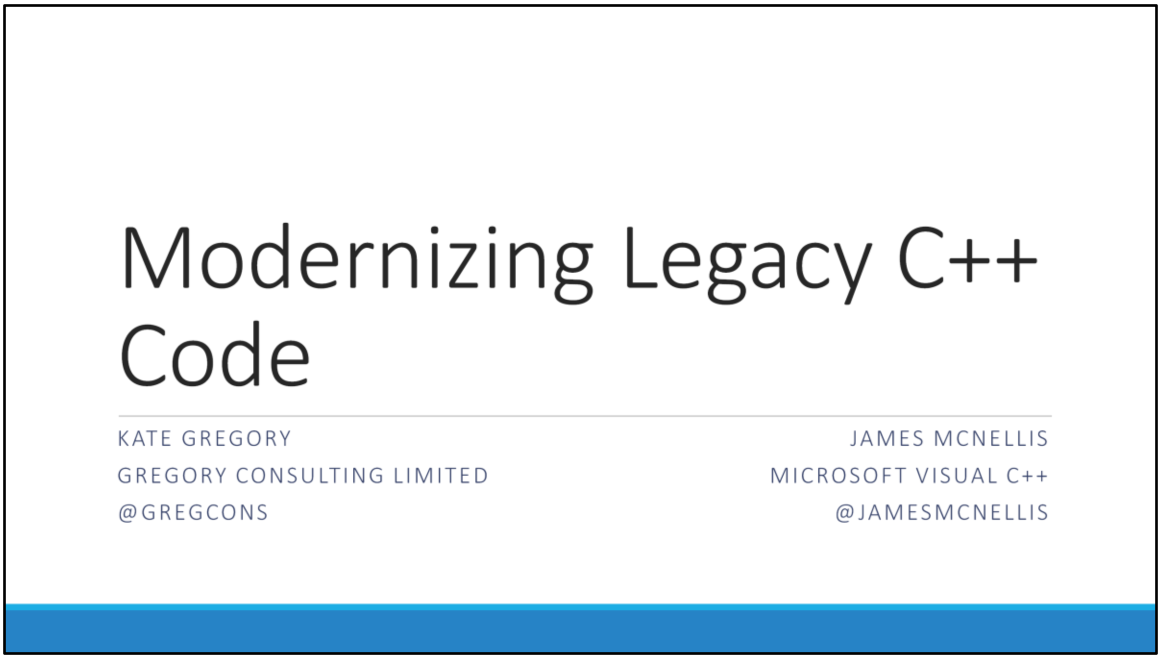


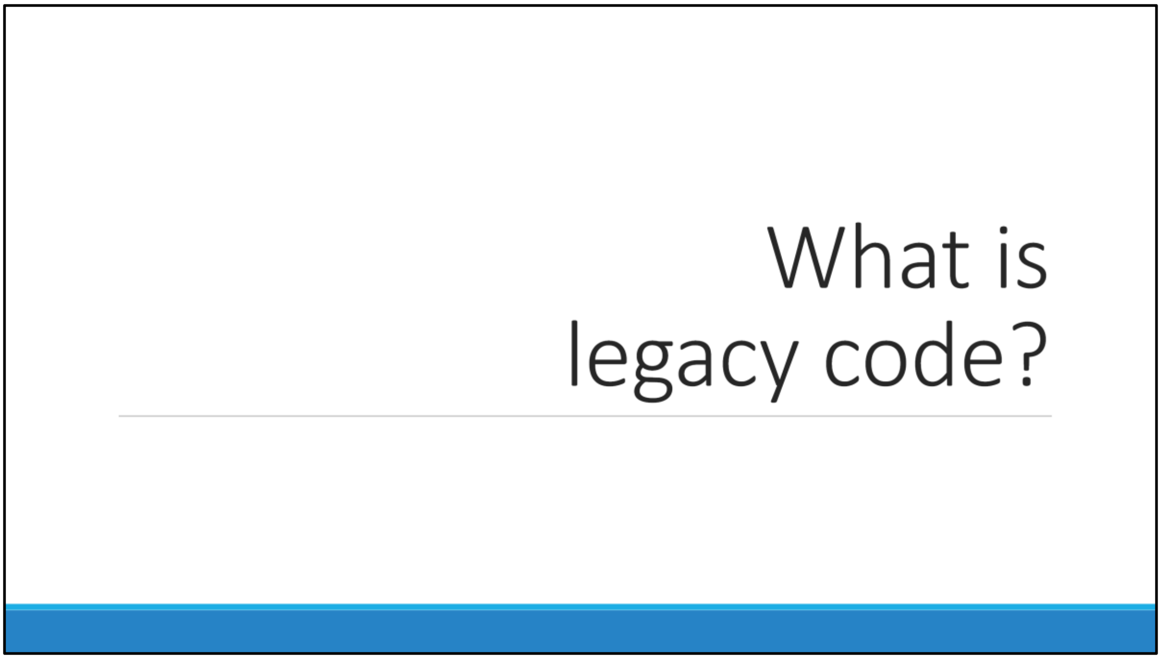
Modernizing Legacy C++ Code

KATE GREGORY
GREGORY CONSULTING LIMITED
@GREGCONS

JAMES MCNELLIS
MICROSOFT VISUAL C++
@JAMESMCNELLIS

INTRO

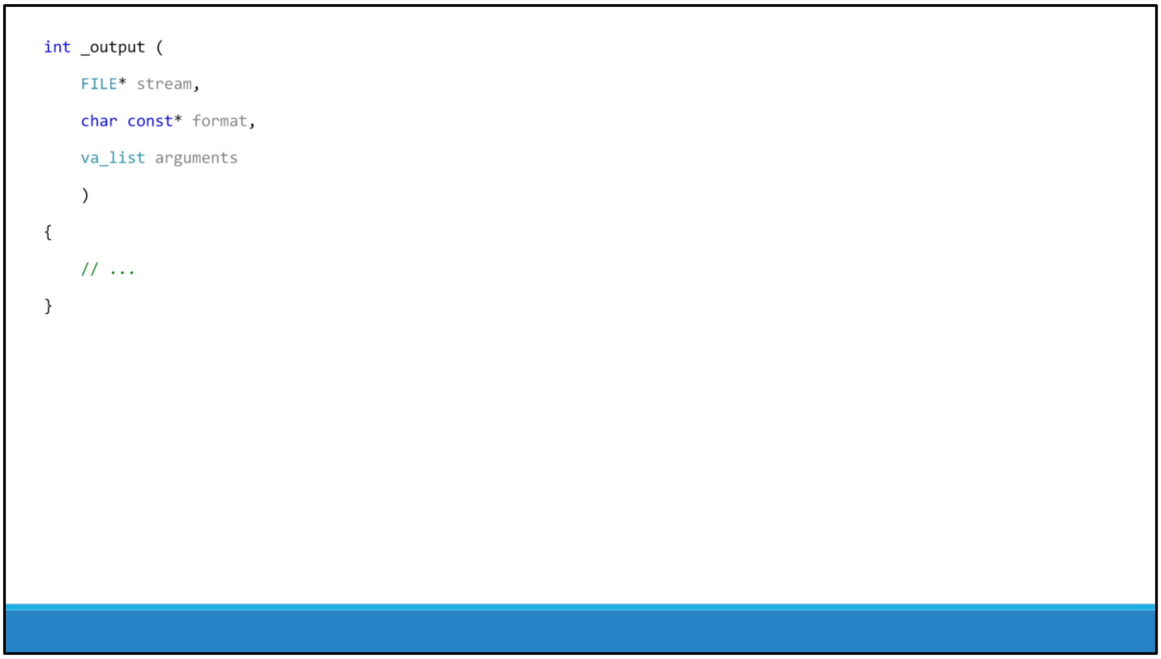
NEXT: What is legacy code?



What is legacy code?

Some people use the name “Legacy Code” to mean code that doesn’t have any tests in it. And certainly it would be a lot easier to refactor and generally mess around with your code if you had the safety net of a suite of tests, whether code based or a human running the app, to confirm that you didn’t break it when you changed it. That’s not what this talk is about, though: this talk is about taking a big unreadable unmaintainable mess, which may have been created that way because the language didn’t support what you really wanted to do, and turning it into something more modern.

NEXT: First example, _output

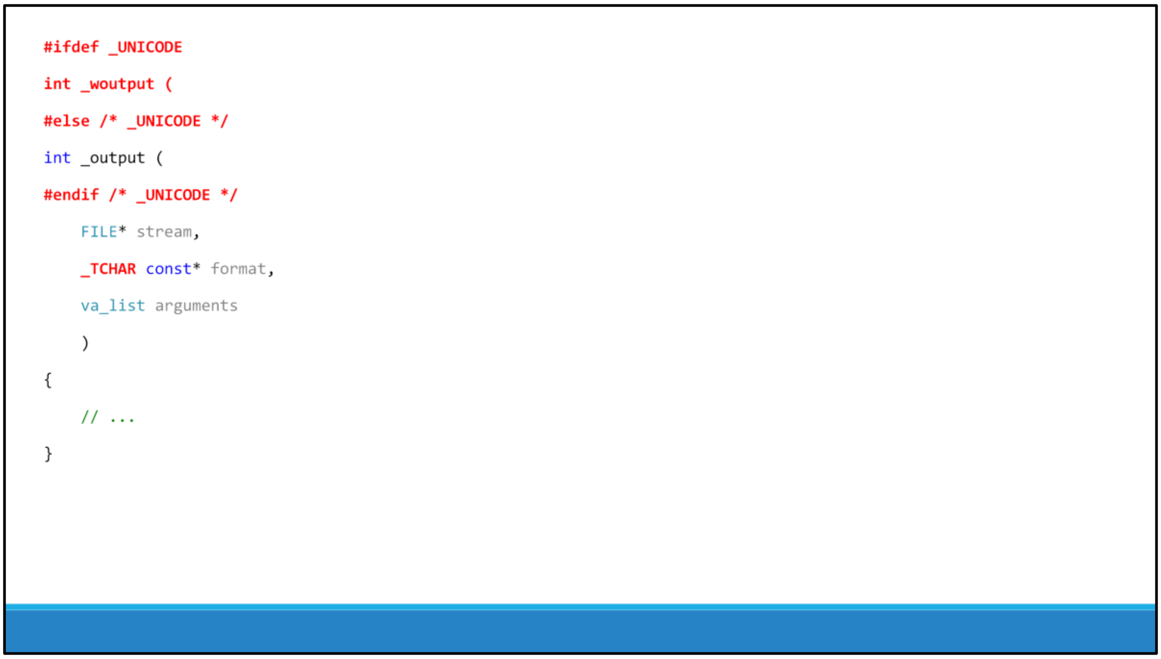


```
int _output (
    FILE* stream,
    char const* format,
    va_list arguments
    )
{
    // ...
}
```

From output.c in the Visual Studio 2013 CRT sources

Example from our C Standard Library
_output function has core implementation for printf, fprintf, sprintf, etc.
We ship the source with Visual Studio
Look at the evolution of just the function declaration
We start here…nice and simple
Someone decides we need to add Unicode (wchar_t)…

NEXT: Add Unicode functions

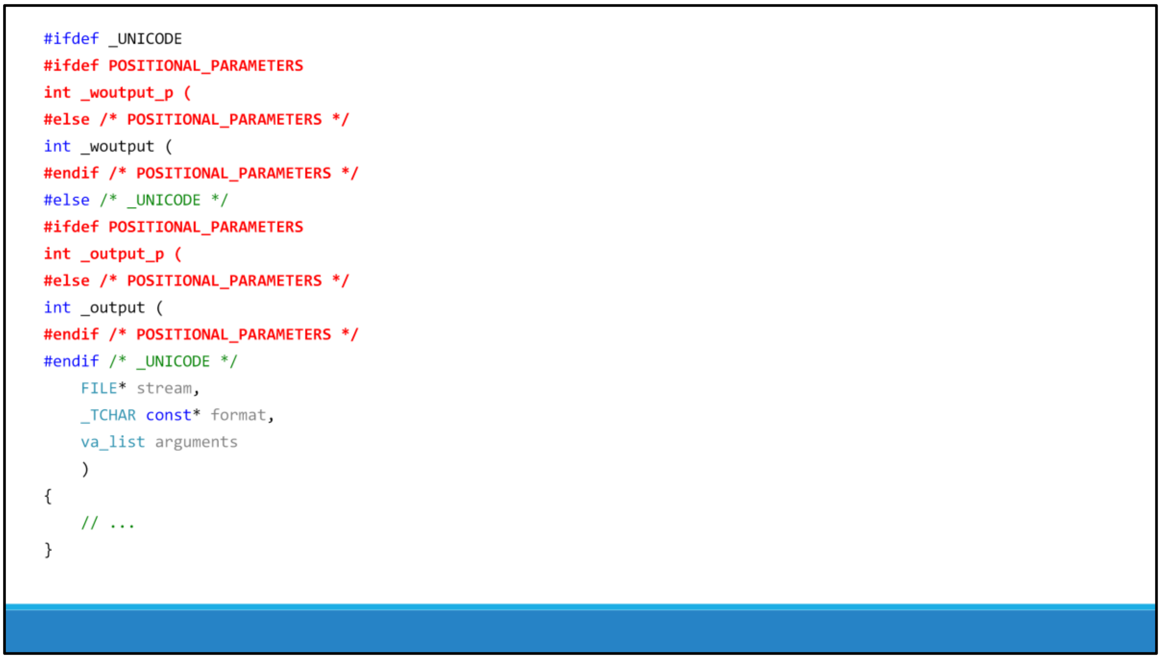


```
#ifdef _UNICODE
int _woutput (
#else /* _UNICODE */
int _output (
#endif /* _UNICODE */
    FILE* stream,
    _TCHAR const* format,
    va_list arguments
    )
{
    // ...
}
```

…so the function is changed
We compile the file twice now, once with UNICODE, once without
We update the definition to use _TCHAR where required
A bunch of #ifdef UNICODE blocks in the function too

Then someone says “we should support positional parameters…

NEXT: Positional parameters added

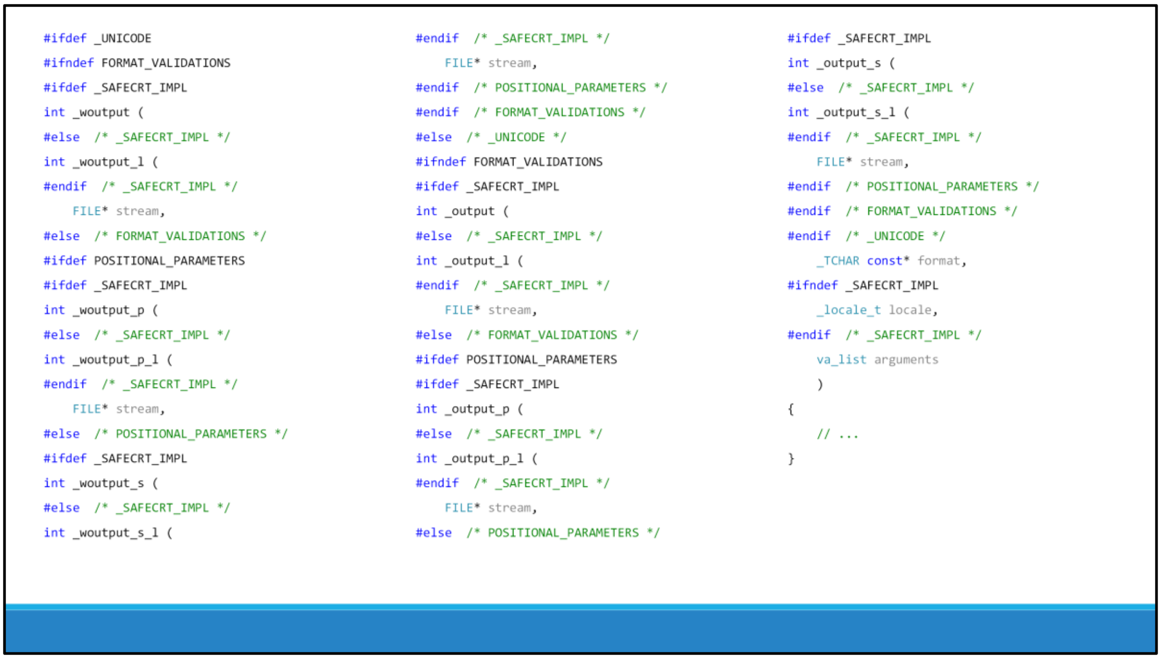


```
#ifdef _UNICODE
#ifdef POSITIONAL_PARAMETERS
int _woutput_p (
#else /* POSITIONAL_PARAMETERS */
int _woutput (
#endif /* POSITIONAL_PARAMETERS */
#else /* _UNICODE */
#ifdef POSITIONAL_PARAMETERS
int _output_p (
#else /* POSITIONAL_PARAMETERS */
int _output (
#endif /* POSITIONAL_PARAMETERS */
#endif /* _UNICODE */
    FILE* stream,
    _TCHAR const* format,
    va_list arguments
    )
{
    // ...
}
```

…So we do that too
Add _p versions of each of these functions, and throughout the file we add more #ifdef blocks

Then, after a couple more iterations, the declaration (JUST the declaration) looks like this…

NEXT: Entire declaration



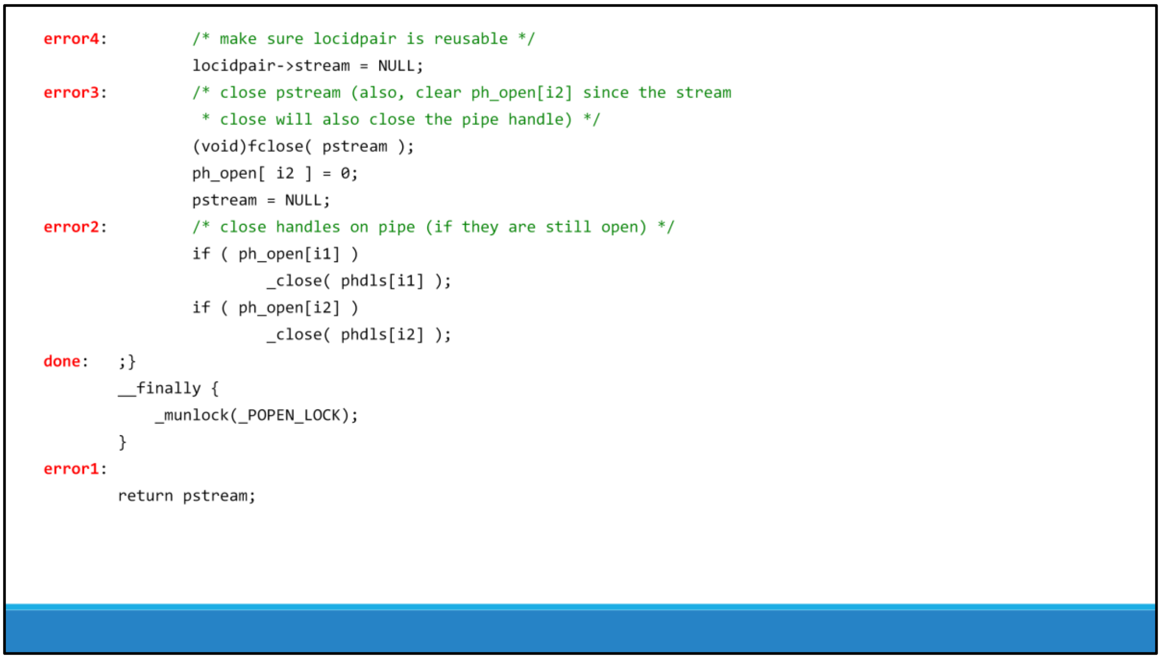
```
#ifdef _UNICODE
#ifndef FORMAT_VALIDATIONS
#ifdef _SAFECRT_IMPL
int _woutput (
#else  /* _SAFECRT_IMPL */
int _woutput_l (
#endif  /* _SAFECRT_IMPL */
    FILE* stream,
#else  /* FORMAT_VALIDATIONS */
#ifdef POSITIONAL_PARAMETERS
#ifdef _SAFECRT_IMPL
int _woutput_p (
#else  /* _SAFECRT_IMPL */
int _woutput_p_l (
#endif  /* _SAFECRT_IMPL */
    FILE* stream,
#else  /* POSITIONAL_PARAMETERS */
#ifdef _SAFECRT_IMPL
int _woutput_s (
#else  /* _SAFECRT_IMPL */
int _woutput_s_l (
#endif  /* _SAFECRT_IMPL */
    FILE* stream,
#endif  /* POSITIONAL_PARAMETERS */
#endif  /* FORMAT_VALIDATIONS */
#else  /* _UNICODE */
#ifndef FORMAT_VALIDATIONS
#ifdef _SAFECRT_IMPL
int _output (
#else  /* _SAFECRT_IMPL */
int _output_l (
#endif  /* _SAFECRT_IMPL */
    FILE* stream,
#else  /* FORMAT_VALIDATIONS */
#ifdef POSITIONAL_PARAMETERS
#ifdef _SAFECRT_IMPL
int _output_p (
#else  /* _SAFECRT_IMPL */
int _output_p_l (
#endif  /* _SAFECRT_IMPL */
    FILE* stream,
#else  /* POSITIONAL_PARAMETERS */
#ifdef _SAFECRT_IMPL
int _output_s (
#else  /* _SAFECRT_IMPL */
int _output_s_l (
#endif  /* _SAFECRT_IMPL */
    FILE* stream,
#endif  /* POSITIONAL_PARAMETERS */
#endif  /* FORMAT_VALIDATIONS */
#endif  /* _UNICODE */
    _TCHAR const* format,
#ifndef _SAFECRT_IMPL
    _locale_t locale,
#endif  /* _SAFECRT_IMPL */
    va_list arguments
    )
{
    // ...
}
```

Ugh.
You can't even at a glance figure out which functions take which parameters.
This is only 2/3 of the declarations too (12 here; 6 more).

This is a 2700 line file
This one function spans 1500 lines
There are 223 conditionally compiled blocks

In short, this code gradually became impossible to maintain
Maintenance difficulty prevented us from adding features in VS2013
Unnoticed performance issues were lurking

NEXT: Another completely different example; popen, gotos

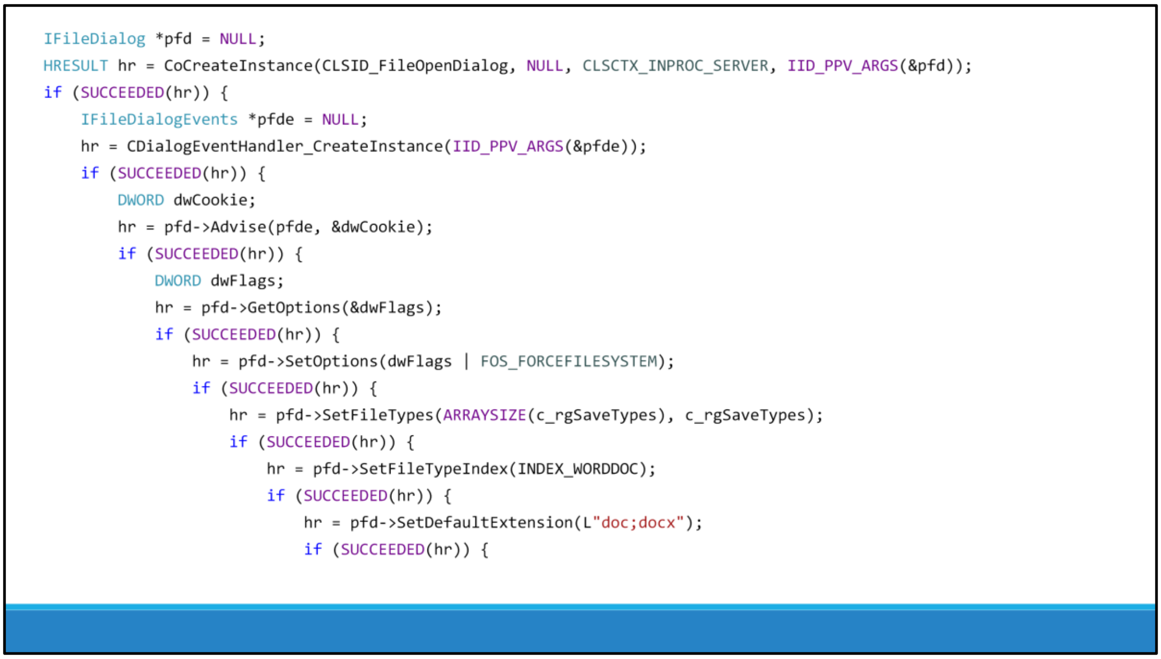


```
error4:         /* make sure locidpair is reusable */
                locidpair->stream = NULL;
error3:         /* close pstream (also, clear ph_open[i2] since the stream
                 * close will also close the pipe handle) */
                (void)fclose( pstream );
                ph_open[ i2 ] = 0;
                pstream = NULL;
error2:         /* close handles on pipe (if they are still open) */
                if ( ph_open[i1] )
                        _close( phdls[i1] );
                if ( ph_open[i2] )
                        _close( phdls[i2] );
done:   ;}
        __finally {
            _munlock(_POPEN_LOCK);
        }
error1:
        return pstream;
```

From popen.c in the Visual Studio 2013 CRT sources

popen starts up a child process with its I/O attached to a pipe in the calling process
Reading our implementation, we get to the end and discover this lovely code…
…and throughout the function there are a bunch of gotos

NEXT: An example from MSDN



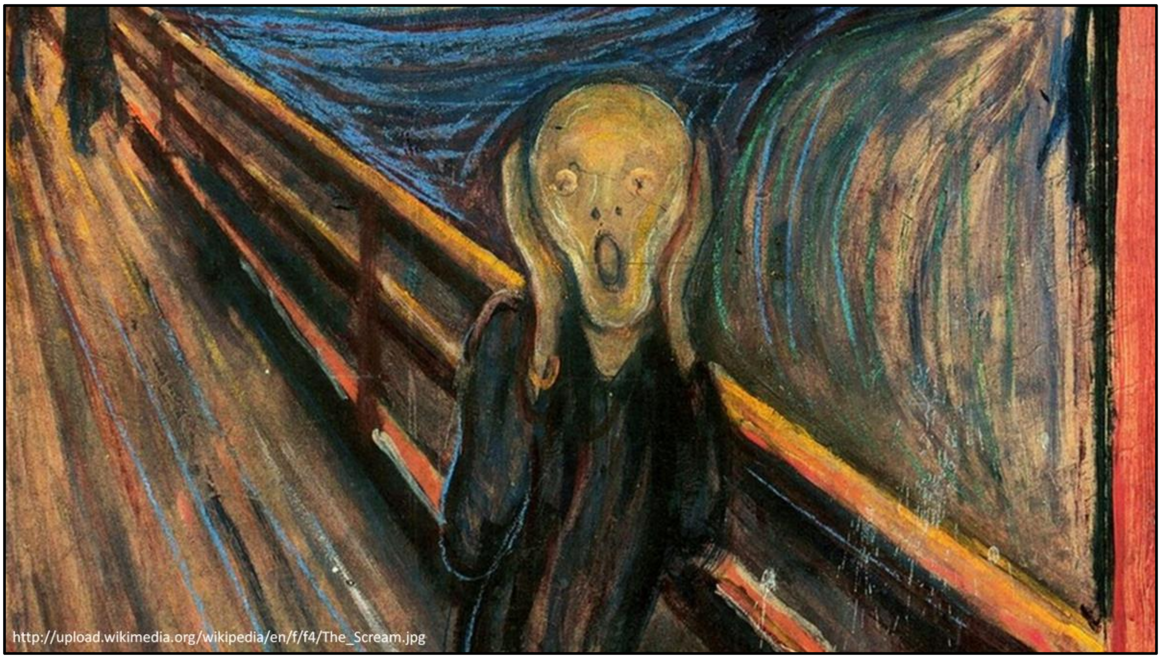
```
IFileDialog *pfd = NULL;
HRESULT hr = CoCreateInstance(CLSID_FileOpenDialog, NULL, CLSCTX_INPROC_SERVER, IID_PPV_ARGS(&pfd));
if (SUCCEEDED(hr)) {
    IFileDialogEvents *pfde = NULL;
    hr = CDialogEventHandler_CreateInstance(IID_PPV_ARGS(&pfde));
    if (SUCCEEDED(hr)) {
        DWORD dwCookie;
        hr = pfd->Advise(pfde, &dwCookie);
        if (SUCCEEDED(hr)) {
            DWORD dwFlags;
            hr = pfd->GetOptions(&dwFlags);
            if (SUCCEEDED(hr)) {
                hr = pfd->SetOptions(dwFlags | FOS_FORCEFILESYSTEM);
                if (SUCCEEDED(hr)) {
                    hr = pfd->SetFileTypes(ARRAYSIZE(c_rgSaveTypes), c_rgSaveTypes);
                    if (SUCCEEDED(hr)) {
                        hr = pfd->SetFileTypeIndex(INDEX_WORDDOC);
                        if (SUCCEEDED(hr)) {
                            hr = pfd->SetDefaultExtension(L"doc;docx");
                            if (SUCCEEDED(hr)) {
```

From http://msdn.microsoft.com/en-us/library/windows/desktop/bb776913(v=vs.85).aspx

This is sample code from MSDN
Fairly recent
Not as bad as the other two
But terrible C++ practice
“Arrow code”
Not using RAII, so we can’t make the code linear

My reaction when I find code like these examples is something like…

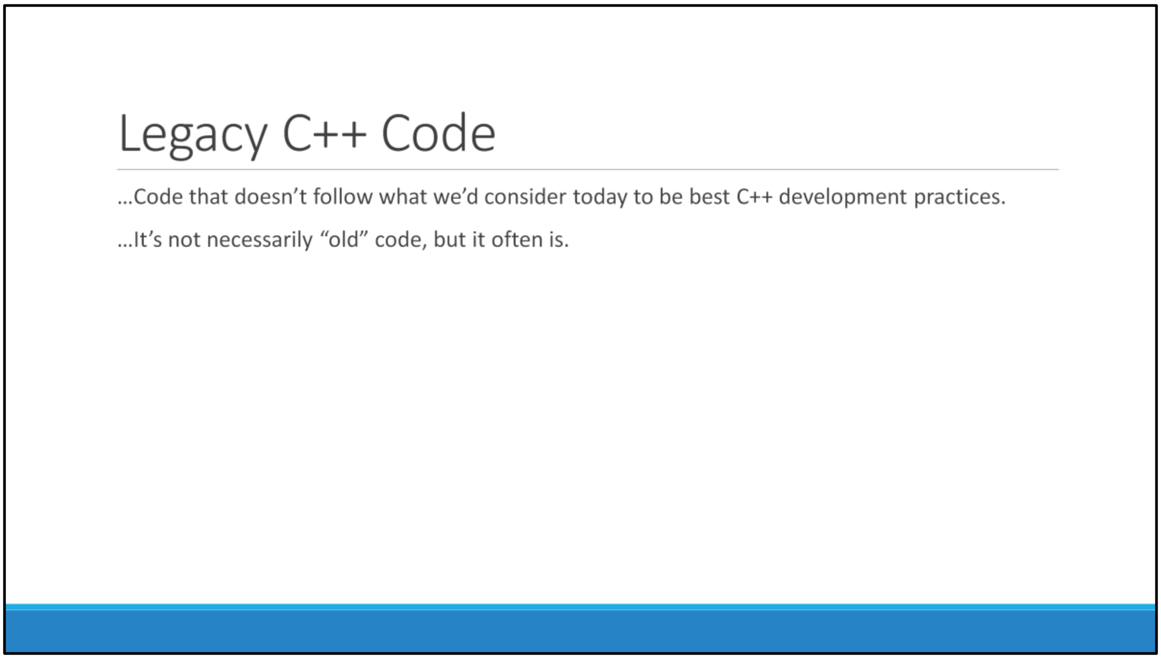
NEXT: The Scream



http://upload.wikimedia.org/wikipedia/en/f/f4/The_Scream.jpg

…this.

NEXT: Legacy C++ Code definition



Legacy C++ Code

…Code that doesn’t follow what we’d consider today to be best C++ development practices.

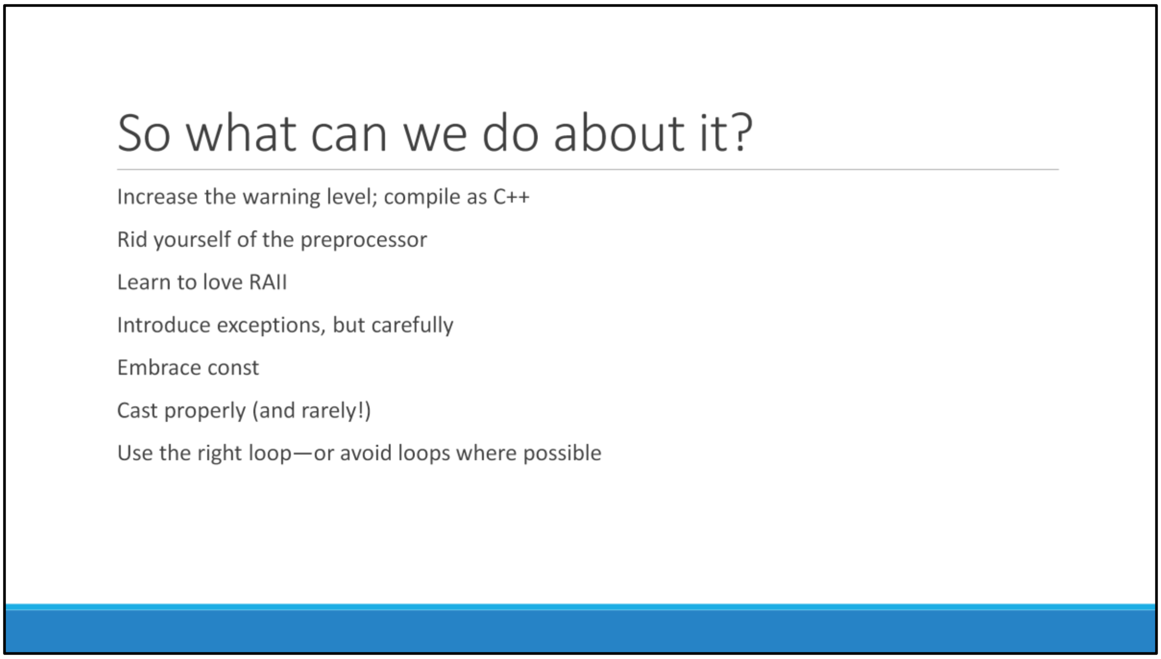
…It’s not necessarily “old” code, but it often is.

For the purposes of this talk, define “legacy code” broadly

Perhaps it…
…overuses the preprocessor or
…doesn’t use RAII

Why change it?
“If it isn’t broken don’t fix it?”
…Update code during regular maintenance
…Update code if its design/impl is harming other development
…Audit critical components regularly
…From time to time consider a major overhaul

NEXT: Resources



So what can we do about it?

Increase the warning level; compile as C++

Rid yourself of the preprocessor

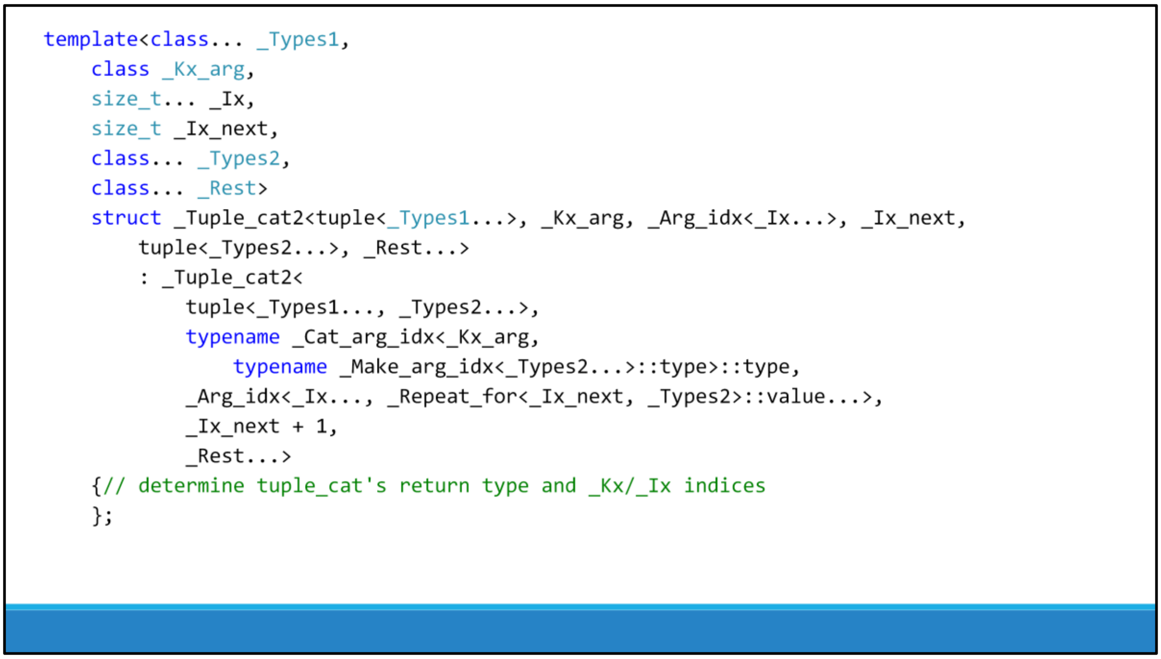
Learn to love RAII

Introduce exceptions, but carefully

Embrace const

Cast properly (and rarely!)

Use the right loop—or avoid loops where possible



```
template<class... _Types1,
    class _Kx_arg,
    size_t... _Ix,
    size_t _Ix_next,
    class... _Types2,
    class... _Rest>
    struct _Tuple_cat2<tuple<_Types1...>, _Kx_arg, _Arg_idx<_Ix...>, _Ix_next,
        tuple<_Types2...>, _Rest...>
        : _Tuple_cat2<
            tuple<_Types1..., _Types2...>,
            typename _Cat_arg_idx<_Kx_arg,
                typename _Make_arg_idx<_Types2...>::type>::type,
            _Arg_idx<_Ix..., _Repeat_for<_Ix_next, _Types2>::value...>,
            _Ix_next + 1,
            _Rest...>
    {// determine tuple_cat's return type and _Kx/_Ix indices
    };
```

C++11 and C++14 add a ton of great new features to C++
…lambdas, variadic templates, rvalue references

But almost everything we discuss here will work even with C++03
“Modernization” doesn’t mean turning your codebase into a terrifying template mess

This example from our implementation of std::tuple
(Libraries, especially C++ Standard Library, are where extensive C++11 use is often beneficial and required.)

NEXT: SECTION 1: If you do nothing else…

This code is here as an example of what we are NOT suggesting you write. It’s from a library, and it’s entirely appropriate for library code, but it’s not simple to read or understand, and it has a lot in common with the wall of code examples from earlier in the talk. We are not trying to lead you to this type of code, quite the opposite.

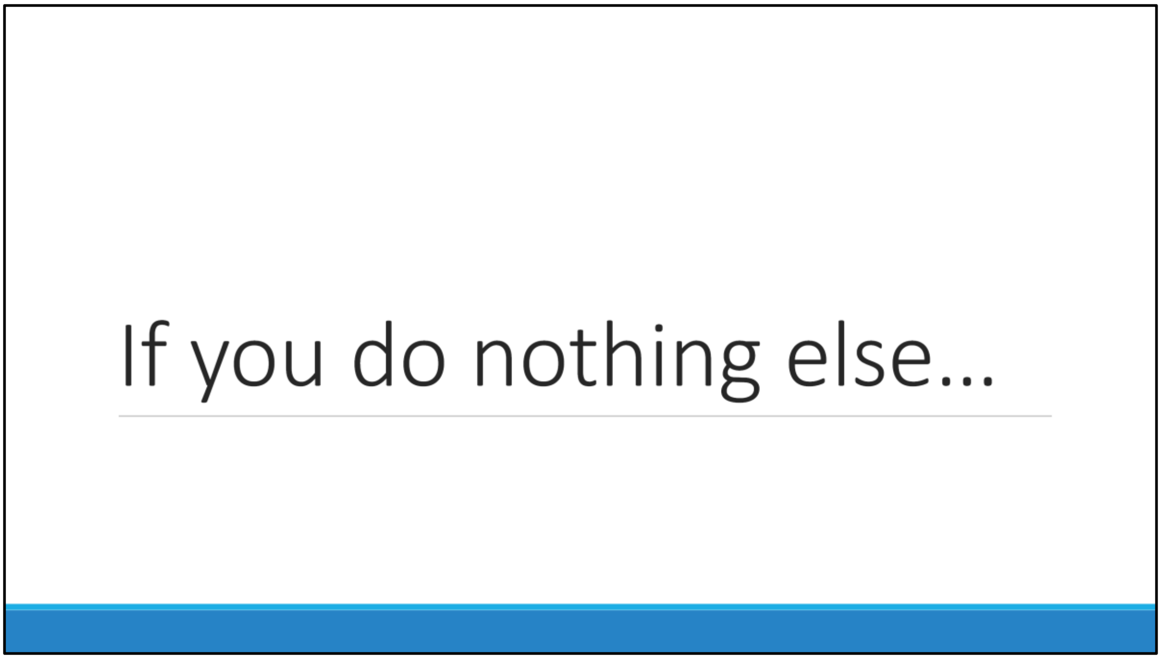
Resources

REFACTORING
IMPROVING THE DESIGN OF EXISTING CODE
MARTIN FOWLER
With Contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts
Foreword by Erich Gamma
Object Technology International Inc.
BOOCH JACOBSON RUMBAUGH
Robert C. Martin Series
WORKING EFFECTIVELY WITH LEGACY CODE
Michael C. Feathers
The Pragmatic Programmers
Modern C++ Programming with Test-Driven Development
Code Better, Sleep Better
Jeff Langr
Foreword by Robert C. Martin (Uncle Bob)
Edited by Michael Swaine

These are a few great resources for general refactoring techniques
This talk: only a few C++-specific techniques, mostly.
…Smorgasbord of suggestions to improve code

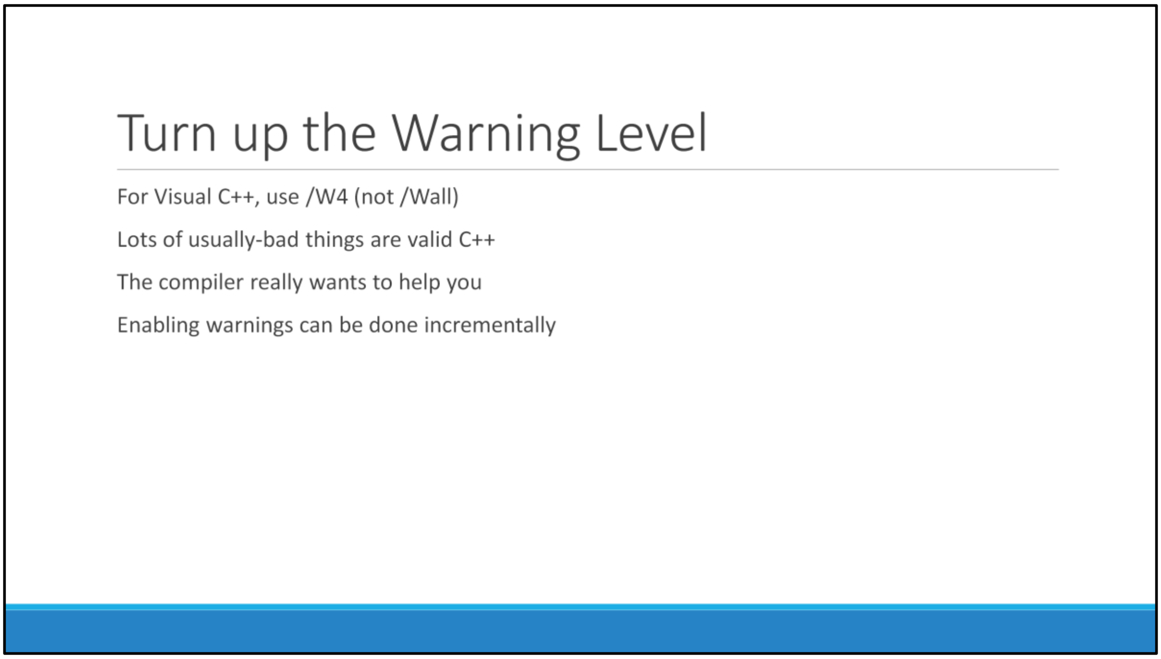
All available at the bookstore.

NEXT: BASICS TITLE



If you do nothing else…

Switch to Kate



Turn up the Warning Level

For Visual C++, use /W4 (not /Wall)

Lots of usually-bad things are valid C++

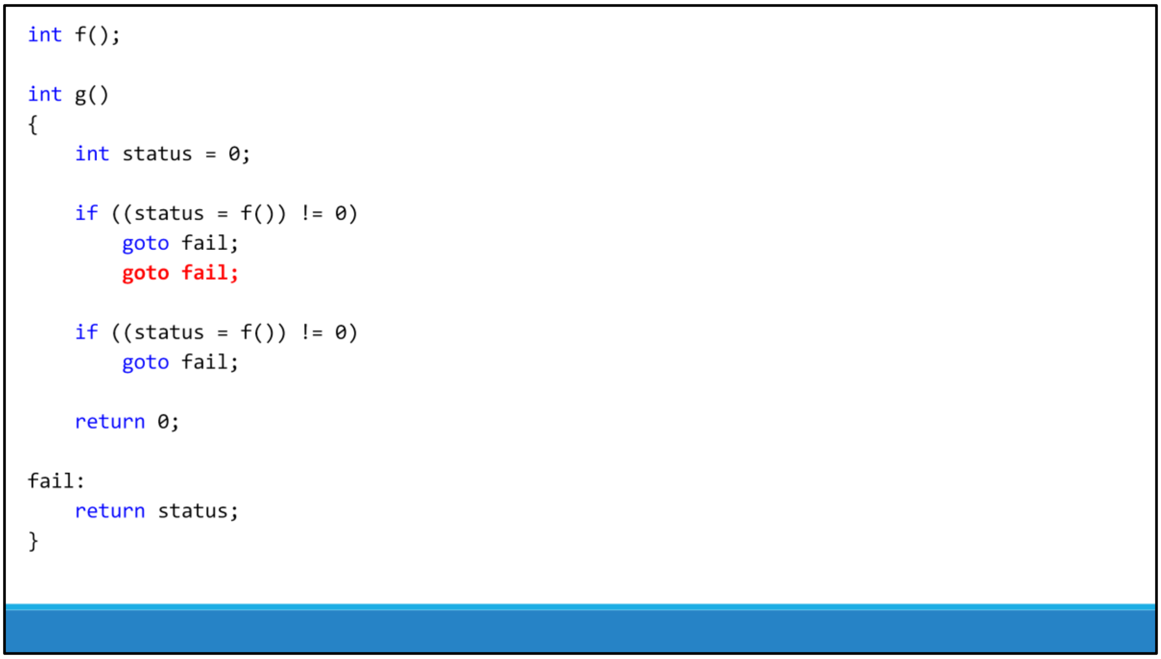
The compiler really wants to help you

Enabling warnings can be done incrementally

“it’s your foot” and “the compiler is your friend”

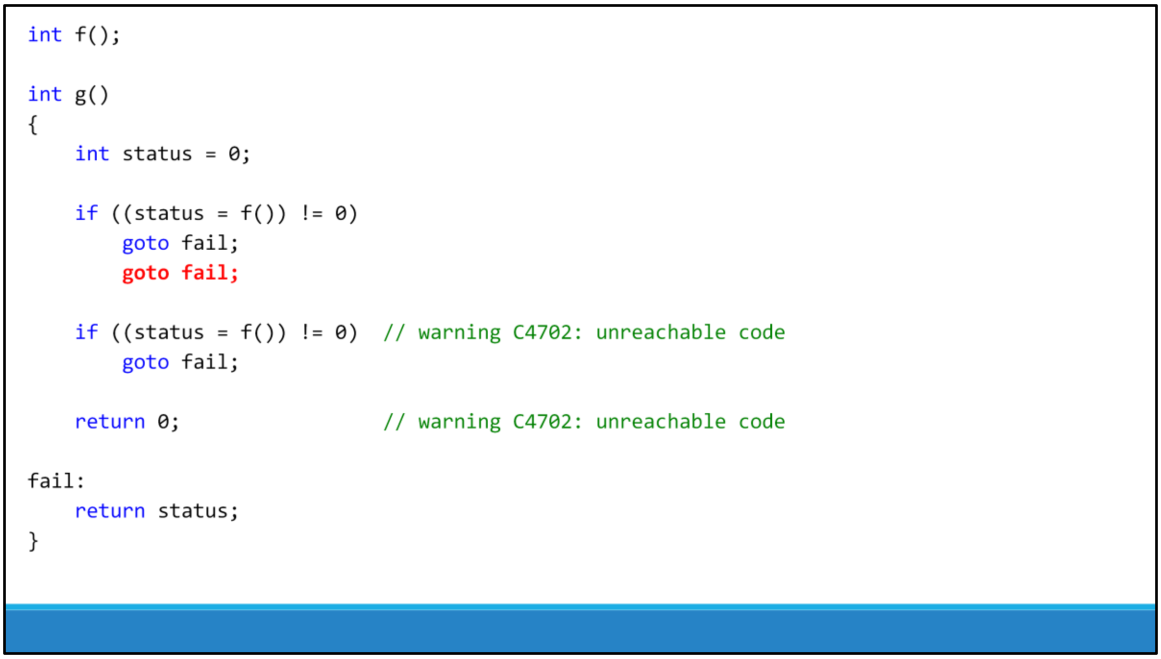
Warnings-as-errors – all the cool kids do it, but wait until you’re compiling clean on /W4 before you turn it on

Leaving artifacts in your code (pragmas, casts, etc) to make warnings go away help others to understand that whatever you’re doing here, you’re doing deliberately



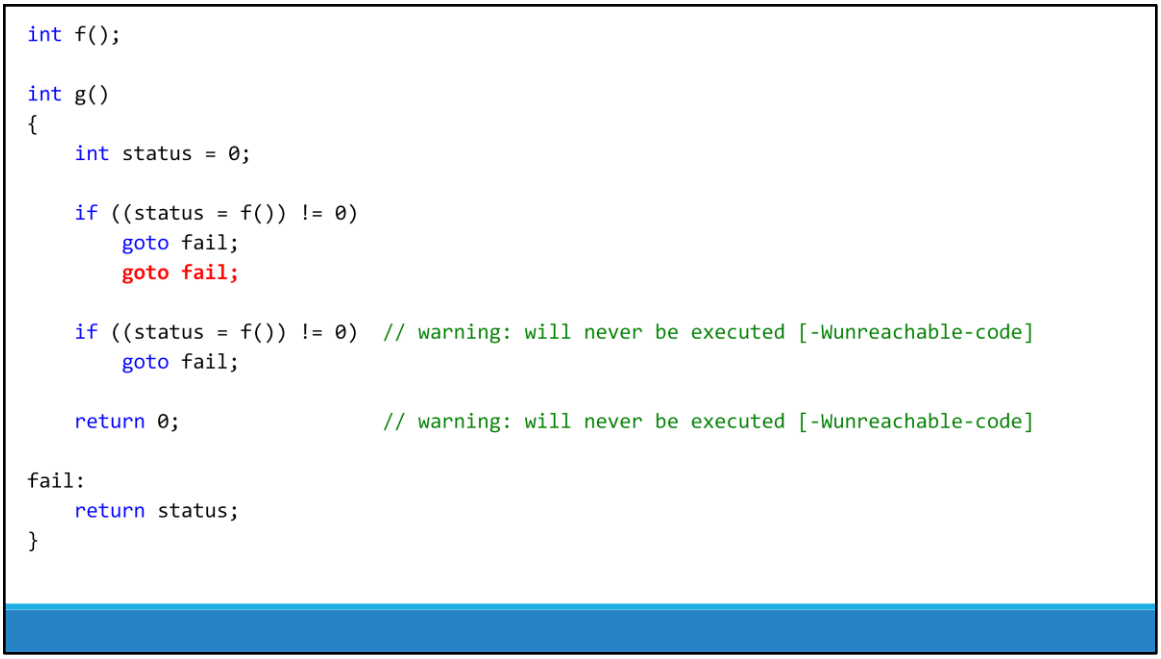
```
int f();

int g()
{
    int status = 0;

    if ((status = f()) != 0)
        goto fail;
        goto fail;

    if ((status = f()) != 0)
        goto fail;

    return 0;

fail:
    return status;
}
```

the compiler doesn’t read indents – but we do



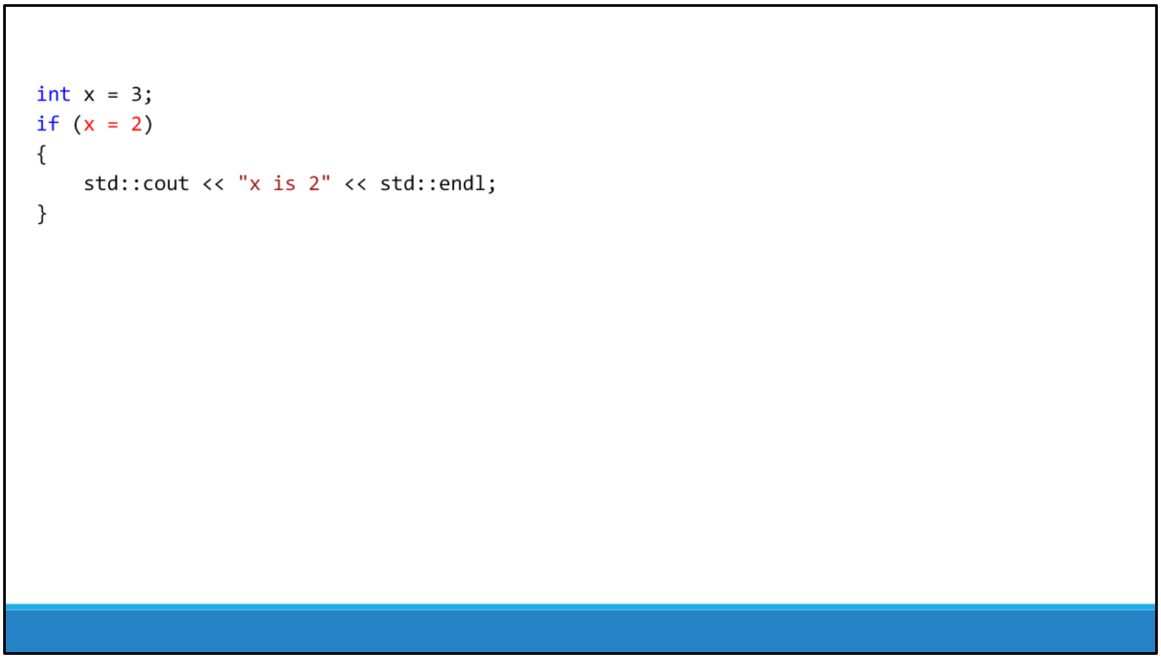
```
int f();

int g()
{
    int status = 0;

    if ((status = f()) != 0)
        goto fail;
        goto fail;

    if ((status = f()) != 0)  // warning C4702: unreachable code
        goto fail;

    return 0;                 // warning C4702: unreachable code

fail:
    return status;
}
```

Visual C++ warns

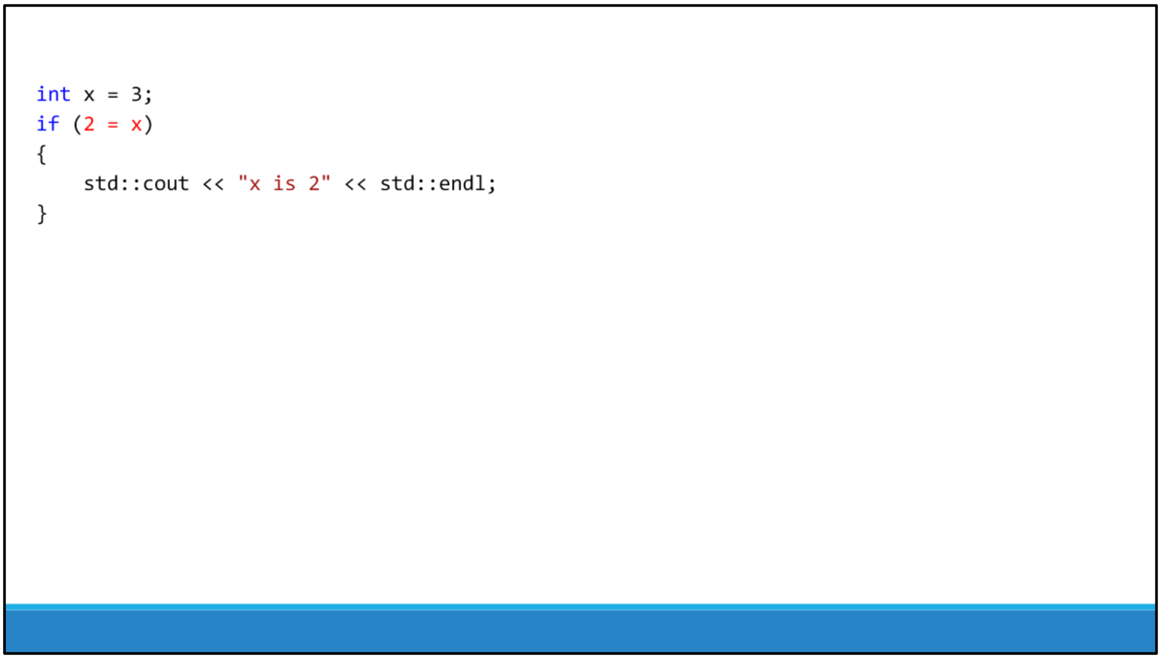


```
int f();

int g()
{
    int status = 0;

    if ((status = f()) != 0)
        goto fail;
        goto fail;

    if ((status = f()) != 0)  // warning: will never be executed [-Wunreachable-code]
        goto fail;

    return 0;                 // warning: will never be executed [-Wunreachable-code]

fail:
    return status;
}
```

GCC and Clang also warn.

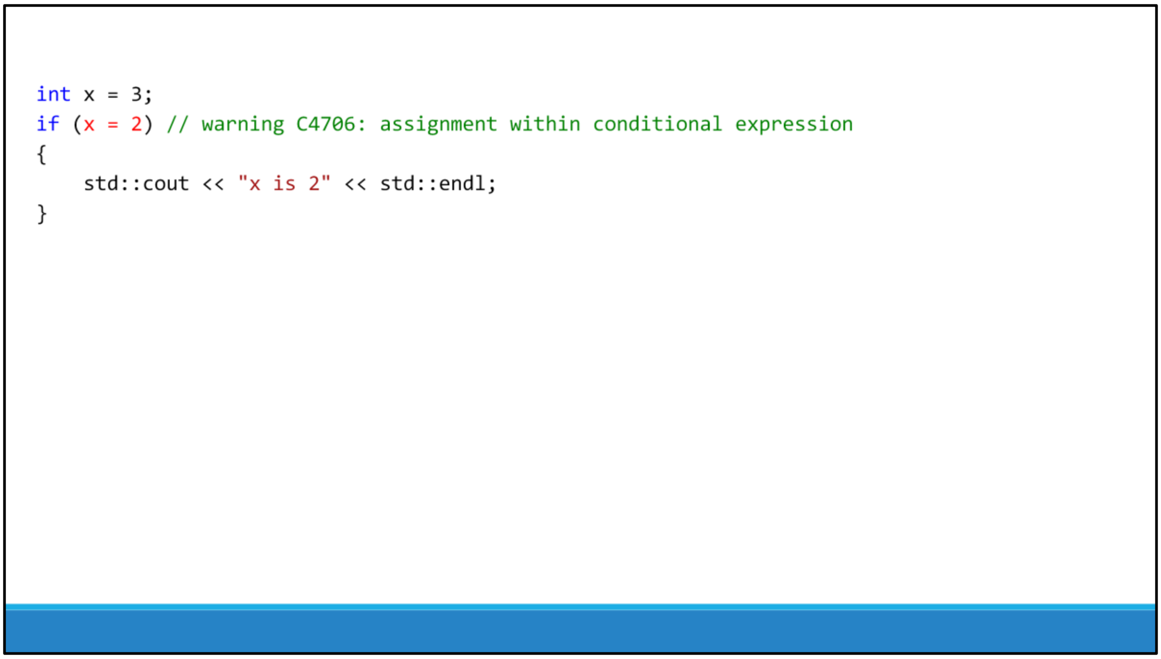


```cpp
int x = 3;
if (x = 2)
{
    std::cout << "x is 2" << std::endl;
}
```

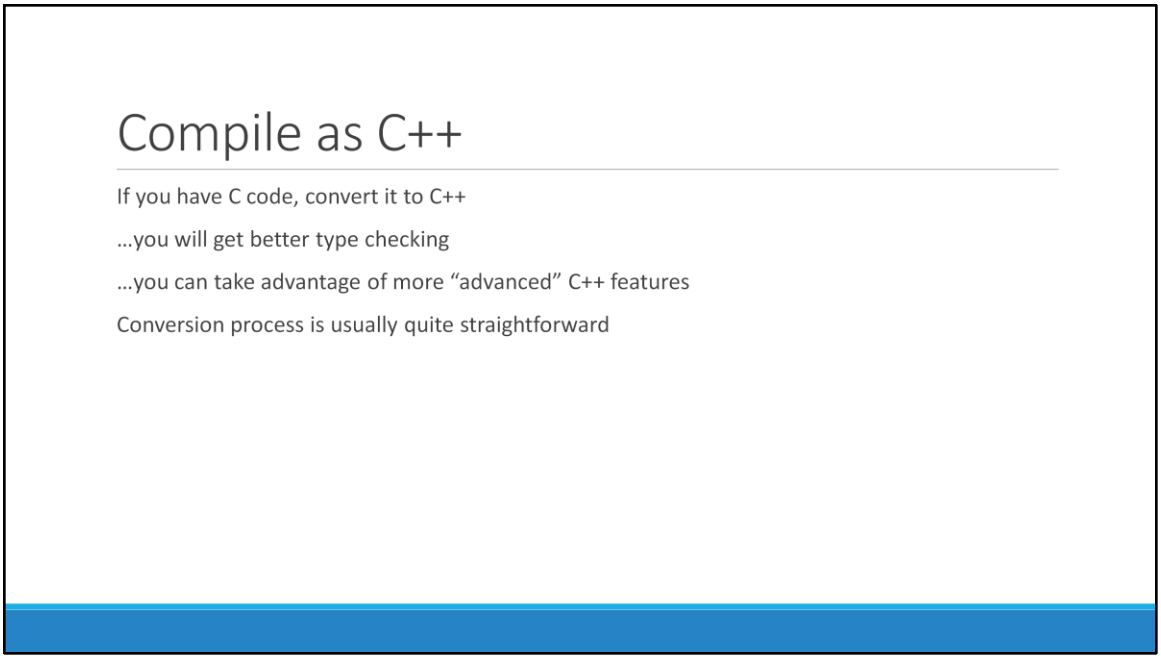


```cpp
int x = 3;
if (2 = x)
{
    std::cout << "x is 2" << std::endl;
}
```

Fortran joke, Yoda conditions



```
int x = 3;
if (x = 2) // warning C4706: assignment within conditional expression
{
    std::cout << "x is 2" << std::endl;
}
```



Compile as C++

If you have C code, convert it to C++

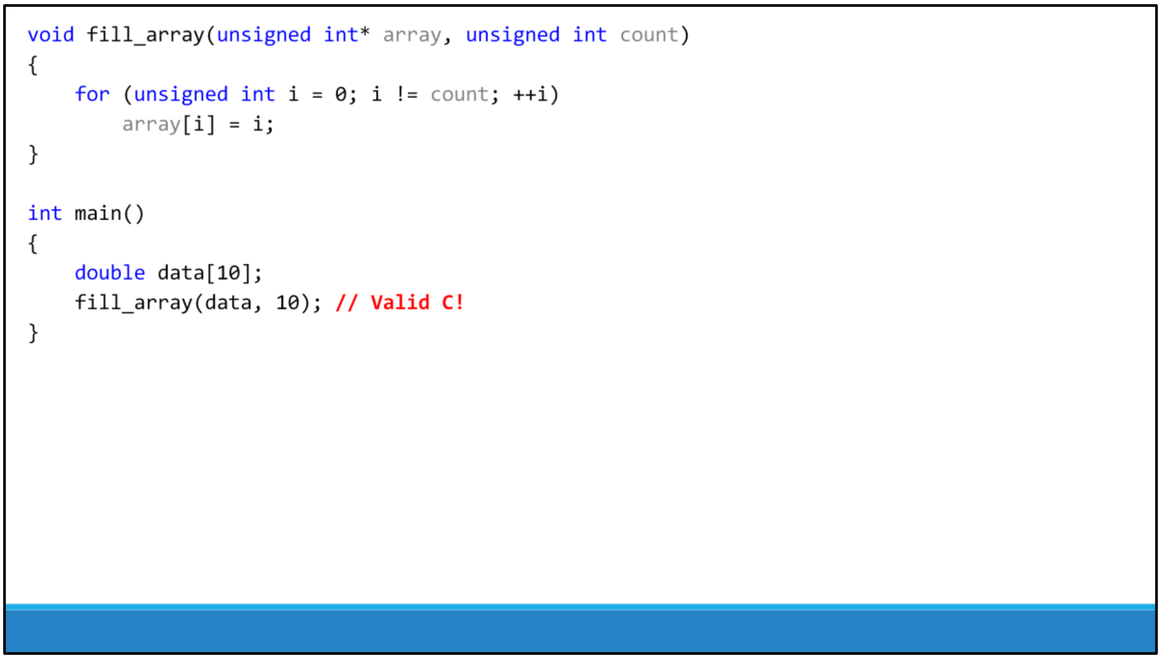
…you will get better type checking

…you can take advantage of more “advanced” C++ features

Conversion process is usually quite straightforward

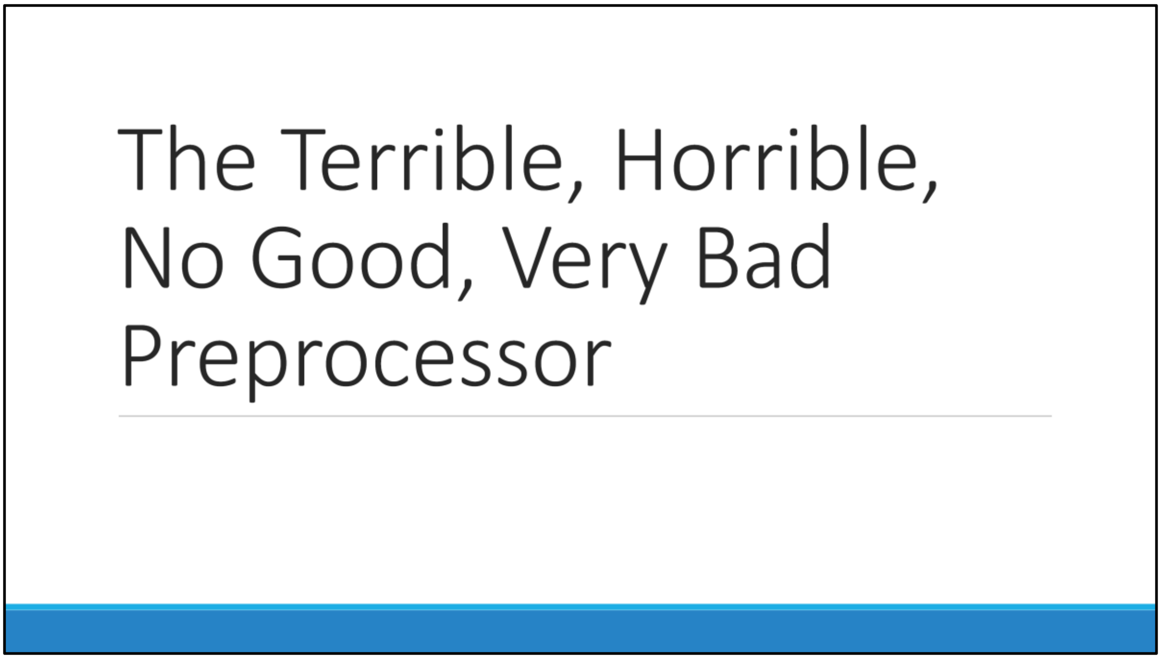
Advanced features eg templates ☺

The conversion process consists mainly of fixing errors you didn’t realize were there



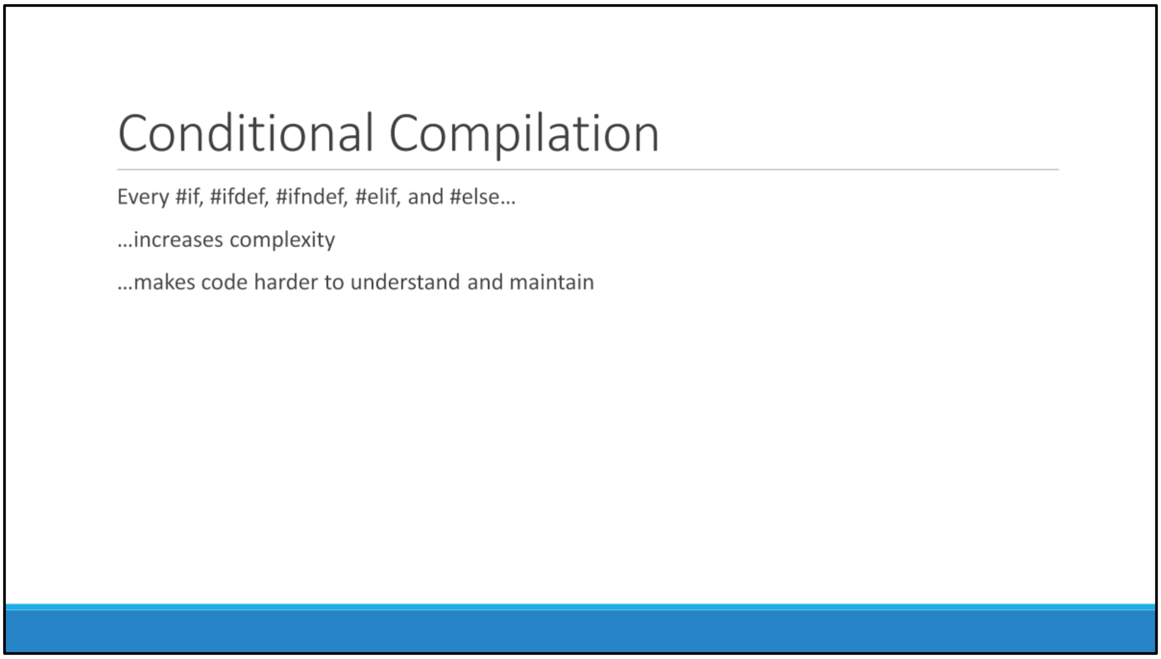
```
void fill_array(unsigned int* array, unsigned int count)
{
    for (unsigned int i = 0; i != count; ++i)
        array[i] = i;
}

int main()
{
    double data[10];
    fill_array(data, 10); // Valid C!
}
```

If your code was bug free with no errors, turning up the warning level and compiling as C++ would at least give you that knowledge about your code. The reality is for most people that doing this will turn up a TON of places you have changes to make. And while you're making those changes, you might want to clean up some other things, that compile fine but that are very hard to read and maintain. So let's talk about some of those now … hand to James.



The Terrible, Horrible, No Good, Very Bad Preprocessor

Switch to James

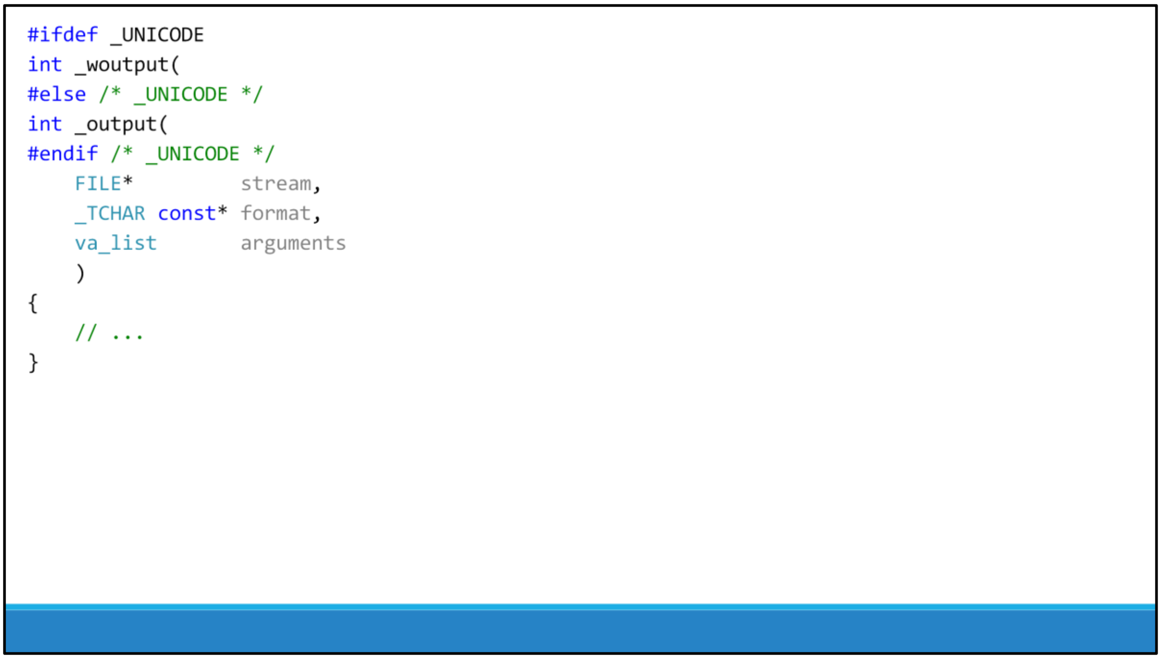


Conditional Compilation

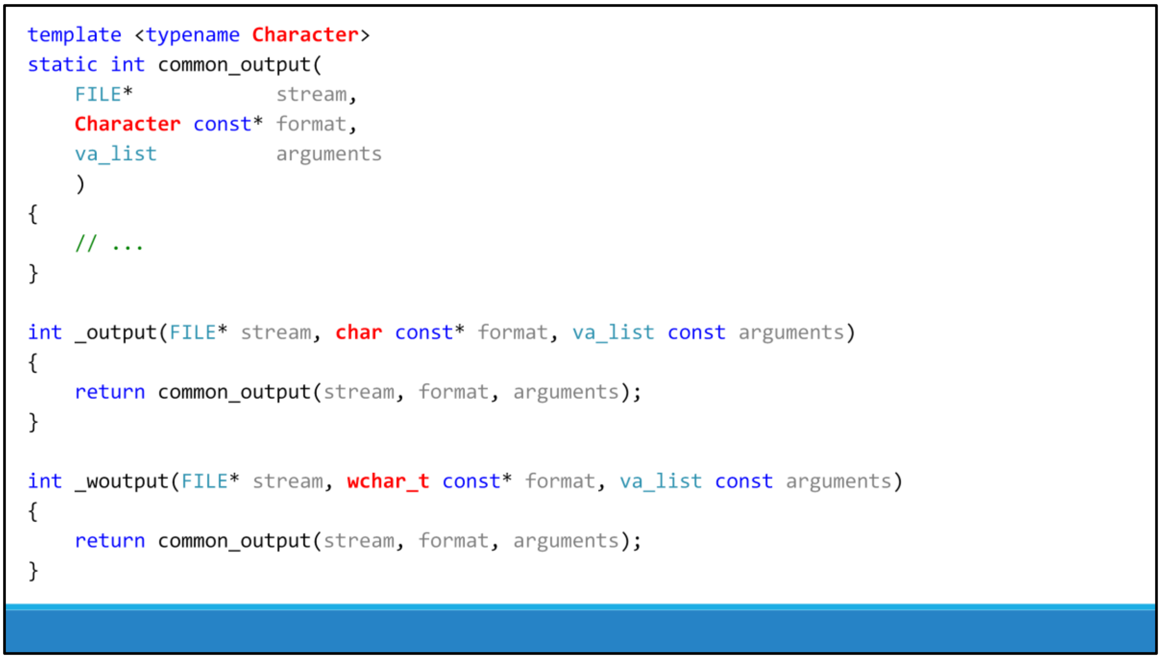
Every #if, #ifdef, #ifndef, #elif, and #else…

…increases complexity

…makes code harder to understand and maintain

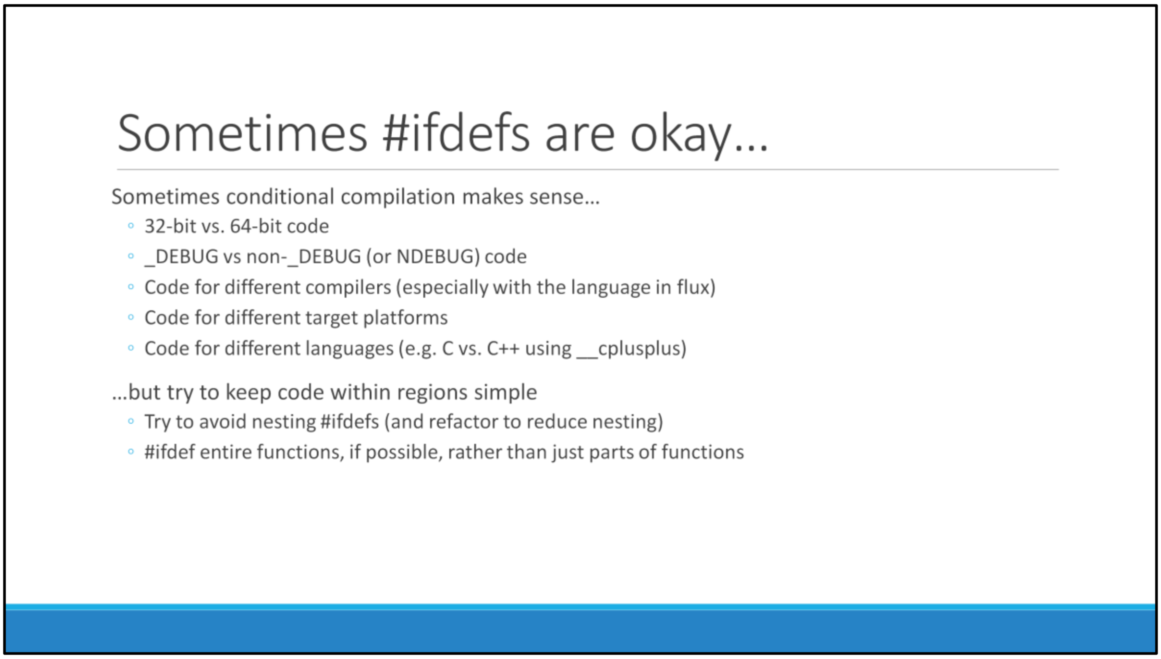


```
#ifdef _UNICODE
int _woutput(
#else /* _UNICODE */
int _output(
#endif /* _UNICODE */
    FILE*         stream,
    _TCHAR const* format,
    va_list       arguments
    )
{
    // ...
}
```



```
template <typename Character>
static int common_output(
    FILE*             stream,
    Character const* format,
    va_list           arguments
    )
{
    // ...
}

int _output(FILE* stream, char const* format, va_list const arguments)
{
    return common_output(stream, format, arguments);
}

int _woutput(FILE* stream, wchar_t const* format, va_list const arguments)
{
    return common_output(stream, format, arguments);
}
```

This makes it more obvious what is the same and what is different



Sometimes #ifdefs are okay…

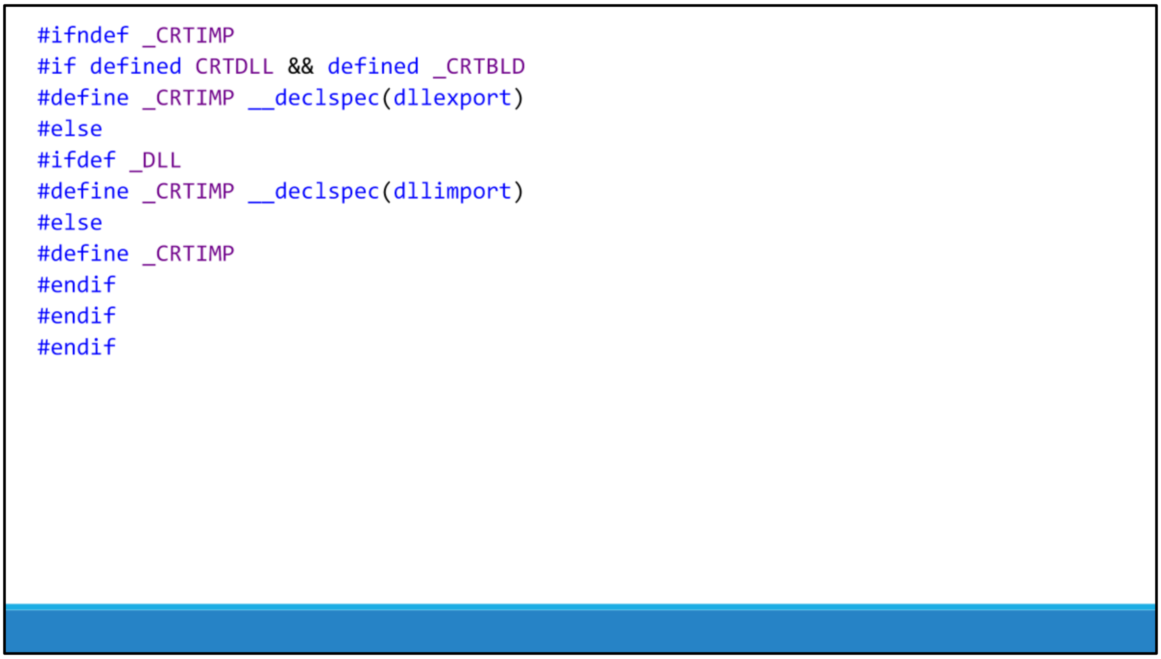
Sometimes conditional compilation makes sense…

- 32-bit vs. 64-bit code
- _DEBUG vs non-_DEBUG (or NDEBUG) code
- Code for different compilers (especially with the language in flux)
- Code for different target platforms
- Code for different languages (e.g. C vs. C++ using __cplusplus)

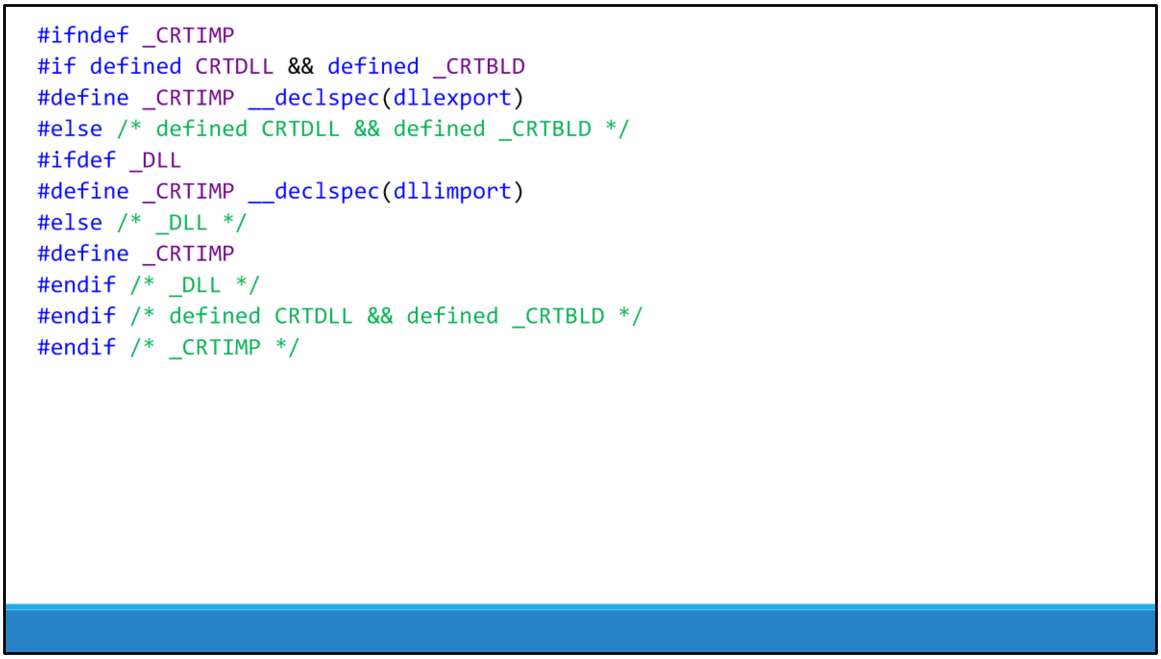
…but try to keep code within regions simple

- Try to avoid nesting #ifdefs (and refactor to reduce nesting)
- #ifdef entire functions, if possible, rather than just parts of functions

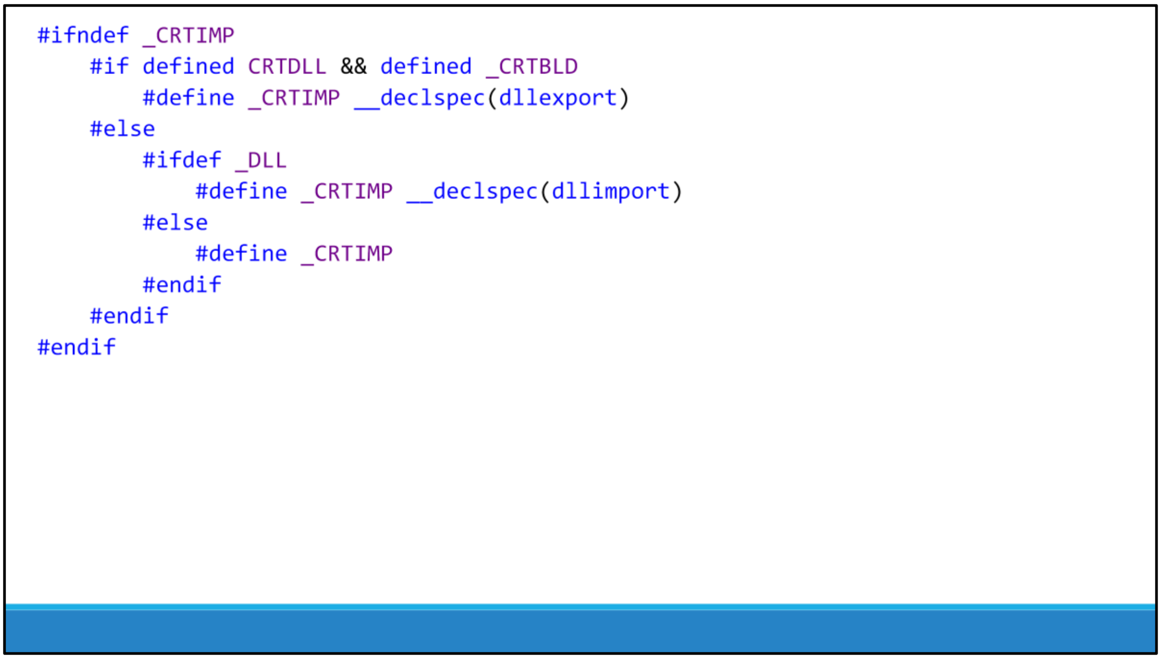
Nesting makes everything harder to follow. Comments can help but remember the compiler doesn’t read comments. Ifdeffing the entire function lets you indent normally to increase readability, for example.



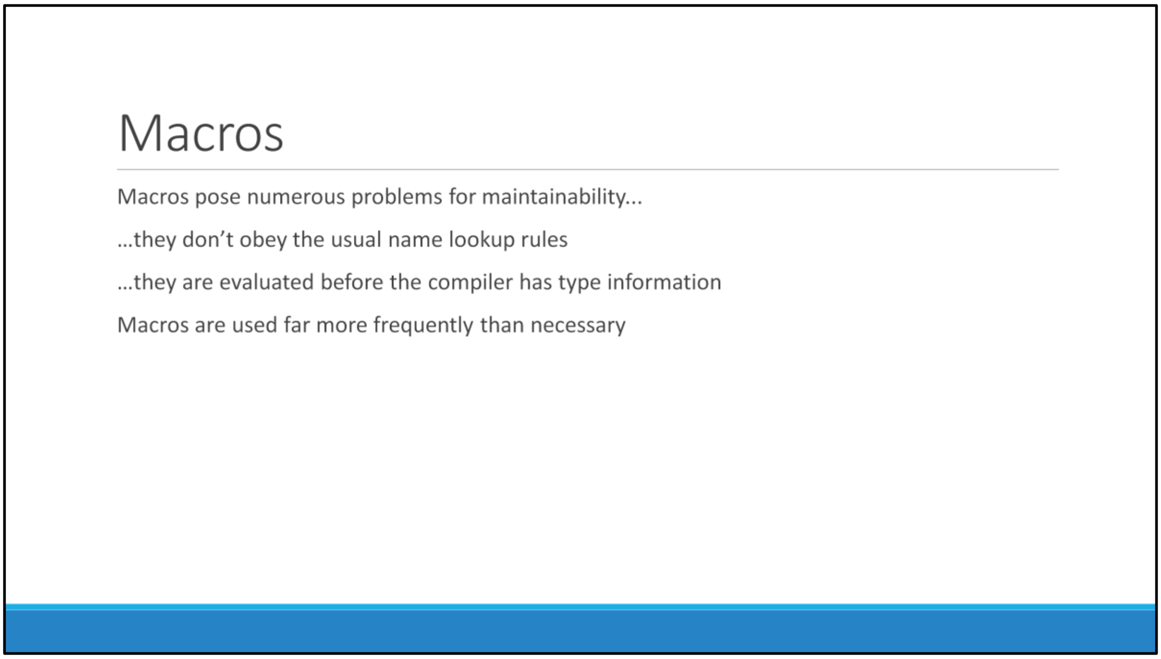
```
#ifndef _CRTIMP
#if defined CRTDLL && defined _CRTBLD
#define _CRTIMP __declspec(dllexport)
#else
#ifdef _DLL
#define _CRTIMP __declspec(dllimport)
#else
#define _CRTIMP
#endif
#endif
#endif
```



```
#ifndef _CRTIMP
#if defined CRTDLL && defined _CRTBLD
#define _CRTIMP __declspec(dllexport)
#else /* defined CRTDLL && defined _CRTBLD */
#ifdef _DLL
#define _CRTIMP __declspec(dllimport)
#else /* _DLL */
#define _CRTIMP
#endif /* _DLL */
#endif /* defined CRTDLL && defined _CRTBLD */
#endif /* _CRTIMP */
```



```
#ifndef _CRTIMP
    #if defined CRTDLL && defined _CRTBLD
        #define _CRTIMP __declspec(dllexport)
    #else
        #ifdef _DLL
            #define _CRTIMP __declspec(dllimport)
        #else
            #define _CRTIMP
        #endif
    #endif
#endif
```



Macros

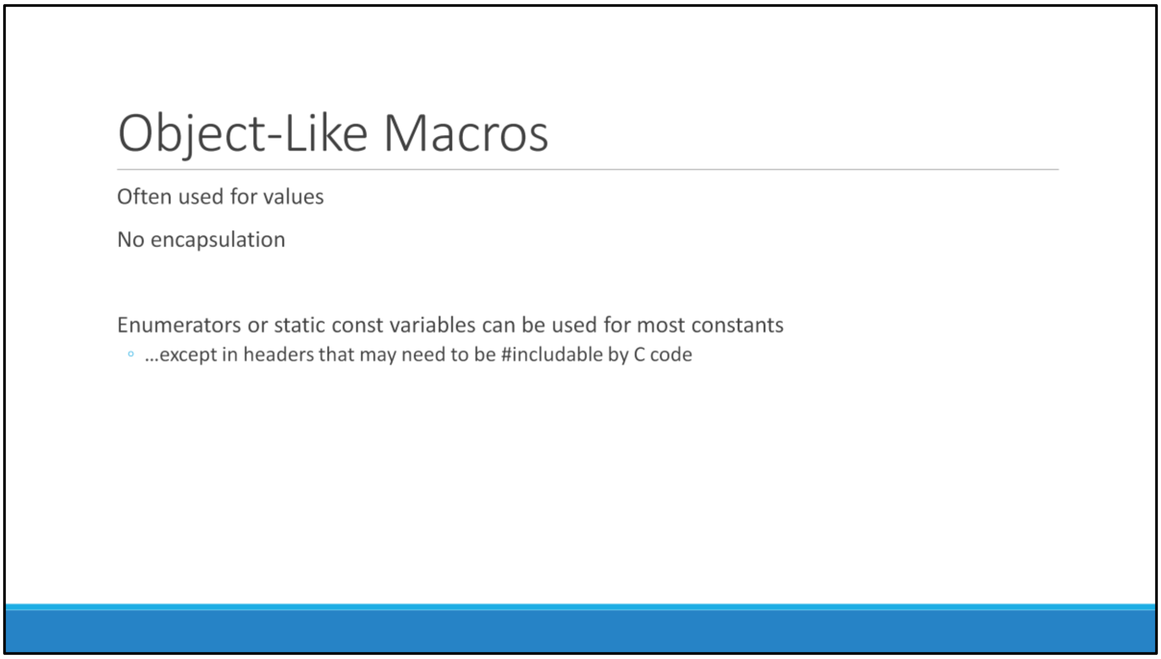
Macros pose numerous problems for maintainability...

...they don’t obey the usual name lookup rules

...they are evaluated before the compiler has type information

Macros are used far more frequently than necessary

We are going to talk about three uses for macros and what to do about them



Object-Like Macros

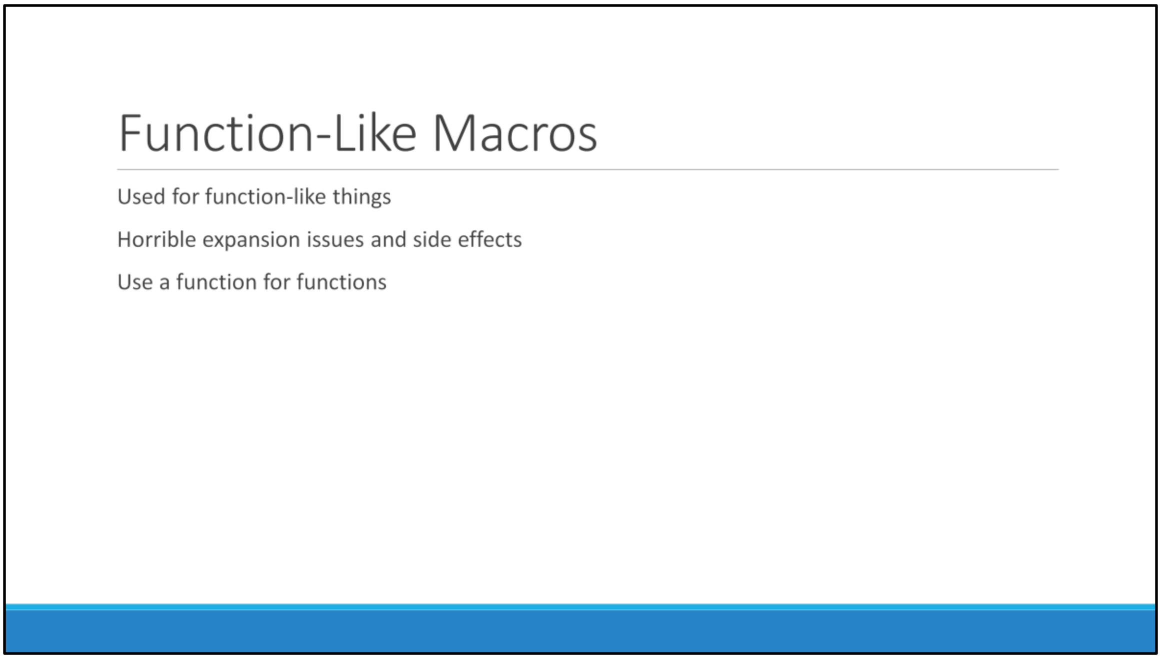
Often used for values

No encapsulation

Enumerators or static const variables can be used for most constants

- …except in headers that may need to be #includable by C code

If you want a value, use a const (it will have a type and everything!)
If you want several similar values, use enums
  - worried about name overlap? Enum classes
http://www.cprogramming.com/c++11/c++11-nullptr-strongly-typed-enum-class.html



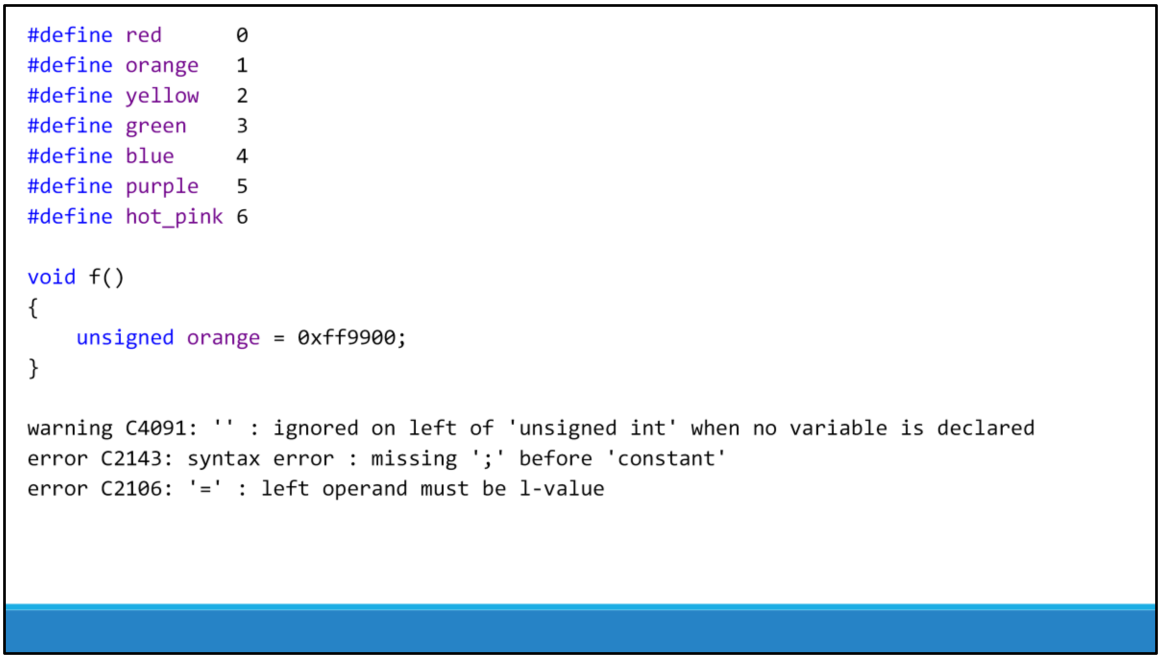
Function-Like Macros

Used for function-like things

Horrible expansion issues and side effects

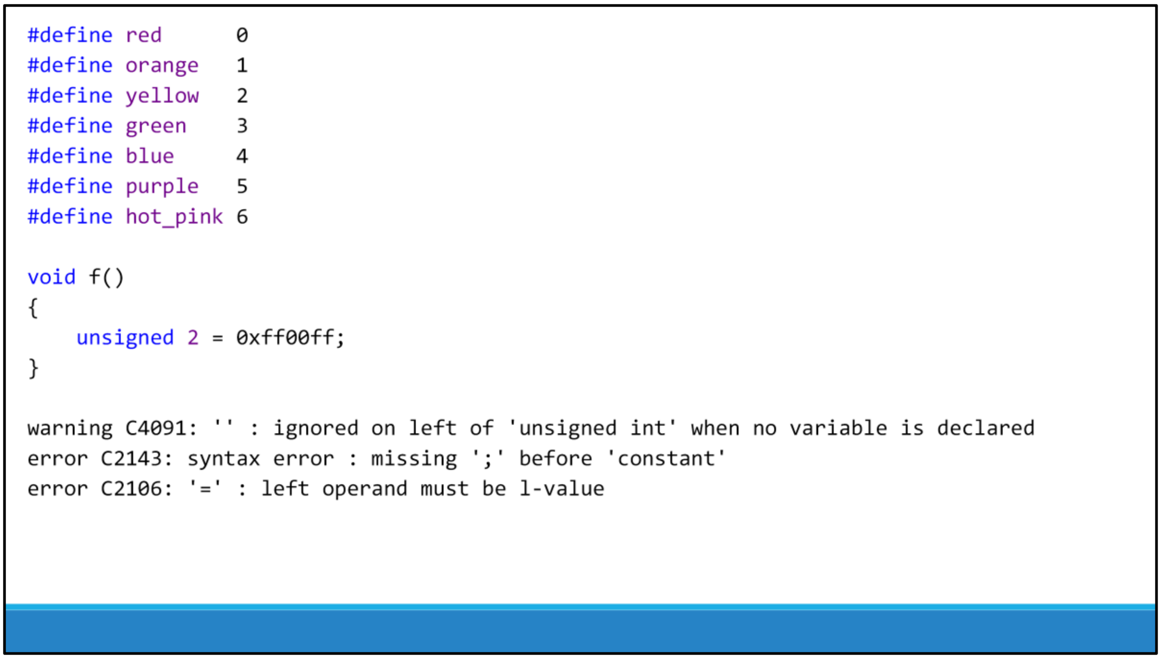
Use a function for functions

Need demo of this

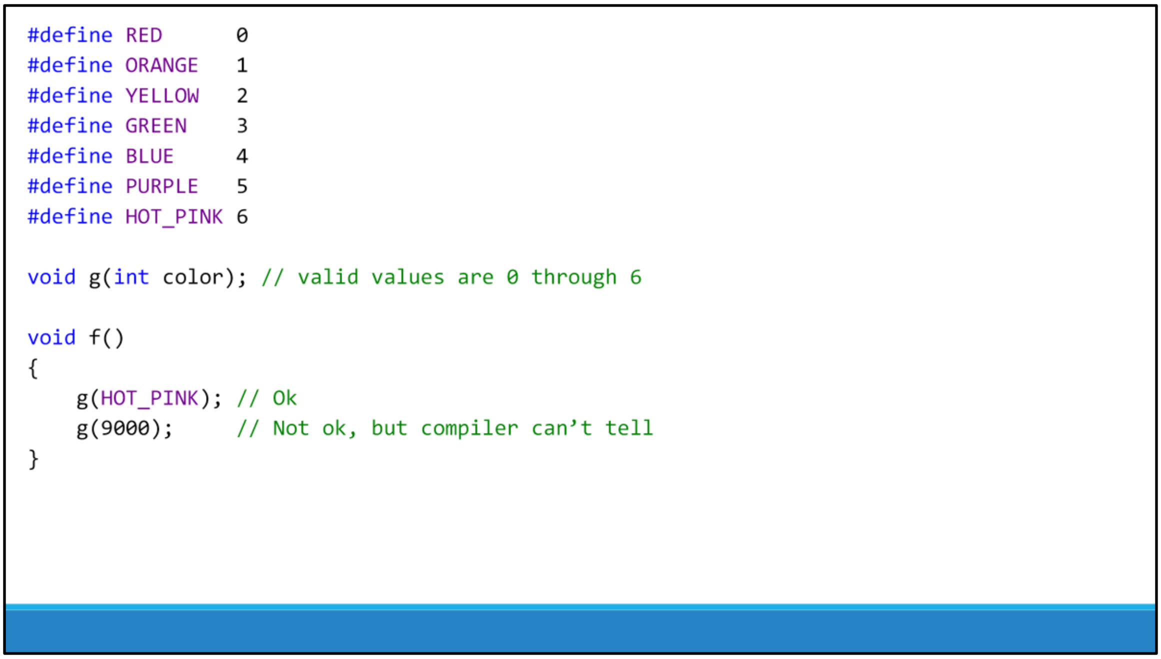


```
#define red      0
#define orange   1
#define yellow   2
#define green    3
#define blue     4
#define purple   5
#define hot_pink 6

void f()
{
    unsigned orange = 0xff9900;
}

warning C4091: '' : ignored on left of 'unsigned int' when no variable is declared
error C2143: syntax error : missing ';' before 'constant'
error C2106: '=' : left operand must be l-value
```

This sort of thing is why many old school C++ developers have rigid strongly held opinions about naming conventions. If “constants” defined as macros are always all upper case, then you can’t accidently declare a local variable that has the same name as a macro. As a safety net, this is not exactly bulletproof. And of course the error message is utterly incomprehensible to someone who has no idea that these macros are defined in some header file that was included as a result of some long chain of includes, if the developer never uses them and didn’t know they existed.

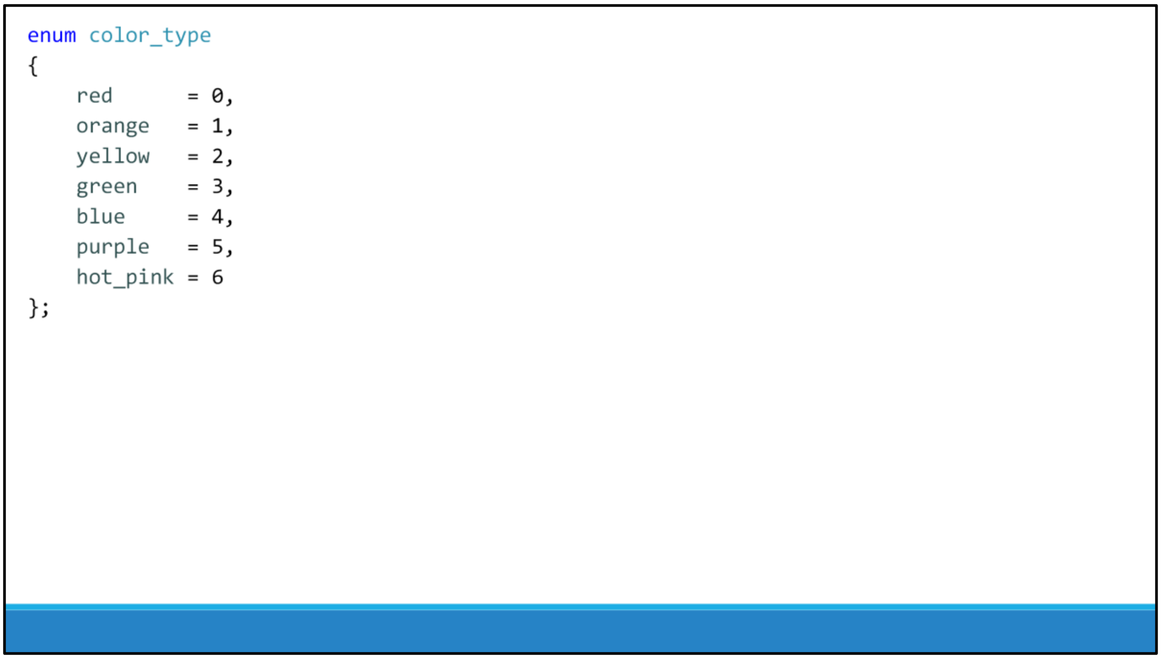


```
#define red      0
#define orange   1
#define yellow   2
#define green    3
#define blue     4
#define purple   5
#define hot_pink 6

void f()
{
    unsigned 2 = 0xff00ff;
}

warning C4091: '' : ignored on left of 'unsigned int' when no variable is declared
error C2143: syntax error : missing ';' before 'constant'
error C2106: '=' : left operand must be l-value
```



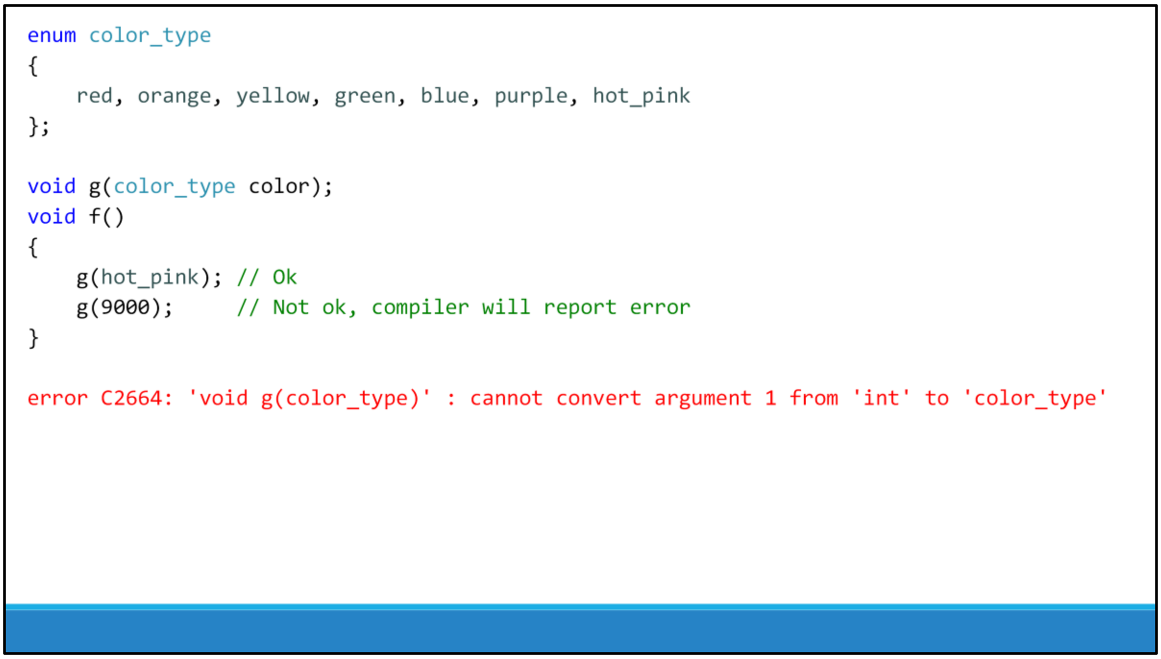
```
#define RED      0
#define ORANGE   1
#define YELLOW   2
#define GREEN    3
#define BLUE     4
#define PURPLE   5
#define HOT_PINK 6

void g(int color); // valid values are 0 through 6

void f()
{
    g(HOT_PINK); // Ok
    g(9000);     // Not ok, but compiler can’t tell
}
```

Again compilers don’t read comments, so a comment telling people to make sure they only pass values in a certain range is not something the compiler can enforce. At best you’ll get an assert or some other runtime error, at worst you’ll get really strange behavior that is difficult to debug. The compiler is your friend, you want to let the compiler help you.

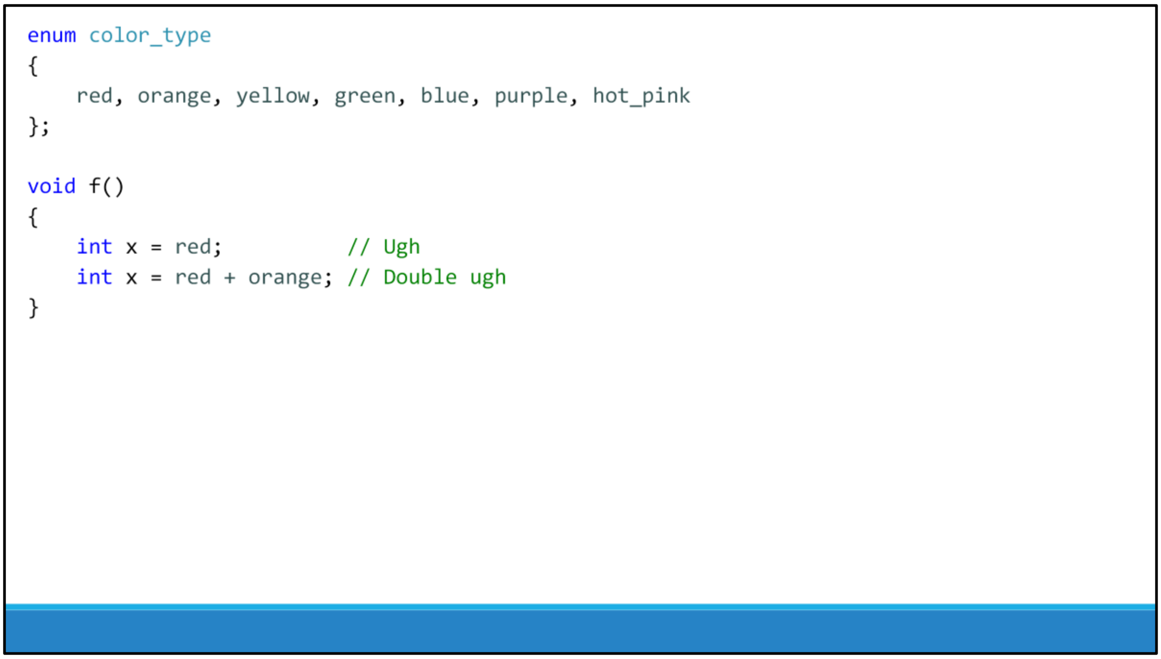


```
enum color_type
{
    red      = 0,
    orange   = 1,
    yellow   = 2,
    green    = 3,
    blue     = 4,
    purple   = 5,
    hot_pink = 6
};
```

So switching to an enum solves a lot of this. You get to make it clear these 7 constants belong together, for one thing. (And if these are the values you want, you don’t even need to type out the values, though in this case since they already exist, what the heck.) Also the compiler will notice if you flub up a copy and paste and give two members of the enum the same value, which the preprocessor never would have.

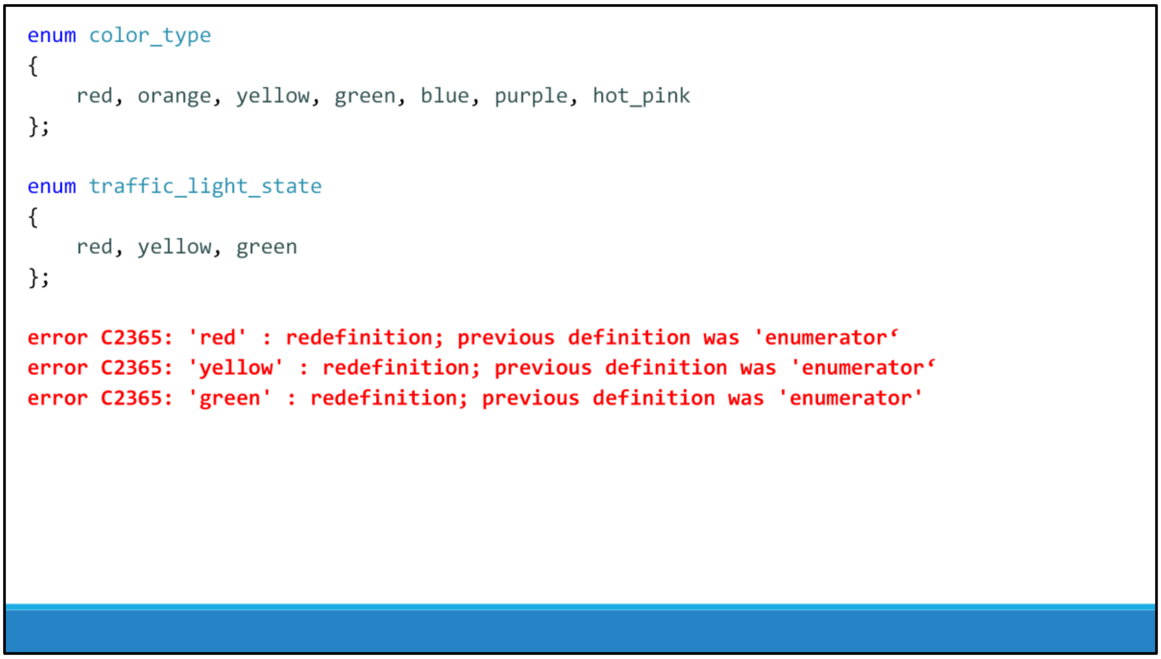


```
enum color_type
{
    red, orange, yellow, green, blue, purple, hot_pink
};

void g(color_type color);
void f()
{
    g(hot_pink); // Ok
    g(9000);     // Not ok, compiler will report error
}

error C2664: 'void g(color_type)' : cannot convert argument 1 from 'int' to 'color_type'
```

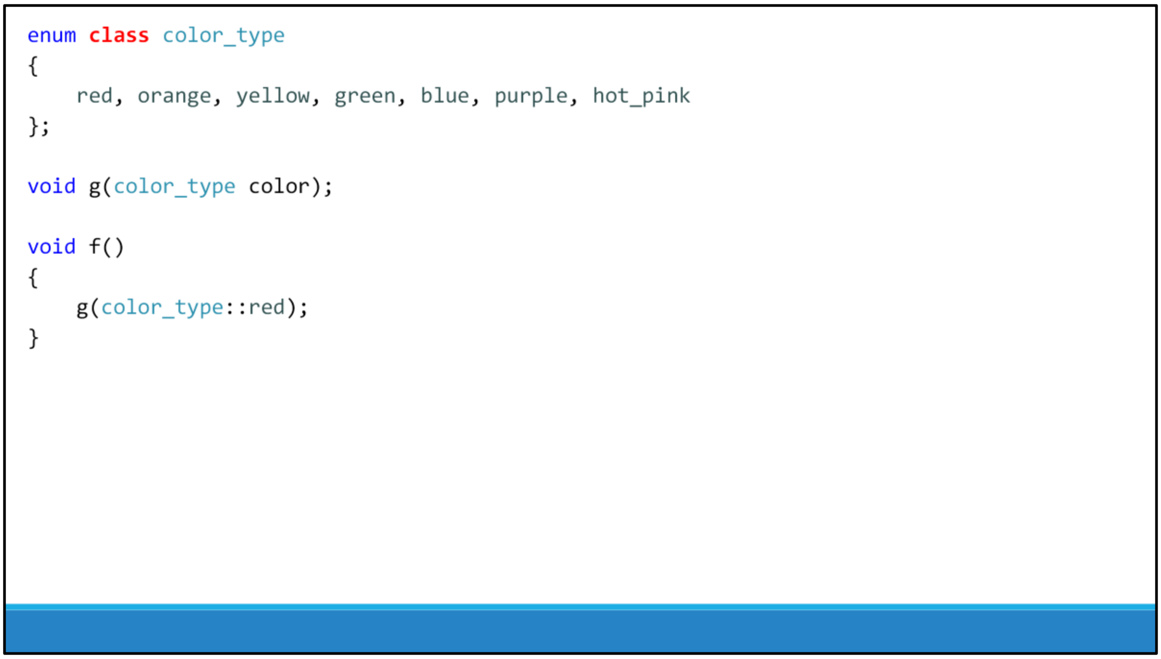


```
enum color_type
{
    red, orange, yellow, green, blue, purple, hot_pink
};

void f()
{
    int x = red;          // Ugh
    int x = red + orange; // Double ugh
}
```

As coded, red+anything = anything since red is 0. yellow+green=purple, and so on. Composing flags by adding bitfields is a possibility but in general, we shouldn’t be adding enums.

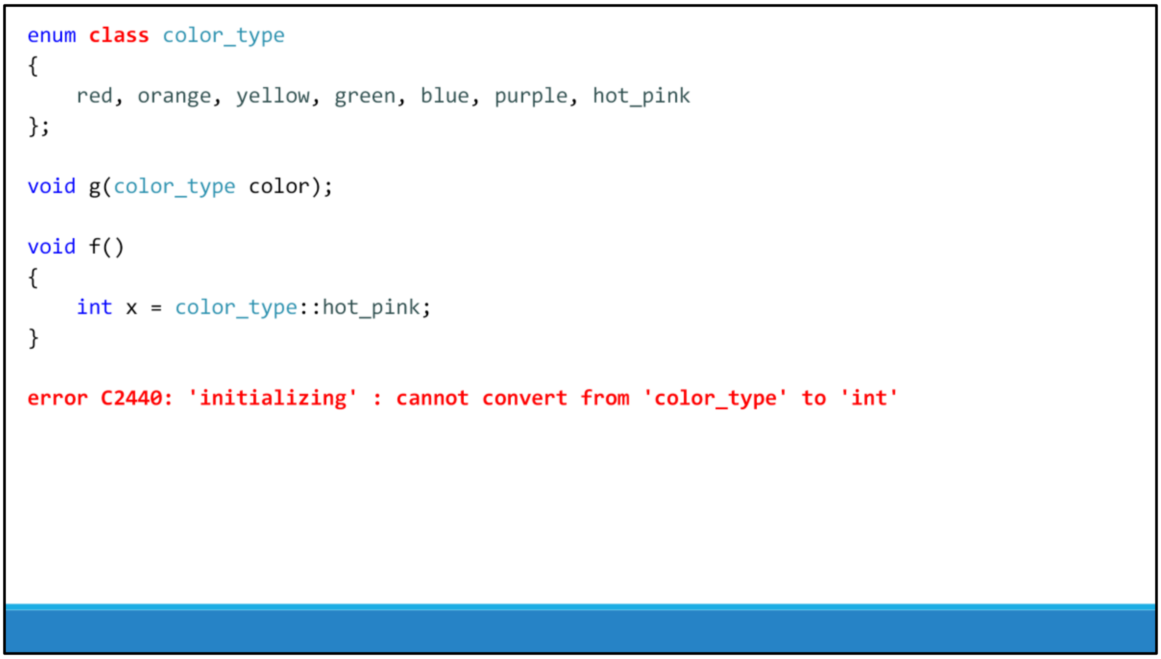


```
enum color_type
{
    red, orange, yellow, green, blue, purple, hot_pink
};

enum traffic_light_state
{
    red, yellow, green
};

error C2365: 'red' : redefinition; previous definition was 'enumerator‘
error C2365: 'yellow' : redefinition; previous definition was 'enumerator‘
error C2365: 'green' : redefinition; previous definition was 'enumerator'
```

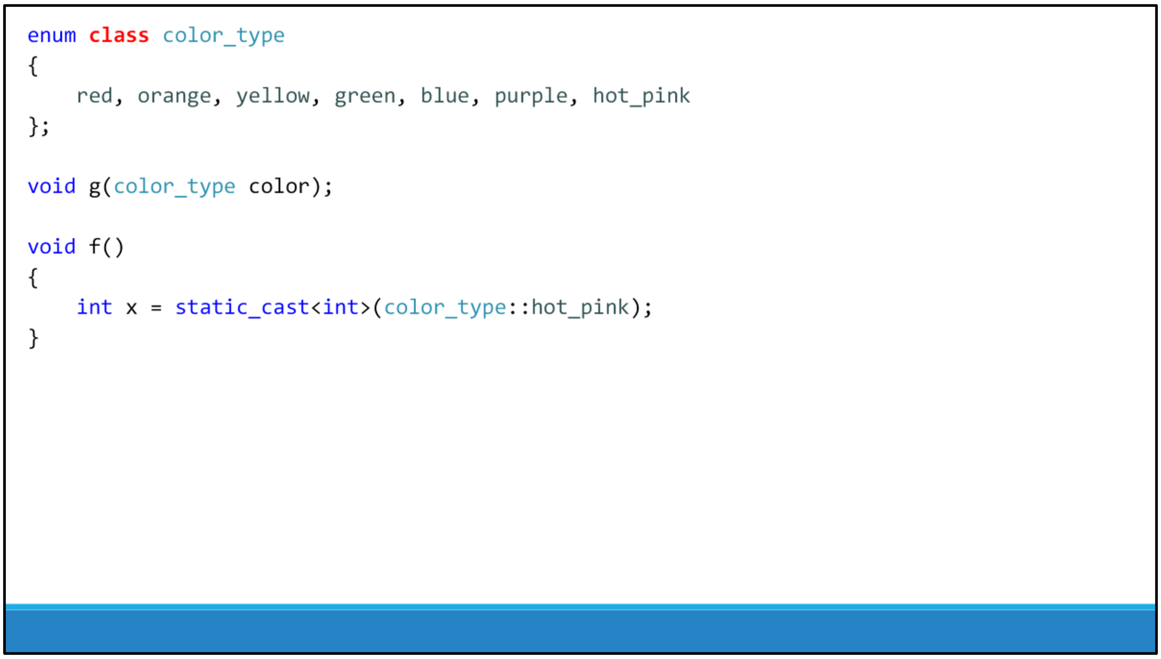


```
enum class color_type
{
    red, orange, yellow, green, blue, purple, hot_pink
};

void g(color_type color);

void f()
{
    g(color_type::red);
}
```

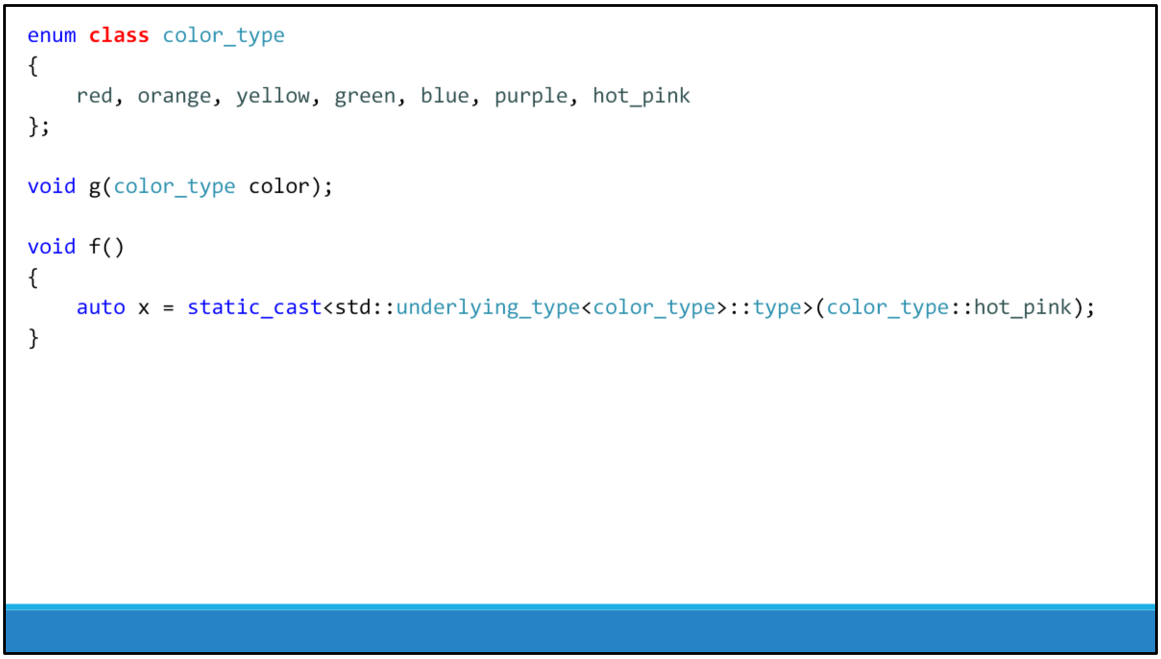
Enum classes solve this. They aren't implicitly convertible to their underlying types, and that's a feature. You can static_cast<> them to the underlying type any time you want.



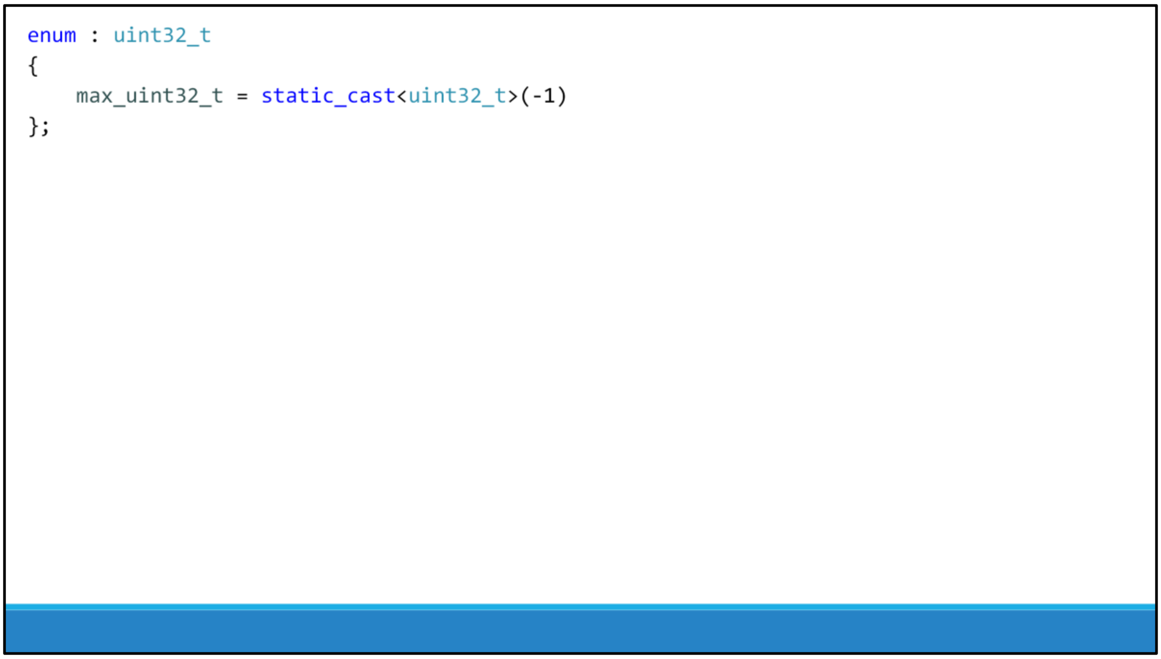
```
enum class color_type
{
    red, orange, yellow, green, blue, purple, hot_pink
};

void g(color_type color);

void f()
{
    int x = color_type::hot_pink;
}

error C2440: 'initializing' : cannot convert from 'color_type' to 'int'
```



```
enum class color_type
{
    red, orange, yellow, green, blue, purple, hot_pink
};

void g(color_type color);

void f()
{
    int x = static_cast<int>(color_type::hot_pink);
}
```

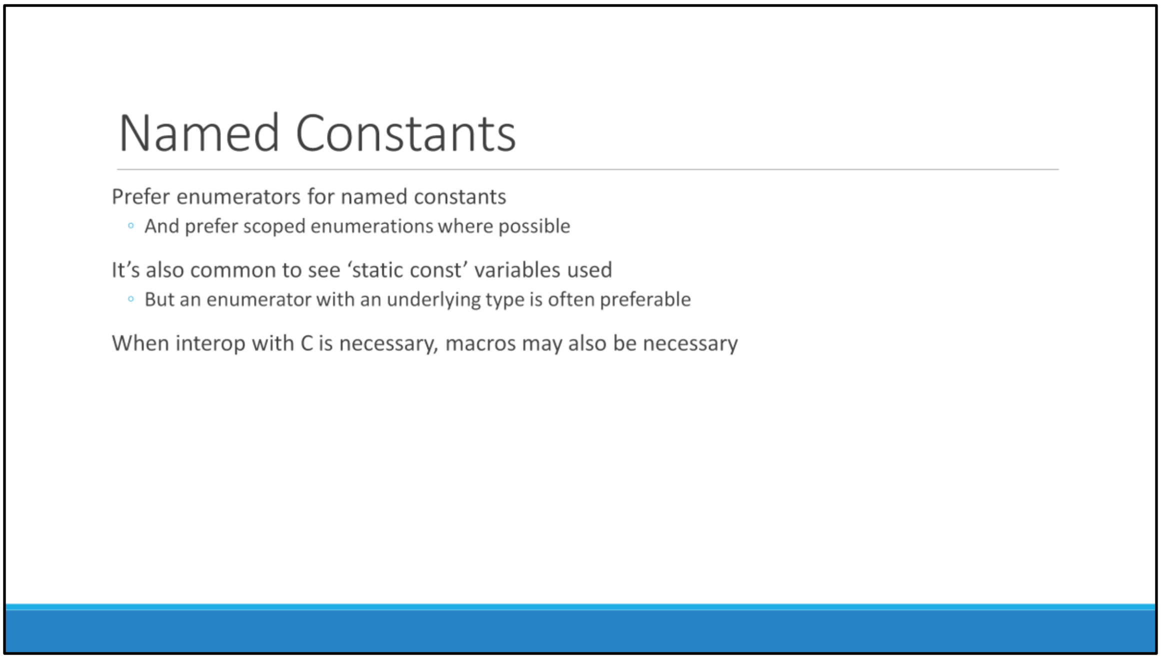


```
enum class color_type
{
    red, orange, yellow, green, blue, purple, hot_pink
};

void g(color_type color);

void f()
{
    auto x = static_cast<std::underlying_type<color_type>::type>(color_type::hot_pink);
}
```



```
enum : uint32_t
{
    max_uint32_t = static_cast<uint32_t>(-1)
};
```

You can even use an enum for a single constant if you want to get some of these advantages

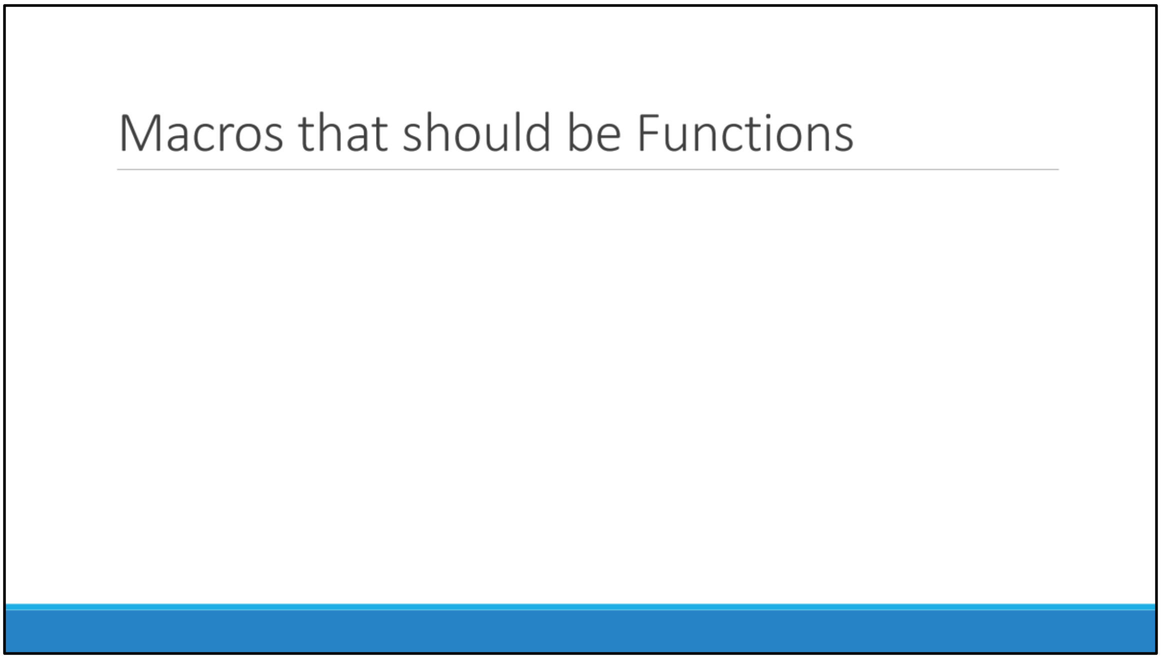


Named Constants

Prefer enumerators for named constants
- And prefer scoped enumerations where possible

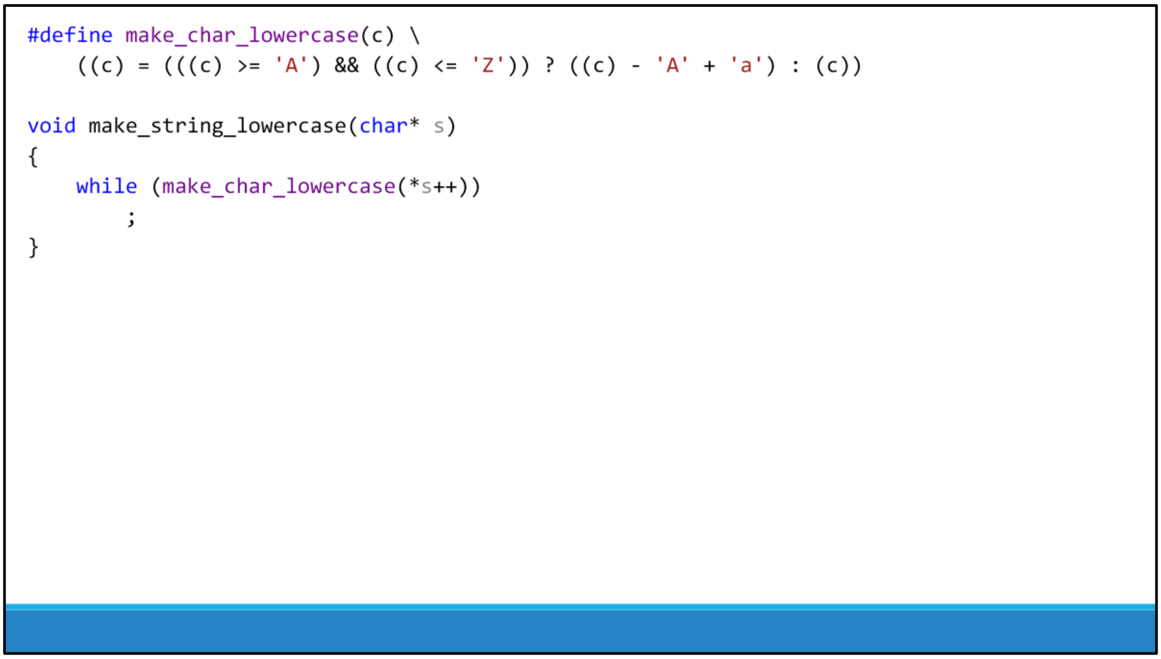
It’s also common to see ‘static const’ variables used
- But an enumerator with an underlying type is often preferable

When interop with C is necessary, macros may also be necessary



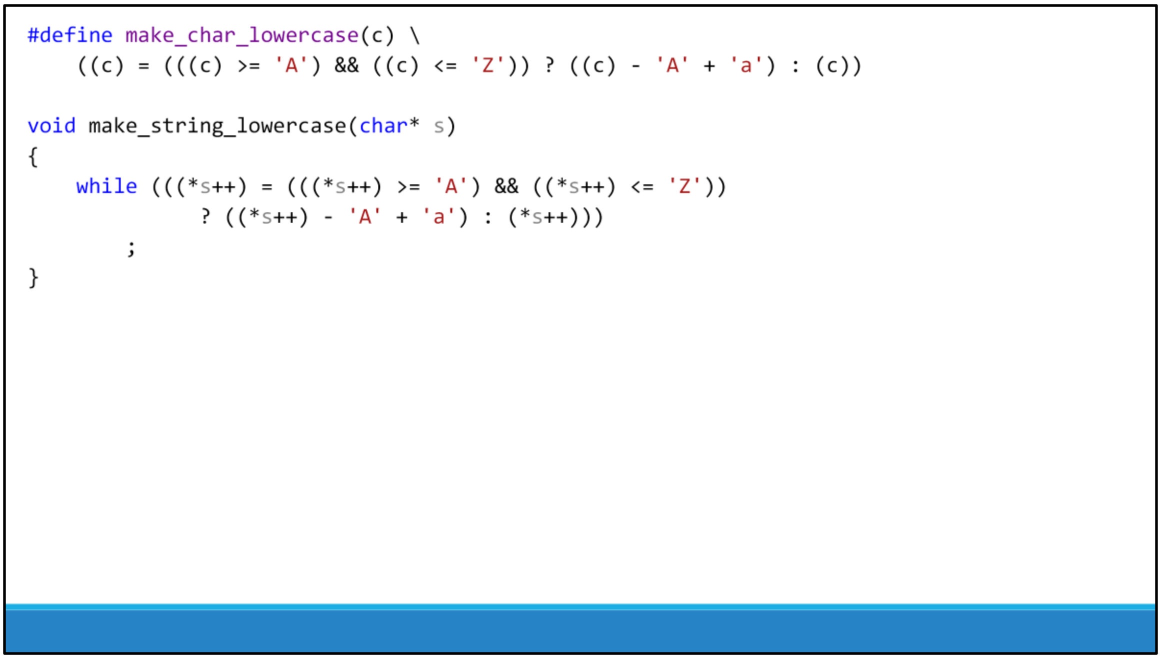
Macros that should be Functions

Add more parentheses, that'll fix it!



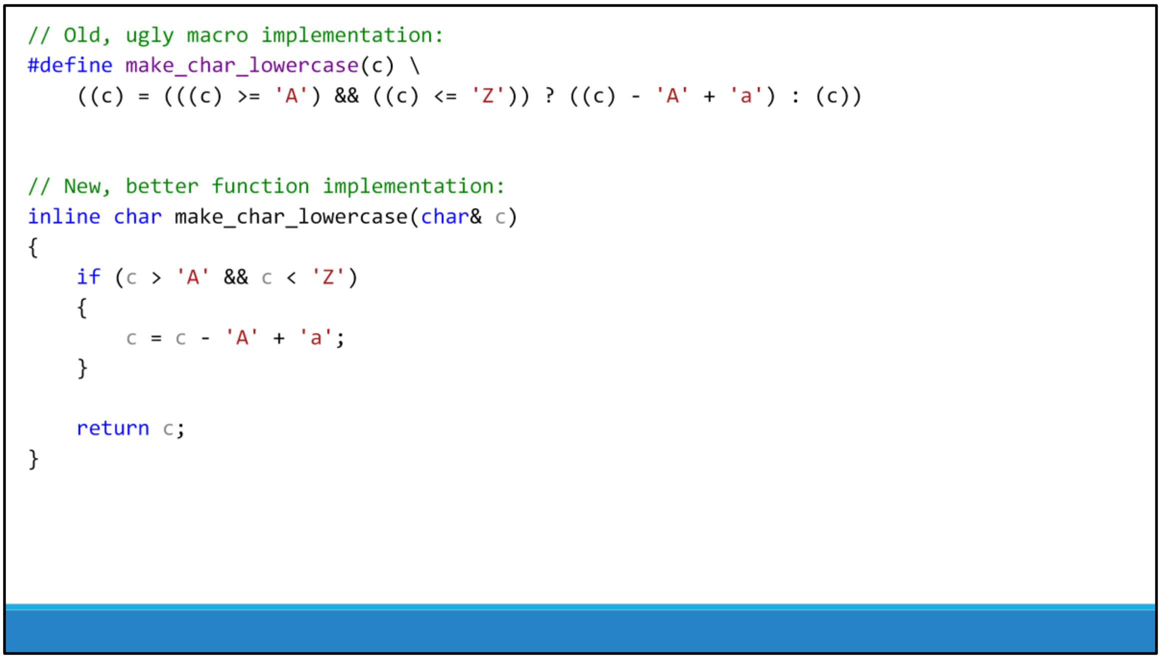
```
#define make_char_lowercase(c) \
    ((c) = (((c) >= 'A') && ((c) <= 'Z')) ? ((c) - 'A' + 'a') : (c))

void make_string_lowercase(char* s)
{
    while (make_char_lowercase(*s++))
        ;
}
```

With this macro to make a single character lowercase, you might think it makes sense to write a function that can make a string lowercase by using the macro in a loop. But macros aren't functions and don't evaluate their parameters. What the compiler actually sees is this:



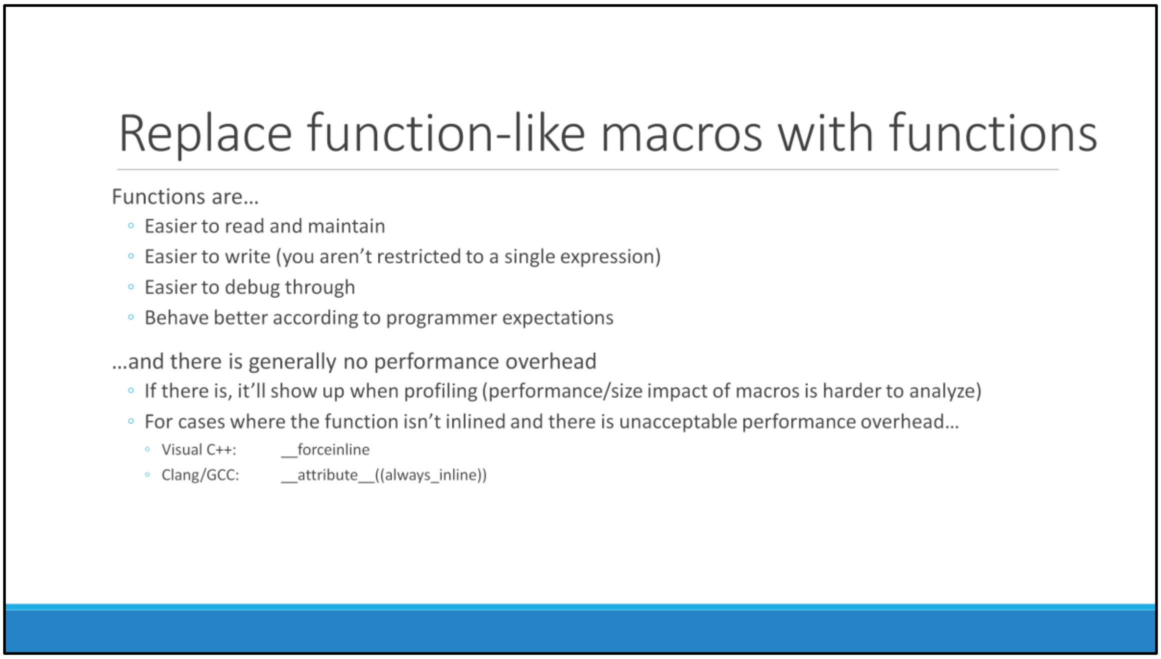
```
#define make_char_lowercase(c) \
    ((c) = (((c) >= 'A') && ((c) <= 'Z')) ? ((c) - 'A' + 'a') : (c))

void make_string_lowercase(char* s)
{
    while (((*s++) = (((*s++) >= 'A') && ((*s++) <= 'Z'))
              ? ((*s++) - 'A' + 'a') : (*s++)))
        ;
}
```

And that means the character pointer is getting incremented a whole lot more than you bargained for.



```
// Old, ugly macro implementation:
#define make_char_lowercase(c) \
    ((c) = (((c) >= 'A') && ((c) <= 'Z')) ? ((c) - 'A' + 'a') : (c))


// New, better function implementation:
inline char make_char_lowercase(char& c)
{
    if (c > 'A' && c < 'Z')
    {
        c = c - 'A' + 'a';
    }

    return c;
}
```

There’s no arguing this is more readable. As well, it gets rid of the weirdness around argument evaluation. Perhaps you think it’s going to be slower? This is going to be inlined which means you are not taking a perf hit doing it this way.



Replace function-like macros with functions

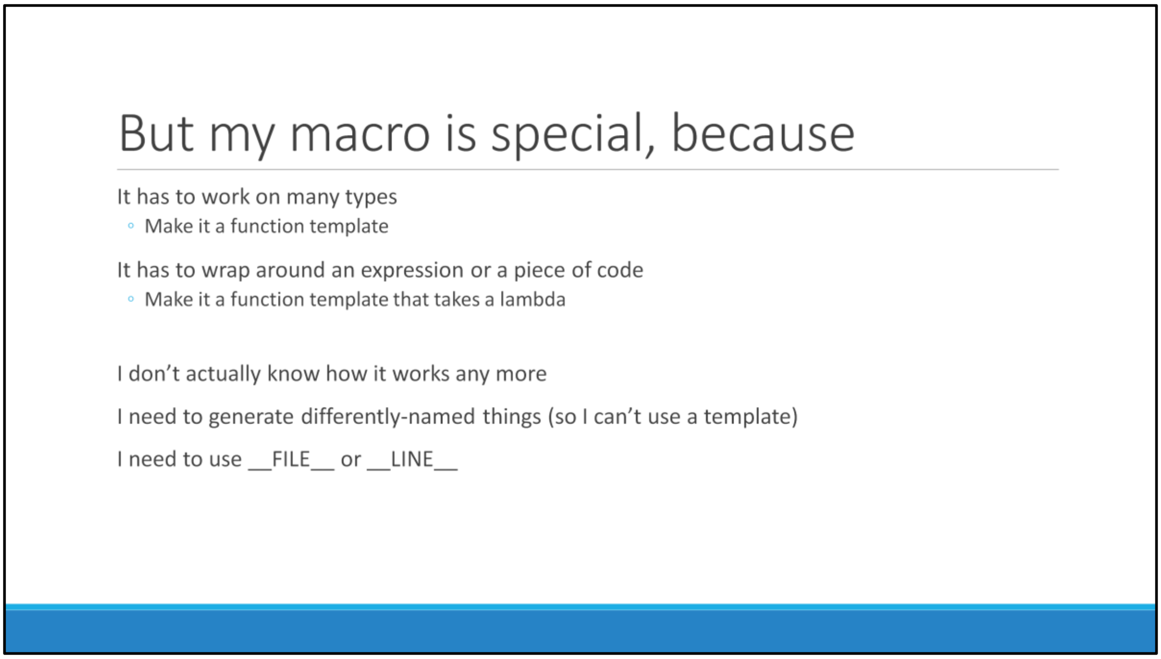
Functions are…

- Easier to read and maintain
- Easier to write (you aren’t restricted to a single expression)
- Easier to debug through
- Behave better according to programmer expectations

…and there is generally no performance overhead

- If there is, it’ll show up when profiling (performance/size impact of macros is harder to analyze)
- For cases where the function isn’t inlined and there is unacceptable performance overhead…
  - Visual C++: __forceinline
  - Clang/GCC: __attribute__((always_inline))

If you were putting up with macros because you thought you didn’t have alternatives – you have alternatives.



But my macro is special, because

It has to work on many types

- Make it a function template

It has to wrap around an expression or a piece of code

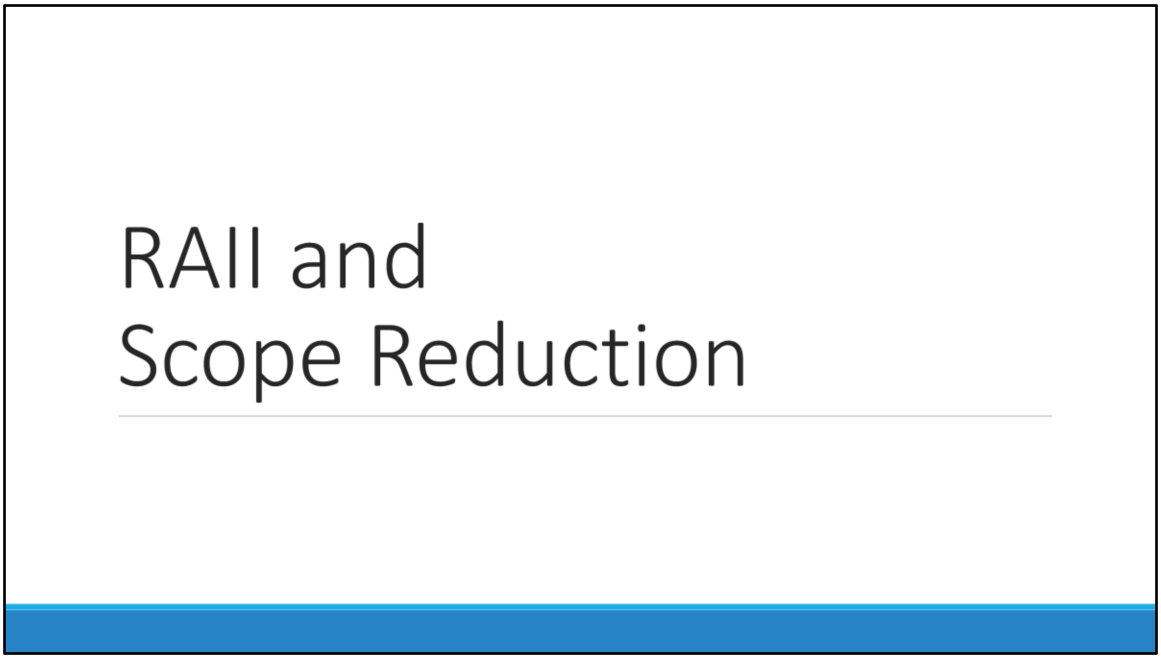
- Make it a function template that takes a lambda

I don’t actually know how it works any more

I need to generate differently-named things (so I can’t use a template)

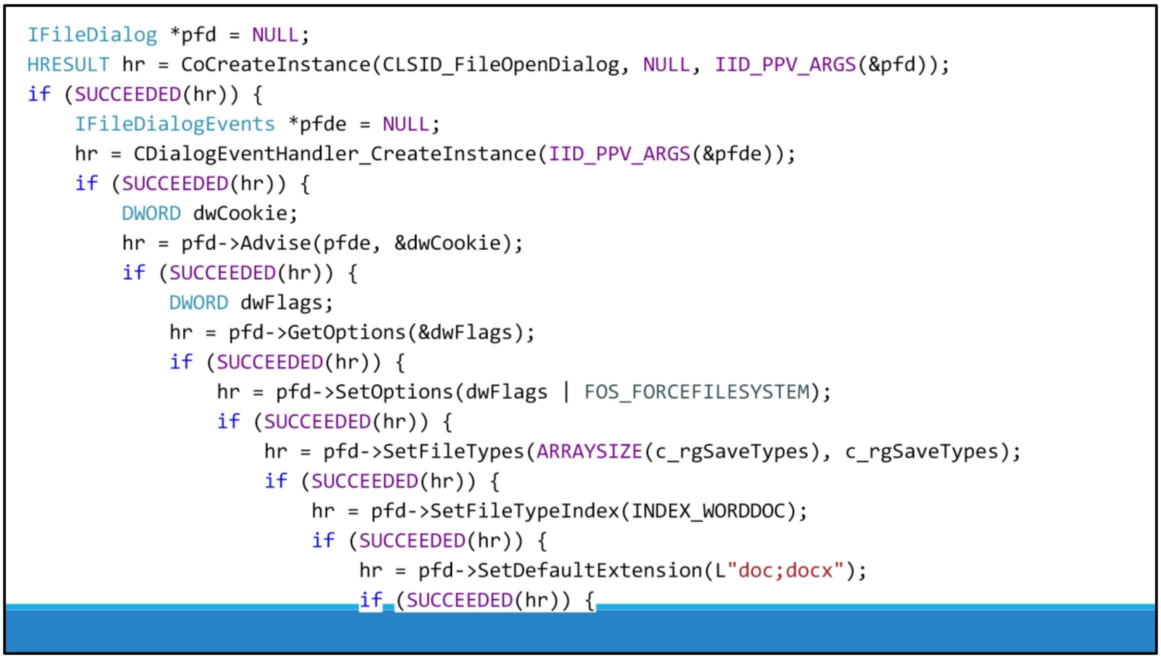
I need to use __FILE__ or __LINE__

The last three, yeah, ok, you probably still need a macro. But anything else? Stop using macros.



RAII and Scope Reduction

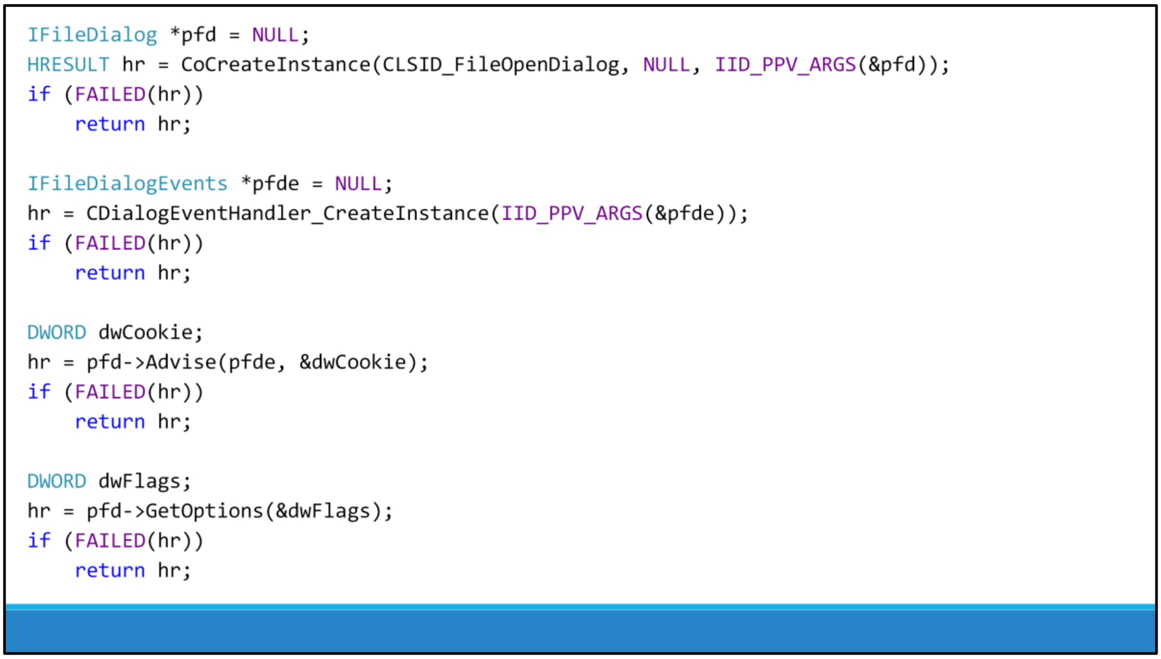
Switch to Kate
Horrible name, wonderful concept
Can’t sprinkle on RAII, have to bake it in. But what you want is natural scope so that cleanup happens for you.



```
IFileDialog *pfd = NULL;
HRESULT hr = CoCreateInstance(CLSID_FileOpenDialog, NULL, IID_PPV_ARGS(&pfd));
if (SUCCEEDED(hr)) {
    IFileDialogEvents *pfde = NULL;
    hr = CDialogEventHandler_CreateInstance(IID_PPV_ARGS(&pfde));
    if (SUCCEEDED(hr)) {
        DWORD dwCookie;
        hr = pfd->Advise(pfde, &dwCookie);
        if (SUCCEEDED(hr)) {
            DWORD dwFlags;
            hr = pfd->GetOptions(&dwFlags);
            if (SUCCEEDED(hr)) {
                hr = pfd->SetOptions(dwFlags | FOS_FORCEFILESYSTEM);
                if (SUCCEEDED(hr)) {
                    hr = pfd->SetFileTypes(ARRAYSIZE(c_rgSaveTypes), c_rgSaveTypes);
                    if (SUCCEEDED(hr)) {
                        hr = pfd->SetFileTypeIndex(INDEX_WORDDOC);
                        if (SUCCEEDED(hr)) {
                            hr = pfd->SetDefaultExtension(L"doc;docx");
                            if (SUCCEEDED(hr)) {
```

Actual example from msdn
“arrow code”

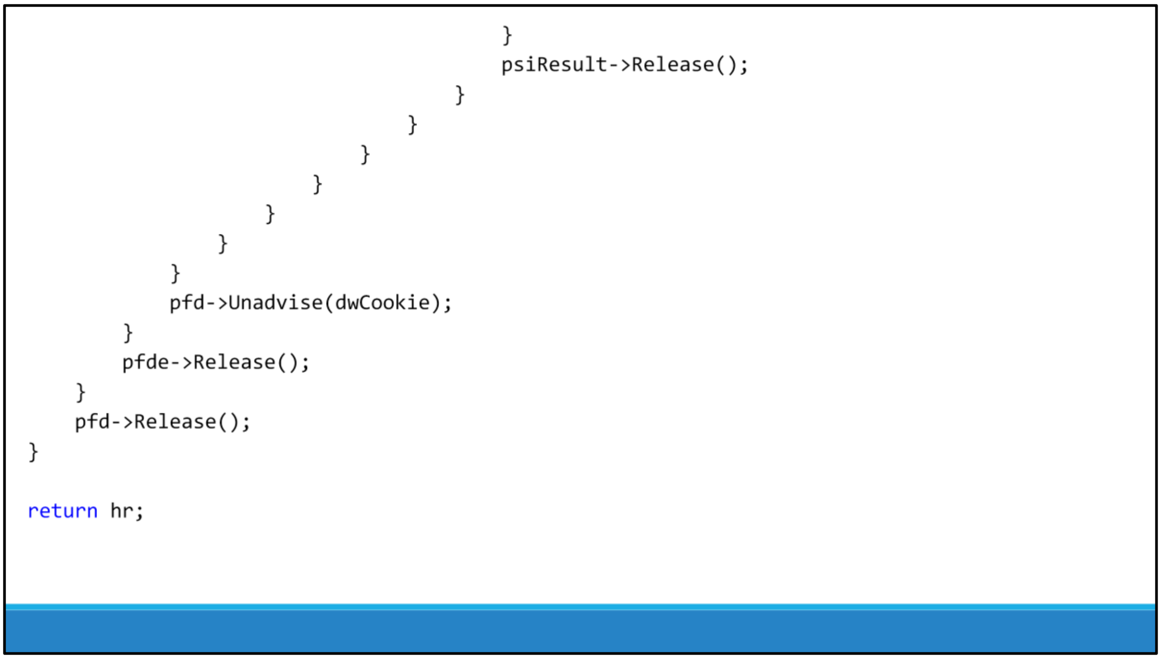
Explain SUCCEEDED(hr)
Horizontal scrolling
Housekeeping is obscuring actual purpose of the code



```
IFileDialog *pfd = NULL;
HRESULT hr = CoCreateInstance(CLSID_FileOpenDialog, NULL, IID_PPV_ARGS(&pfd));
if (FAILED(hr))
    return hr;

IFileDialogEvents *pfde = NULL;
hr = CDialogEventHandler_CreateInstance(IID_PPV_ARGS(&pfde));
if (FAILED(hr))
    return hr;

DWORD dwCookie;
hr = pfd->Advise(pfde, &dwCookie);
if (FAILED(hr))
    return hr;

DWORD dwFlags;
hr = pfd->GetOptions(&dwFlags);
if (FAILED(hr))
    return hr;
```

We would prefer this – linear code

Certainly less scrolly. And your pattern matching skills may enable you to better ignore the housekeeping in amongst the good stuff. But there’s something missing

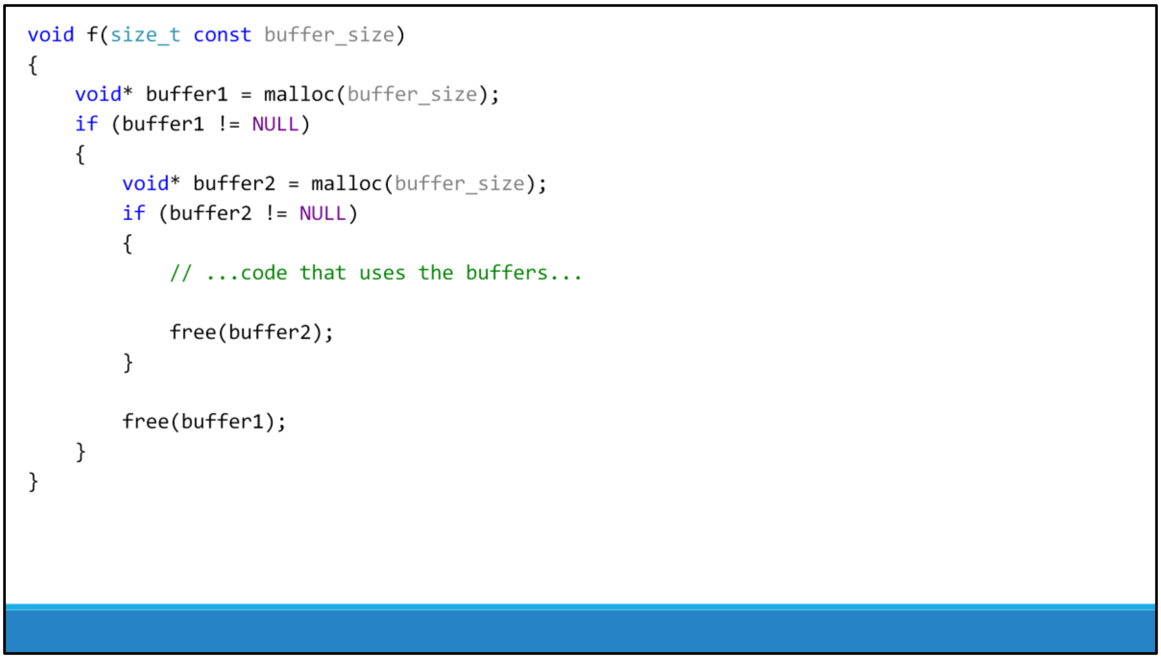


```
                                        }
                                        psiResult->Release();
                                    }
                                }
                            }
                        }
                    }
                }
            }
            pfd->Unadvise(dwCookie);
        }
        pfde->Release();
    }
    pfd->Release();
}

return hr;
```

But here's the bottom half of the arrow
So you can't just bail, you have to release or unadvised or whatever according to how far you got
You could put each of these "cleanups" before the return statements, but you would be repeating them – if you return after completing two things, clean up after two things, if you return after completing three things, clean up after three things, etc. It's really hard to write and maintain this code.

This pattern happens outside of COM calls or Windows programming. Any time you make a change of some kind and are committed to changing it back when you're done (for better or worse) then you're in a great place to let a destructor be the way you do that cleanup. Of course, it's not super efficient for all of us to write these little classes whose destructors call Release() or Unadvise() or whatever it is that needs to be called, but you know once one is written, using it will make your code neater, more readable, and probably eliminate some kind of leak, resource leak, handle leak or whatnot.

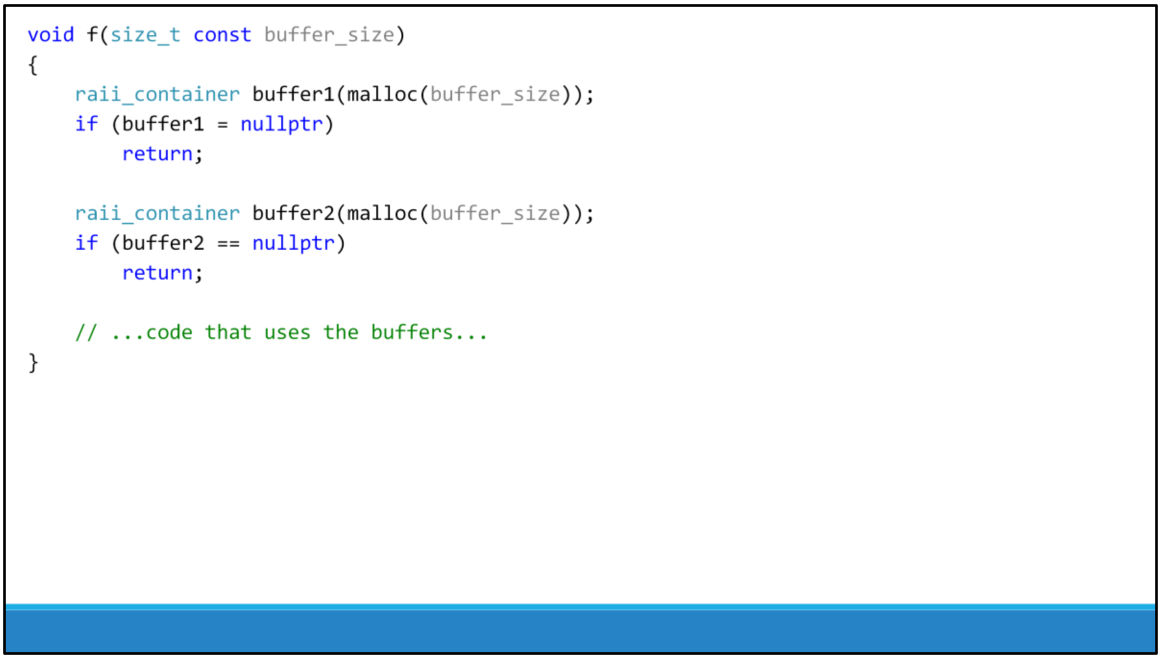
Let's switch to a slightly simpler and slightly less nested example



```
void f(size_t const buffer_size)
{
    void* buffer1 = malloc(buffer_size);
    if (buffer1 != NULL)
    {
        void* buffer2 = malloc(buffer_size);
        if (buffer2 != NULL)
        {
            // ...code that uses the buffers...

            free(buffer2);
        }

        free(buffer1);
    }
}
```

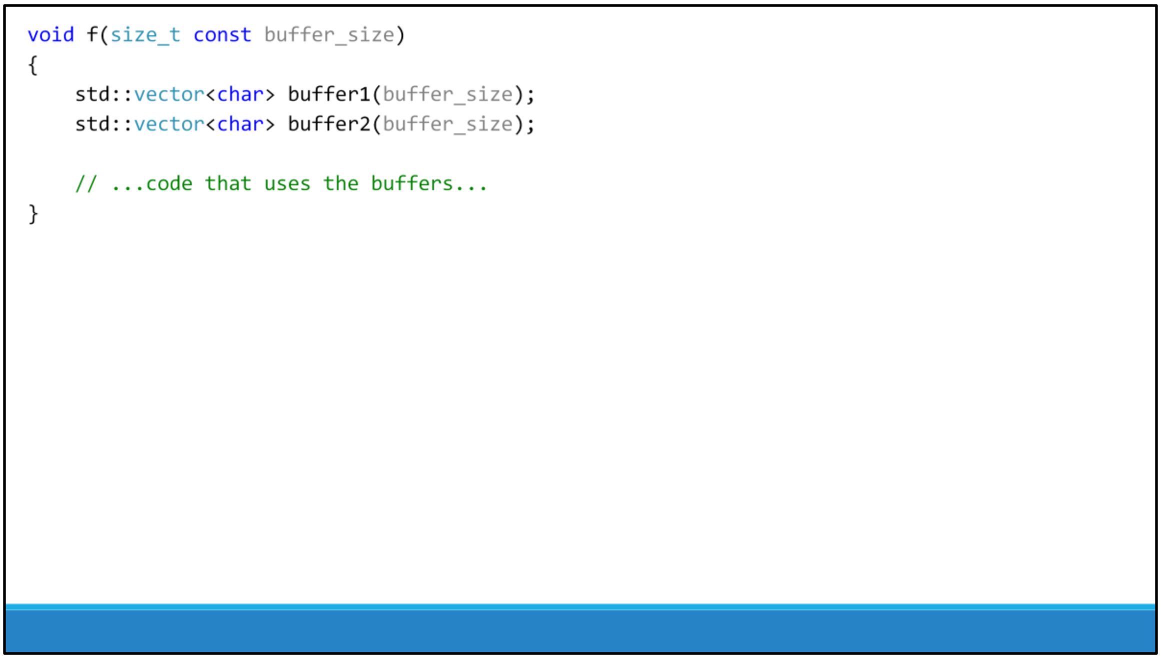
Let’s switch to a much simpler example because that huge one is too big for a slide. This code is fine, it works. But you have to remember to free the buffers according to whether you got them or not.

So imagine someone wrote an RAII container for you whose destructor will call free.



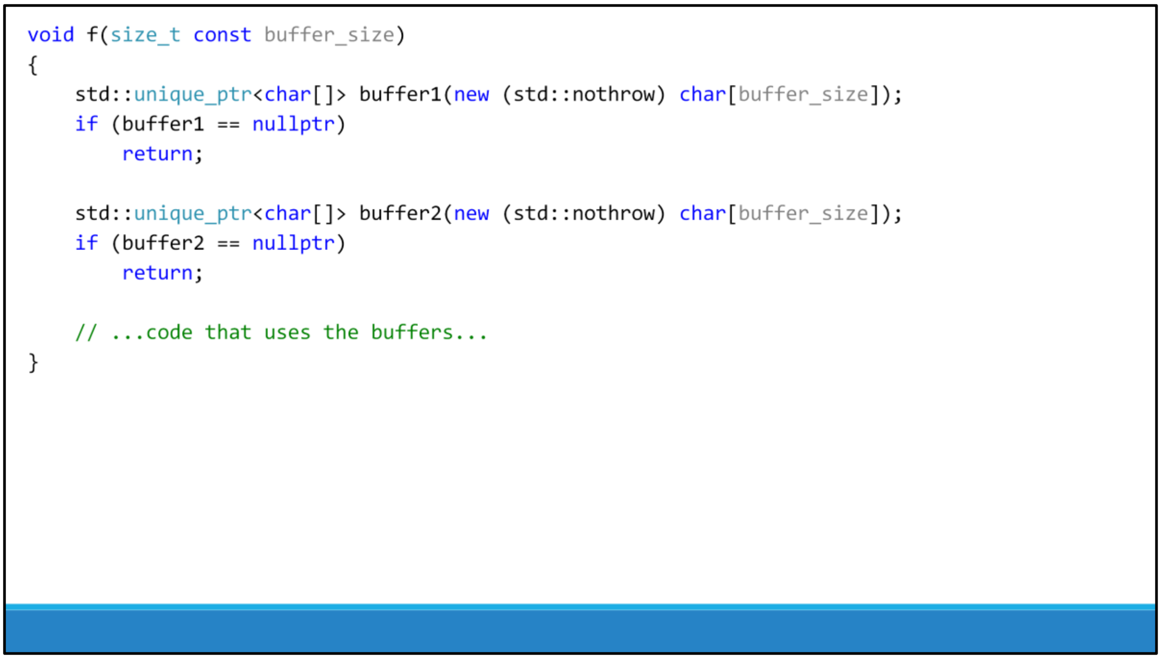
```
void f(size_t const buffer_size)
{
    raii_container buffer1(malloc(buffer_size));
    if (buffer1 = nullptr)
        return;

    raii_container buffer2(malloc(buffer_size));
    if (buffer2 == nullptr)
        return;

    // ...code that uses the buffers...
}
```

So this is a classic RAII example. You no longer call free – the destructor of the imaginary raii_container will do so, either on the two return statements you see here or at the end when the function is finished. Yay.



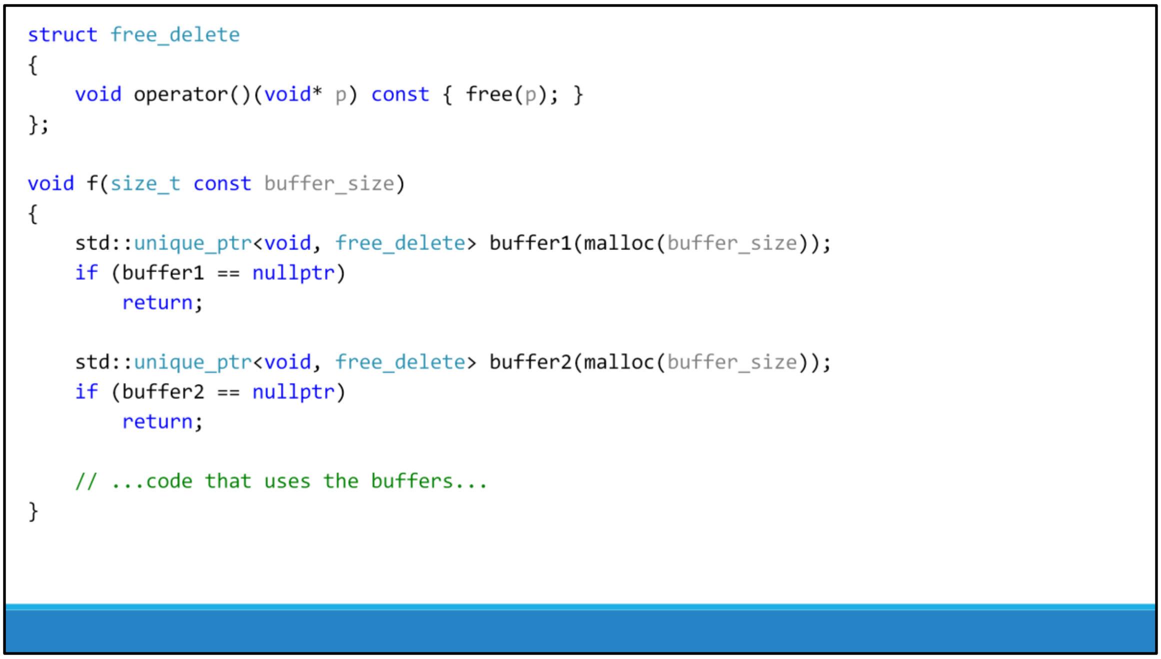
```cpp
void f(size_t const buffer_size)
{
    std::vector<char> buffer1(buffer_size);
    std::vector<char> buffer2(buffer_size);

    // ...code that uses the buffers...
}
```

Hey, what about std::vector? It’s all RAII and stuff right?
Yes, BUT….
It uses new
And new throws when things go bad
And code that is full of manual memory management is probably not exception safe (notice we had nothing on the previous slides about try/catch etc) so chances are we would be introducing bugs here.



```
void f(size_t const buffer_size)
{
    std::unique_ptr<char[]> buffer1(new (std::nothrow) char[buffer_size]);
    if (buffer1 == nullptr)
        return;

    std::unique_ptr<char[]> buffer2(new (std::nothrow) char[buffer_size]);
    if (buffer2 == nullptr)
        return;

    // ...code that uses the buffers...
}
```

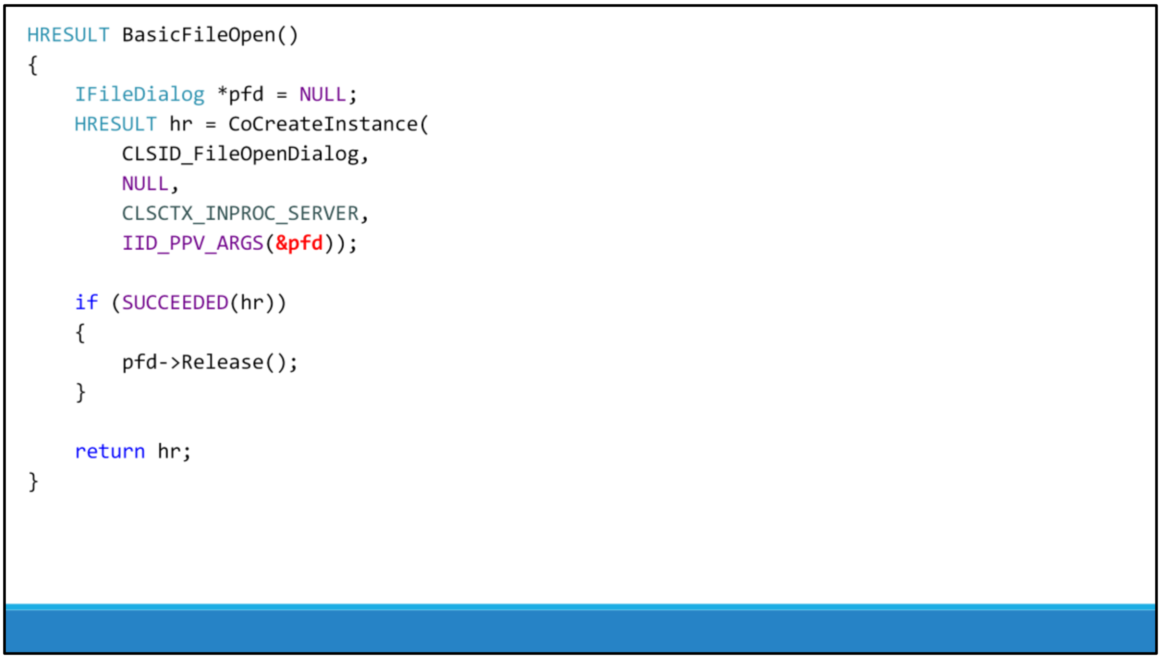
But that doesn’t mean C++ has nothing to offer us. How about we use the nonthrowing new to make our buffers, and then harness unique_ptr? This will totally work.



```
struct free_delete
{
    void operator()(void* p) const { free(p); }
};

void f(size_t const buffer_size)
{
    std::unique_ptr<void, free_delete> buffer1(malloc(buffer_size));
    if (buffer1 == nullptr)
        return;

    std::unique_ptr<void, free_delete> buffer2(malloc(buffer_size));
    if (buffer2 == nullptr)
        return;

    // ...code that uses the buffers...
}
```

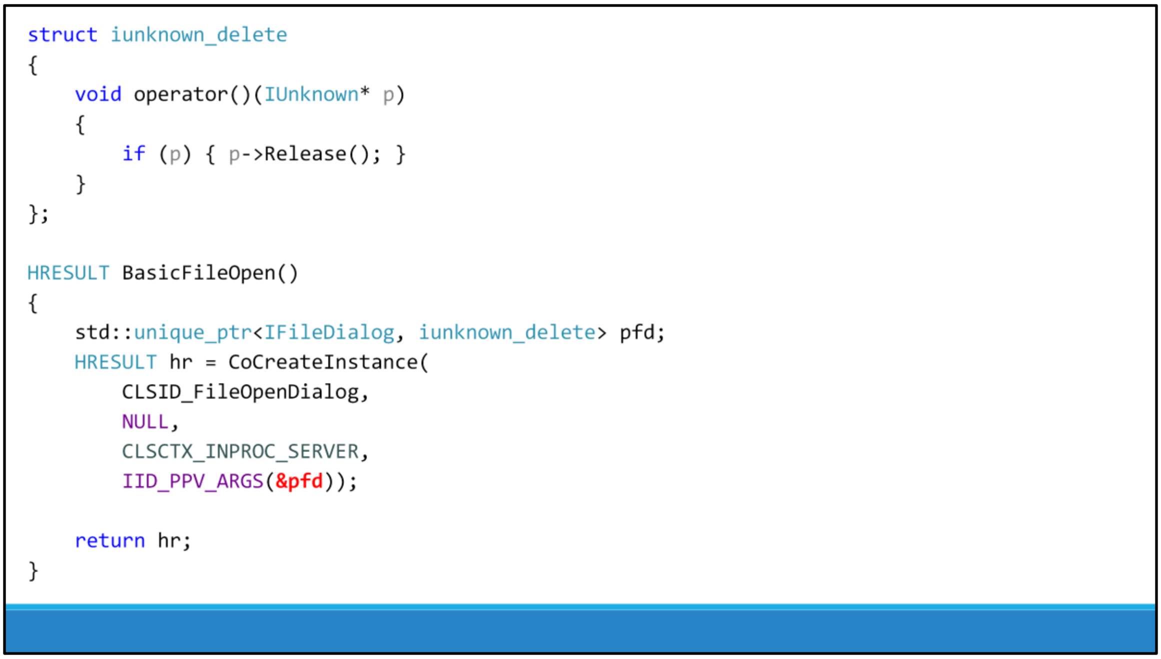
If you have reason to prefer malloc over new for reasons other than the throwing, you can use malloc with unique_ptr, you just have to provide your own deleter as a second template argument and write that deleter to use free.

So wow cool just look for any memory allocation and hand that to unique_ptr? That will solve all my problems? Not always. Unique_ptr is great for allocations that have a single owner. This code (or the previous slide) create the buffers, use them, clean them up. Perfect. Not all code does that.



```
HRESULT BasicFileOpen()
{
    IFileDialog *pfd = NULL;
    HRESULT hr = CoCreateInstance(
        CLSID_FileOpenDialog,
        NULL,
        CLSCTX_INPROC_SERVER,
        IID_PPV_ARGS(&pfd));

    if (SUCCEEDED(hr))
    {
        pfd->Release();
    }

    return hr;
}
```

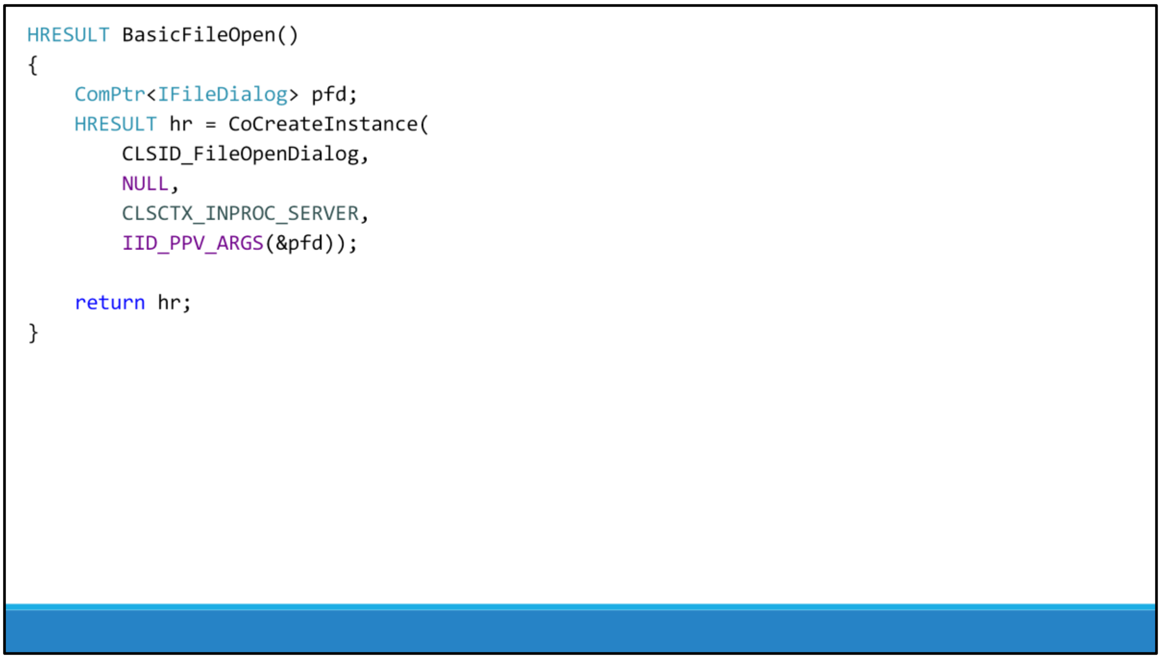
Let’s go back to that file open dialog example. This code just calls CoCreateInstance and then Release. But notice that when it passes the pointer, pfd, to CoCreateInstance it actually passes the address. The address of a unique pointer is not that same as the address of a raw pointer so just making pfd a unique_ptr is not going to work.



```
struct iunknown_delete
{
    void operator()(IUnknown* p)
    {
        if (p) { p->Release(); }
    }
};

HRESULT BasicFileOpen()
{
    std::unique_ptr<IFileDialog, iunknown_delete> pfd;
    HRESULT hr = CoCreateInstance(
        CLSID_FileOpenDialog,
        NULL,
        CLSCTX_INPROC_SERVER,
        IID_PPV_ARGS(&pfd));

    return hr;
}
```

So although I can make a unique_ptr with my own deleter that does the COM release, using it will mean passing the address of a unique_ptr and that's not the type CoCreateInstance expects. And casting won't help unless the address of the unique_ptr also happens to be the address of the raw pointer it's holding inside, which you cannot know.

No worries, and no need to write all this yourself anyway. You're hardly the first COM programmer after all

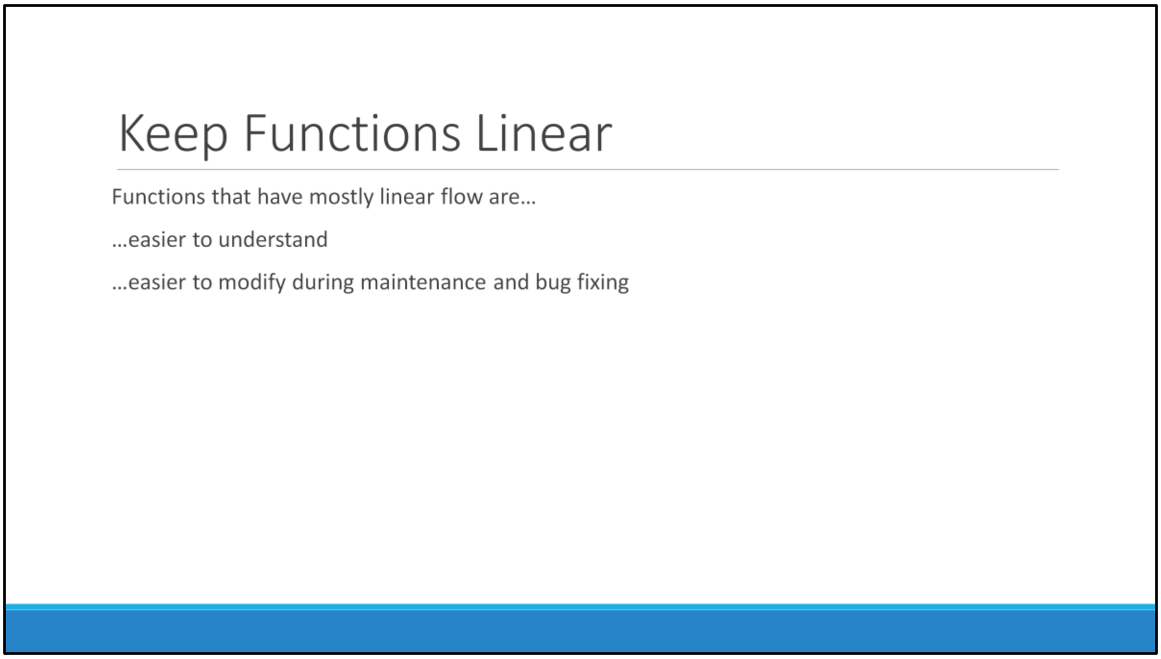


```
HRESULT BasicFileOpen()
{
    ComPtr<IFileDialog> pfd;
    HRESULT hr = CoCreateInstance(
        CLSID_FileOpenDialog,
        NULL,
        CLSCTX_INPROC_SERVER,
        IID_PPV_ARGS(&pfd));

    return hr;
}
```

ComPtr is an RAII container that does the release for you when it goes out of scope, and has an overload of the address-of operator that does what you expect. It’s in the WRL library. There is also a CComPtr in ATL that is similar.

You don’t need to write your own RAII containers for everything, but you do need to know a little bit about how these containers work and which one is right for you.

The moral of this section is not “when to use unique_ptr, when to use unique_ptr with a deleter, when to use ComPtr, when to write your own class” etc. The moral of this section is that RAII will save your butt. It will save your butt because the beauty of invisible code makes the real work of your function more obvious, and because you won’t have leaks if things go wrong partway through. Once you embrace RAII then you will have the specific task of finding a class that someone has written that does what you need, and if you’re doing work that lots of people do, and COM is a great example of that, then you’re likely to find a class that someone else also wrote. But if you have to write the class yourself, that’s hardly a problem: you have the cleanup code you need right there at the bottom of the arrow, so this is a relatively mechanical exercise.



Keep Functions Linear

Functions that have mostly linear flow are…

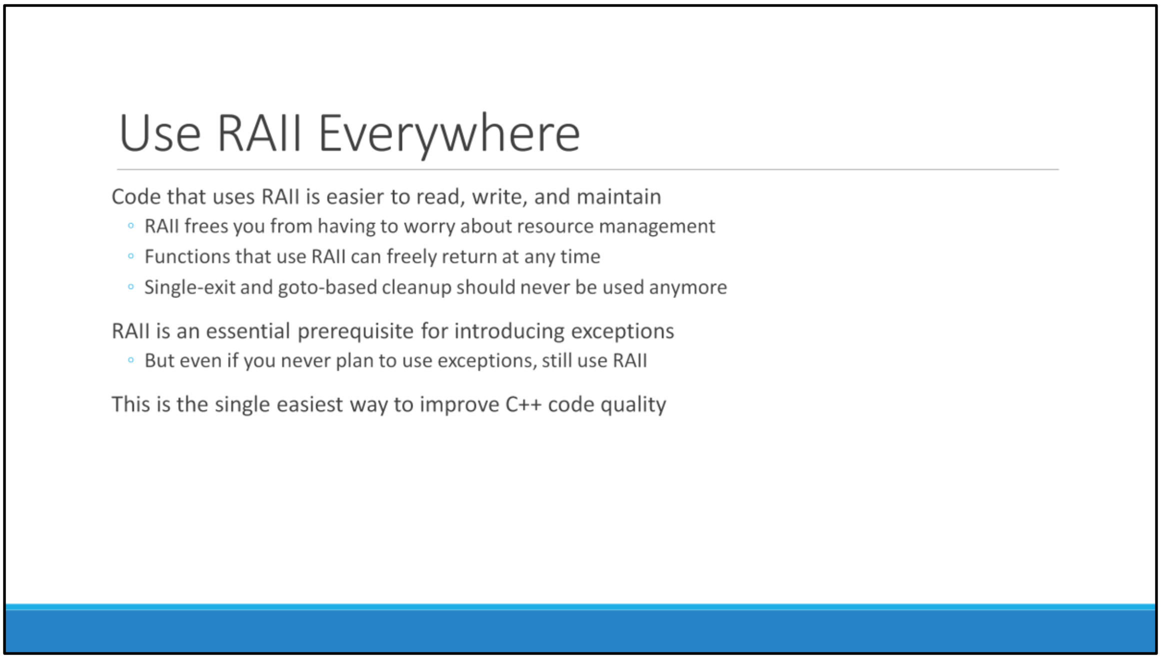
…easier to understand

…easier to modify during maintenance and bug fixing

Just don’t try to achieve linearity through endless copy and paste of similar cleanup code

You can only safely bail in the middle if you know the cleanup will happen.

People can read your code and understand what it does and not be surprised.



Use RAII Everywhere

Code that uses RAII is easier to read, write, and maintain

- RAII frees you from having to worry about resource management
- Functions that use RAII can freely return at any time
- Single-exit and goto-based cleanup should never be used anymore

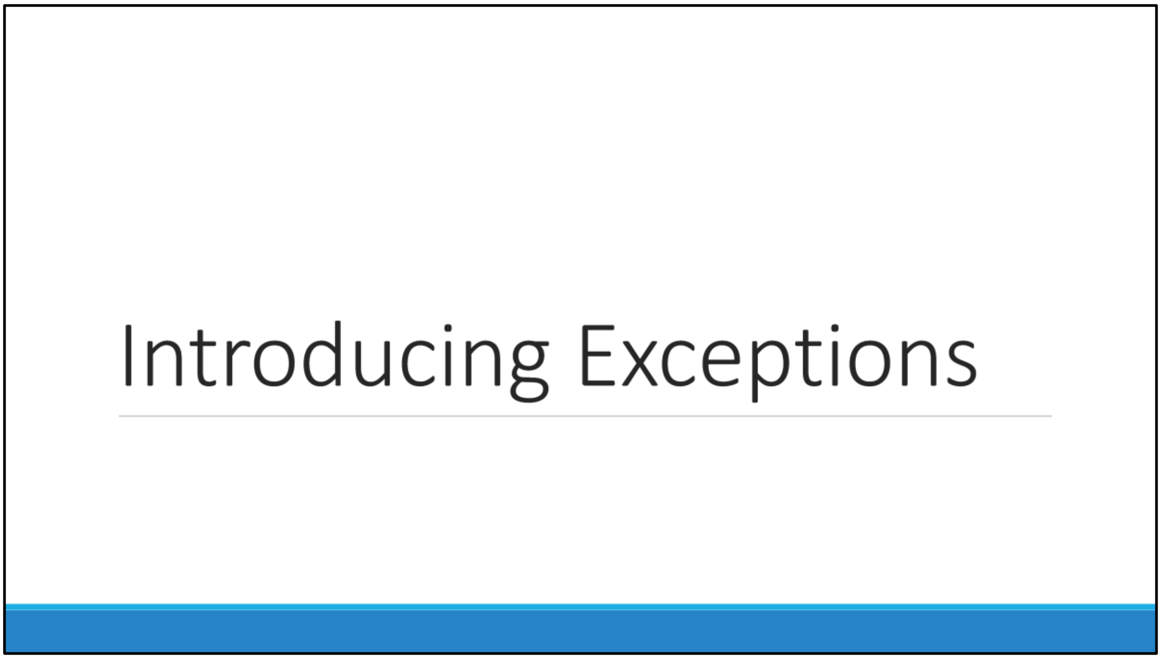
RAII is an essential prerequisite for introducing exceptions

- But even if you never plan to use exceptions, still use RAII

This is the single easiest way to improve C++ code quality

Again just emphasizing that even though most of us were taught RAII in the context of exceptions, they still provide tremendous value in an exception free world. They make cleanup code both invisible and guaranteed to happen. Both of those are major features. After all, encapsulation is still and always a good thing. Detailed inside knowledge like “this needs to be released” or “you need to call Close on this” should have been buried in a class all along and that’s what we’re advocating for.

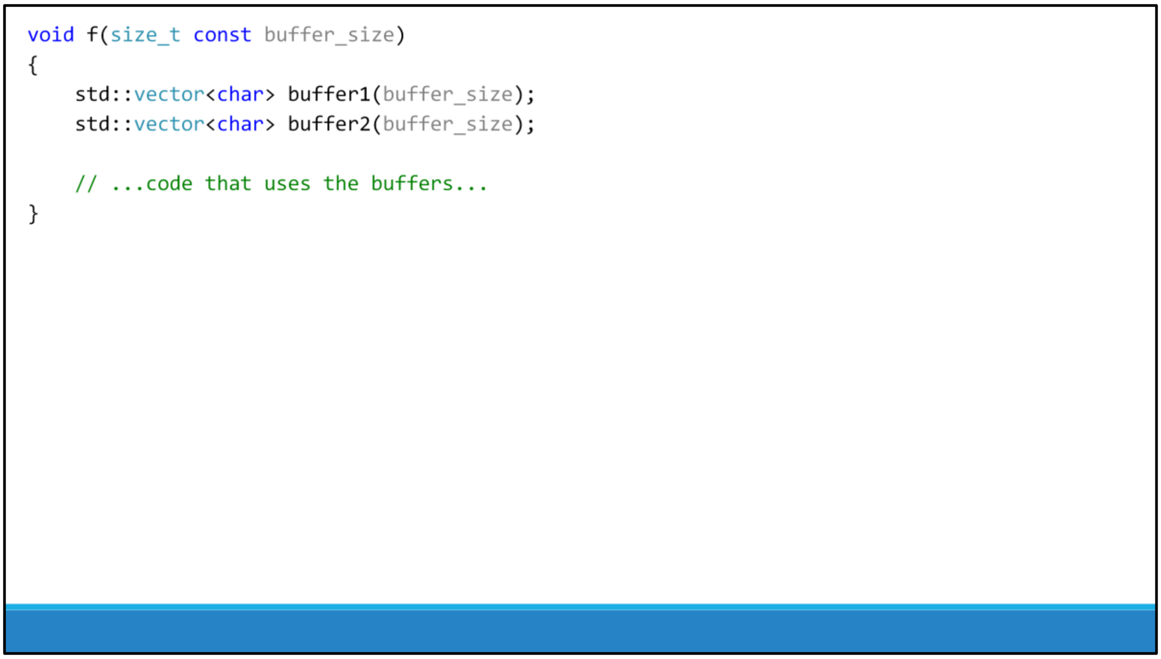
But speaking of exceptions… switch to James



Introducing Exceptions

Switch to James

NEXT: vector example from previous section

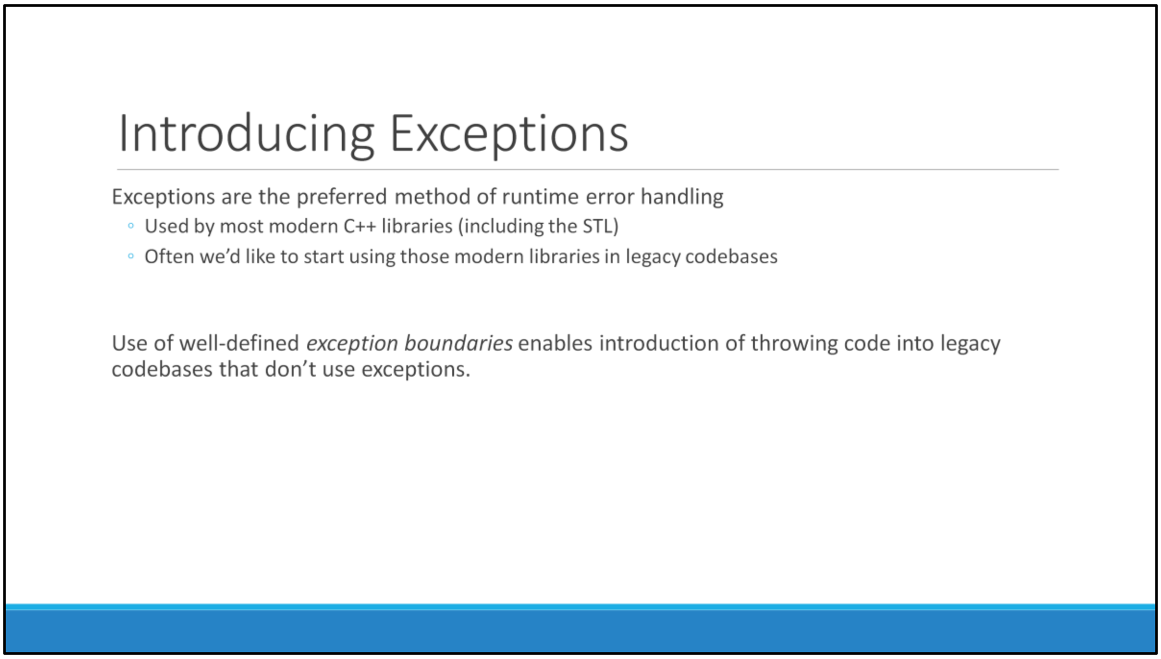


```
void f(size_t const buffer_size)
{
    std::vector<char> buffer1(buffer_size);
    std::vector<char> buffer2(buffer_size);

    // ...code that uses the buffers...
}
```

Let’s go back to a slide that we had in the previous section…
…ideally, we really should use std::vector for those heap-allocated buffers.
…but we can’t just start using the STL in code that isn’t exception-safe
…that would be a disaster.

But let’s say that we really want to start using exceptions in some part of our library

NEXT: Introducing exceptions intro



Introducing Exceptions

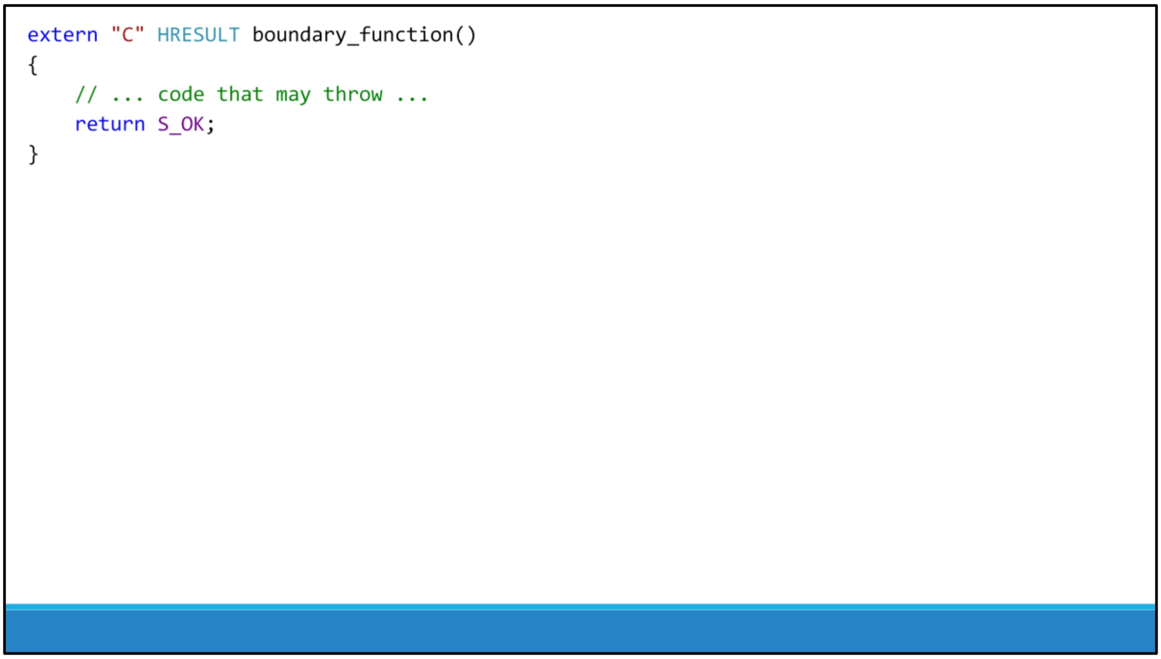
Exceptions are the preferred method of runtime error handling

- Used by most modern C++ libraries (including the STL)
- Often we'd like to start using those modern libraries in legacy codebases

Use of well-defined *exception boundaries* enables introduction of throwing code into legacy codebases that don't use exceptions.

In many cases it's not so much that you want to use exceptions, it's that you want to use new or you want to use a library class like vector<> that might throw, and that means that you're using exceptions whether you actually intended to or not.

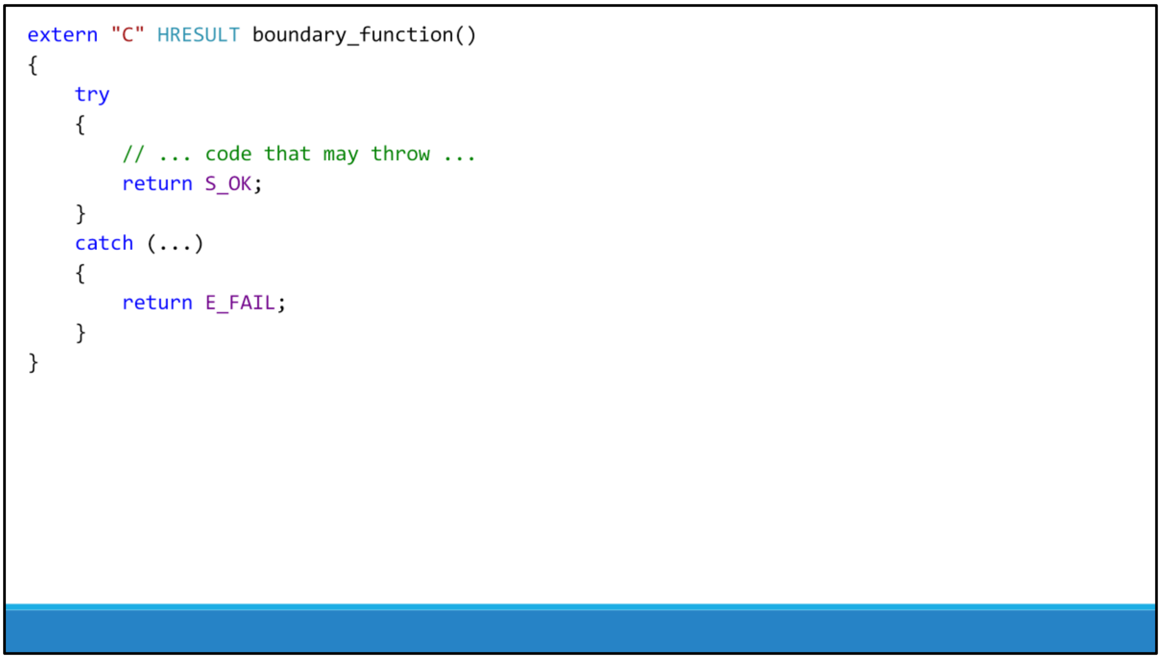
NEXT: Basic boundary function



```
extern "C" HRESULT boundary_function()
{
    // ... code that may throw ...
    return S_OK;
}
```

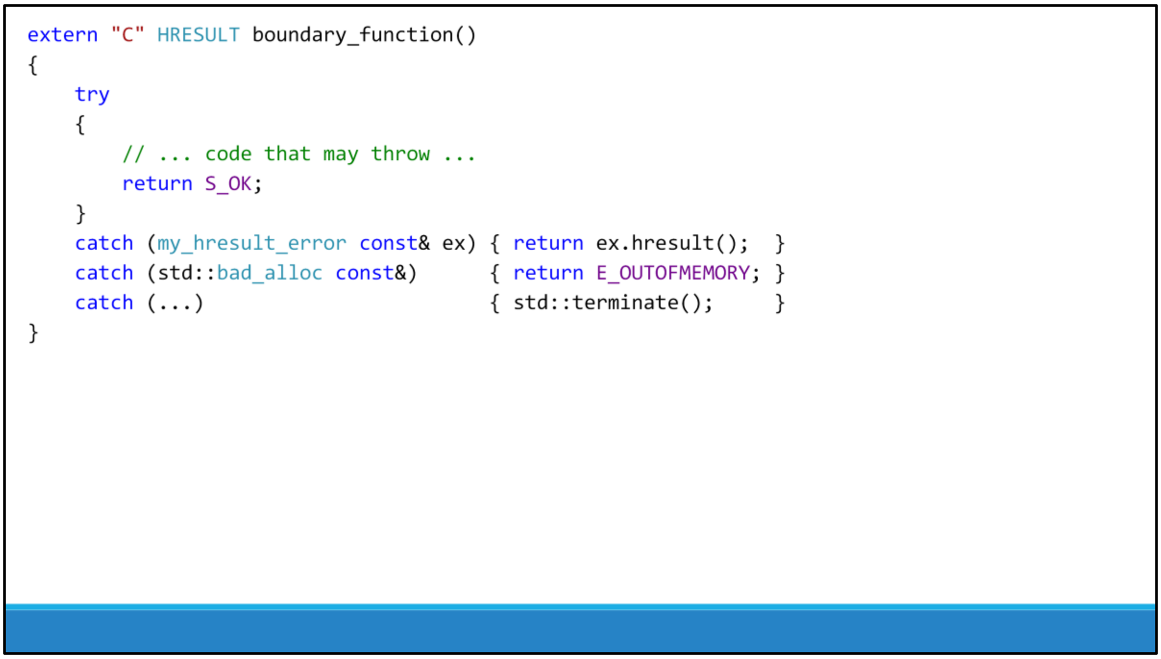
An exception boundary function uses things that may throw, but it itself does not throw.

NEXT: Basic try/catch wrapper



```
extern "C" HRESULT boundary_function()
{
    try
    {
        // ... code that may throw ...
        return S_OK;
    }
    catch (...)
    {
        return E_FAIL;
    }
}
```

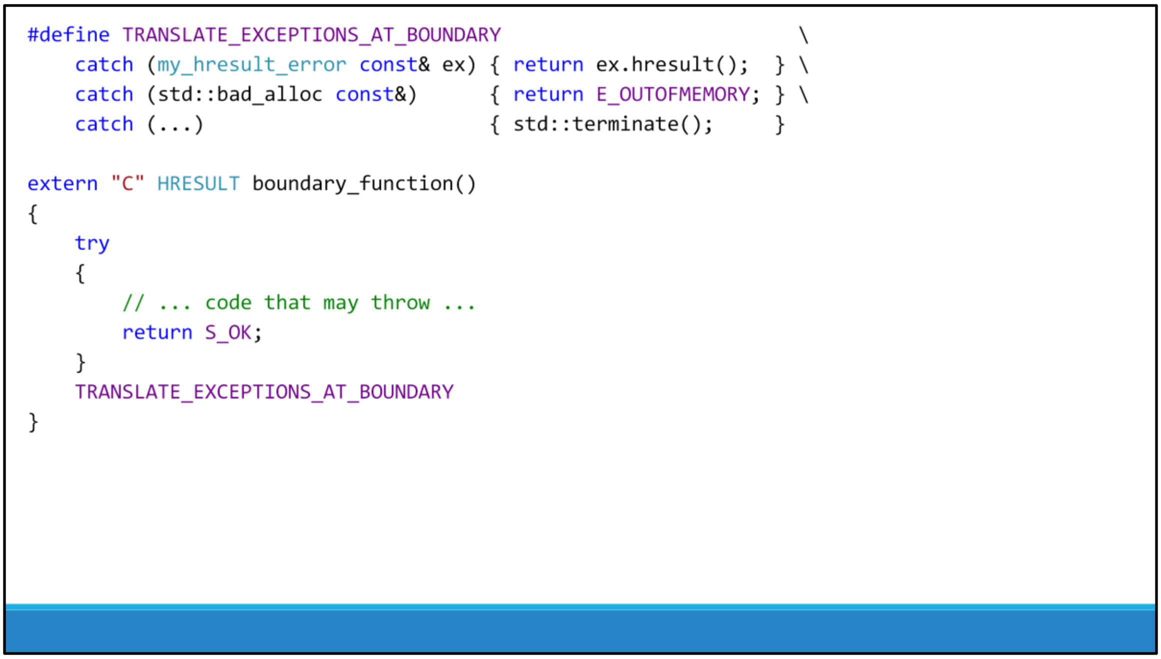
NEXT: Translating to actual error codes



```cpp
extern "C" HRESULT boundary_function()
{
    try
    {
        // ... code that may throw ...
        return S_OK;
    }
    catch (my_hresult_error const& ex) { return ex.hresult();  }
    catch (std::bad_alloc const&)      { return E_OUTOFMEMORY; }
    catch (...)                        { std::terminate();     }
}
```

Too much code here; we’re likely to repeat this code everywhere and to need to update or expand this code in hundreds of places to take care of a new expected exception that doesn’t warrant calling terminate.

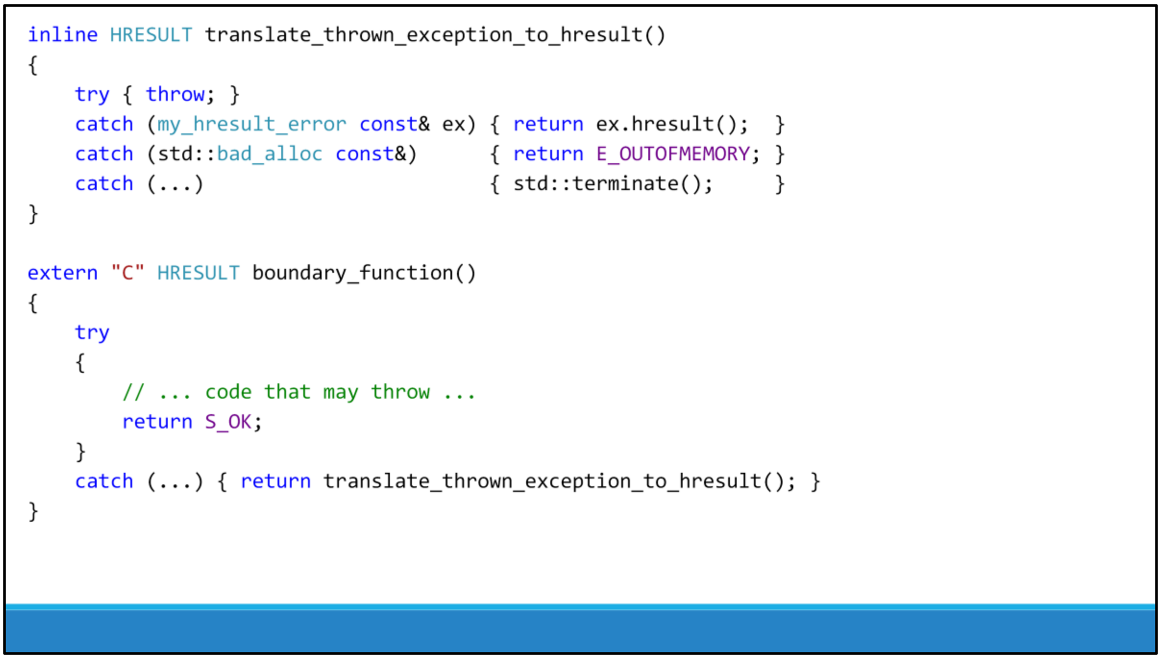
NEXT: Macro to encapsulate catches



```
#define TRANSLATE_EXCEPTIONS_AT_BOUNDARY                         \
    catch (my_hresult_error const& ex) { return ex.hresult();  } \
    catch (std::bad_alloc const&)      { return E_OUTOFMEMORY; } \
    catch (...)                        { std::terminate();     }

extern "C" HRESULT boundary_function()
{
    try
    {
        // ... code that may throw ...
        return S_OK;
    }
    TRANSLATE_EXCEPTIONS_AT_BOUNDARY
}
```

OK, yuck, but hands up if you’ve done it? People have.

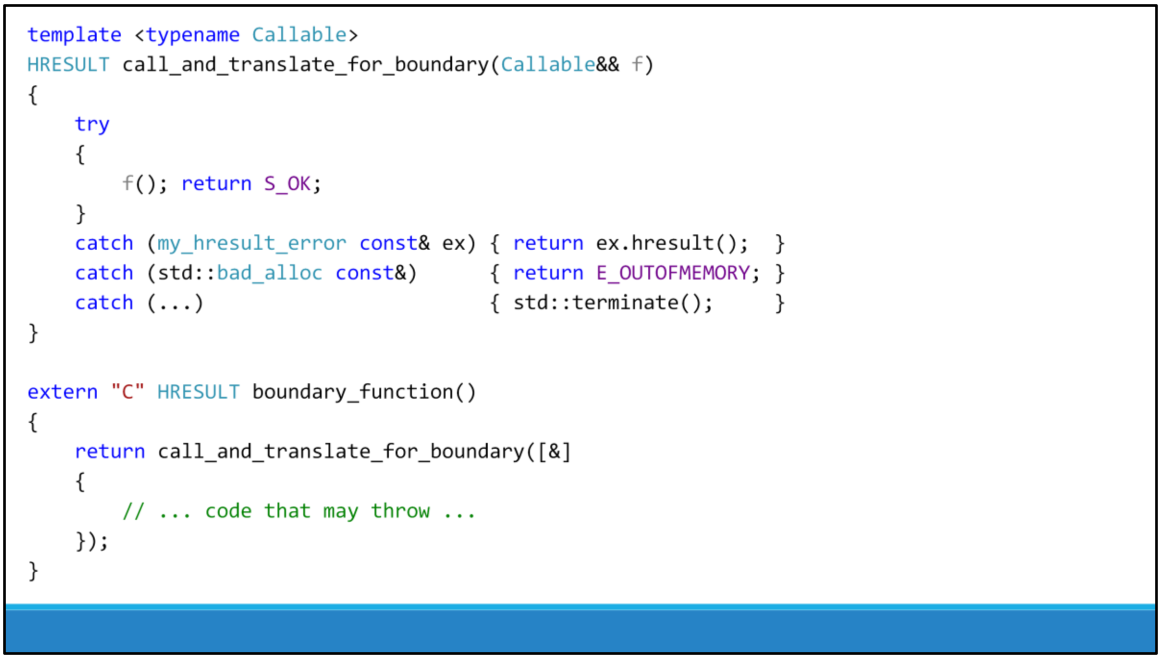
NEXT: Function to encapsulate catches



```
inline HRESULT translate_thrown_exception_to_hresult()
{
    try { throw; }
    catch (my_hresult_error const& ex) { return ex.hresult();  }
    catch (std::bad_alloc const&)      { return E_OUTOFMEMORY; }
    catch (...)                        { std::terminate();     }
}

extern "C" HRESULT boundary_function()
{
    try
    {
        // ... code that may throw ...
        return S_OK;
    }
    catch (...) { return translate_thrown_exception_to_hresult(); }
}
```

Lippincott function. Same effect and much cleaner.

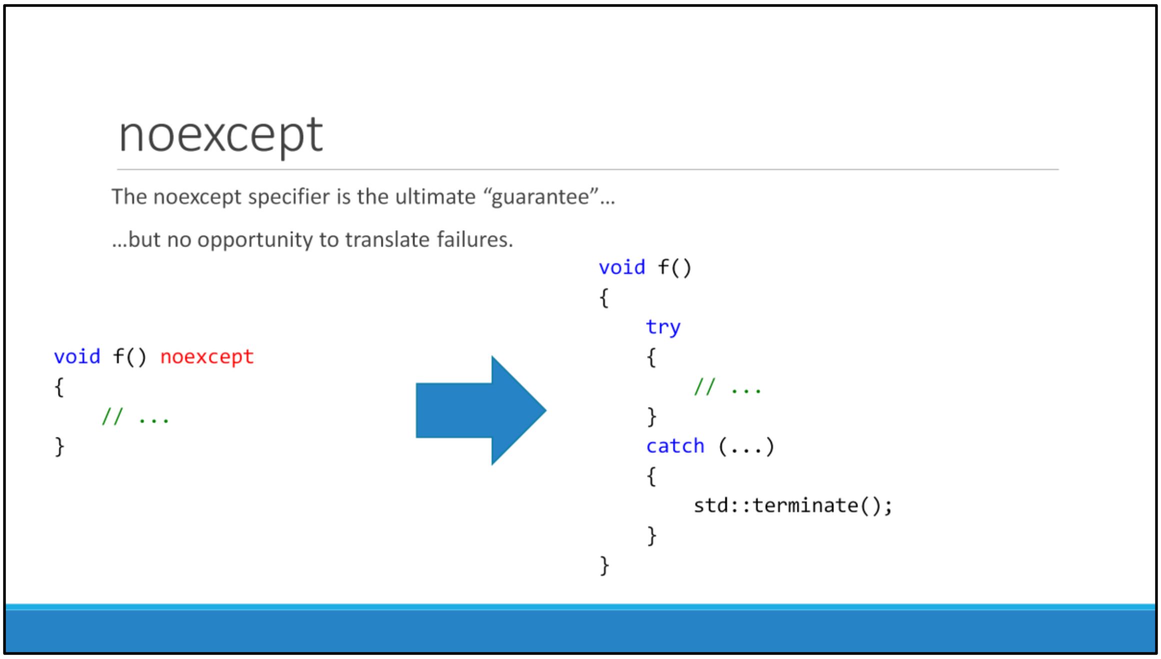
NEXT: Final beautiful lambda example



```cpp
template <typename Callable>
HRESULT call_and_translate_for_boundary(Callable&& f)
{
    try
    {
        f(); return S_OK;
    }
    catch (my_hresult_error const& ex) { return ex.hresult();  }
    catch (std::bad_alloc const&)      { return E_OUTOFMEMORY; }
    catch (...)                        { std::terminate();     }
}

extern "C" HRESULT boundary_function()
{
    return call_and_translate_for_boundary([&]
    {
        // ... code that may throw ...
    });
}
```

This makes the call site nice and clean. You don’t have to use a lambda, the Callable could be an entire throwing function also.

Unlike the macro, you can debug through this. You can adapt this approach for anything where you need to run some code in a different context.



noexcept

The noexcept specifier is the ultimate "guarantee"…

…but no opportunity to translate failures.

```
void f() noexcept
{
    // ...
}
```

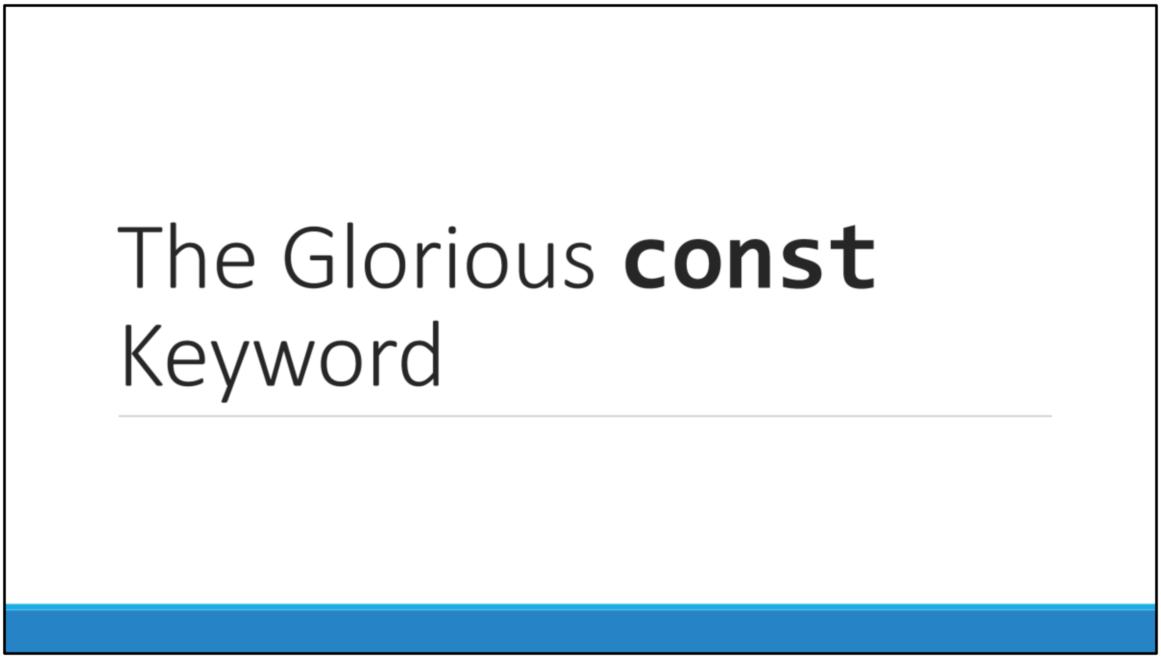
```
void f()
{
    try
    {
        // ...
    }
    catch (...)
    {
        std::terminate();
    }
}
```

This is an imperfect example here: Whether the stack is unwound is an implementation detail
 * In the Visual C++ implementation, it is not, in order to preserve the call stack in crash dumps

The compiler can optimize away the try/catch in places where it knows the code can't throw (but this is a general optimization, not limited to noexcept)

noexcept may offer some performance benefit in callers (EH state needs not be tracked across the call)

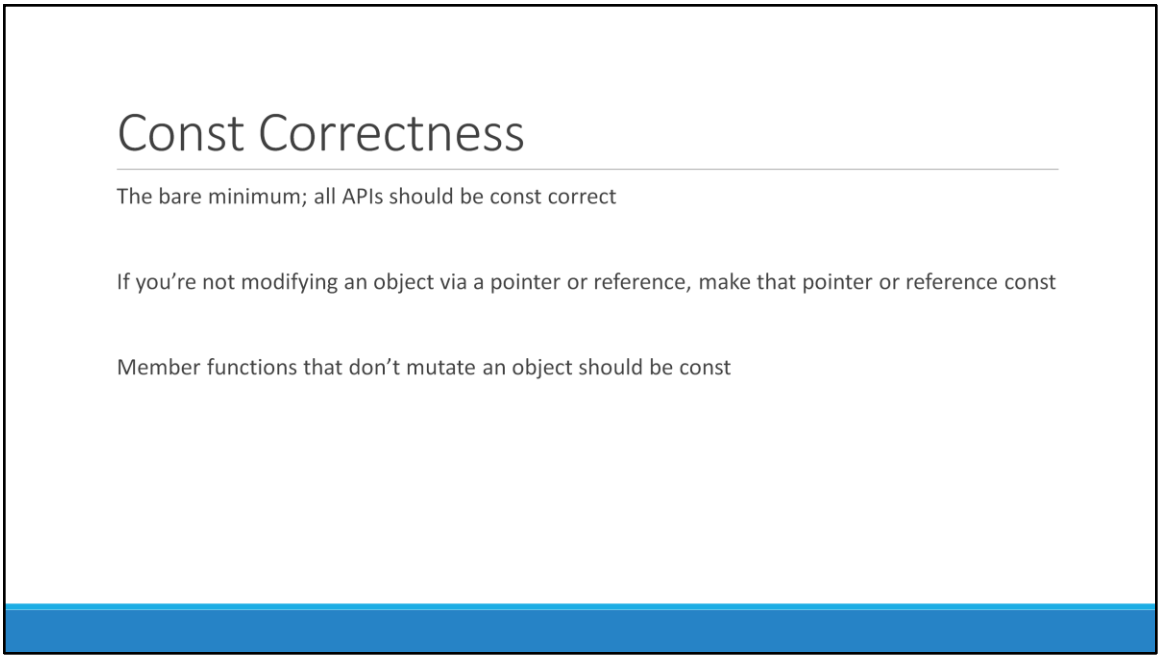
NEXT: SECTION: const



The Glorious **const** Keyword

Switch to Kate

Protects you from errors of thought
Enables some optimizations
viral



Const Correctness

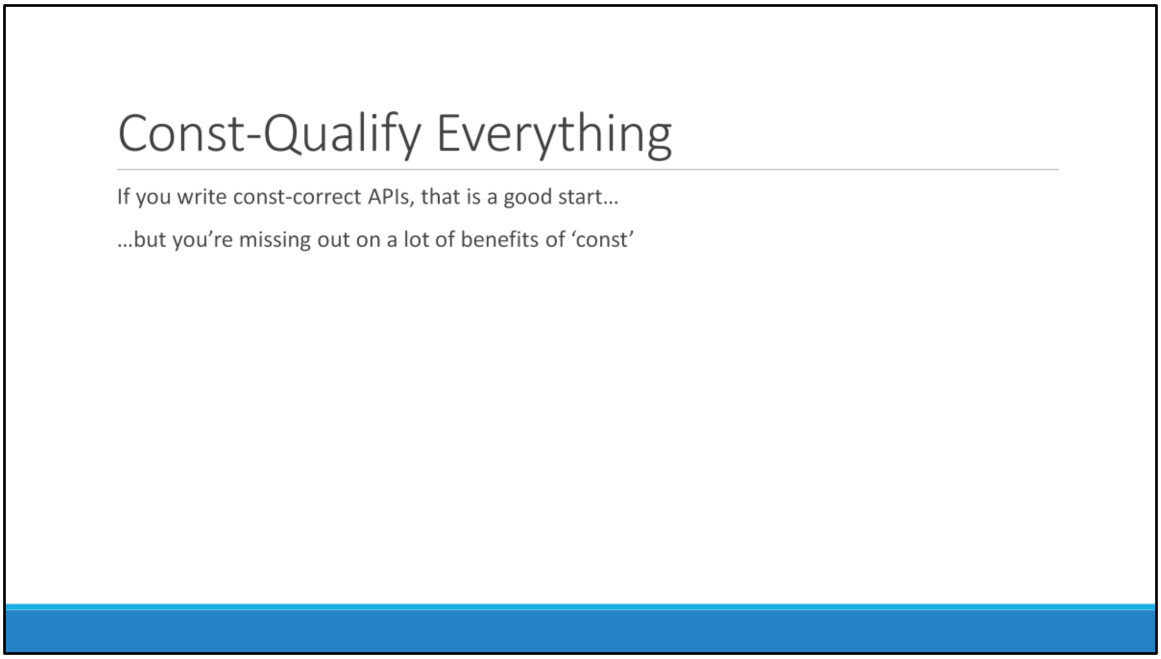
The bare minimum; all APIs should be const correct

If you’re not modifying an object via a pointer or reference, make that pointer or reference const

Member functions that don’t mutate an object should be const

Assumption: you’re at this point already. Your code compiles and you’ve marked member functions and parameters const where you **had to** in order to get it to compile. What we’re proposing in this section is that you go beyond that, and that it’s worth your time and effort to go beyond that.

This will enable you to understand and safely change the legacy code you’re working on.



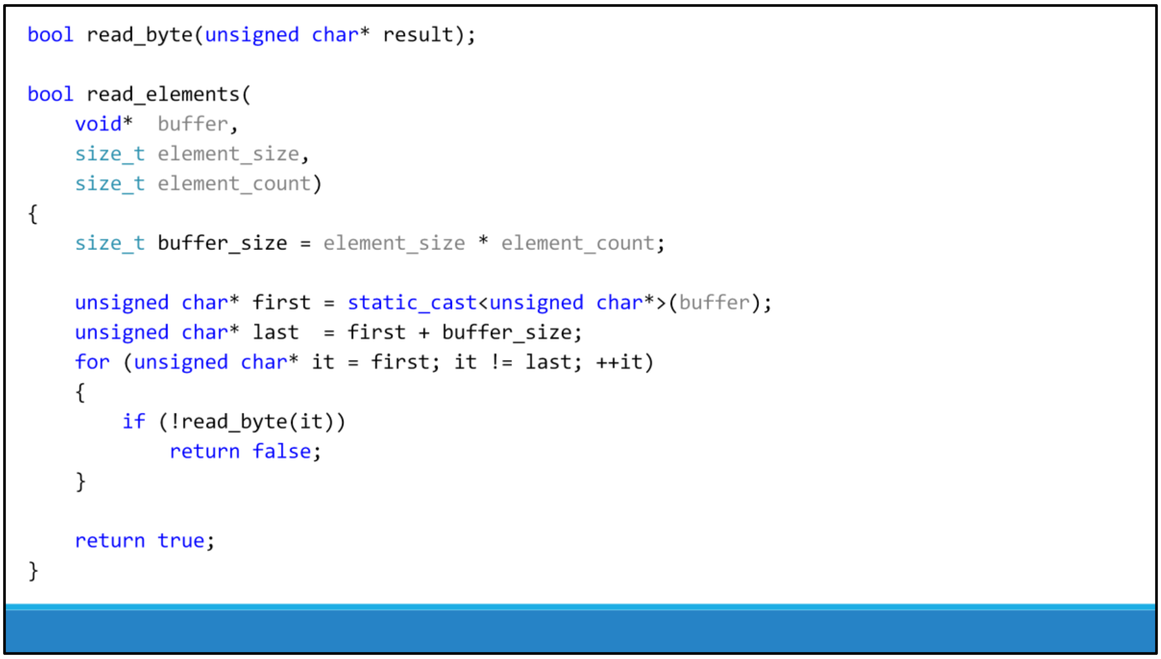
Const-Qualify Everything

If you write const-correct APIs, that is a good start…

…but you're missing out on a lot of benefits of 'const'

Adding const and seeing what breaks is actually a useful way to understand legacy code. Mark everything const that you possibly can, not just that you have to.

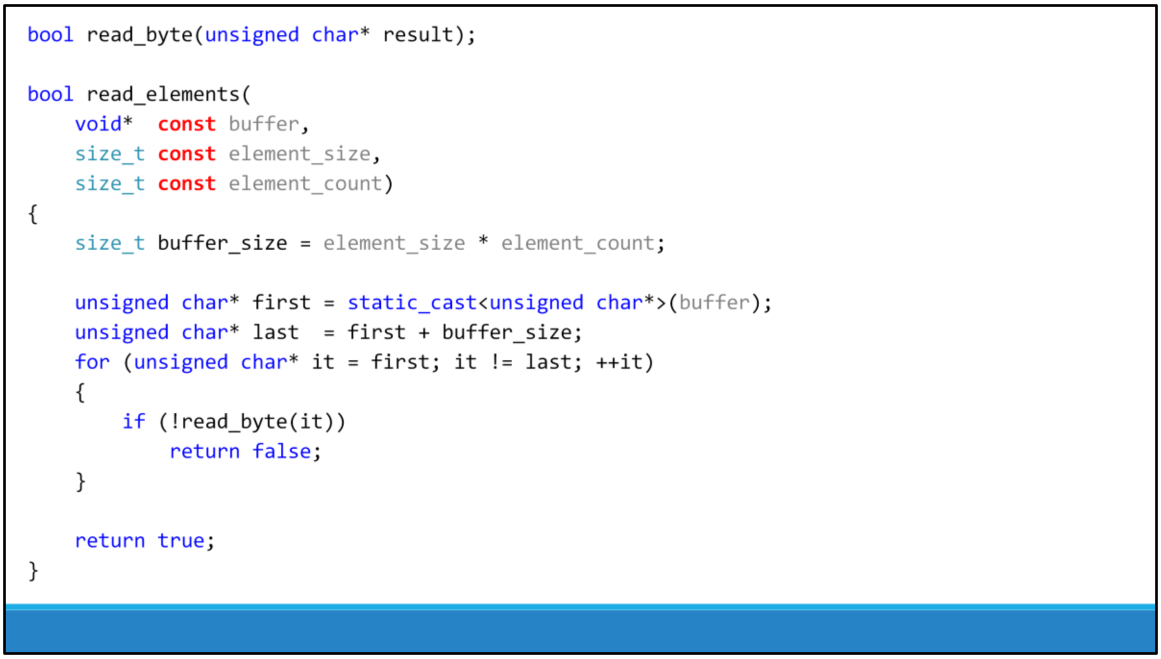
Why? Because the compiler is your friend.



```
bool read_byte(unsigned char* result);

bool read_elements(
    void*  buffer,
    size_t element_size,
    size_t element_count)
{
    size_t buffer_size = element_size * element_count;

    unsigned char* first = static_cast<unsigned char*>(buffer);
    unsigned char* last  = first + buffer_size;
    for (unsigned char* it = first; it != last; ++it)
    {
        if (!read_byte(it))
            return false;
    }

    return true;
}
```

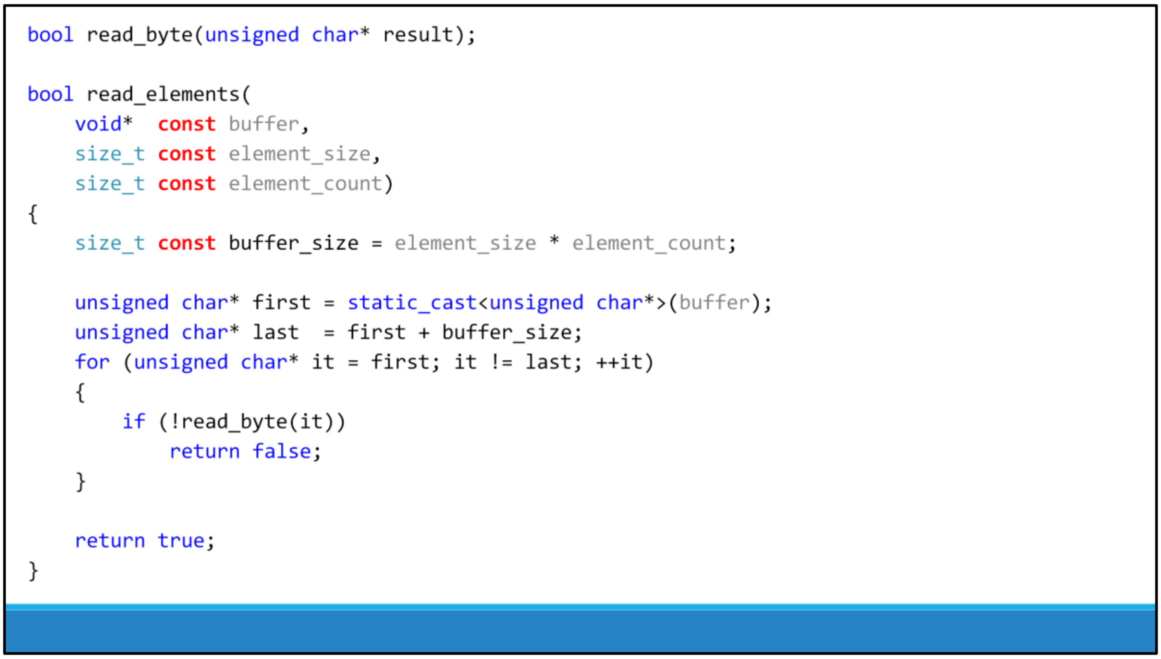
Skipping overflow check on buffer size

What can we const qualify? First, do we change the parameters? Clearly we're going to change the contents of the memory buffer points to, by reading into it, but we don't intend to change the pointer, or the size or count.



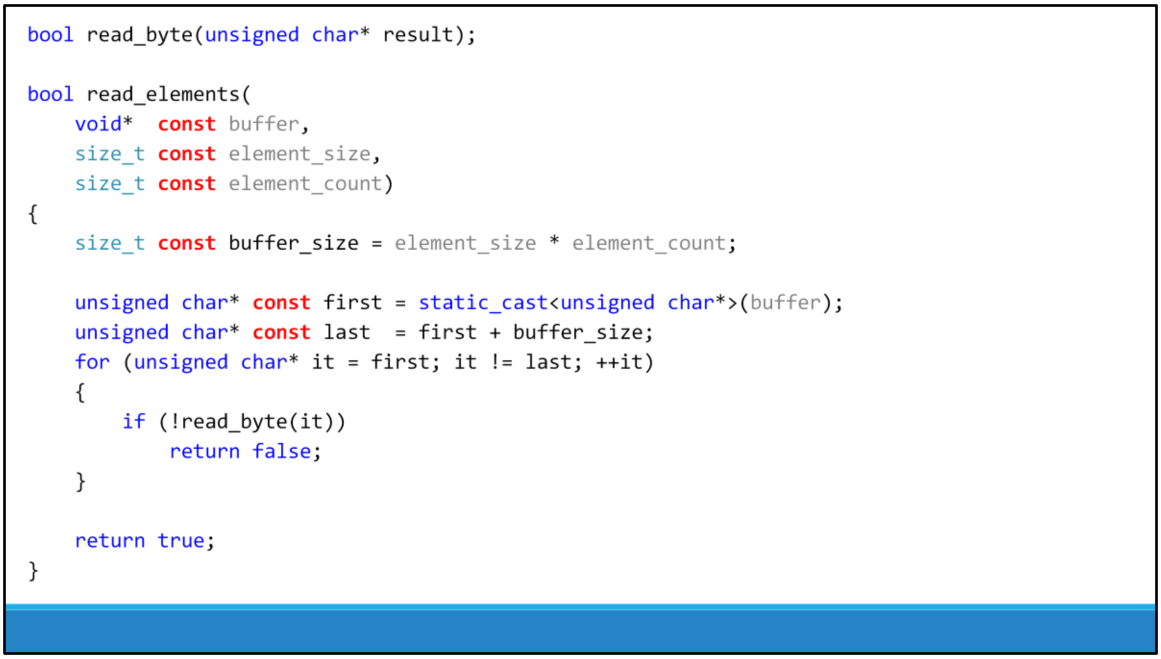
```
bool read_byte(unsigned char* result);

bool read_elements(
    void*  const buffer,
    size_t const element_size,
    size_t const element_count)
{
    size_t buffer_size = element_size * element_count;

    unsigned char* first = static_cast<unsigned char*>(buffer);
    unsigned char* last  = first + buffer_size;
    for (unsigned char* it = first; it != last; ++it)
    {
        if (!read_byte(it))
            return false;
    }

    return true;
}
```

What else? The size, once calculated, isn’t going to change.



```
bool read_byte(unsigned char* result);

bool read_elements(
    void*  const buffer,
    size_t const element_size,
    size_t const element_count)
{
    size_t const buffer_size = element_size * element_count;

    unsigned char* first = static_cast<unsigned char*>(buffer);
    unsigned char* last  = first + buffer_size;
    for (unsigned char* it = first; it != last; ++it)
    {
        if (!read_byte(it))
            return false;
    }

    return true;
}
```

And the “iterators” (pointers really, but mentally they’re iterators) for first and last aren’t going to change either



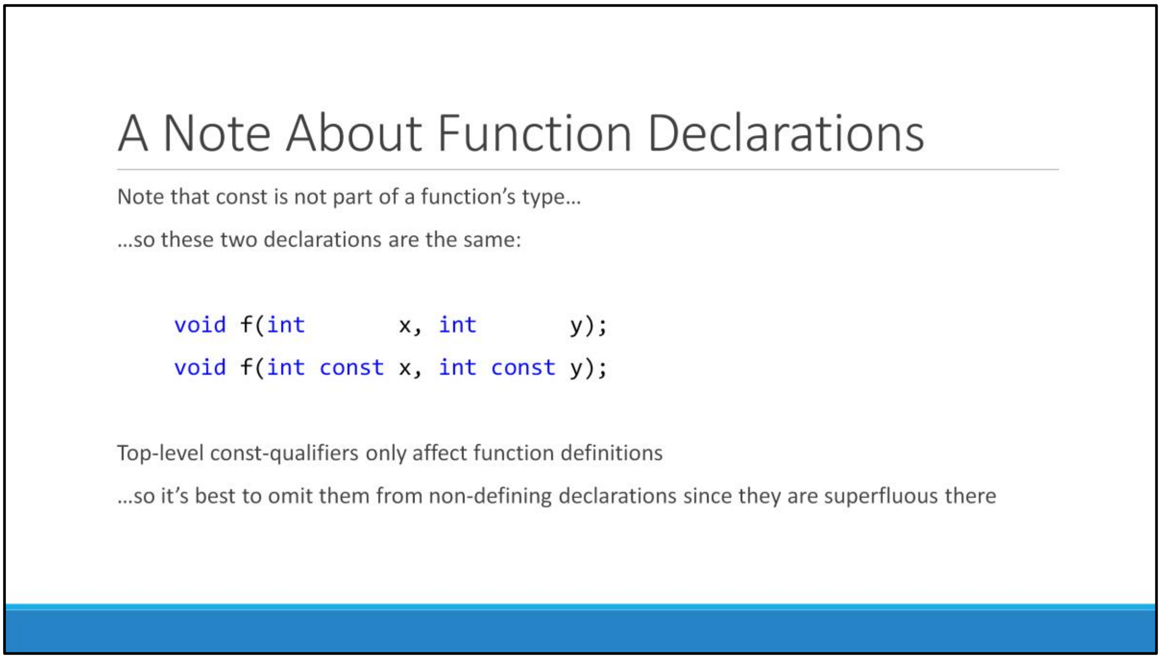
```
bool read_byte(unsigned char* result);

bool read_elements(
    void*  const buffer,
    size_t const element_size,
    size_t const element_count)
{
    size_t const buffer_size = element_size * element_count;

    unsigned char* const first = static_cast<unsigned char*>(buffer);
    unsigned char* const last  = first + buffer_size;
    for (unsigned char* it = first; it != last; ++it)
    {
        if (!read_byte(it))
            return false;
    }

    return true;
}
```

So?
First of all, if there is any actual bug that changes some of these things, you’ll get a compiler error
Code with less moving parts is easier to debug and understand. Here we’re labelling the parts that don’t move. You’re reducing the surface area of what you have to hold in your head while you refactor.

We haven’t done anything viral here because the only thing we pass into some other function is it, and we didn’t mark that const because we increment it ourselves.

By going through, marking things const, making sure it still builds, marking more things const, making sure it still builds, you’re making sure that your mental picture of this code is actually correct.

Insert chit chat here about const on the left/right before/after
My heart says “const int” but my fingers type “int const” and yours should too



A Note About Function Declarations

Note that const is not part of a function’s type…

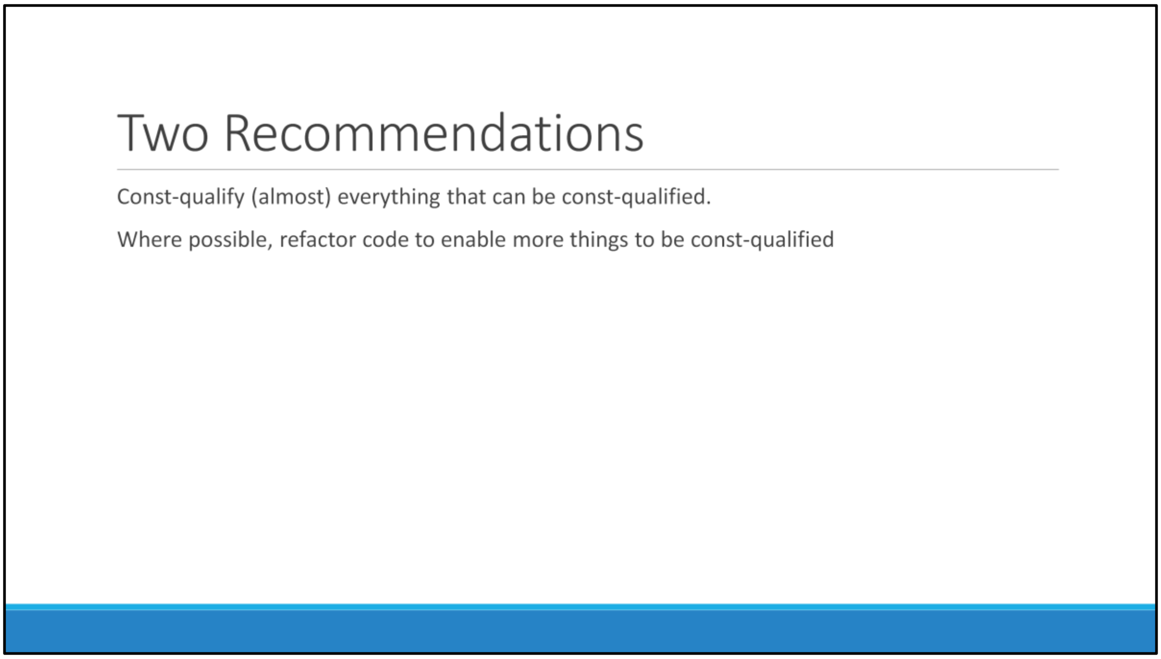
…so these two declarations are the same:

```
void f(int       x, int       y);
void f(int const x, int const y);
```

Top-level const-qualifiers only affect function definitions

…so it’s best to omit them from non-defining declarations since they are superfluous there

Basically put it everywhere you can, it will make your life easier

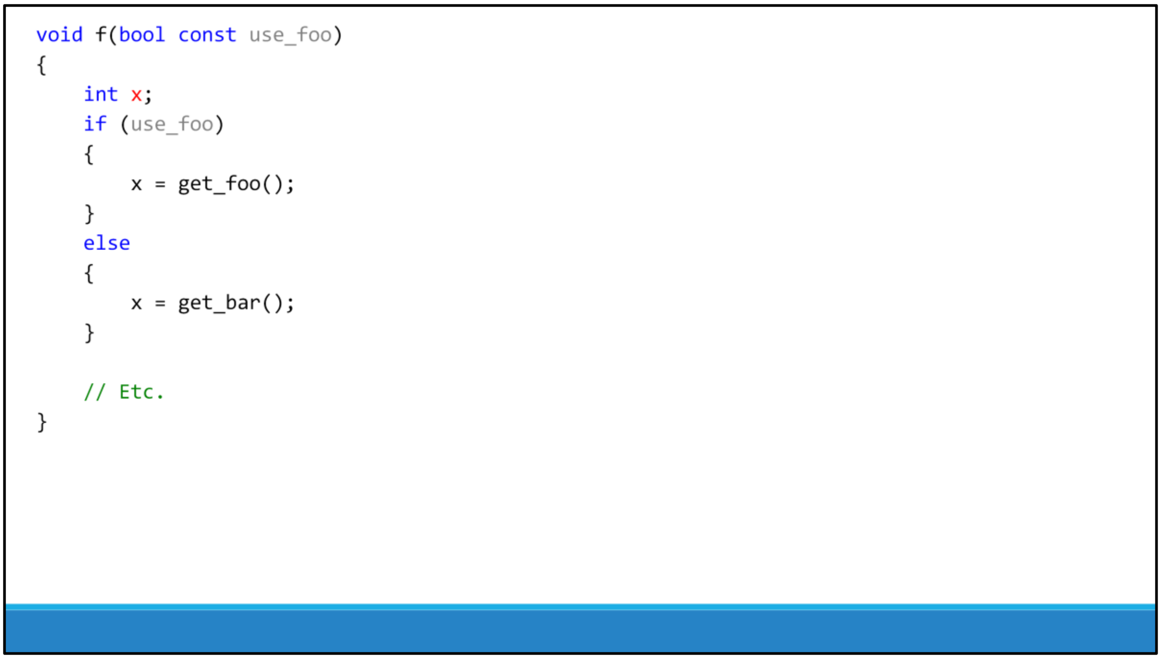


Two Recommendations

Const-qualify (almost) everything that can be const-qualified.

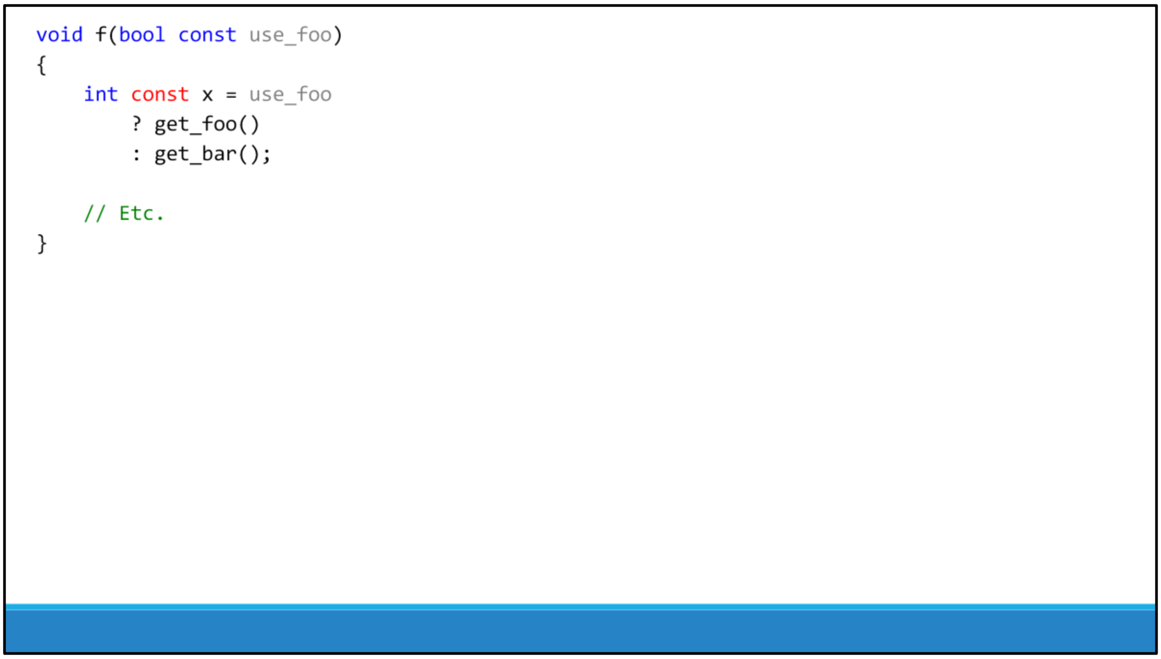
Where possible, refactor code to enable more things to be const-qualified

Basically put it everywhere you can, it will make your life easier



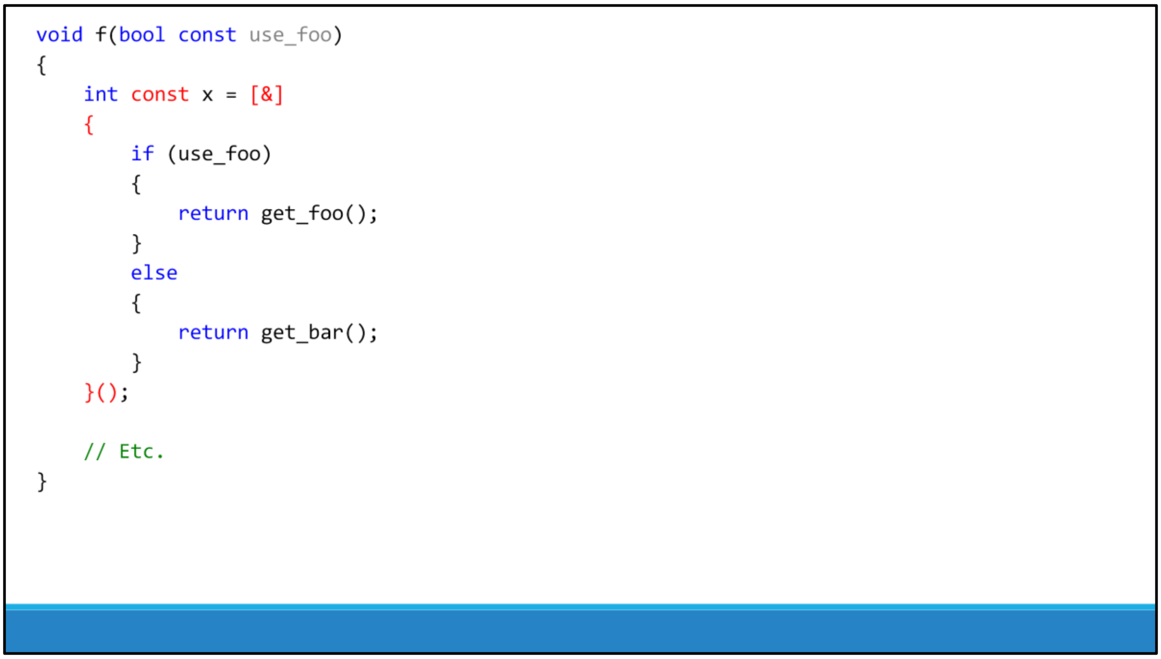
```cpp
void f(bool const use_foo)
{
    int x;
    if (use_foo)
    {
        x = get_foo();
    }
    else
    {
        x = get_bar();
    }

    // Etc.
}
```

Here’s an example: we may have some variable that requires some “complex” initialization



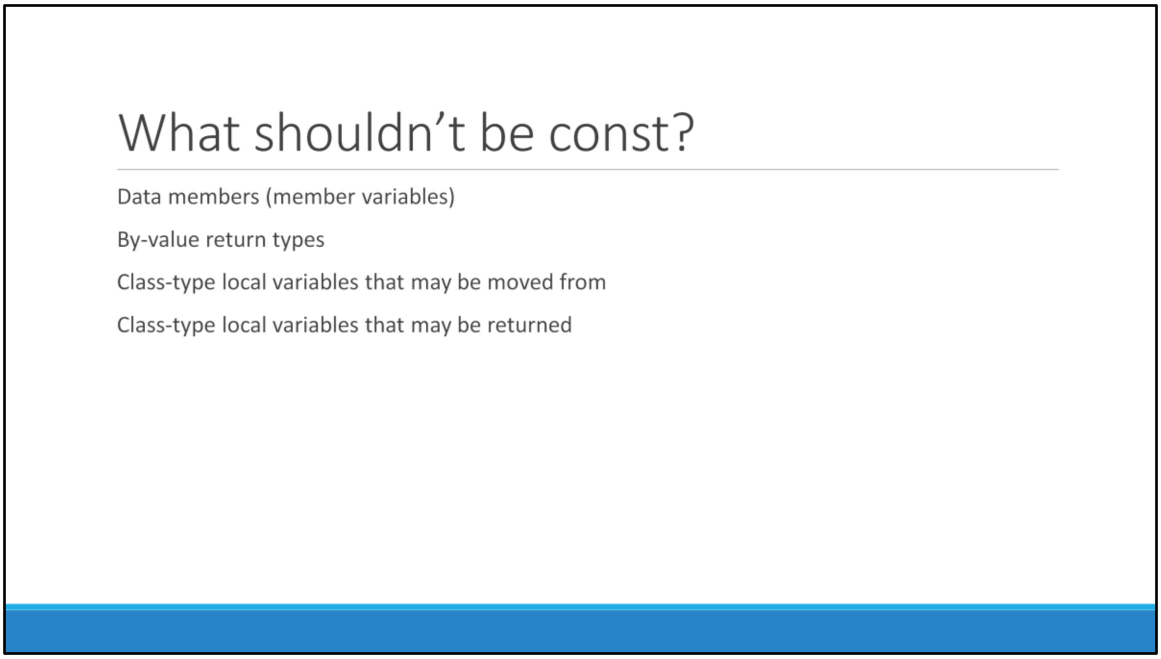
```
void f(bool const use_foo)
{
    int const x = use_foo
        ? get_foo()
        : get_bar();

    // Etc.
}
```

In this not-so-complex case, we could use the conditional operator
(James loves this.)



```
void f(bool const use_foo)
{
    int const x = [&]
    {
        if (use_foo)
        {
            return get_foo();
        }
        else
        {
            return get_bar();
        }
    }();

    // Etc.
}
```

In more complex cases, consider using a lambda expression



What shouldn’t be const?

Data members (member variables)

By-value return types

Class-type local variables that may be moved from

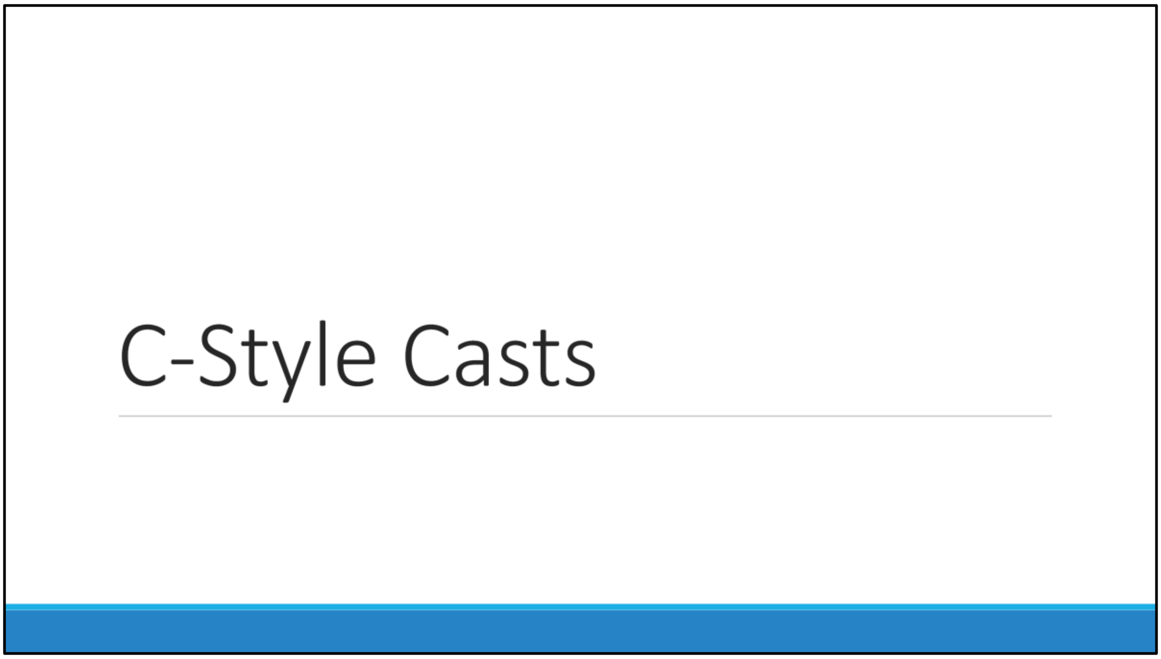
Class-type local variables that may be returned

Making a data member const prevents assignment of objects. If that’s what you want (immutability) it’s best to express that through the interface of the object. If your class is for some reason already non copyable this is less of an issue.

Returning a const value ties the hands of code that calls you for no good reason (pointer or ref is different)

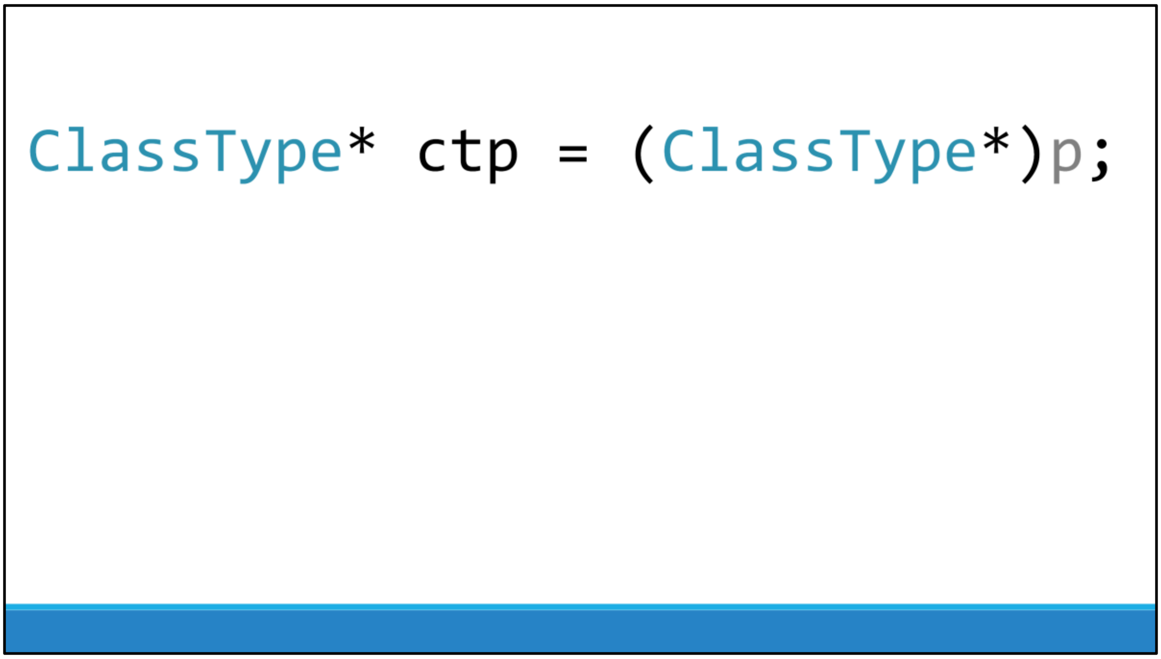
Move semantics and rvalue references are super cool but moving changes an object so it can’t be const. That means const temporaries will get copied from instead of moved (with no warning) for a perf hit.

When you return by value there appears to be a copy but modern compilers have RVO, return value optimization, that saves you the copy with a move or just by eliding it, however if you declared a const local variable and then return it, you will have to pay for the copy, same as any other const temporary. So don’t overuse const.



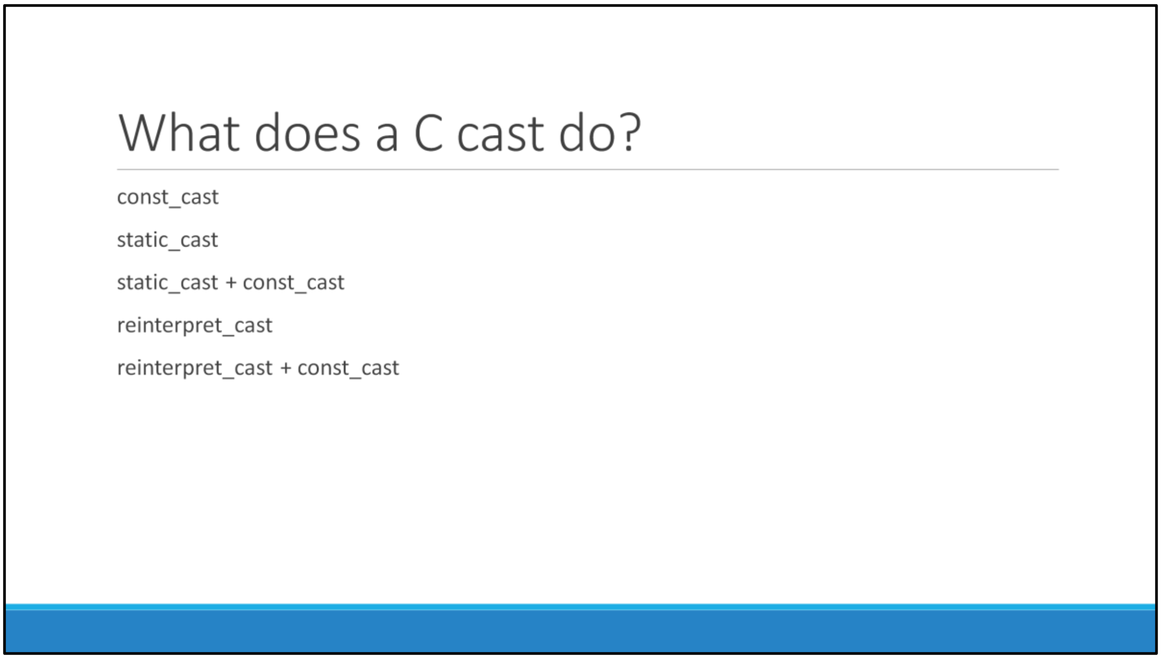
C-Style Casts

Switch to James



```
ClassType* ctp = (ClassType*)p;
```

What does this do? Can you know without knowing what p is and seeing the definition of ClassType?



What does a C cast do?

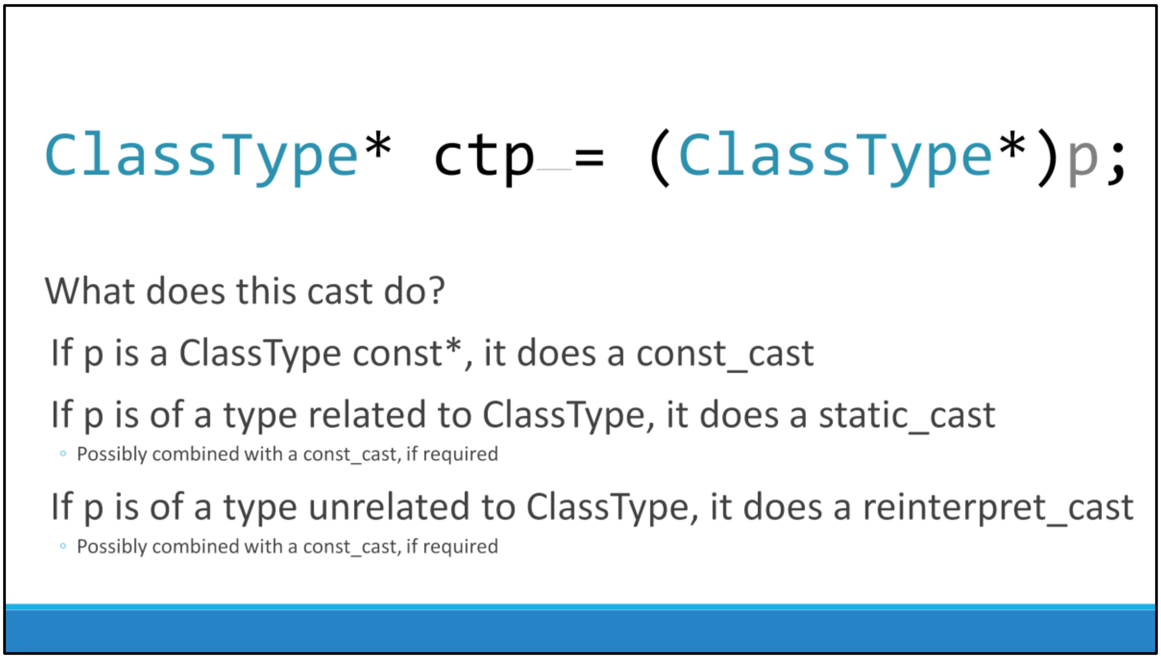
const_cast

static_cast

static_cast + const_cast

reinterpret_cast

reinterpret_cast + const_cast



```
ClassType* ctp = (ClassType*)p;
```

What does this cast do?

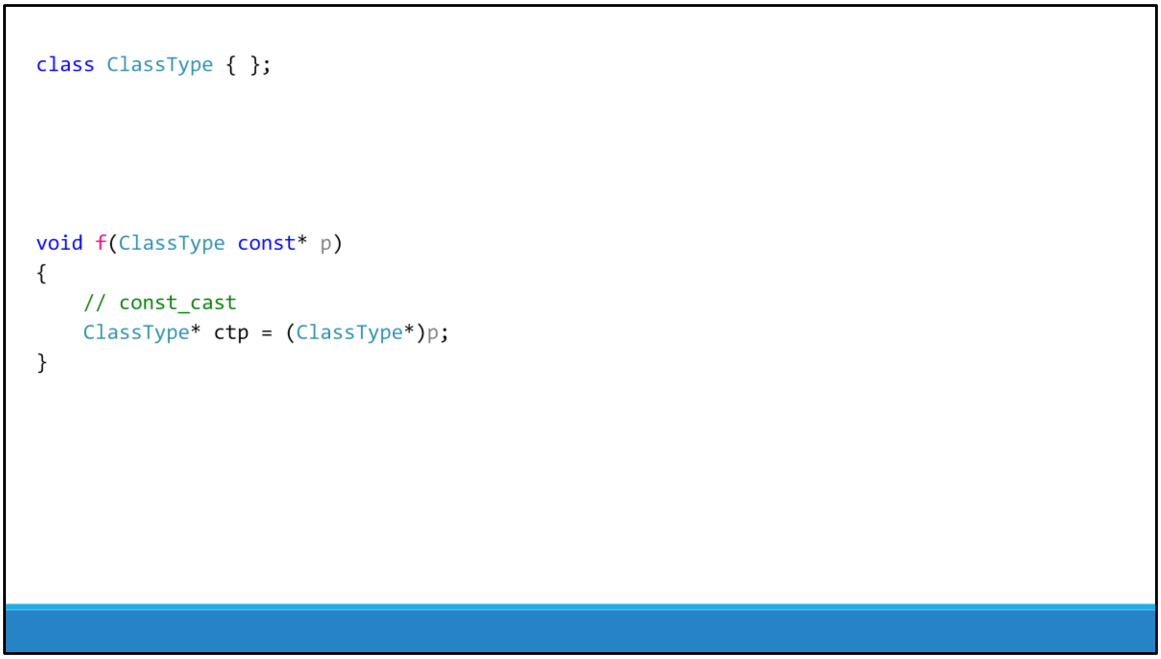
If p is a ClassType const*, it does a const_cast

If p is of a type related to ClassType, it does a static_cast

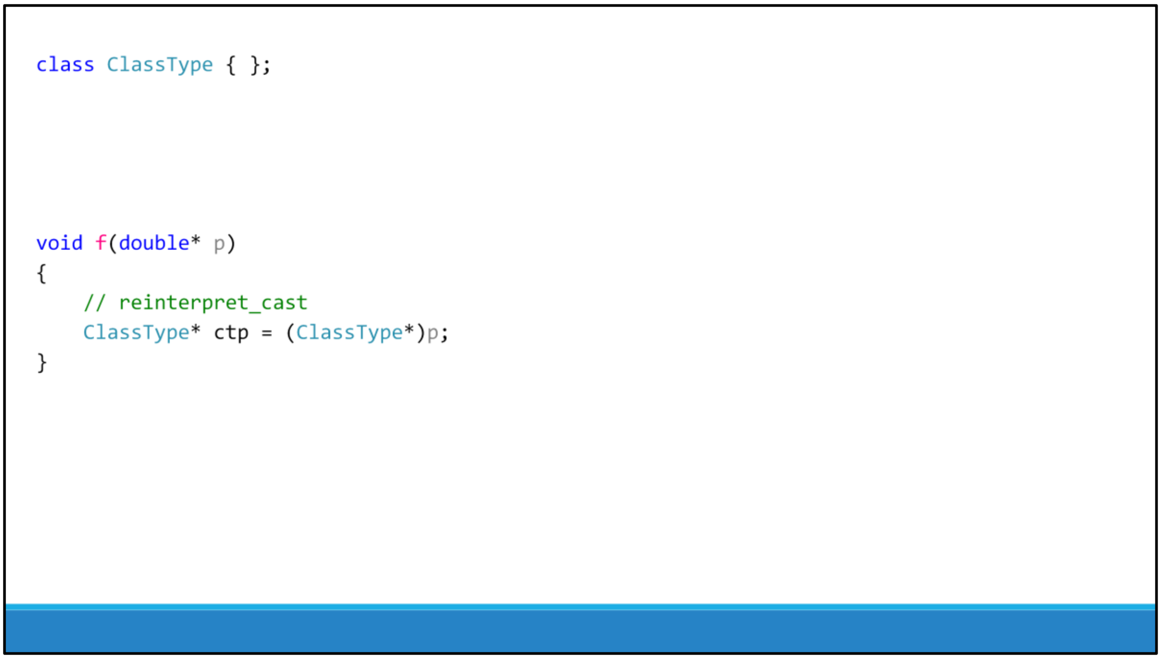
- Possibly combined with a const_cast, if required

If p is of a type unrelated to ClassType, it does a reinterpret_cast

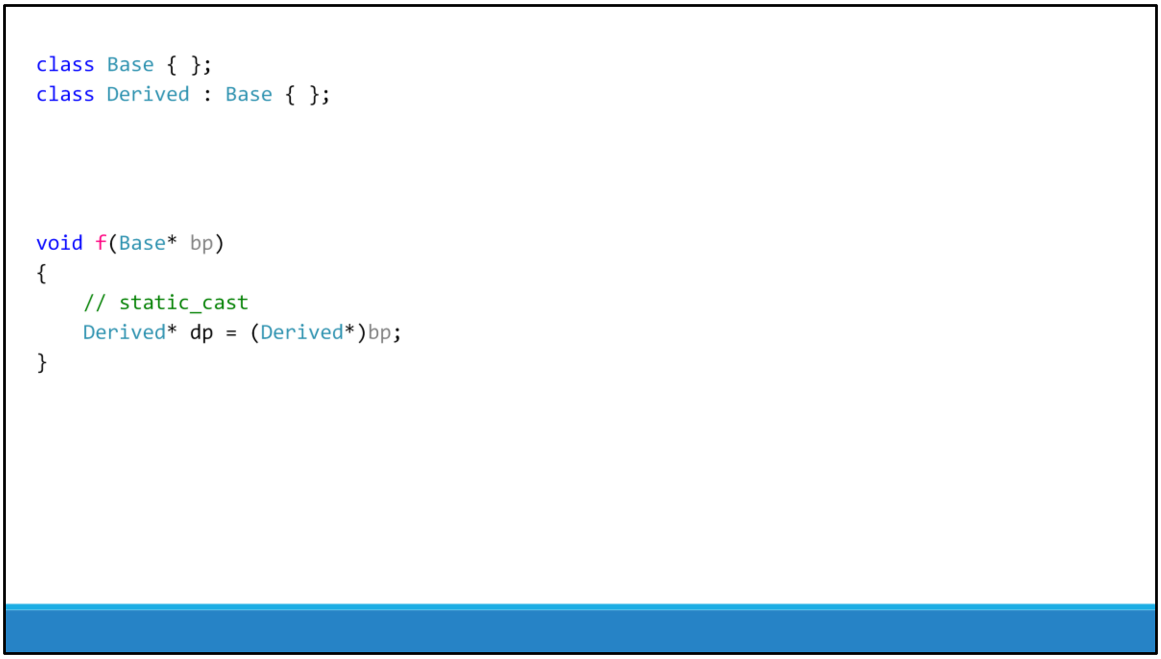
- Possibly combined with a const_cast, if required



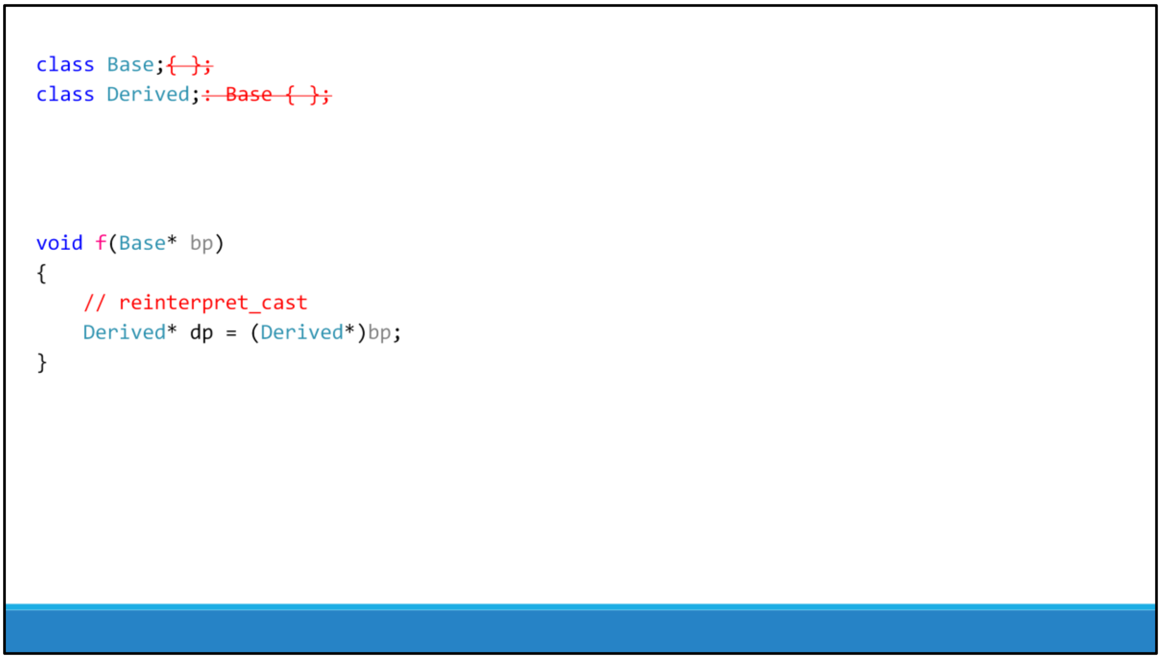
```
class ClassType { };




void f(ClassType const* p)
{
    // const_cast
    ClassType* ctp = (ClassType*)p;
}
```



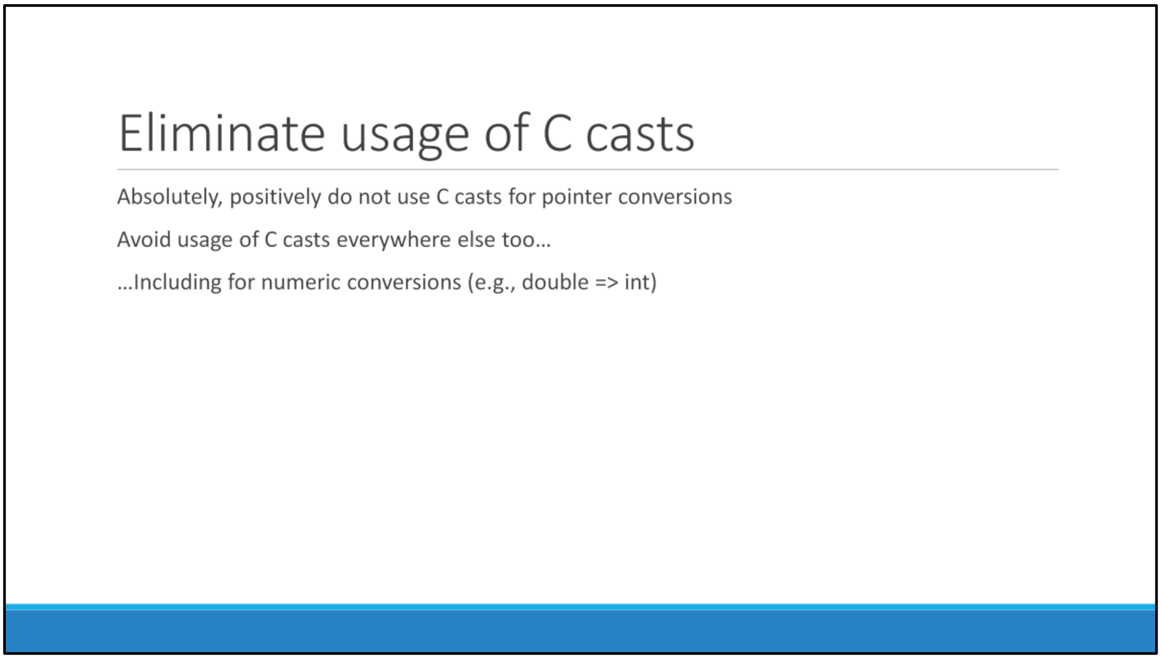
```
class ClassType { };




void f(double* p)
{
    // reinterpret_cast
    ClassType* ctp = (ClassType*)p;
}
```



```
class Base { };
class Derived : Base { };



void f(Base* bp)
{
    // static_cast
    Derived* dp = (Derived*)bp;
}
```



```
class Base;~~{ };~~
class Derived;~~: Base { };~~



void f(Base* bp)
{
    // reinterpret_cast
    Derived* dp = (Derived*)bp;
}
```



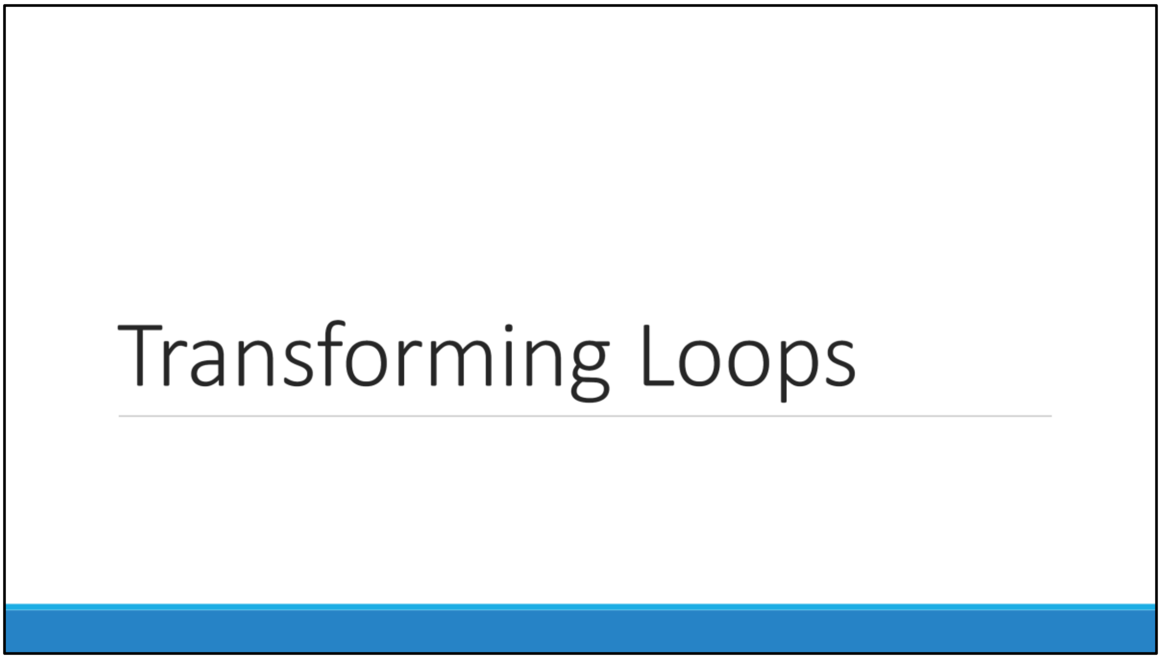
Eliminate usage of C casts

Absolutely, positively do not use C casts for pointer conversions

Avoid usage of C casts everywhere else too…

…Including for numeric conversions (e.g., double => int)

In addition to matters of correctness, just consider the searching issues trying to find casts if you use C casts. Just don’t. Tell people who are reading your code what you think you’re doing, and your code will be better for it.

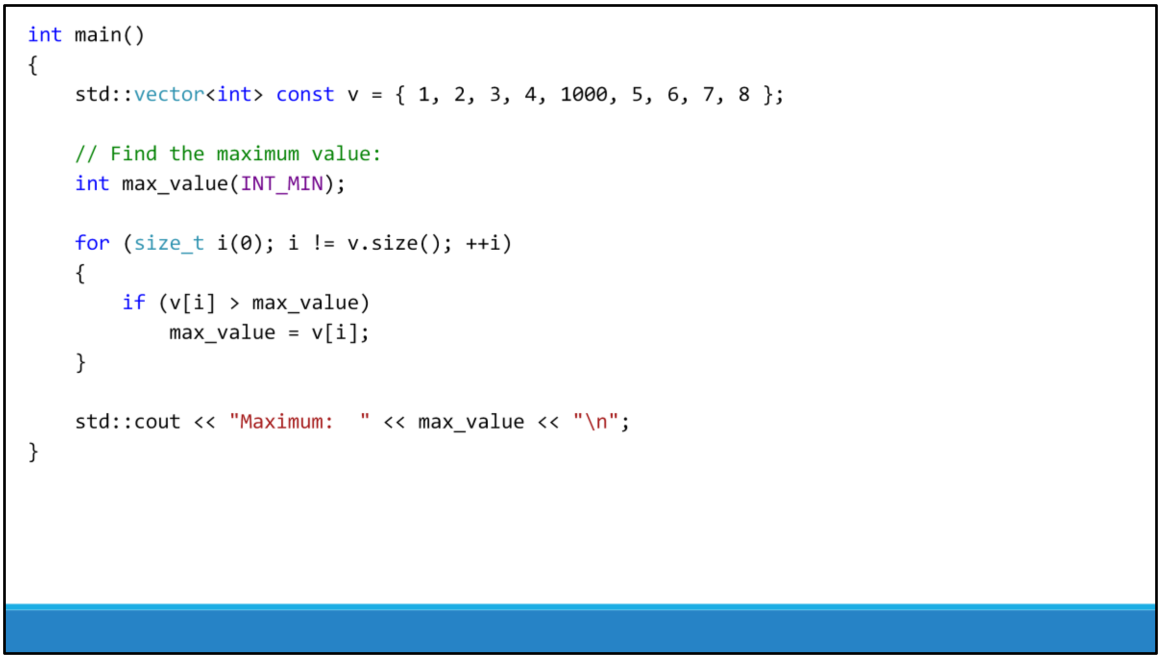


Transforming Loops

Switch to Kate

The action items are getting harder as we go through the recommendations. There is no doubt that this may be the hardest work to do, but then again it may have the biggest benefit, whether that is clearing away hidden bugs, improving performance, or making it possible to add a capability because you can now understand and perhaps change a section of your code.

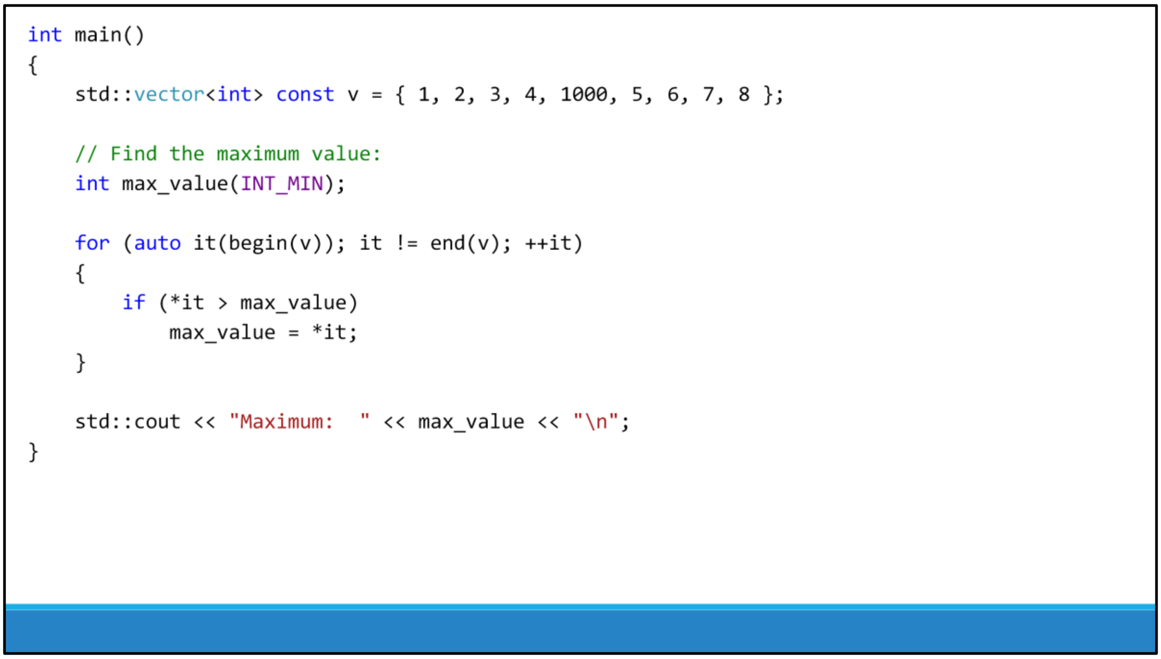
Some code you have the freedom to make bigger changes to. Making these changes isn’t right for every project, but will make the code easier to read, understand, check, debug, and reuse. Let me show you.



```
int main()
{
    std::vector<int> const v = { 1, 2, 3, 4, 1000, 5, 6, 7, 8 };

    // Find the maximum value:
    int max_value(INT_MIN);

    for (size_t i(0); i != v.size(); ++i)
    {
        if (v[i] > max_value)
            max_value = v[i];
    }

    std::cout << "Maximum:  " << max_value << "\n";
}
```

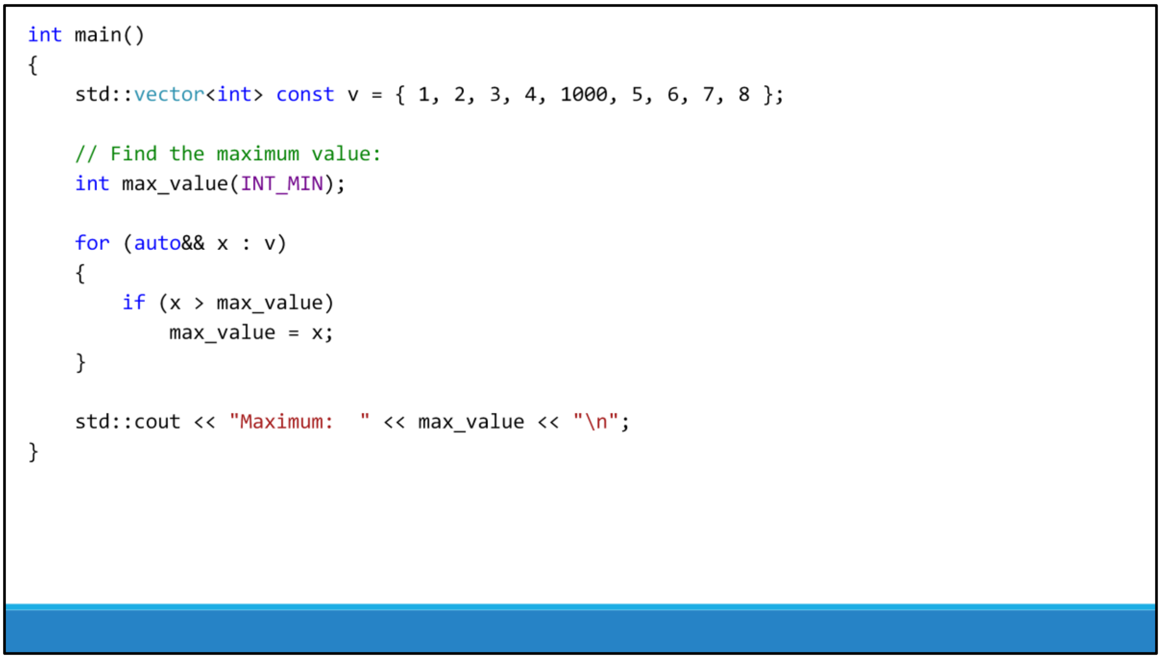
Two hard problems: cache invalidation, naming things, and off by one errors

Fencepost errors



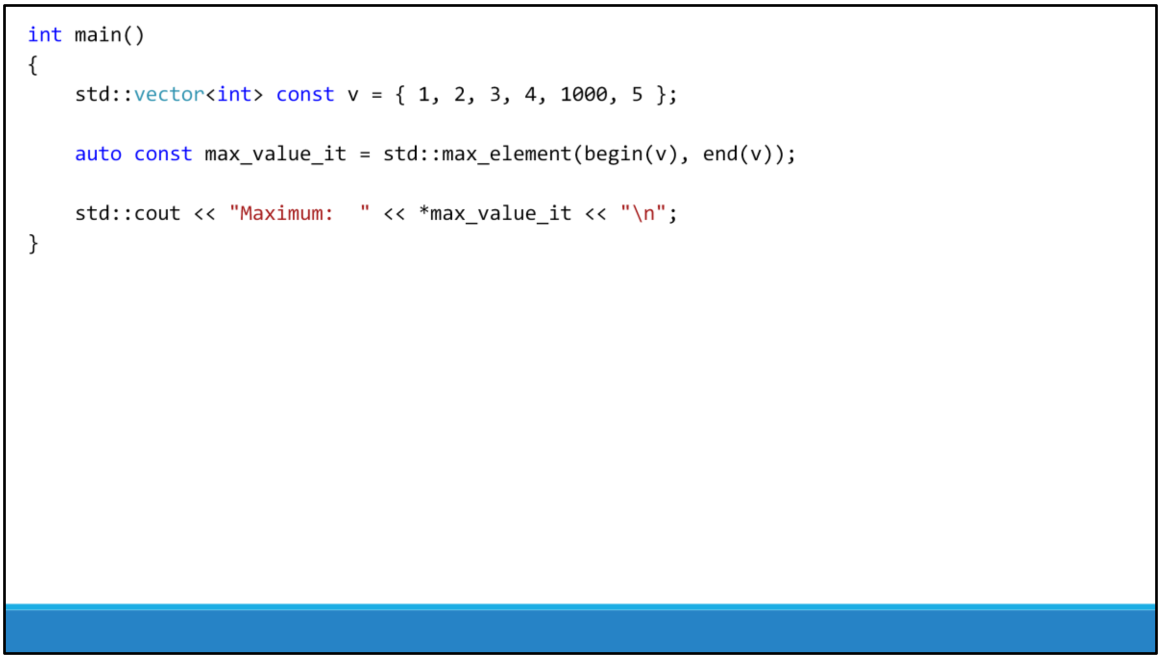
```
int main()
{
    std::vector<int> const v = { 1, 2, 3, 4, 1000, 5, 6, 7, 8 };

    // Find the maximum value:
    int max_value(INT_MIN);

    for (auto it(begin(v)); it != end(v); ++it)
    {
        if (*it > max_value)
            max_value = *it;
    }

    std::cout << "Maximum:  " << max_value << "\n";
}
```

Iterator code is better than indexing. For one thing people do weird stuff with indexes that they don’t do with iterators.



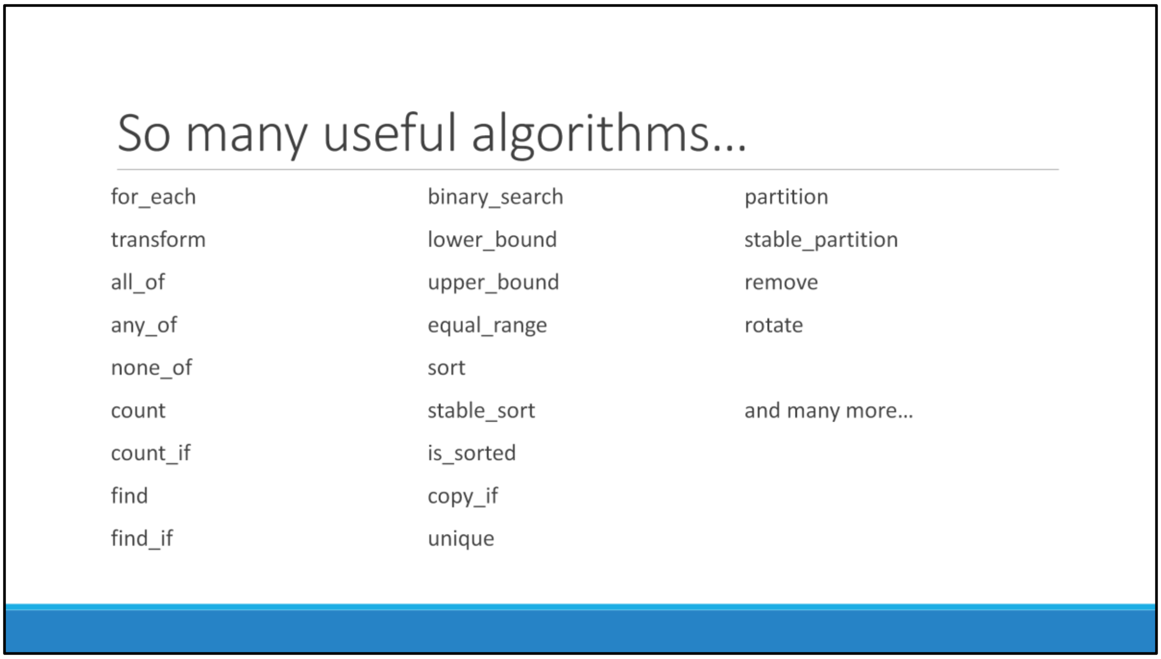
```
int main()
{
    std::vector<int> const v = { 1, 2, 3, 4, 1000, 5, 6, 7, 8 };

    // Find the maximum value:
    int max_value(INT_MIN);

    for (auto&& x : v)
    {
        if (x > max_value)
            max_value = x;
    }

    std::cout << "Maximum:  " << max_value << "\n";
}
```

Now it’s safer and there’s so much less to get wrong



```cpp
int main()
{
    std::vector<int> const v = { 1, 2, 3, 4, 1000, 5 };

    auto const max_value_it = std::max_element(begin(v), end(v));

    std::cout << "Maximum:  " << *max_value_it << "\n";
}
```

<algorithm> is glorious. Finding the largest value in a collection is not exactly ground breaking research. Nor is finding the smallest value, sorting, counting, and so on.



So many useful algorithms…

for_each
transform
all_of
any_of
none_of
count
count_if
find
find_if

binary_search
lower_bound
upper_bound
equal_range
sort
stable_sort
is_sorted
copy_if
unique

partition
stable_partition
remove
rotate

and many more…

You don’t need a comment because “sort” and “find” are [not really subtle] clues about what is going on.

The key to me around using these is the addition of lambdas in C++11. Using a lambda for the predicate that the _if functions take (and some sort overloads etc) makes it all readable and usable. If you memorized that you hate these things back in the days of function pointers and then never thought about them again, it’s time to take another look.

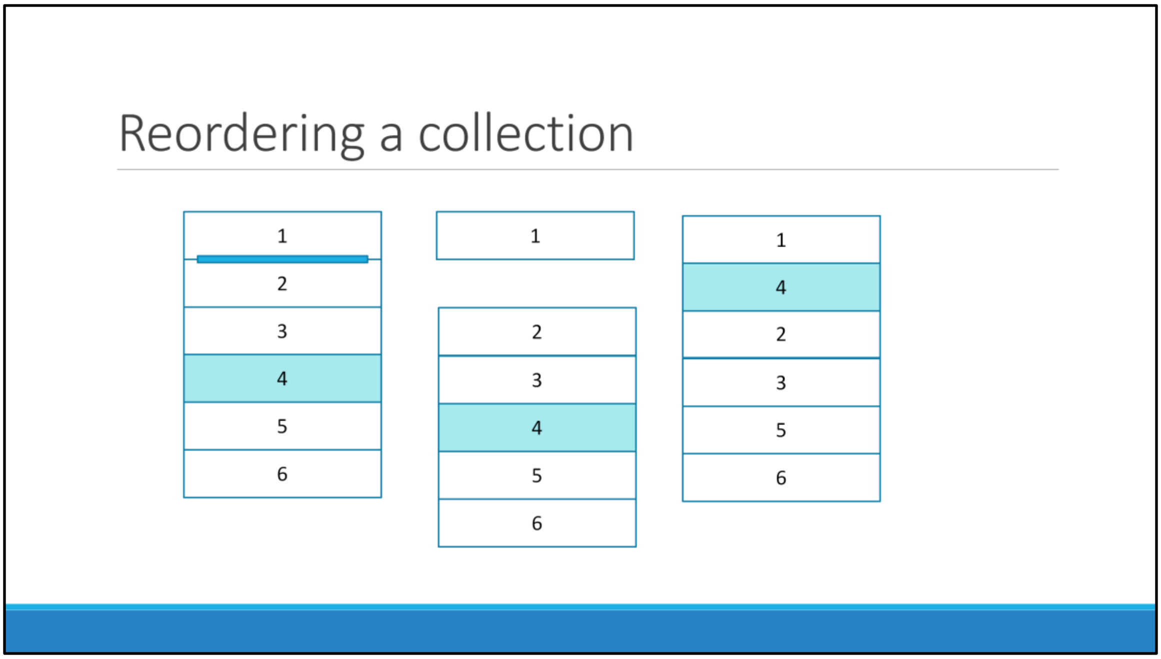
C++ Seasoning

http://channel9.msdn.com/Events/GoingNative/2013/Cpp-Seasoning

Opened my (and many people's) eyes to partition and rotate

A lot of problems are solved problems.

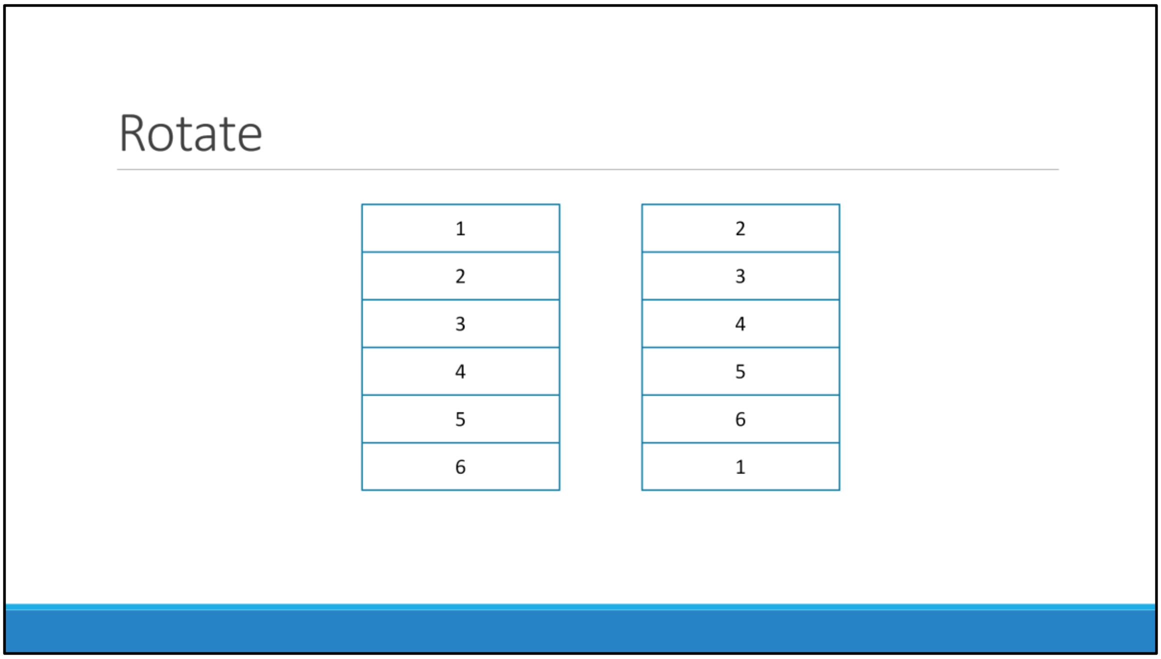
I built this canoe in 1985. (It is a few months older than James.) Everyone admires it, it’s beautiful. But a C++ developer once asked me: “did you rip the cedar into strips yourself?” No, I didn’t. And nor did I grow the tree. You can write a beautiful app with someone else’s library functionality.



Reordering a collection

Imagine you’re writing a ToDo application. You’ve got some sort of user interface elements that represent things the user wants to do, and they’re dragging and dropping them for the priority. So there are tasks 1 through whatever, and someone grabs task 4 and pulls it up between 1 and 2. You’ve got some sort of vector, and you need to rearrange the elements of the vector in response to this user interaction you’ve received about the dragging and the dropping. If you write this yourself, you might do something like “insert a copy of it after the place it was dragged, shoving everything else down. Then delete it from where it used to be, pulling everything else back up.” If it was dragged upwards, “where it used to be” has been pushed down one, and if it was dragged downwards, “where it was dropped” has been pulled up one, so most people end up writing this in two halves, one for an upward pull and one for a downward pull. It’s kind of persnickety work.

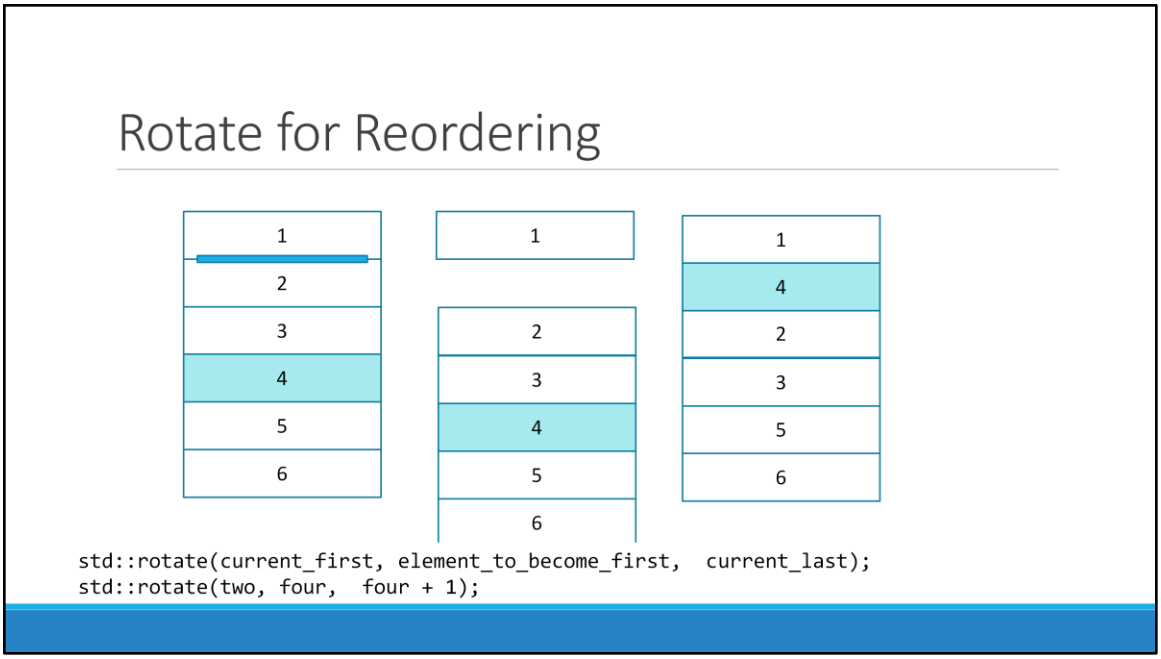
I’m not going to show you the hundreds of lines of code to do that by hand with loops of my own. Instead, I’m going to skip to the punchline and show you rotate and stable partition.



Rotate

1	2
2	3
3	4
4	5
5	6
6	1

This is actually a job for std::rotate. You probably don’t think so, because pretty much every example of rotate on the planet looks like this. Move all the elements up one, or down one. It’s hard to imagine much of a use for that. But in fact, rotate takes an arbitrary range within a collection and moves it up to some other point within the collection. If you want to move down, just change your opinion of what you’re moving.



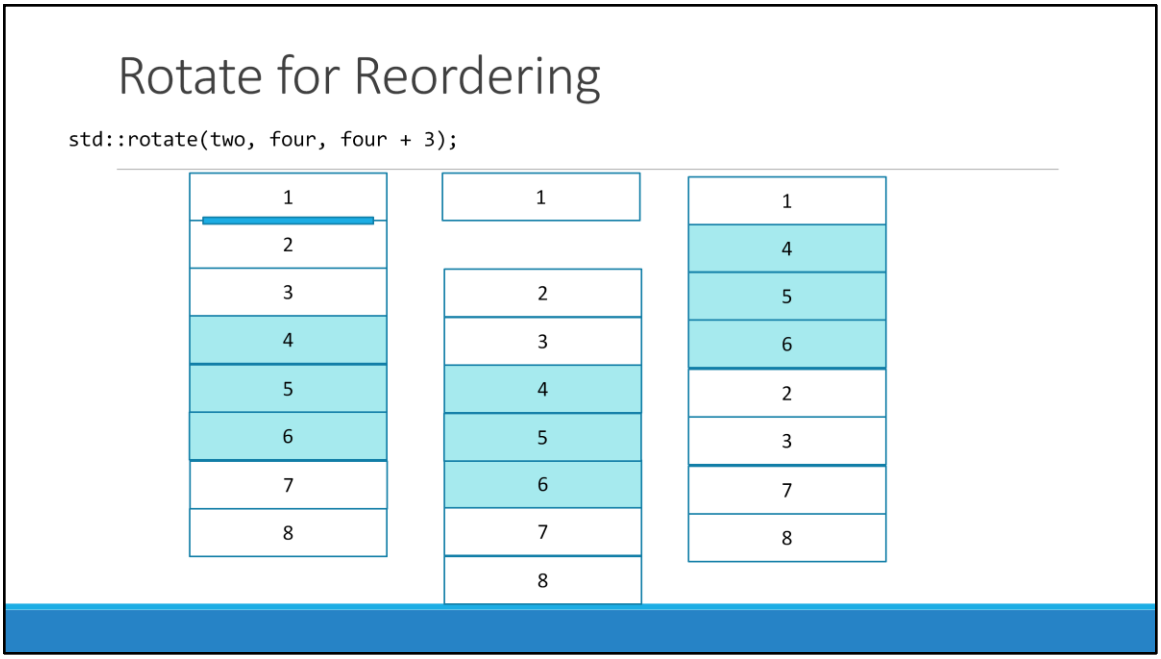
Rotate for Reordering

```
std::rotate(current_first, element_to_become_first,  current_last);
std::rotate(two, four,  four + 1);
```

So back to the Todo list, here’s how ONE LINE OF CODE takes care of that drag-and-drop action by the user:

I’m not going to show you the hundreds of lines of code to do that by hand with loops of my own. Instead, I’m going to skip to the punchline and show you rotate and stable partition.

Actually the parameter names are “Start of section of collection you are rotating (we’re ignoring element one and starting the rotate at 2)”, “element that will be the new first element in the section” “just past end of section of collection you’re rotating (we’re ignoring elements 5 and 6 and ending the rotate after 4)”.

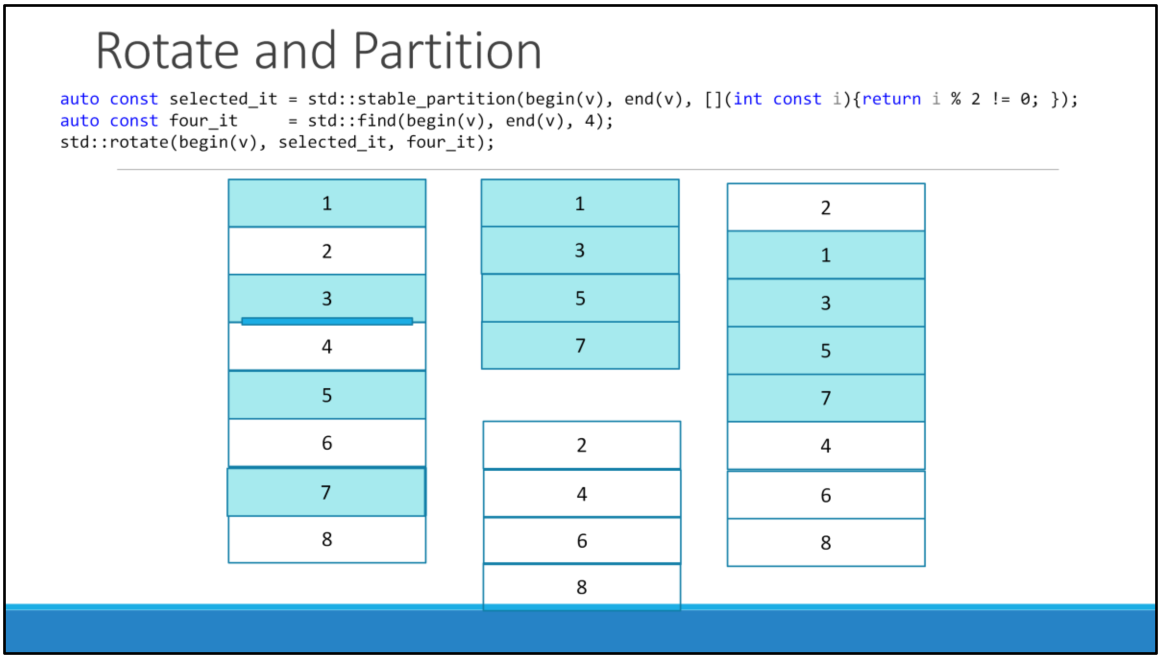


Rotate for Reordering

```
std::rotate(two, four, four + 3);
```

Now, imagine that the user is allowed to shift-click, or whatever it is on your favourite OS, and grab a range of tasks. Maybe tasks 4, 5, and 6 are all errands that will be run together and the user wants to drag them down after task 6, or up before task 2. You can see how this makes your function a lot more complicated. In fact, I learned about rotate in a talk by Sean Parent and he had an example of real code from a Google code review that was doing this for windows a user could drag around, and it was pages long.

Switch back to code and show the three-item rotate

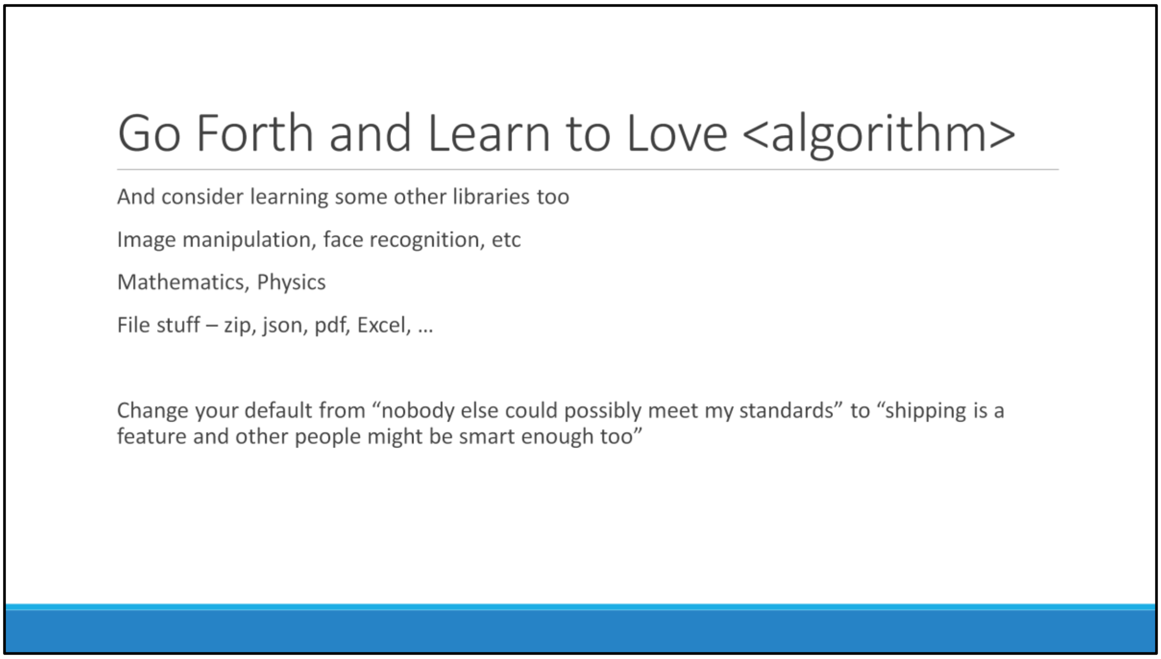


Rotate and Partition

```
auto const selected_it = std::stable_partition(begin(v), end(v), [](int const i){return i % 2 != 0; });
auto const four_it     = std::find(begin(v), end(v), 4);
std::rotate(begin(v), selected_it, four_it);
```

What’s more, whether it’s windows or todo items, you can imagine that a user might want to control-click or command-click and pick up several disjoint elements and drag them all together to one place in the collection, a gathering operation, and if it’s pages of code to move a contiguous range what is it to move a disjoint selection?

To pull certain items out of a collection is to partition the collection, basically into the selected and unselected items. A **stable** partition ensures that the items in each part of the collection stay in the same order they were in before, just like stable sort. In a real UI the condition that separated our partitions would be whether they were selected or not. Here I’m going to use whether they are odd or not. The iterator you get back from stable_partition points to the element after the partition, in this case 2. I’m going to put all the odd elements before element four. I’m rotating the partitioned collection from the beginning, making the element after the selected items (2) the new start of the collection, and the rotation is going from the beginning to just before element 4, so after all the selected ones the original remaining elements will carry on from there. I could have done another find or whatever to find which I wanted to be the new start of the rotated collection.



Go Forth and Learn to Love <algorithm>

And consider learning some other libraries too

Image manipulation, face recognition, etc

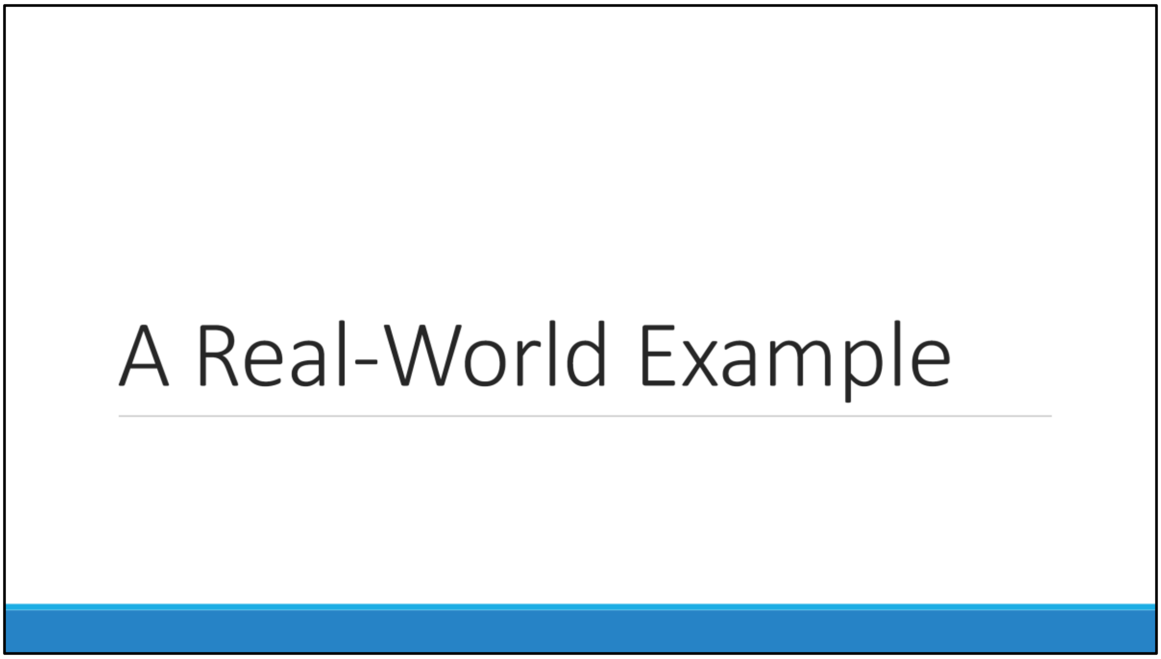
Mathematics, Physics

File stuff – zip, json, pdf, Excel, …

Change your default from “nobody else could possibly meet my standards” to “shipping is a feature and other people might be smart enough too”

In other words, consider buying the strips of wood and then making a beautiful canoe from them.

Hand to James



A Real-World Example

Switch to James

a five- to seven-minute example (maybe "demo" style in the IDE) talking about the rewrite that we did of printf as part of the Visual Studio 14 CRT refactoring. This will be a brief walkthrough of the old sources to show how horrible they were, then a walk through the new implementation pointing out places where we've strategically used modern C++ features (even lambdas and policy based design) to make the code more maintainable. The nice thing about this "case study" is that we can finish by addressing one of the biggest concerns that people have about using these C++ features--performance. The new C++ized sprintf is up to 60x faster

There’s Just One Problem…

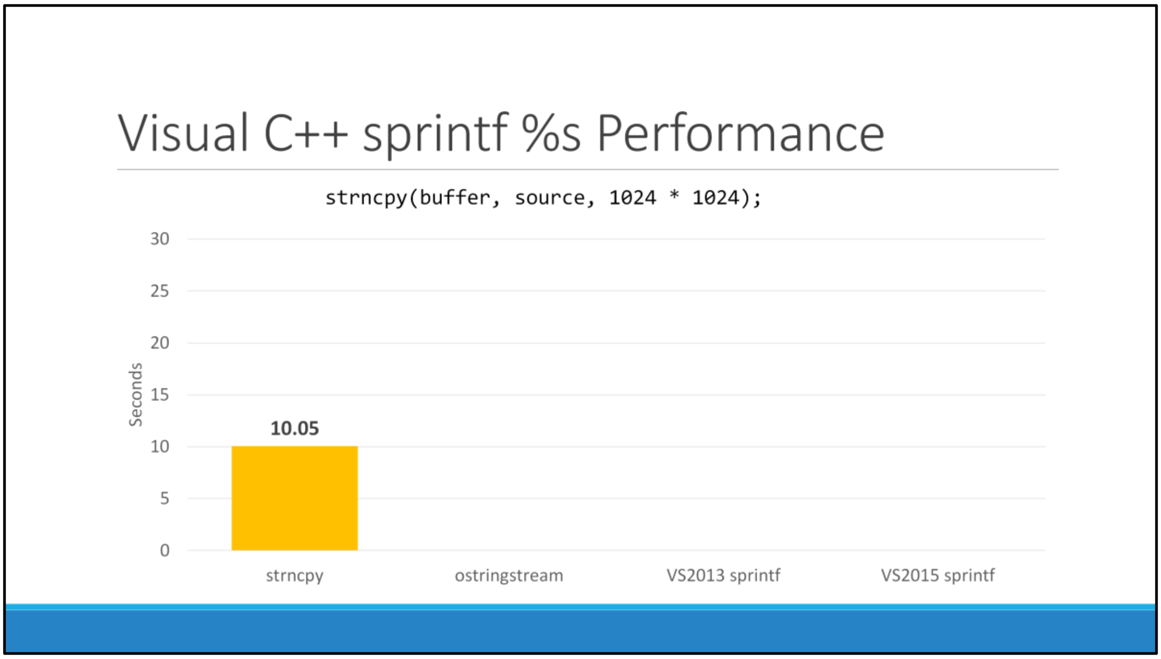
We’ve gone and replaced really fast C code with C++ code.

Everyone knows C++ is slower than C.

…Right?

…Right?

LOL

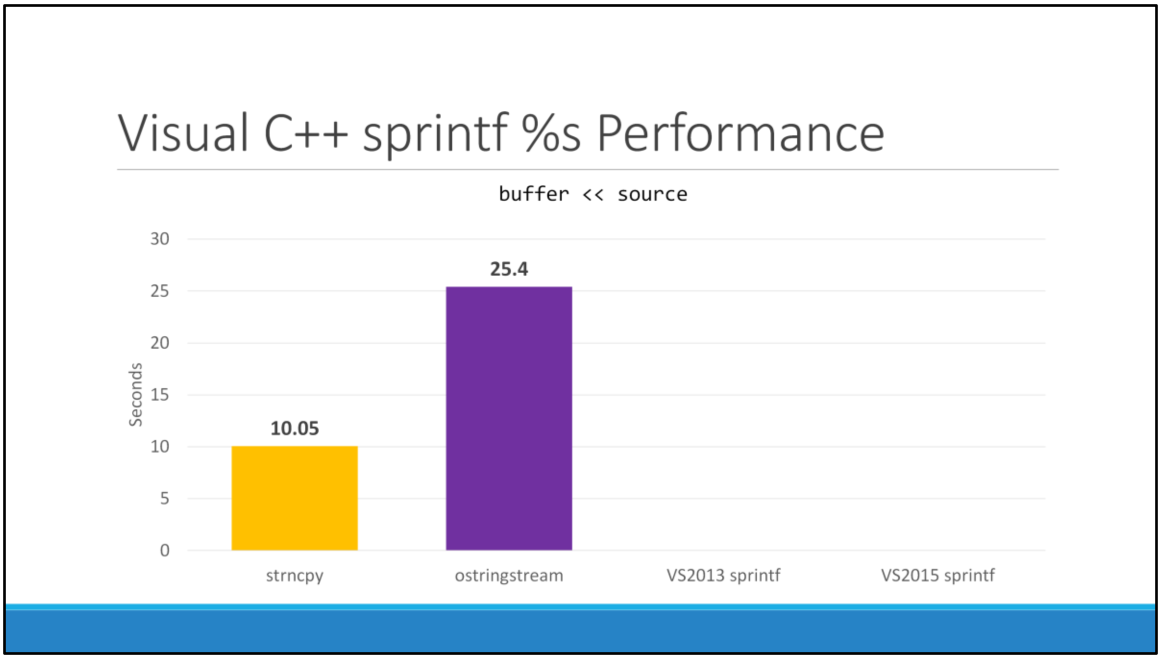


Visual C++ sprintf %s Performance

```
strncpy(buffer, source, 1024 * 1024);
```

```
void run_test(char const* const name)
{
   char* const source = (char*)malloc(1024 * 1024);
   char* const buffer = (char*)malloc(1024 * 1024);

   for (int i = 0; i != 1024 * 1024; ++i)
   {
      source[i] = '0' + (i % 10);
   }

   source[1024 * 1024 - 1] = '\0';

   auto const start_time = get_time();

   for (int i = 0; i != 16000; ++i)
   {
                    sprintf_s(buffer, 1024 * 1024, "%s", source);
   }
```

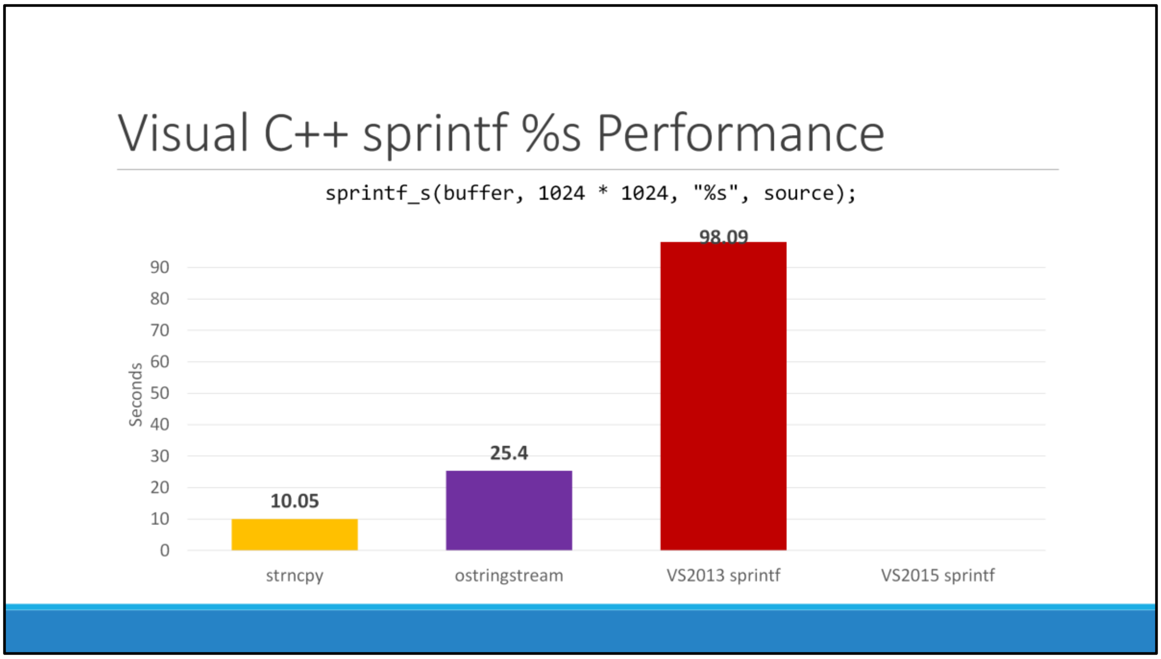
```
    auto const end_time = get_time();

    printf("elapsed time (%s):  %f\n", name, ((double)end_time - start_time) /
get_frequency());
}
```



Visual C++ sprintf %s Performance

```
void run_test(char const* const name)
{
   char* const source = (char*)malloc(1024 * 1024);
   char* const buffer = (char*)malloc(1024 * 1024);

   for (int i = 0; i != 1024 * 1024; ++i)
   {
      source[i] = '0' + (i % 10);
   }

   source[1024 * 1024 - 1] = '\0';

   auto const start_time = get_time();

   for (int i = 0; i != 16000; ++i)
   {
                    sprintf_s(buffer, 1024 * 1024, "%s", source);
   }
```

```
    auto const end_time = get_time();

    printf("elapsed time (%s):  %f\n", name, ((double)end_time - start_time) /
get_frequency());
}
```

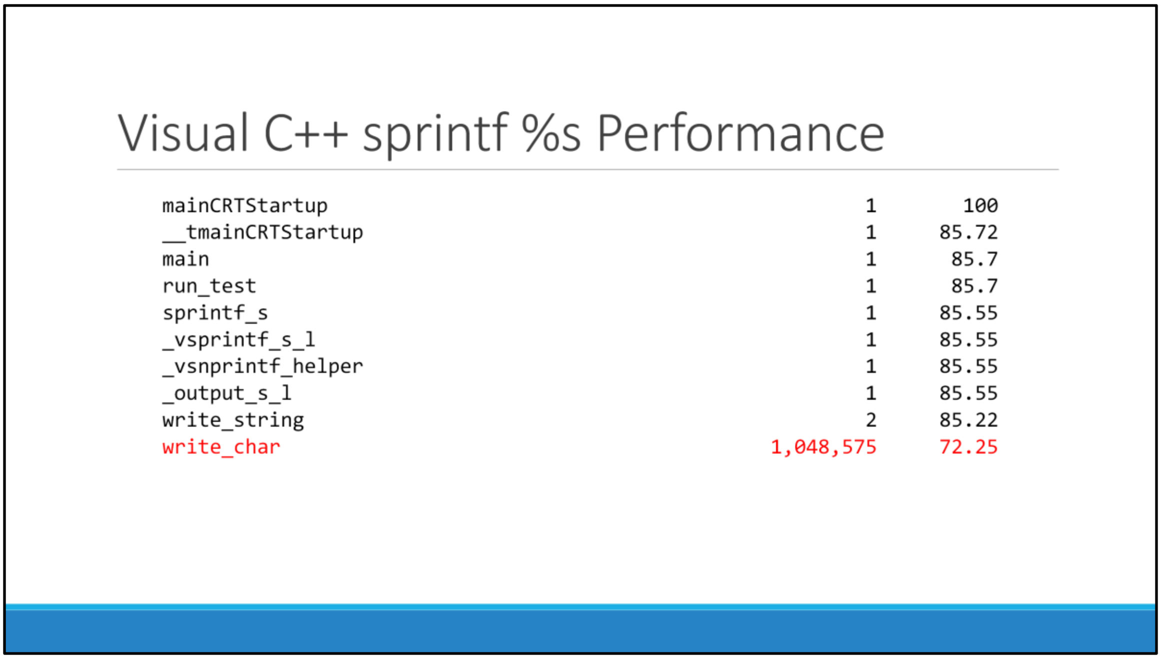


Visual C++ sprintf %s Performance

```
sprintf_s(buffer, 1024 * 1024, "%s", source);
```

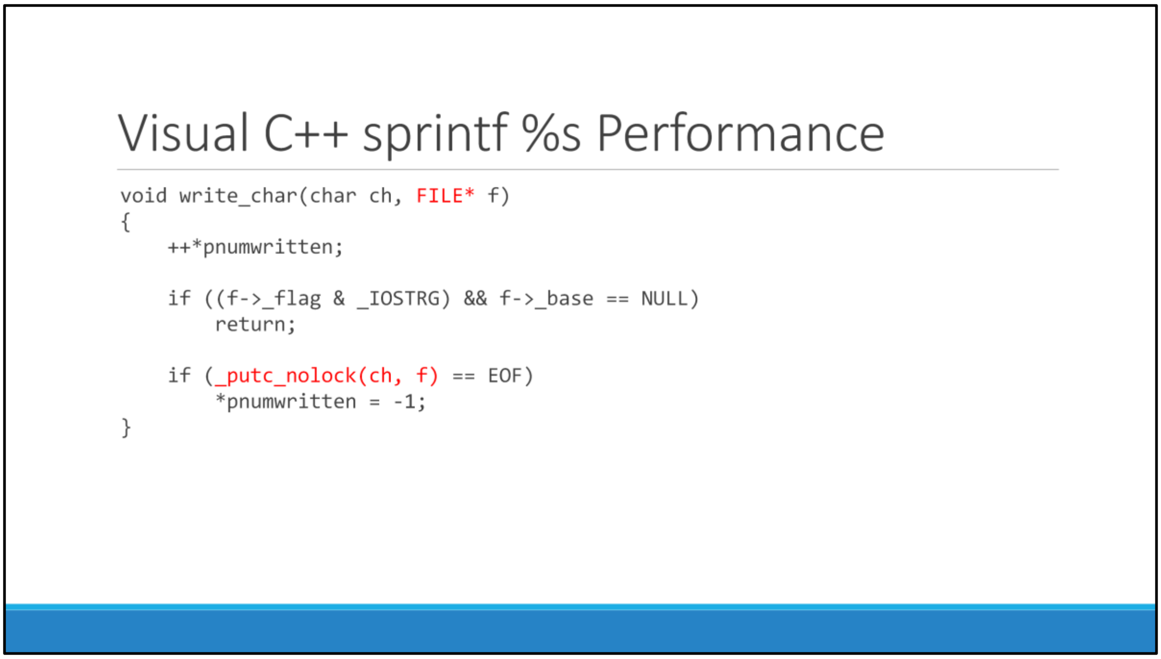
```
void run_test(char const* const name)
{
   char* const source = (char*)malloc(1024 * 1024);
   char* const buffer = (char*)malloc(1024 * 1024);

   for (int i = 0; i != 1024 * 1024; ++i)
   {
      source[i] = '0' + (i % 10);
   }

   source[1024 * 1024 - 1] = '\0';

   auto const start_time = get_time();

   for (int i = 0; i != 16000; ++i)
   {
                    sprintf_s(buffer, 1024 * 1024, "%s", source);
   }
```

```
    auto const end_time = get_time();

    printf("elapsed time (%s):  %f\n", name, ((double)end_time - start_time) /
get_frequency());
}
```



Visual C++ sprintf %s Performance

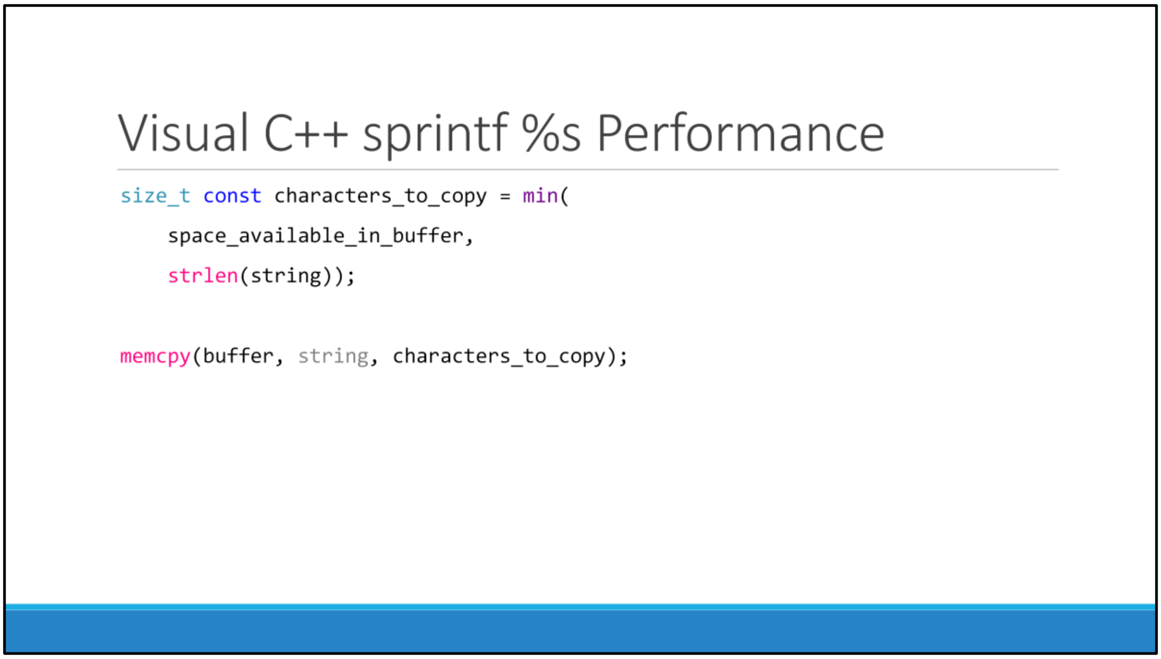
```
mainCRTStartup                                    1       100
__tmainCRTStartup                                 1     85.72
main                                              1      85.7
run_test                                          1      85.7
sprintf_s                                         1     85.55
_vsprintf_s_l                                     1     85.55
_vsnprintf_helper                                 1     85.55
_output_s_l                                       1     85.55
write_string                                      2     85.22
write_char                                1,048,575     72.25
```



Visual C++ sprintf %s Performance

```
void write_char(char ch, FILE* f)
{
    ++*pnumwritten;

    if ((f->_flag & _IOSTRG) && f->_base == NULL)
        return;

    if (_putc_nolock(ch, f) == EOF)
        *pnumwritten = -1;
}
```

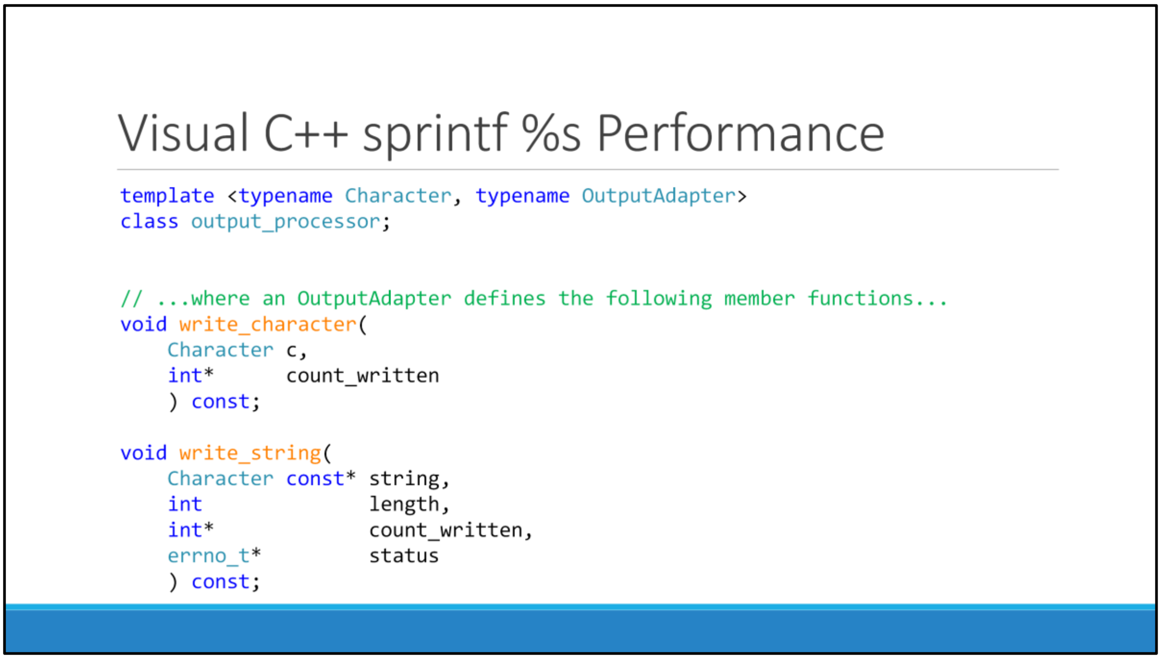
What’s a FILE* doing here
That _putc_nolock expands into an ugly expression
The resulting write_char (after that expansion) was “too big” for the compiler to naturally inline



Visual C++ sprintf %s Performance

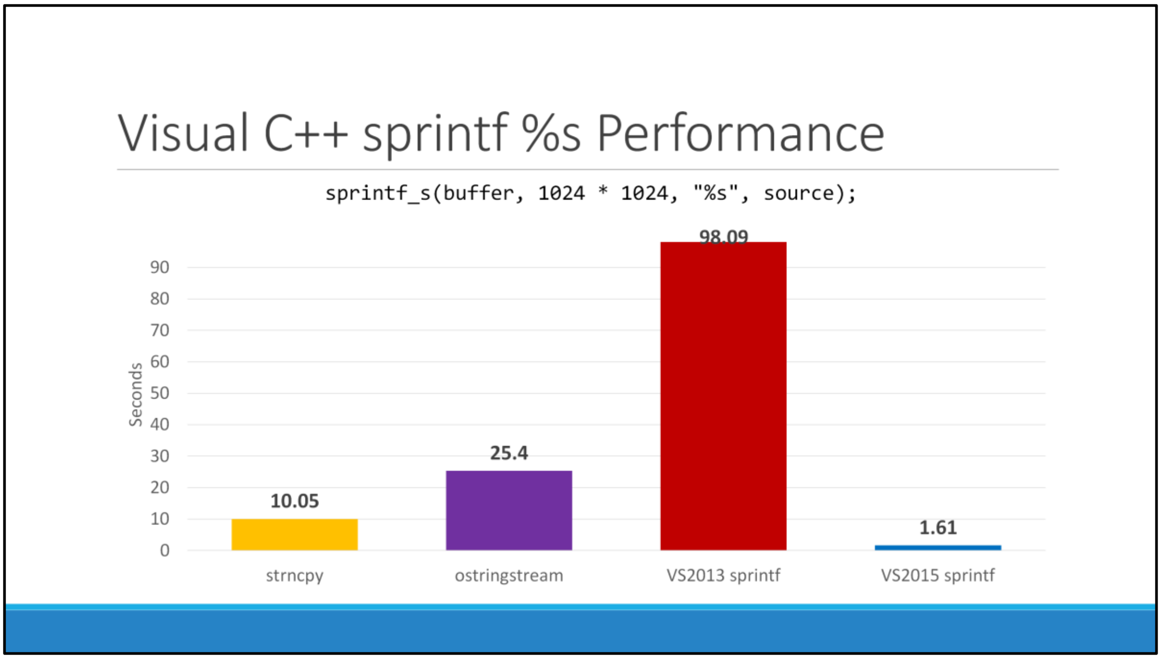
```
size_t const characters_to_copy = min(
    space_available_in_buffer,
    strlen(string));

memcpy(buffer, string, characters_to_copy);
```

Really, what we want is to do this…
Why not just call strncpy?
We could, but we have to handle field widths.



Visual C++ sprintf %s Performance

```
template <typename Character, typename OutputAdapter>
class output_processor;


// ...where an OutputAdapter defines the following member functions...
void write_character(
    Character c,
    int*      count_written
    ) const;

void write_string(
    Character const* string,
    int              length,
    int*             count_written,
    errno_t*         status
    ) const;
```

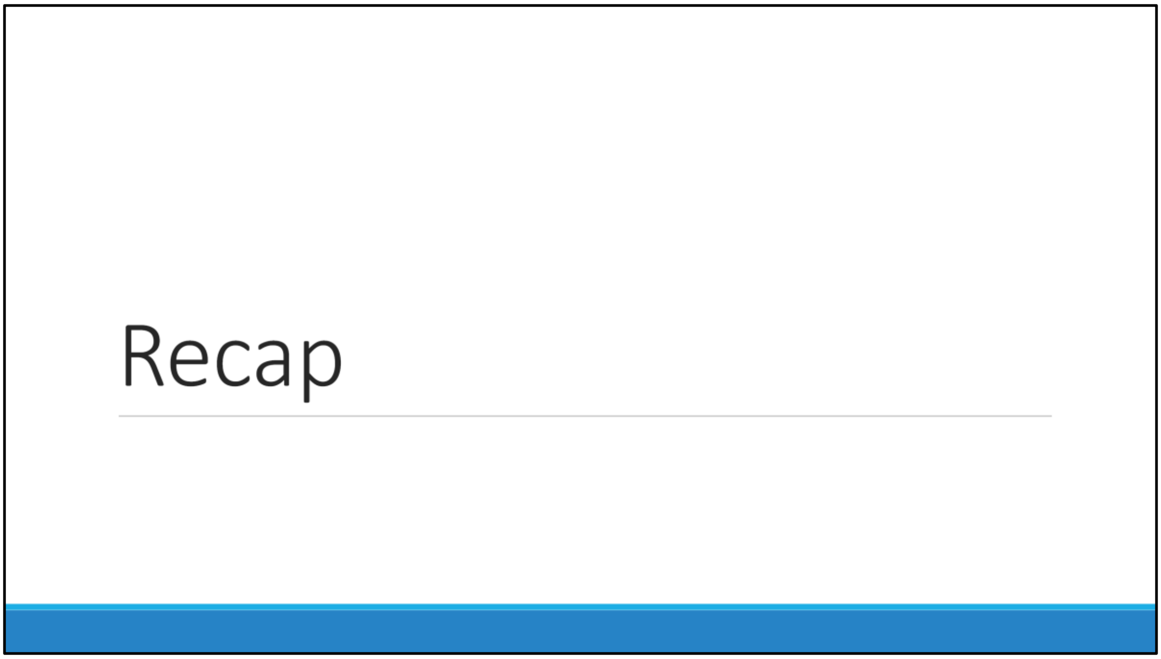


Visual C++ sprintf %s Performance

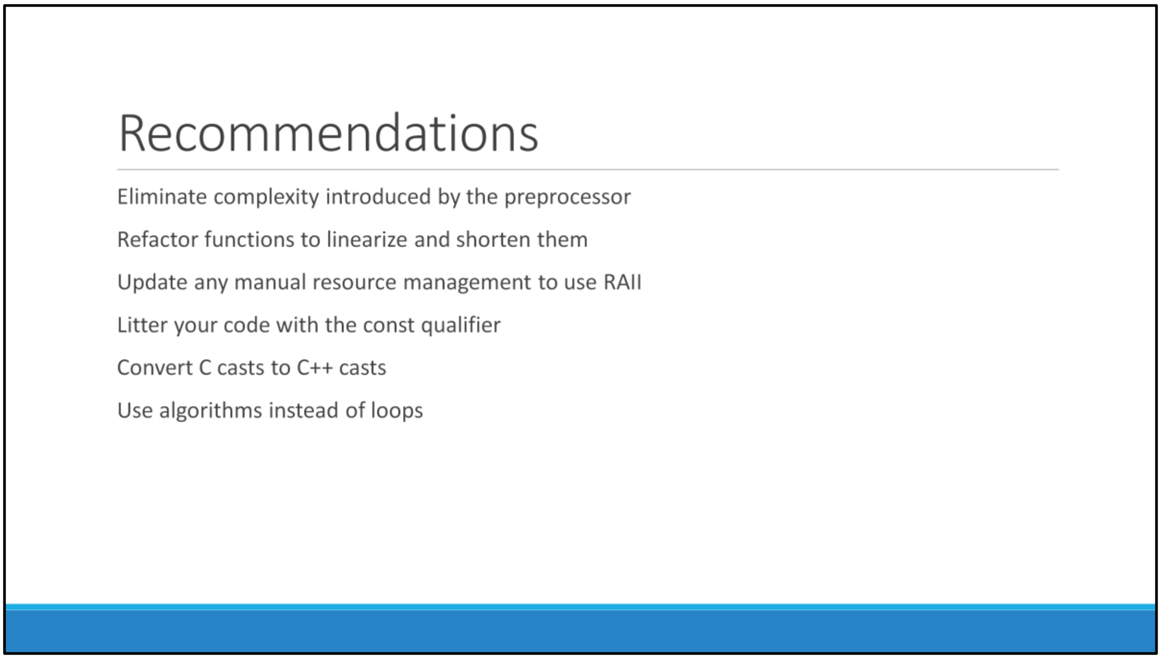
```
sprintf_s(buffer, 1024 * 1024, "%s", source);
```

```
void run_test(char const* const name)
{
   char* const source = (char*)malloc(1024 * 1024);
   char* const buffer = (char*)malloc(1024 * 1024);

   for (int i = 0; i != 1024 * 1024; ++i)
   {
      source[i] = '0' + (i % 10);
   }

   source[1024 * 1024 - 1] = '\0';

   auto const start_time = get_time();

   for (int i = 0; i != 16000; ++i)
   {
                  sprintf_s(buffer, 1024 * 1024, "%s", source);
   }
```

```
    auto const end_time = get_time();

    printf("elapsed time (%s):  %f\n", name, ((double)end_time - start_time) /
get_frequency());
}
```



Recap



Recommendations

Eliminate complexity introduced by the preprocessor

Refactor functions to linearize and shorten them

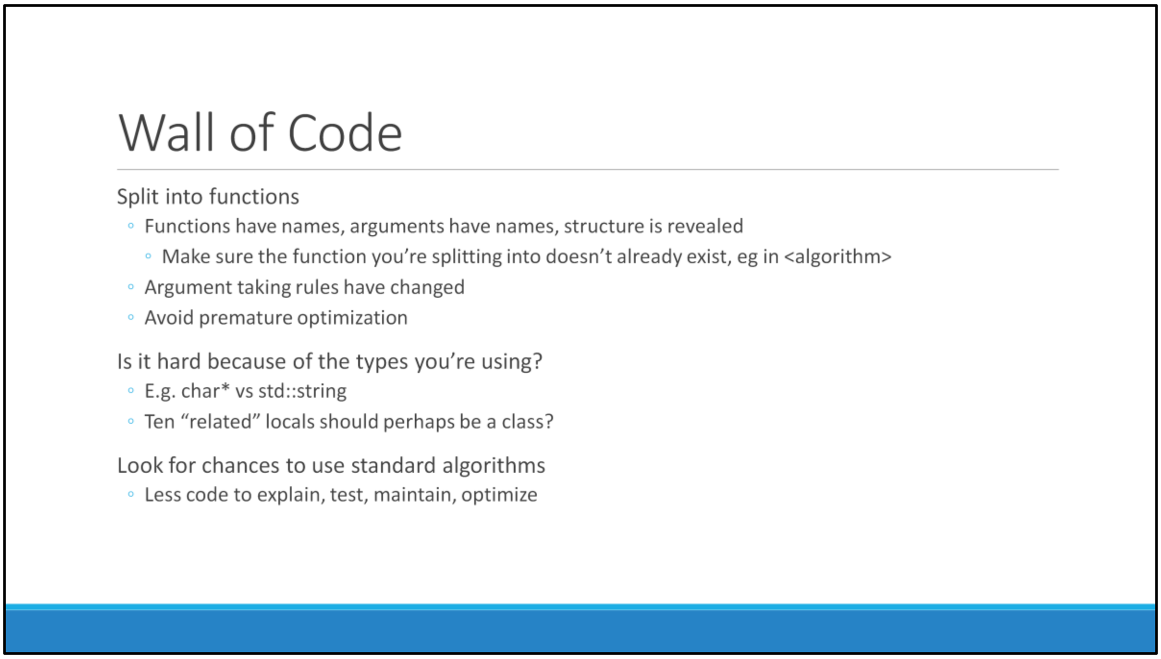
Update any manual resource management to use RAII

Litter your code with the const qualifier

Convert C casts to C++ casts

Use algorithms instead of loops

This is a summary. But all of it applies to one problem we all face when we meet legacy code



Wall of Code

Split into functions

- Functions have names, arguments have names, structure is revealed
  - Make sure the function you're splitting into doesn't already exist, eg in <algorithm>
- Argument taking rules have changed
- Avoid premature optimization

Is it hard because of the types you're using?

- E.g. char* vs std::string
- Ten "related" locals should perhaps be a class?

Look for chances to use standard algorithms

- Less code to explain, test, maintain, optimize

Also mention arrow code

In my experience if you meet a 1000 line function whose structure is opaque to you then you need to figure out what it does before you start to pull it apart.

One reason to write a wall of code is you have a type in your head that is not in your code. You have a bunch of local variables that together constitute some thing. And you have big blocks of code that do stuff to those locals as a way to work on that thing. So it does something to employee name, and then to employee phone number, and then to employee email address, and so on. This code is showing you where encapsulation is dying to emerge.

\0