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.

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
   }
};
```

A brief primer on const

To the compiler, the `const` qualifier on a method refers to *physical constness*: calling this method does not change the bits in this object. What we want is *logical constness*, which is only partly overlapping: calling this method does not affect the object in ways callers will notice, nor does it give you a handle with the ability to do so.

Mismatches between these concepts can occur in both directions. When something is logically but not physically const, C++ provides the `mutable` keyword to silence compiler complaints. This is valuable for e.g. cached calculations, where the cache is an implementation detail callers do not care about. When something is physically but not logically const, however, the compiler will happily accept it, and there are no tools that will automatically save you. This discrepancy usually involves pointers. For example,

```
void T::Cleanup() const { delete pointer_member_; }
```

Deleting a member is a change callers are likely to care about, so this is probably not logically const. But because `delete` does not affect the pointer itself, but only the memory it points to, this code is physically const, so it will compile.

Or, more subtly, consider this pseudocode from a node in a tree:

```
class Node {
 public:
  void RemoveSelf() { parent_->RemoveChild(this); }
  void RemoveChild(Node* node) {
    if (node == left_child_)
      left_child_ = nullptr;
    else if (node == right_child_)
      right_child_ = nullptr;
  }
  Node* left_child() const { return left_child_; }
  Node* right_child() const { return right_child_; }

 private:
  Node* parent_;
  Node* left_child_;
  Node* right_child_;
};
```

The left_child() and right_child() methods don't change anything about |this|, so making them const seems fine. But they allow code like this:

```
void SignatureAppearsHarmlessToCallers(const Node& node) {
  node.left_child()->RemoveSelf();
  // Now |node| has no |left_child_|, despite having been passed in by const
ref.
}
```

The original class definition compiles, and looks locally fine, but it's a timebomb: a const method returning a handle that can be used to change the system in ways that affect the original object. Eventually, someone will actually modify the object, potentially far away from where the handle is obtained.

These modifications can be difficult to spot in practice. As we see in the previous example, splitting related concepts or state (like "a tree") across several objects means a change to one object affects the behavior of others. And if this tree is in turn referred to by yet more objects (e.g. the DOM of a web page, which influences all sorts of other data structures relating to the page), then small changes can have visible ripples across the entire system. In a codebase as complex as Chromium, it can be almost impossible to reason about what sorts of local changes could ultimately impact the behavior of distant objects, and vice versa.

"Logically const correct" code assures readers that const methods will not change the system, directly or indirectly, nor allow callers to easily do so. They make it easier to reason about large-scale behavior. But since the compiler verifies physical constness, it will not guarantee that code is actually logically const. Hence the recommendations here.