".

The simplest question to ask yourself to determine what belongs where is "if I change this, will I have to change code in other files to make things compile again?" If the answer is "yes" it probably belongs in the header file; if the answer is "no" it probably belongs in the code file.

All good, but with inaccurate terminology. In a word, "declarations" - the term "definition" is synonymous with "implementation". Only declarative code, inline code, macro definitions, and template code should be in a header; i.e. nothing that instantiates code or data.

You use the terms declaration and definition rather loosly and somewhat interchangably. But they have precise meanings in C++. Examples: A class declaration introduces a name of a class but doesn't say what's in it. A class definition lists all members and friend functions. Both can be put into header files without problems. What you call "function prototype" is a function *declaration*. But a function *definition* is that thing that contains the function's code and should be placed into a cpp-file -- unless it's inline or (part of) a template.

What compiles into nothing (zero binary footprint) goes into header file.
Variables do not compile into nothing, but type declarations do (coz they only describe how variables behave).
functions do not, but inline functions do (or macros), because they produce code only where called.
templates are not code, they are only a recipe for creating code. so they also go in h files.

Question: In general, you put declarations in the header (.h) file and definitions in the implementation (.cpp) file, with the exception of what?

a. static variables.
b. templates, where the definition must also go in the header.
c. inline functions.
d. pre-processor macros.

Header (.h)

- Macros and includes needed for the interfaces (as few as possible)

- The declaration of the functions and classes

- Documentation of the interface

- Declaration of inline functions/methods, if any

- extern to global variables (if any)

Body (.cpp)

- Rest of macros and includes

- Include the header of the module

- Definition of functions and methods

- Global variables (if any)

As a rule of thumb, you put the "shared" part of the module on the .h (the part that other modules needs to be able to see) and the "not shared" part on the .cpp

Headers and Includes: Why and How

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
** 0) Introduction **
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

This article is meant to address a common newbie problem regarding failure to understand #include, headers, and source file interaction. Several good practices are outlined and explained to show how to avoid some ugly snags. Later sections get into more advanced topics (inlining and templates), so even C++ coders with some experience under their belt might benefit from a read!

If you are already familiar with the basics, feel free to skip ahead to section 4. That is where practices and design strategies are discussed.

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
** 1) Why we need header files. **
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

If you're just starting out in C++, you might be wondering why you need to #include files and why you would want to have multiple .cpp files for a program. The reasons for this are simple:

(1) It speeds up compile time. As your program grows, so does your code, and if everything is in a single file, then everything must be fully recompiled every time you make any little change. This might not seem like a big deal for small programs (and it isn't), but when you have a reasonable size project, compile times can take *several minutes* to compile the entire program. Can you imagine having to wait that long between every minor change?

Compile / wait 8 minutes / "oh crap, forgot a semicolon" / compile / wait 8 minutes / debug / compile / wait 8 minutes / etc

(2) It keeps your code more organized. If you seperate concepts into specific files, it's easier to find the code you are looking for when you want to make modifications (or just look at it to remember how to use it and/or how it works).

(3) It allows you to separate *interface* from *implementation*. If you don't understand what that means, don't worry, we'll see it in action throughout this article.

Those are the upsides, but the big, obvious downside is that is makes things a little more complicated if you don't understand how it all works (in reality, though, it's simpler than the alternatives as the project gets larger)

C++ programs are built in a two stage process. First, each source file is *compiled* on its own. The compiler generates intermediate files for each compiled source file. These intermediate files are often called *object files* -- but they are not to be confused with objects in your code. Once all the files have been individually compiled, it then *links* all the object files together, which generates the final binary (the program).

This means that *each source file is compiled separately* from other source files. As a result of this, in terms of compiling, "a.cpp" is clueless as to what's going on inside of "b.cpp". Here's a quick example to illustrate:

```
1  // in myclass.cpp
2
```

```
 3 class MyClass
 4 {
 5 public:
 6   void foo();
 7   int bar;
 8 };
 9
10 void MyClass::foo()
11 {
12   // do stuff
13 }
```

```
1 // in main.cpp                                              Edit & Run
2
3 int main()
4 {
5   MyClass a; // Compiler error: 'MyClass' is unidentified
6   return 0;
7 }
```

Even though MyClass is declared *in your program*, it is not declared *in main.cpp*, and therefore when you compile main.cpp you get that error.

This is where header files come in. Header files allow you to make the *interface* (in this case, the class MyClass) visible to other .cpp files, while keeping the *implementation* (in this case, MyClass's member function bodies) in its own .cpp file. That same example again, but tweaked slightly:

```
1 // in myclass.h
2
3 class MyClass
4 {
5 public:
6   void foo();
7   int bar;
8 };
```

```
1 // in myclass.cpp
2 #include "myclass.h"
3
4 void MyClass::foo()
5 {
6 }
```

```
1 //in main.cpp                                               Edit & Run
2 #include "myclass.h"   // defines MyClass
3
4 int main()
5 {
6   MyClass a; // no longer produces an error, because MyClass is
7 defined
8   return 0;
}
```

The #include statement is basically like a copy/paste operation. The compiler will "replace" the

#include line with the actual contents of the file you're including when it compiles the file.

```
*************************************************************
** 2) What is the difference between .h/.cpp/.hpp/.cc/etc **
*************************************************************
```

All files are fundamentally the same in that they're all text files, however different kinds of files should have different extensions:

- *Header files* should use a .h__ extension (.h / .hpp / .hxx). Which of those you use doesn't matter.
- *C++ Source files* should use a .c__ extention (.cpp / .cxx / .cc). Which of those you use doesn't matter.
- *C Source files* should use .c (.c only).

The reason C and C++ source files are seperated here is because it makes a difference in some compilers. In all likelyhood (since you're reading this on a C++ site), you'll be using C++ code, so refrain from using the .c extension. Also, files with header extensions might be ignored by the compiler if you try to compile them.

So what's the difference between Header files and Source files? Basically, header files are #included and not compiled, whereas source files are compiled and not #included. You can try to side-step these conventions and make a file with a source extension behave like a header or vice-versa, but you shouldn't. I won't list the many reasons why you shouldn't (other than the few I already have) -- just don't.

The one exception is that it is sometimes (although *very rarely*) useful to include a source file. This scenario has to do with instantiating templates and is outside the scope of this article. For now... just put it in your brain: "*do not #include source files*"

<div style="text-align:right;">Last edited on May 2, 2009 at 11:38am May 2, 2009 at 5:38pm UTC</div>

May 2, 2009 at 9:13am May 2, 2009 at 3:13pm UTC

Disch (13769)

```
*************************************************
** 3) Include guards **
*************************************************
```

C++ compilers do not have brains of their own, and so they will do exactly what you tell them to. If you tell them to include the same file more than once, then that is exactly what they will do. And if you don't handle it properly, you'll get some crazy errors as a result:

```
1 // myclass.h
2
3 class MyClass
4 {
5   void DoSomething() { }
6 };
```

```
1 // main.cpp                                                    Edit & Run
2 #include "myclass.h"    // define MyClass
3 #include "myclass.h"    // Compiler error - MyClass already defined
```

Now you might be saying to yourself "well that's stupid, why would I include the same file twice?" Believe it or not, it will happen a lot. Not quite as illustrated above, though. Usually it happens when you include two files that each include the same file. Example:

```
1 // x.h
2 class X { };
```

```
1  // a.h
2  #include "x.h"
3
4  class A { X x; };
```

```
1  // b.h
2  #include "x.h"
3
4  class B { X x; };
```

```
1  // main.cpp                                    Edit & Run
2
3  #include "a.h"  // also includes "x.h"
4  #include "b.h"  // includes x.h again!   ERROR
```

Because of this scenario, many people are told not to put #include in header files. However *this is bad advice and you should not listen to it*. Sadly, some people are actually *taught* this in C++ courses that they are *paying money for*. If your C++ instructor tells you not to #include in header files, then [grudgingly] follow his instructions in order to pass the course, but once you're out of his/her class, shake that habit.

The truth is there is nothing wrong with putting #include in header files -- and in fact it is very benefitial. *Provided you take two precautions*:

1) Only #include things you *need* to include (covered next section)
2) Guard against incidental multiple includes with include guards.

An Include Guard is a technique which uses a unique identifier that you #define at the top of the file. Here's an example:

```
1  //x.h
2
3  #ifndef __X_H_INCLUDED__   // if x.h hasn't been included yet...
4  #define __X_H_INCLUDED__   //   #define this so the compiler knows it has
5  been included
6
7  class X { };
8
   #endif
```

This works by skipping over the entire header if it was already included. __X_H_INCLUDED__ is #defined the first time x.h is included -- and if x.h is included a second time, the compiler will skip over the header because the #ifndef check will fail.

Always guard your headers. Always always always. It doesn't hurt anything to do it, and it will save you some headaches. For the rest of this article, it is assumed all header files are include guarded (even if I don't explicitly put it in the example).

You do not need to guard your .cpp files, because they are not #included (Or at least they shouldn't be... right? *RIGHT?*)

```
****************************************************
** 4) The "right way" to include **
****************************************************
```

Classes you create will often have dependencies on other classes. A derived class, for example, will

always be dependent on its parent, because in order to be derived from the parent, it must be aware of its parent at compile time.

There are two basic kinds of dependencies you need to be aware of:
1) stuff that can be forward declared
2) stuff that needs to be #included

If, for example, class A uses class B, then class B is one of class A's dependencies. Whether it can be forward declared or needs to be included depends on how B is used within A:

```
- do nothing if: A makes no references at all to B
- do nothing if: The only reference to B is in a friend declaration
- forward declare B if: A contains a B pointer or reference: B* myb;
- forward declare B if: one or more functions has a B object/pointer/reference
as a parementer, or as a return type: B MyFunction(B myb);
- #include "b.h" if: B is a parent class of A
- #include "b.h" if: A contains a B object: B myb;
```

You want to do the least drastic option possible. Do nothing if you can, but if you can't, forward declare if you can. But if it's necessary, then #include the other header.

Ideally, the dependencies for the class should be layed out in the header. Here is a typical outline of how a "right way" header might look:

myclass.h

```cpp
 1 //===============================
 2 // include guard
 3 #ifndef __MYCLASS_H_INCLUDED__
 4 #define __MYCLASS_H_INCLUDED__
 5
 6 //===============================
 7 // forward declared dependencies
 8 class Foo;
 9 class Bar;
10
11 //===============================
12 // included dependencies
13 #include <vector>
14 #include "parent.h"
15
16 //===============================
17 // the actual class
18 class MyClass : public Parent  // Parent object, so #include "parent.h"
19 {
20 public:
21   std::vector<int> avector;    // vector object, so #include <vector>
22   Foo* foo;                    // Foo pointer, so forward declare Foo
23   void Func(Bar& bar);         // Bar reference, so forward declare Bar
24
25   friend class MyFriend;       // friend declaration is not a dependency
26                                //   don't do anything about MyFriend
27 };
28
29 #endif // __MYCLASS_H_INCLUDED__
```

This shows the two different kinds of dependencies and how to handle them. Because MyClass only uses a pointer to Foo and not a full Foo object, we can forward declare Foo, and don't need to #include "foo.h". *You should always forward declare what you can -- don't #include unless it's necessary*. Unnecessary #includes can lead to trouble.

If you stick to this system, you will bulletproof yourself, and will minimize #include related hazards.

```
*****************************************************
** 5) Why that is the "right way" to include **
*****************************************************
```

Note: in this section I refer to the "right way" method outlined above as "mine". While I did come up with it on my own (after struggling through the mucky muck for a while) -- I can't say I was the first person who ever thought of it, so it isn't really "mine". But for purposes of this article, I call it "mine" for simplicity.

You: "So-and-so says that #including in a header is dangerous, but you say it's not! Why is your way so much better than what so-and-so says?"

So-and-so is partially right, but is explaining it wrong. Frivilous and careless #includes *can* lead to trouble. And one way to avoid those troubles *is* to never #include inside a header file. So yeah, so-and-so's heart is in the right place. But ultimately, using so-and-so's approach is going to give yourself TONS of additional work and headaches.

The concept I'm illustrating is very OO, and enhances encapsulation. The general idea is that it makes "myclass.h" fully self-contained and doesn't require any other area of the program (other than MyClass's implementation/source file) to know how MyClass works internally. If some other class needs to use MyClass, it can just #include "myclass.h" and be done with it!

The alternative (so-and-so's method), would require you to #include all of MyClass's dependencies *before* #including "myclass.h", since myclass.h can't include its dependencies itself. This is headache city, because now using a class isn't so straightforward.

Here is an example of why my method is good:
```
1 //example.cpp
2
3 //  I want to use MyClass
4 #include "myclass.h"   // will always work, no matter what MyClass looks
5 like.
6                        // You're done
7              //  (provided myclass.h follows my outline above and does
                //   not make unnecessary #includes)
```

Here is an example of why so-and-so's method is bad:
```
1 //example.cpp
2
3 //  I want to use MyClass
4 #include "myclass.h"
5   // ERROR 'Parent' undefined
```

so-and-so: "Hrm... okay...."
```
1 #include "parent.h"
2 #include "myclass.h"
3   // ERROR 'std::vector' undefined
```

```
1 #include "parent.h"
2 #include <vector>
3 #include "myclass.h"
4     // ERROR 'Support' undefined
```

so-and-so: "WTF? MyClass doesn't even *use* Support! But alright..."

```
1 #include "parent.h"
2 #include <vector>
3 #include "support.h"
4 #include "myclass.h"
5     // ERROR 'Support' undefined
```

so-and-so: "Give me a break! I'm including it! What else do you want!"

Believe it or not, the above *does* happen. Little did poor so-and-so know, but "parent.h" uses Support, and therefore you must #include "support.h" *before* "parent.h".

And what happens if support.h needs something else? What if *that something else* needs something else? We're already up to 4 #includes just to use a single class! With so-and-so's method, not only do you have to remember which includes are needed for each class, but also *the order in which you need to #include them*. This becomes a *huge* nightmare *very* quickly.

And what happens if you want to tweak MyClass? Let's say you decide that it would be better to use std::list instead of std::vector. With so-and-so's method, you now have to go back and change *every single file* that #includes "myclass.h" and change it to include <list> instead of <vector> (which might be dozens of files depending on the size of the project and how often MyClass is used), whereas with my method you only have to change "myclass.h", and maybe "myclass.cpp".

The "right way" I illustrated above is all about encapsulation. Files that want to use MyClass don't need to be aware of what MyClass uses in order for it to work, and don't need to #include any MyClass dependencies. All you need to do to get MyClass to work is #include "myclass.h". Period. The header file is set up to be completely self contained. It's all very OO friendly, very easy to use, and very easy to maintain.

```
**************************************************
** 6) Circular Dependencies **
**************************************************
```

A circular dependency is when two (or more) classes depend on each other. For example, class A depends on class B, and class B depends on class A. *If you stick to "the right way"* and forward declare when you can instead of #including needlessly, this usually isn't a problem. As long as the circle is broken with a forward declaration at some point, you're fine.

Here's the perfect example of why you should only #include what is necessary:

```
1 // a.h -- assume it's guarded
2 #include "b.h"
3
4 class A { B* b; };
```

```
1 // b.h -- assume it's guarded
2 #include "a.h"
3
4 class B { A* a };
```

An initial glance might see nothing wrong with this. B is a dependency of A, so you include it, and A is a dependency of B, so you include it. So what's wrong with this?

This is a circular inclusion (also called an infinite inclusion) and is the result of one or more includes that shouldn't be there. Say for example you compile "a.cpp":

```
1 // a.cpp
2 #include "a.h"
```

The compiler will do the following:

```
1 #include "a.h"
2
3    // start compiling a.h
4    #include "b.h"
5
6        // start compiling b.h
7        #include "a.h"
8
9            // compilation of a.h skipped because it's guarded
10
11       // resume compiling b.h
12       class B { A* a };          // <--- ERROR, A is undeclared
```

Even though you're #including "a.h", the compiler is not seeing the A class until after the B class gets compiled. This is because of the circular inclusion problem. This is why you should *always forward declare* when you're only using a pointer or reference. Here, "a.h" should not be #including b.h, but instead should just be forward declaring B. Likwise, b.h should be forward declaring A. If you make those changes, the problem is solved.

The circular inclusion problem may persist if both dependencies are #include dependencies (ie: they can't be forward declared). Here's an example:

```
1 // a.h (guarded)
2
3 #include "b.h"
4
5 class A
6 {
7    B b;    // B is an object, can't be forward declared
8 };
```

```
1 // b.h (guarded)
2
3 #include "a.h"
4
5 class B
6 {
7    A a;    // A is an object, can't be forward declared
8 };
```

You may note, however, that this situation is *conceptually impossible*. There is a fundamental design

flaw. If A has a B object, and B has an A object, then A contains a B, which contains another A, which contains another B, which contains another A, which contains another B, etc, etc. You have an infinite recursion problem, and either class is simply impossible to instantiate. The solution is to have one or both classes contain a *pointer or reference* to the other, rather than a full object. Then you can forward declare, and then you can get around the circular inclusion problem.

Last edited on May 2, 2009 at 11:40am May 2, 2009 at 5:40pm UTC

May 2, 2009 at 9:15am May 2, 2009 at 3:15pm UTC

Disch (13769)

```
**************************************************
** 7) Function inlining **
**************************************************
```

The catch-22 with inline functions is that their function body needs to exist in every cpp file which calls them, otherwise you get linker errors (since they can't be linked during the linker process -- they need to be compiled into the code during the compiler process).

This might open circular reference holes or other scenarios that might complicate the "right way" outline.

```
1  class B
2  {
3  public:
4    void Func(const A& a)    // parameter, so forward declare is okay
5    {
6      a.DoSomething();       // but now that we've dereferenced it, it
7                             //  becomes an #include dependency
8                  // = we now have a potential circular inclusion
9    }
10 };
```

The key is that while inline function need to exist in the header, they *do not* need to exist in the class definition itself. This lets us exploit a loophole:

```
1  // b.h  (assume its guarded)
2
3  //-----------------
4  class A;  // forward declared dependency
5
6  //-----------------
7  class B
8  {
9  public:
10   void Func(const A& a);  // okay, A is forward declared
11 };
12
13 //-----------------
14 #include "a.h"         // A is now an include dependency
15
16 inline void B::Func(const A& a)
17 {
18   a.DoSomething();     // okay!  a.h has been included
19 }
```

While you might not think so at first glance... this is *perfectly safe*. The circular inclusion problem is avoided completely, even if a.h includes b.h. This is because the additional #includes don't come up until AFTER class B is fully defined, and they are therefore harmless.

You: "But putting function bodies at the end of my header is ugly. Isn't there a way to avoid that?"

Me: Yep! Just move the function bodies to another header:

```
1 // b.h
2
3     // blah blah
4
5 class B { /* blah blah */ };
6
7 #include "b_inline.h"  // or I sometimes use "b.hpp"
```

```
1  // b_inline.h (or b.hpp -- whatever)
2
3  #include "a.h"
4  #include "b.h"  // not necessary, but harmless
5                  //  you can do this to make this "feel" like a source
6                  //  file, even though it isn't
7
8  inline void B::Func(const A& a)
9  {
10   a.DoSomething();
11 }
```

This seperates the interface from the implementation, while still allowing the implementation to be inlined. You can also have a normal "b.cpp" file for the implementation that isn't inlined.

```
****************************************************
** 8) Forward declaring templates **
****************************************************
```

Forward declaring is pretty straight-forward when it comes to simple classes, but when dealing with template classes, things aren't so simple. Consider the following scenario:

```
1  // a.h
2
3  // included dependencies
4  #include "b.h"
5
6  // the class template
7  template <typename T>
8  class Tem
9  {
10   /*...*/
11    B b;
12 };
13
14 // class most commonly used with 'int'
15 typedef Tem<int> A;   // typedef'd as 'A'
```

```
1  // b.h
2
3  // forward declared dependencies
4  class A;   // error!
```

```
 5
 6 // the class
 7 class B
 8 {
 9   /* ... */
10   A* ptr;
11 };
```

While this seems perfectly logical, it doesn't work! (Although, logically you really think it should. This is an irritation of the language). Because 'A' isn't really a class, but rather a typedef, the compiler will bark at you. Also notice that we can't just #include "a.h" here because of a circular dependency problem.

In order to forward declare 'A', we need to typedef it. Which means we need to forward declare what it's typedef'd as. The way to forward declare it is like so:

```
1 template <typename T> class Tem;   // forward declare our template
2 typedef Tem<int> A;                // then typedef 'A'
```

This is quite a bit uglier than `class A;` , but nonetheless is a necessary evil. This, however, makes templated classes hard to encapsulate, because it requires every class which forward declares them to know exactly how the template is layed out. If that ever changes you have a big mess to clean up.

A practical solution to this problem is to create an alternative header which has the forward declarations of your templated classes and their typedefs. Here's a more elegant way to approach the above example:

```
 1 //a.h
 2
 3 #include "b.h"
 4
 5 template <typename T>
 6 class Tem
 7 {
 8   /*...*/
 9   B b;
10 };
```

```
1 //a_fwd.h
2
3 template <typename T> class Tem;
4 typedef Tem<int> A;
```

```
1 //b.h
2
3 #include "a_fwd.h"
4
5 class B
6 {
7   /*...*/
8   A* ptr;
9 };
```

This allows B to include a header which forward declares A without including the entire class definition.

May 2, 2009 at 10:22am May 2, 2009 at 4:22pm UTC

helios (13682)

You should mention that only files that match source code patterns (*.c, *.cpp, *.cc, *.C, *.cxx, etc) are compiled, even if other files are passed to the compiler. For example, if you use the command line
`g++ main.cpp file.c file.h`
Only main.cpp and file.cpp will be compiled. A side effect of this is that header extensions are arbitrary. Also mention that the compiler to use is inferred from the extension, except when the compiler is explicitly told to compile something as C or C++.

About section 7, are you sure forward declaration doesn't work when passing parameters by value, yet it works when passing pointers/references and manually making copies?

Finally, about templates, I'd say it's better practice to put the template definition in the class declaration. Particularly if there are non-template methods in the class.

May 2, 2009 at 10:56am May 2, 2009 at 4:56pm UTC

Disch (13769)

> g++ main.cpp file.c file.h
> Only main.cpp and file.cpp will be compiled. A side effect of this is that header extensions are arbitrary.

I wasn't sure that was the case. iirc, you *could* compile headers in VS. I haven't tried it since i switched to CodeBlocks+GCC. But that's a valid point.

> About section 7

Oh crap! That's what i get for not testing enough. You're totally right, forward declaring works fine. Only problem happens if its implicitly inlined, but that's another matter.

> Finally, about templates, I'd say it's better practice to put the template definition in the class declaration.

Well -- I'm not a big fan of putting implementation in the class itself (unless it's a really small get() function or some other kind of 1-liner). I guess with templates it's alright because any dependencies can be forward declared and included after the class body (at least I think so, I'd have to actually test that).

There are other considerations, too, though. Like if the template class is exceedingly large and you want to ease compile time (though it would have to be pretty freaking big to make a difference)

Anyway overall I agree. I just included that bit out of completeness. I figured I should focus more on the instantiating method since everybody knows how to do the inlining method. But really -- the more I think about it, the more I think that should belong in another article (like one specifically talking about templates).

In response to that, I've decided to cut sections 7 and 9 completely, and touch up a few related things. I'll edit the posts once I get it straightened out on my local copy.

Thanks for the feedback!

May 2, 2009 at 11:21am May 2, 2009 at 5:21pm UTC

helios (13682)

Visual C++ doesn't compile headers, either.

May 2, 2009 at 11:33am May 2, 2009 at 5:33pm UTC

Disch (13769)

Then I guess I didn't recall correctly XD

Oromis (80)

Thanks for the article.

When using short memberfunctions, the body is often included, like this:

```
1  // MyClass.h
2
3  class MyClass
4  {
5    private:
6      int a;
7
8    public:
9      void set(int n) { a = n;}
10     int  get()      { return a;}
11 };
```

This results in an error when the header is included in both main.cpp and MyClass.cpp. Does this mean it should always be avoided?

Last edited on May 3, 2009 at 11:54am May 3, 2009 at 5:54pm UTC

helios (13682)

Yes. it costs nothing to put the function definitions in a separate file and you avoid errors errors.

Oromis (80)

Oke, thanks :)

Disch (13769)

I disagree, actually.

There is nothing wrong with shortcutting functions like that. It's called "implicit inlining", and doing it may actually provide a performance boost for small functions like get/set members. Taking the functions out of the class definition and putting them in a .cpp file means they can no longer be inlined, which may result in [ever-so-slightly] less efficient code (though you'll only really notice if you're calling the funciton *a whole lot* -- otherwise it doesn't really matter).

That above MyClass.h example you gave *should not* generate any errors, no matter how many .cpp files you include it in. If you are getting errors it must be due to something else. Can you provide a minimal compilable example of where you are getting errors doing this?

helios (13682)

Taking the functions out of the class definition and putting them in a .cpp file means they can no longer be inlined

Functions without the inline keyword are never inlined. Functions with the inline keyword are inlined as the compiler sees fit.

Disch (13769)

I have read and heard otherwise from various sources.

Here is one of them:

http://www.parashift.com/c++-faq-lite/inline-functions.html#faq-9.8

helios (13682)

It says defining the function inside the class declaration is an alias for the inline keyword.

Disch (13769)

Er right.

Meaning you can have functions inlined without using the inline keyword. IE implicit inlining.
helios (13682)
Right. It still doesn't guarantee the compiler will inline, and it's possible to still have them in a separate file by using the inline keyword.

Meh. Haven't had a good argument in a while. I think I'll start a thread and try to defend a purposely wrong premise.
Disch (13769)
Well basically what I'm saying is this:

- If he shortcuts and puts the function bodies in the class definition, he's implicitly inlining (*yes*)
- If he moves them to a .cpp file, they're no longer inlined (*no*)
- You can't inline a function that isn't #included in all source files that use it. So even if he tries to add the inline keyword, if he moves the functions to a .cpp file, they won't be inlined (*no*)

I think I'll start a thread and try to defend a purposely wrong premise.

hahahaah. I can't tell if you're being serious or sarcastic or what. Either way it's funny. XD
Oromis (80)
I thought I tested it, but it compiles fine now... I must have made some other mistake. The fact that it's implicitly inlined explains why there is no problem with defining the function twice, I guess. Thanks.
translore (31)
I had a teacher "so and so" tell me not to include inside header files and I think I even read this in a C++ text but I really do like the method in this article. It seems like it will help more in the OOP approach and will help make the code more robust! Thanks for the article...

How should global variables be handled when using seperated sourcefiles?

I found you need to declare the variable again using the keyword "extern" when you need access to a global variable declared in another sourcefile. I'm to lazy to repeat that for each sourcefile in my project, so I'm using the following construction:

```
1 //global_variables.cpp
2
3 int global_var;
```

```
1 //global_variables.h
2
3 extern int global_var;
```

```
1 //sourcefile.cpp
2
3 #include "global_variables.h"
4
5 /* use the variable somewhere */
```

It compiles, but is it the best way to do this?

It's best to avoid globals completely for many reasons. But if you *really need to*, there are a few things you can do.

One way is to manipulate #defines to put the `extern` in:

```cpp
1 // globals.cpp
2 #define GLOBAL           // #defining it here as nothing prevents 'extern'
3 #include "globals.h"
```

```cpp
1 // globals.h
2 #ifndef GLOBAL
3 #define GLOBAL extern
4 #endif
5
6 GLOBAL int global_var;
```

```cpp
1 // whatever.cpp
2 #include "globals.h"
```

Or, instead of making them all global, you could throw them in a struct/class and make a singleton instance of that struct/class:

```cpp
1 //globals.h
2 struct Globals
3 {
4 public:
5   int v1;
6   int v2;
7
8   inline static Globals& get()
9   {
10     static Globals the_globals;
11     return the_globals;
12   }
13 };
```

```cpp
1 // whatever.cpp
2 #include "globals.h"
3
4 void somefunc()
5 {
6   Globals::get().v1 = 5; // shorthand
7
8   // alternatively
9   Globals& g = Globals::get();
10   g.v1 = 5;
11   g.v2 = 6;
12 }
```

Latter method preferred. Though globals are hideously unorganized and should really be avoided where possible.

May 24, 2009 at 3:04pm May 24, 2009 at 9:04pm UTC

Oromis (80)

Oke, thanks. I will try to work around globals whenever possible, and those solutions will help me out when I can't.

May 24, 2009 at 3:26pm May 24, 2009 at 9:26pm UTC

Disch (13769)

Added section 8 about template classes.

weeee

May 31, 2009 at 9:15am May 31, 2009 at 3:15pm UTC

kempofighter (1183)

I had a teacher "so and so" tell me not to include inside header files and I think I even read this in a C++ text but I really do like the method in this article. It seems like it will help more in the OOP approach and will help make the code more robust! Thanks for the article...

I don't understand this part of the discussion. It is impossible not to include headers inside of other headers in all cases. What does the teacher expect you to do, dynamically allocate every single object in the entire system on the heap?

You obviously have to have a strategy and include as much as possible within .cpp files. I think that it would be very difficult and unwise to never include headers within headers in a large scale app.

May 31, 2009 at 12:48pm May 31, 2009 at 6:48pm UTC

jsmith (5804)

Yes, that's exactly right. That used to be a paradigm some folks kept to when developing C software. A company I used to work for tried at some level to adhere to that. It basically meant you could never add anything to a "common" header file that would suddenly require every .c file to include a new header.

I agree, though, in a C++ world it makes no sense at all.

(Ok, even in a C world it still doesn't make sense to me.)