HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

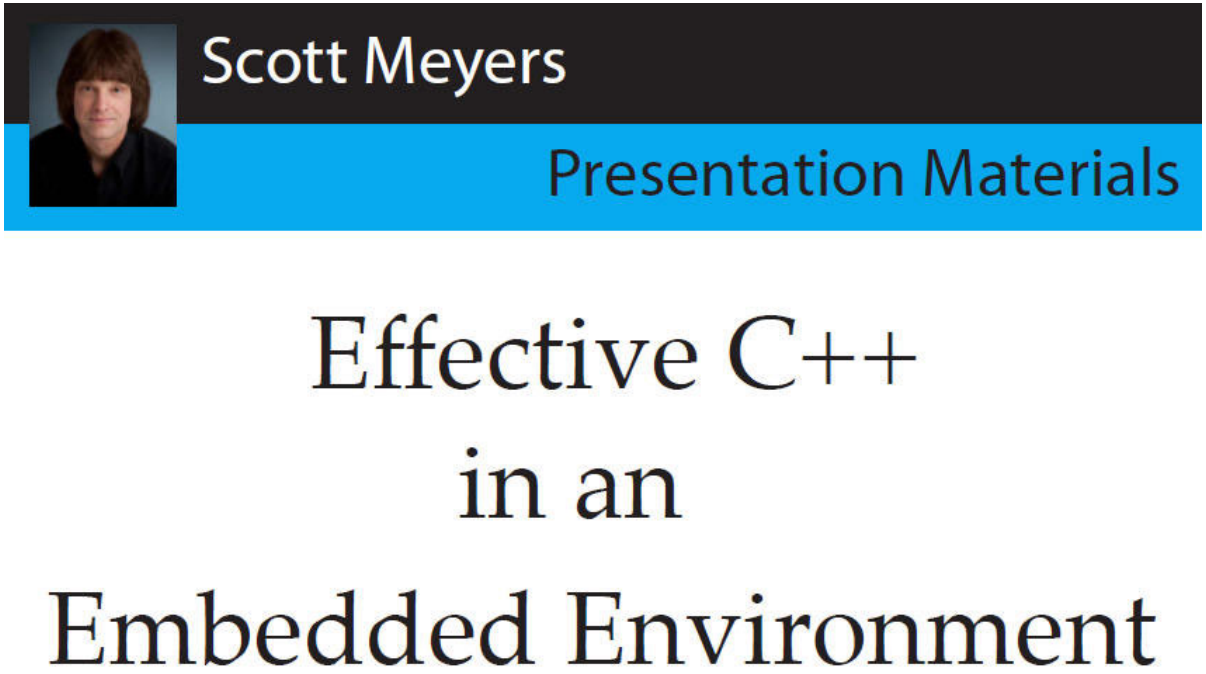
Embedded Software Engineering 2
Avoiding Code Bloat

Prof. Reto Bonderer
HSR Hochschule für Technik Rapperswil
reto.bonderer@hsr.ch

April 2020

Effective C++ in an Embedded Environment

Die meisten der folgenden Informationen stammen aus einem Seminar von Scott Meyers



Code Bloat in C++

C++ has a few features you pay for (in code size and/or runtime speed), even if you don't use them:

- Support for exceptions.
- Support for generalized customizable iostreams, i.e. streams of other than `char` or `wchar_t`.

These things may reasonably be considered bloat.

Possible workarounds:

- Disable exceptions during compilation.
  - Practical only if you know that no code (including libraries, plug-ins, etc.) throws.
- Use stdio instead of iostreams.

Code Bloat in C++ (cont'd)

However, most bloat accusations are unfair, traceable to either:

- **Comparing functionality in C++ with lesser functionality in C**
  - E.g., C++ virtual functions do more than C functions.
- **Improper use of the language**
  - E.g., Putting inessential code in constructors/destructors.

The feature most associated with bloat is templates.

- That's what I'll focus on here.
- Most problems with "template code bloat" arise from:
  - Misunderstandings of template rules.
  - Improper use of templates.

Templates, Header Files, and Inlining

Consider

```
template<typename T>    // header file for a class template
class SomeClass {
public:
  SomeClass() { ... }   // implicitly declared inline
  void mf1() { ... }    // implicitly declared inline
  void mf2();           // not implicitly declared inline
  ...
};

template<typename T>
void SomeClass<T>::mf2() { ... }
// template funcs are typically defined in header files, but this does not automatically declare
// them inline
```

Critical: Don't declare template functions inline simply because they are defined in headers.

Unnecessary inlining will lead to bloat.

Templates, Header Files, and Inlining (cont'd)

Templates need not be defined in headers:

```
template<typename T>
class SomeClass {
public:
  SomeClass() { ... }   // still implicitly inline
  void mf1() { ... }    // still implicitly inline
  void mf2();           // declaration only; no definition
  ...                   // provided in this file
};
```

Code using this header will compile fine.

- But if `SomeClass::mf2()` is called, it won't link.
  - We'll cover how to fix that in a moment.
- Templates are typically defined in header files to avoid such problems

Templates, Header Files, and Inlining (cont'd)

- The convention of putting all template code in headers has an advantage:
  - Single point of change for client-visible code, e.g., function declarations.
  - No need to change both header and implementation files.

- And some disadvantages:
  - Increased compilation times.
  - Increased compilation dependencies for clients.

Instantiating Templates

Templates that aren't used aren't instantiated.

- They thus generate no code and no data.
- But the need to read template headers usually slows compilation. So what?
- Templates can thus generate less code than non-templates!

```
class C {             // Even if C is never used, object files typically contain f1..fn. Few
public:               // linkers will remove all code and data related to uncalled functions.
  void f1();
  ...
  void fn();
};

template<typename T> // Object files should contain only those functions that are called.
class C {
public:
  void f1();
  ...
  void fn();
};
```

Dead code

Safety-critical systems often require the elimination of dead code, so the fact that templates can avoid generating it in the first place is attractive to people developing such systems.

Templates can help avoid linking dead code into executables

Instantiating Templates (cont'd)

Instantiated templates may generate both code and data:

```
SomeClass<int> sc;      // SomeClass<int> instantiated.
                        // Some code generated, memory for static class data set aside
```

Instantiating a class shouldn’t instantiate all its member functions:

- Only member functions that are used should be instantiated.
  - You shouldn’t pay for what you don’t use.

- A few compilers (typically older ones) get this wrong.
  - They instantiate all member functions of a class if any is used.
  - We’ll discuss how to avoid this in a moment.

Instantiating Templates (cont'd)

Most templates are *implicitly instantiated*:

- Compiler notes used functions, instantiates them automatically.
- To create the functions, it needs access to their definitions.
  - This is why template code is typically in header files.
- Without a definition, compiler generates reference to external symbol.
  - Hence `SomeClass::mf2()` callable w/o a definition, but a link-time error will result.

Templates can also be *explicitly instantiated*:

- You can force a class or function template to be instantiated.
  - For class templates, *all* member functions are instantiated.
  - Individual member functions can also be instantiated.

Explicit Instantiation

In a .h file:

```
template<typename T>            // as before
class SomeClass {
public:
  SomeClass() { ... }
  void mf1() { ... }
  void mf2();
  ...
};
```

In a .cpp file:

```
...                          // Definitions of SomeClass's non-inline functions go here

template                     // explicitly instantiate all SomeClass mem funcs for T=double
class SomeClass<double>;     // compiled code will go in this .cpp's .obj file

template                     // explicitly instantiate SomeClass::mf2 for T=int
void SomeClass<int>::mf2(); // compiled code will go in this .cpp's .obj file
```

Explicit Instantiation (cont'd)

Explicit instantiation can be a lot of work:

- You must manually list each template and set of instantiation parameters to be instantiated.

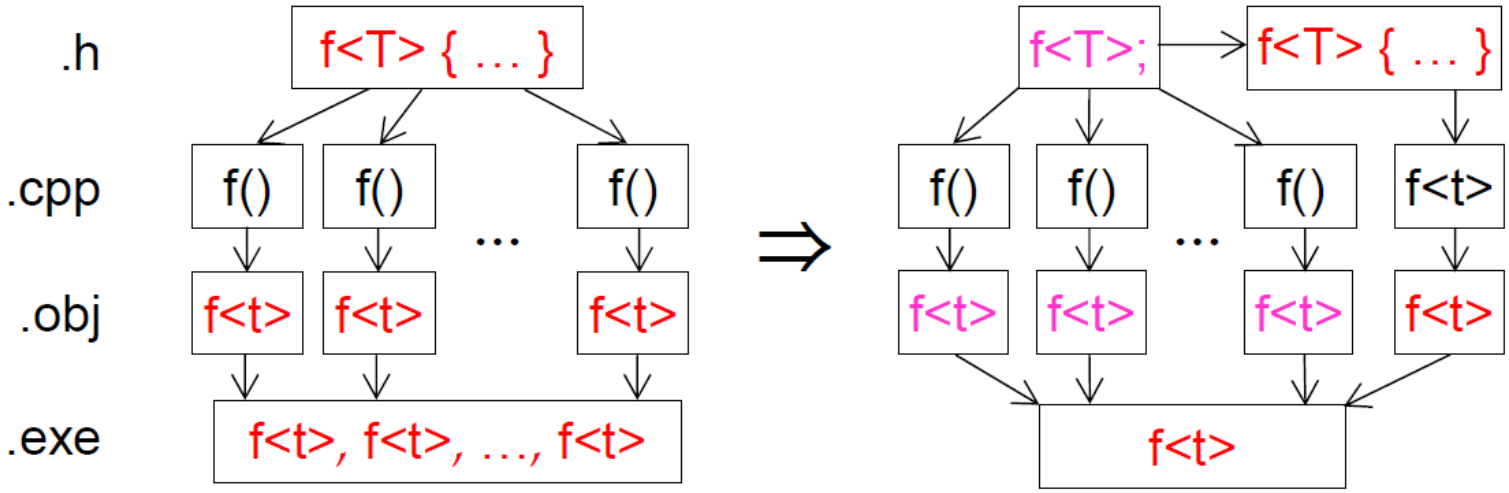
But it can be useful:

- To create libraries of instantiations.
- To put instantiations into particular code sections.
- To avoid code bloat arising from bad compilers/linkers.
  - Details on next page.

Explicit Instantiation (cont'd)

Your executable might end up with multiple copies of an instantiation:

- If your compiler (incorrectly) instantiates all class template member functions when only some are used.
- If you use dynamic linking.
- If your linker is bad:



Avoiding Code Duplication

Consider

```
class IntClass {
public:
  void usageInfo(std::ostream& s); // summarize usage info to s
  ...
};
class DoubleClass {
public:
  void usageInfo(std::ostream& s); // summarize usage info to s
  ...
};
```

Both usageInfo functions will do essentially the same thing.

- This is code duplication.
- It leads to code bloat.

Note that no templates are involved here.

Avoiding Code Duplication (cont'd)

A common way to eliminate such duplication is to move the duplicated code to a base class:

```
class Base {
public:
  void usageInfo(std::ostream& s); // summarize usage info to s
  ...
protected:
  ... // data storing usage info
};

class IntClass: public Base {
  ... // no declaration of usageInfo
};

class DoubleClass: public Base {
  ... // no declaration of usageInfo
};
```

Now there's only one copy of usageInfo in the program, regardless of how many classes inherit from Base.

Avoiding Code Duplication (cont'd)

Exactly the same reasoning applies when writing templates:

```
template<typename T>                  // a template leading to bloat
class SomeClass {
  ...
  void usageInfo(std::ostream& s);  // leads to code duplication if usageInfo makes no use of T
};
```

The solution is the same:

```
class Base {                          // same as on previous page
public:
  void usageInfo(std::ostream& s);  // summarize usage info to s
  ...
};

template<typename T>                  // a template avoiding bloat
class SomeClass: public Base {
  ... // no declaration of usageInfo
};
```

Code Hoisting

Moving type-invariant code into a base class sometimes called *code hoisting*.

It can help avoid code bloat due to multiple pointer types:

```
template<typename T>              // general template
class Stack { ... };

class GenericPtrStack { ... };   // non-template using void*s

template<typename T>             // partial specialization for pointers
class Stack<T*>:                 // uses void*-based base class for all real work
private GenericPtrStack {
  ...                            // all inline casting functions;
};                               // they generate no code
```

All Stack instantiations for pointer types thus share their code.

- We'll see this example in detail later.

Code Hoisting

Code hoisting works well with inlining to avoid duplication arising from non-type template parameters:

```
template<typename T, std::size_t bufSz> // Suspect design: each bufSz value will yield a new
class Buffer {                         // set of member functions
  T buffer[bufSz];
public:
  ...
};

template<typename T>                   // Better design:
class BufferBase {                     // BufferBase is independent of bufSz
...
};
template<typename T, std::size_t bufSz> // Buffer does only bufSz dependent operations.
class Buffer: public BufferBase<T> {
...                                    // Ideally, all are inline, so
};                                     // Buffer classes cost nothing
```

Avoiding Code Duplication

Avoiding code bloat with templates fundamentally calls for disciplined *commonality and variability analysis*:

- The parts of a template that don't depend on the template parameters (the *common* parts) should be moved out of the template.
- The remaining parts (the *variable* parts) should stay in the template.

This kind of analysis is critical to avoiding code duplication in any guise:

- Features common to multiple classes should be moved out of the classes.
  - Maybe to a base class.
  - Maybe to a class template.
- Features common to multiple functions should be moved out of the functions:
  - Maybe to a new function.
  - Maybe to a function template.

Code Bloat Summary

Most bloat can be eliminated by careful design. Arrows in your quiver:

- **Consider disabling support for exceptions.**
- **Consider stdio instead of iostreams.**
- **Avoid excessive inlining, especially with templates.**
- **Judiciously use explicit instantiation to avoid code duplication.**
- **Hoist parameter-independent code out of templates.**

Dealing with Function Templates

We've discussed only class templates, but bloat elimination techniques for function templates are similar:

```
template<typename T>                    // a template leading to bloat
void doSomething(const T& obj)
{
  ...                                   // code making use of T or obj
  ...                                   // code independent of T or obj
  ...                                   // code making use of T or obj
}
```

A "hoisting" alternative:

```
void doSomethingHelper();               // "hoisted" code in non-template
                                        // function; not inline
template<typename T>                    // revised template avoiding bloat
void doSomething(const T& obj)
{
  ...                                   // code making use of T or obj
  doSomethingHelper();
  ...                                   // code making use of T or obj
}
```

Data Bloat

Not all bloat is due to code. Unnecessary classes can yield data bloat, too:

- Some classes have a vtbl, so unnecessary classes ⇒ unnecessary vtbls.
  - Such unnecessary classes could come from templates.
- Functions must behave properly when exceptions are thrown, so unnecessary non-inline functions ⇒ unnecessary EH tables.
  - Such unnecessary functions could come from templates.
  - This applies only to the Table Approach to EH.

An important exception to these issues are class templates that:

- Contain only inline functions.
  - Hence no extra EH tables.
- Contain no virtual functions.
  - Hence no extra vtbls.

We'll see examples of such "bloat-free" templates later.