Adventures in a Legacy Codebase

JAMES MCNELLIS

SENIOR SOFTWARE ENGINEER

MICROSOFT VISUAL C++

James's Vacation Photos

JAMES MCNELLIS
SENIOR SOFTWARE ENGINEER
MICROSOFT VISUAL C++

Adventures in a Legacy Codebase

JAMES MCNELLIS

SENIOR SOFTWARE ENGINEER

MICROSOFT VISUAL C++

The Project: The Universal CRT

The Universal CRT

- Project originally started out as the “CRT Refactoring and Unification Effort”

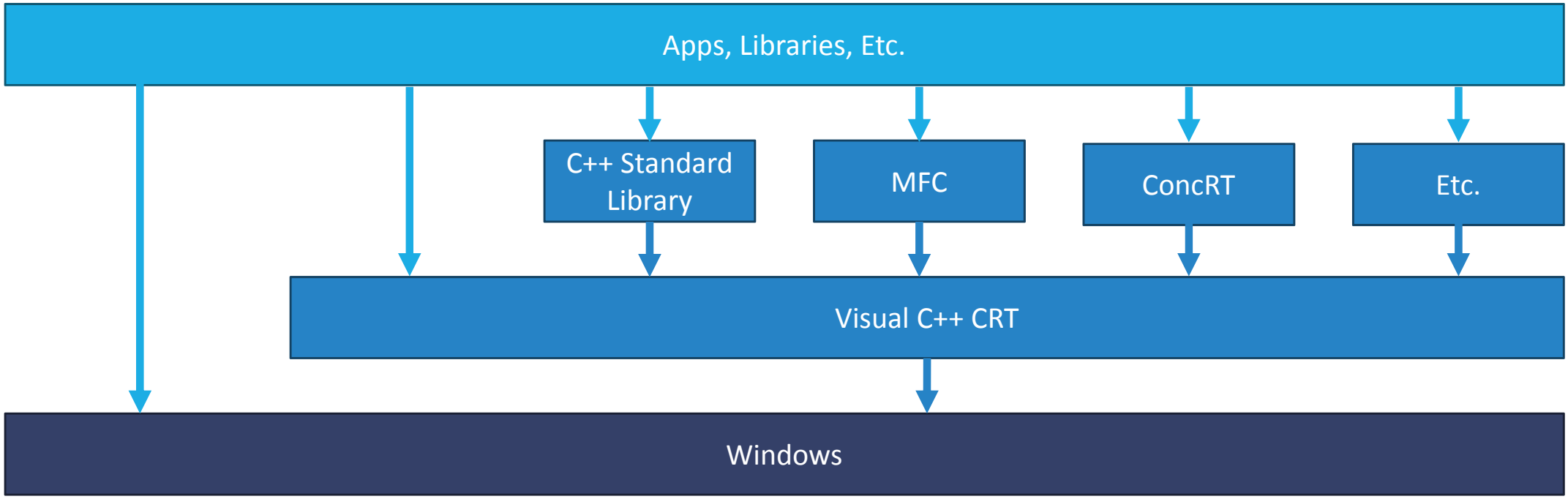
- The CRT is Microsoft’s “C Runtime” (C Run-Time => CRT)
  - The C Standard Library implementation, plus many extensions
  - Various C and C++ runtime support functionality

- The project had two parts, *Refactoring* and *Unification*
  - We’ll look at the *Unification* part first, with the rationale for what we were trying to accomplish…
  - …and then we’ll look at some of what we did in the *Refactoring* to accomplish those goals.

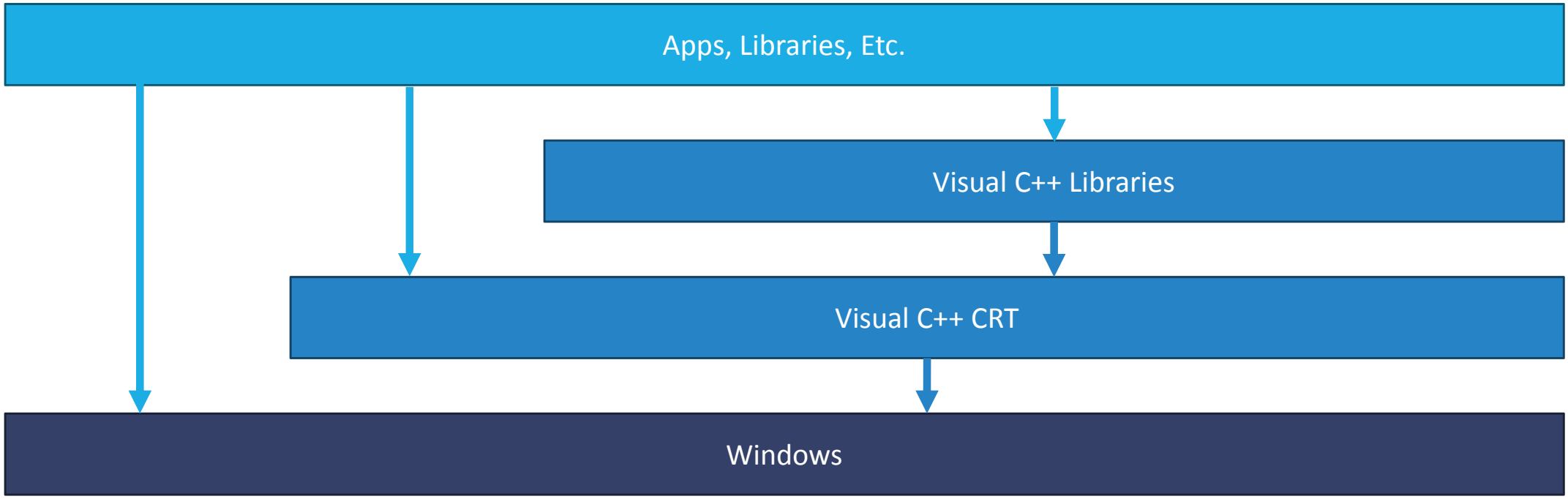
- This is not the story of how we wrote awesome, amazing, terrific C++ code
- This is the story of how we took a legacy codebase and made gradual improvements

Part I
Unification

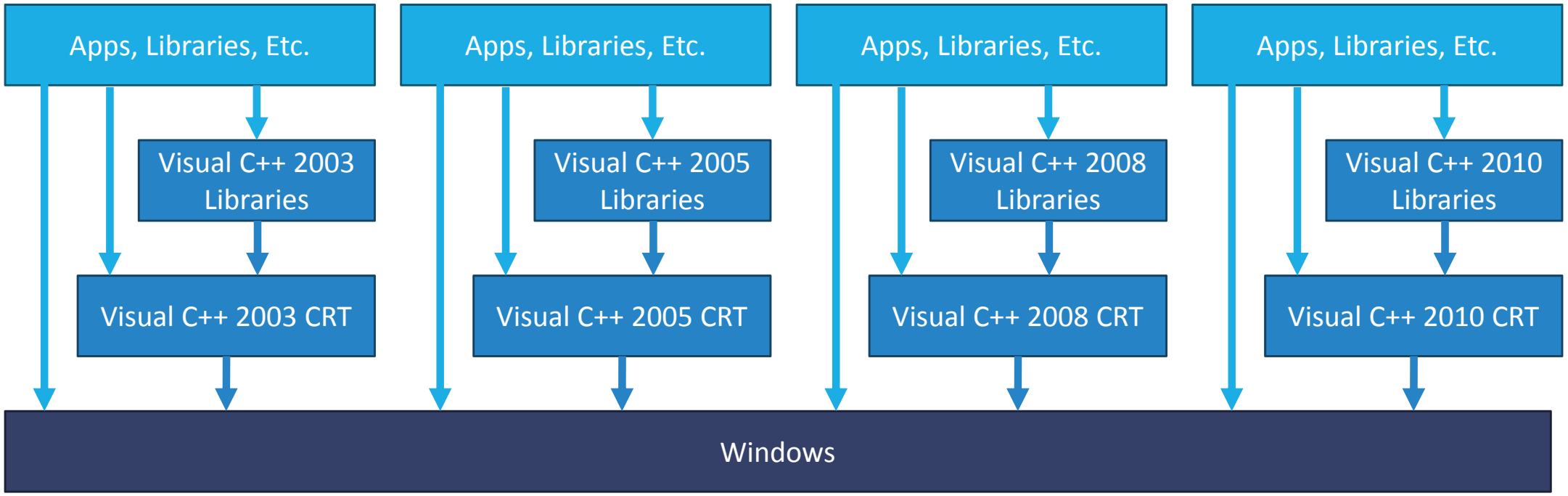
How Things *Should* Look



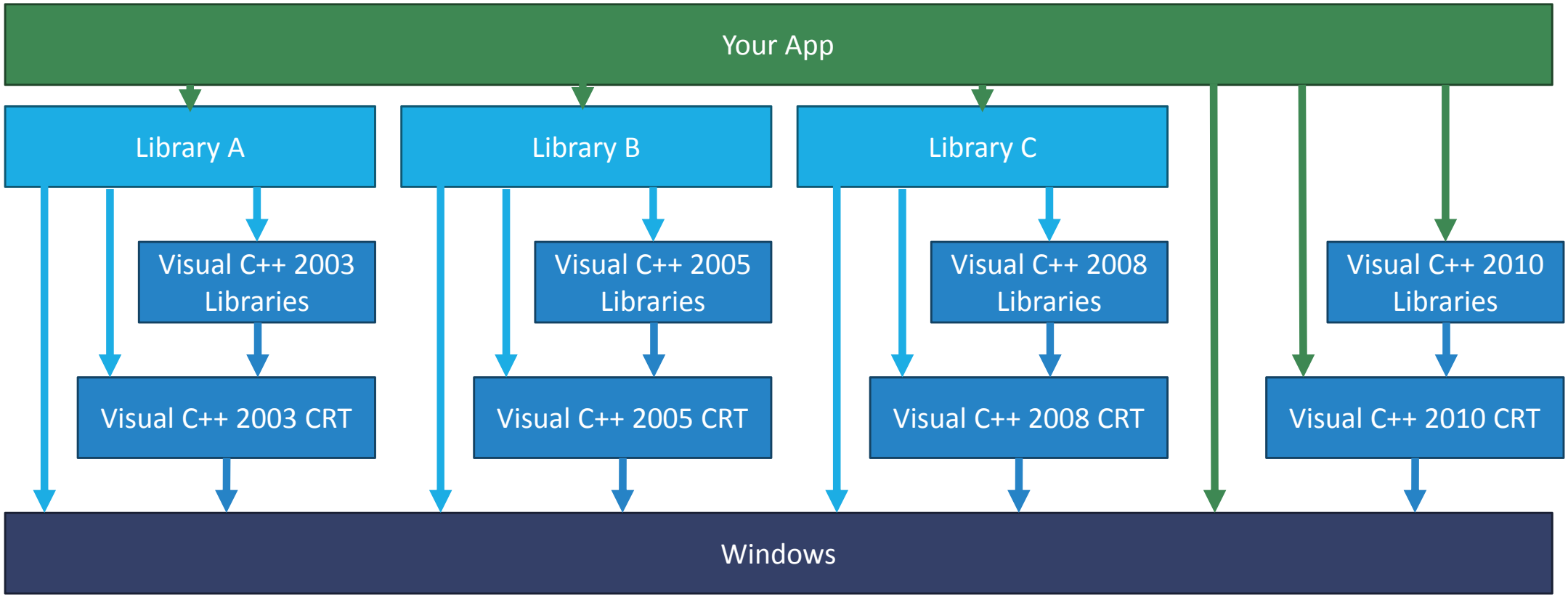
How Things *Should* Look



How Things *Actually* Look

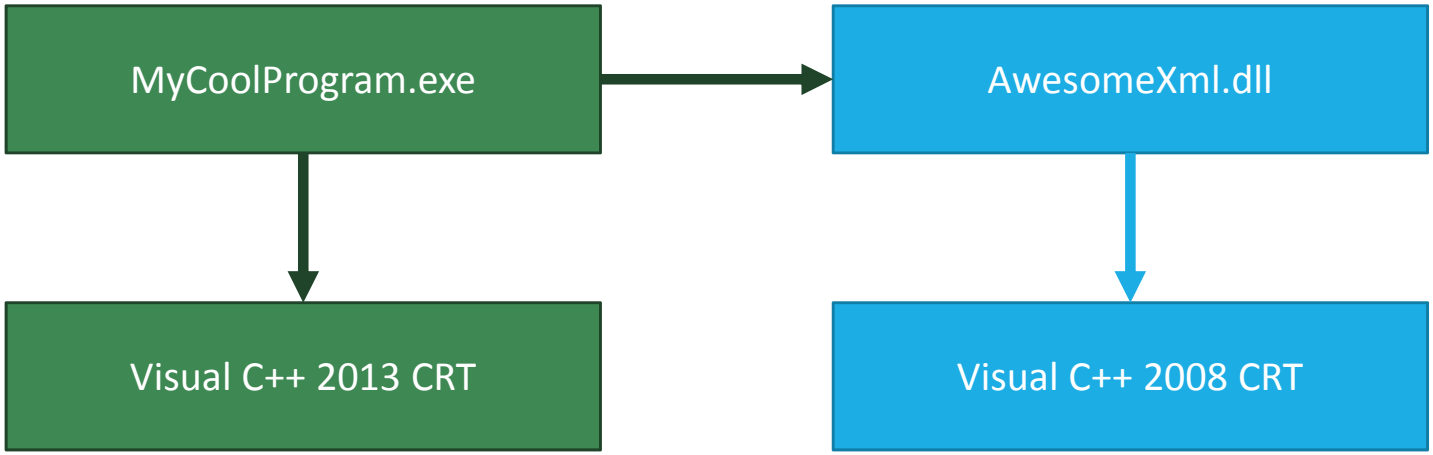


How Things *Actually* Look



How Things *Actually* Look

```
xml_document read_xml_from_file(FILE* fp);
```

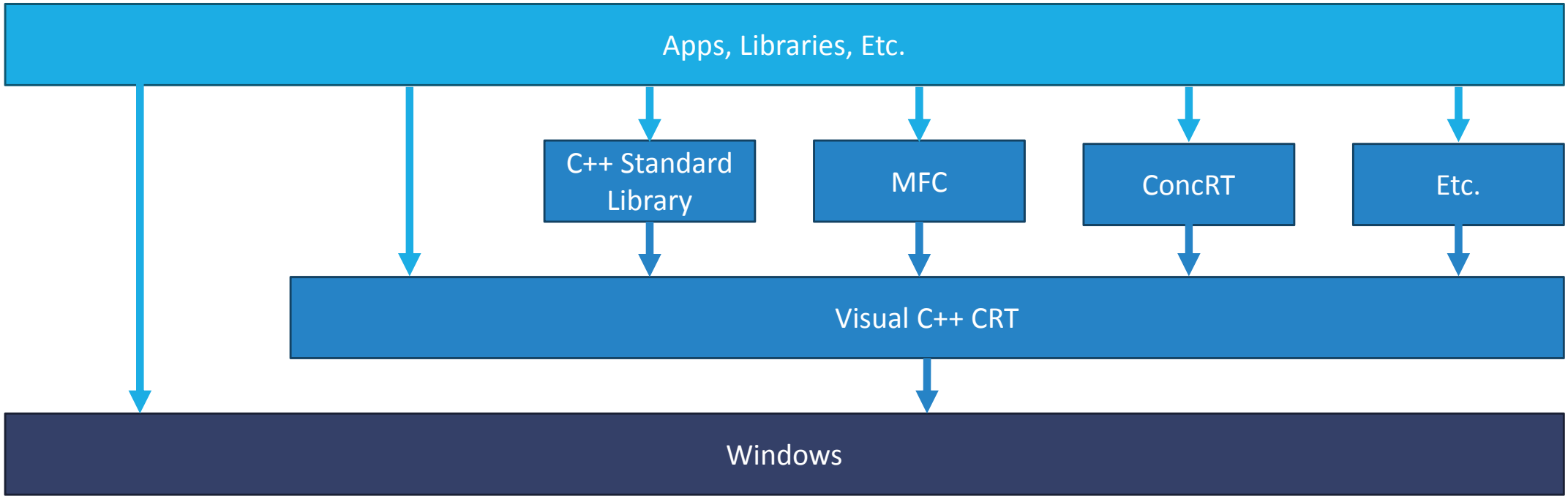


How Things *Actually* Look

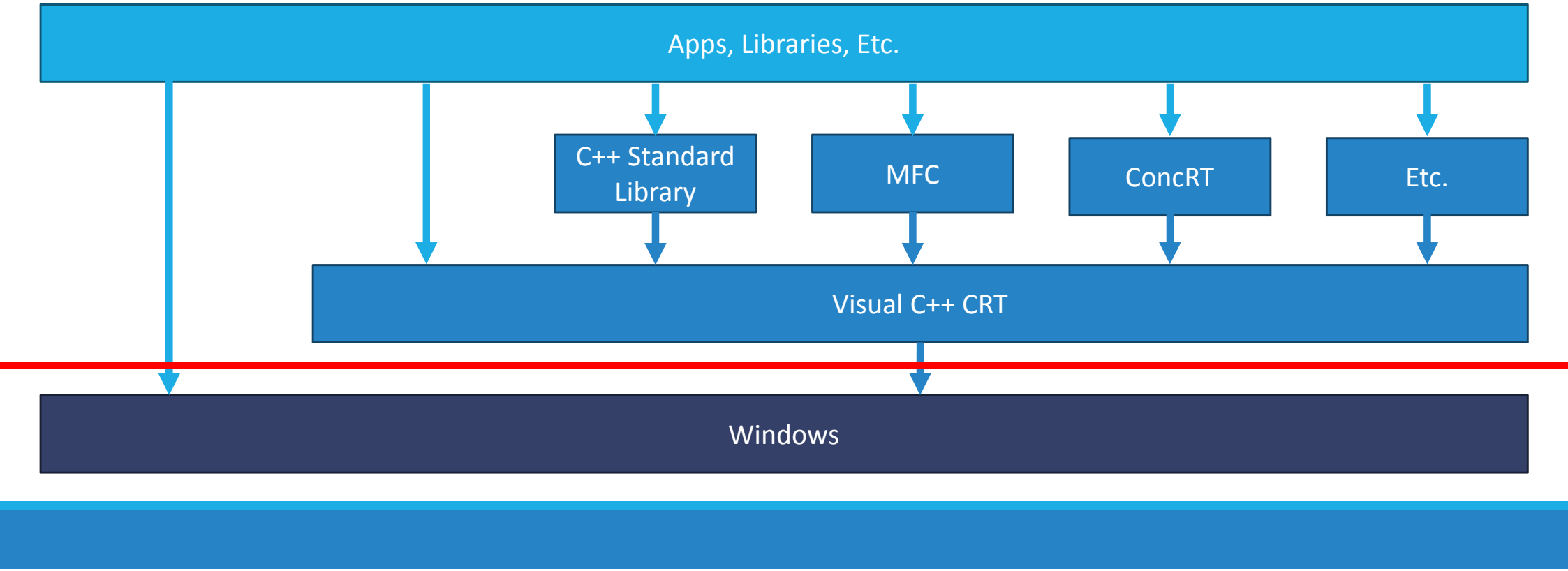
- As an app developer…
  - If you have source for AwesomeXml.dll, you can rebuild it
  - If you don't have source for AwesomeXml.dll,
    - You can ask the author of AwesomeXml.dll to build a new version using Visual C++ 2013
    - You can try to write an abstraction layer over AwesomeXml.dll to bridge the gap
    - You can get stuck on Visual C++ 2008 because of your dependency on AwesomeXml.dll

- As a library developer…
  - You can re-release your library each time a new version of Visual C++ is released
  - You can ignore the problem and force your customers to use the same Visual C++ version you used
  - You can design your library's API to avoid the problem altogether

How Things *Should* Look



How Things *Should* Look



The Universal CRT

- The Universal CRT is the new version of the CRT for use with Visual C++ 2015
  - But this will be the “last” *version* (named ucrtbase.dll, no version number in name)
  - Future versions of Visual C++ will use this same CRT
  - This CRT will be updated in-place over time, not side-by-side in new DLLs
- The Universal CRT is a Windows operating system component
  - It’s part of Windows 10
  - It’s distributed to older operating systems via Windows Update (or offline redistributables)
  - In Windows 10, many OS components use it instead of the old msvcrt.dll
- The Universal CRT “solves” the FILE* problem described previously
  - …but only for the future
- Other Benefits
  - Reduced resource usage (disk space, address space, TLS/FLS)
  - Far easier for us to make bug fixes and improvements

What is “Compatibility” Anyway?

- Binary Compatibility
  - If you build a DLL or EXE today, it will continue to work in the future.
  - This is what we call “app compat”
  - Must-have; absolutely critical
- Object Compatibility
  - If you build a static library today and later link it into a DLL, the code in that static library will work correctly
  - Highly desirable
- Source Compatibility
  - If you have source code that you build with today’s SDK, it will still build with tomorrow’s SDK
  - Desirable, but really a nice-to-have

Part II
Refactoring

Why?

Why refactor at all? Why not just take what we had in Visual C++ 2013 and declare it “stable”?

- Substantial technical debt needed to be addressed
- Known C and C++ standards conformance issues
- Some APIs impossible to declare as being “stable” in their existing form
  - For example, APIs exported with C++ linkage
- Some APIs expose many internal implementation details

Substantial Technical Debt

```
*erccode _chsize_s(filedes, size) - change size of a file
*
*Purpose:
*       Change file size. Assume file is open for writing, or we can't do it.
*       The DOS way to do this is to go to the right spot and write 0 bytes. The
*       Xenix way to do this is to make a system call. We write '\0' bytes because
*       DOS won't do this for you if you lseek beyond eof, though Xenix will.
```

Base Improvements

- Converted most CRT source files to compile as C++
- Updated *most* code to use RAII for resource management
- Fixed innumerable warnings to make code compile cleanly at /W4, plus some off-by-default warnings

- Removed over half of the #ifdefs from the source code (over 3,000 #ifdefs)
- Eliminated usage of _TCHAR; started using templates for generality

- Reduced number of MSBuild projects required to build the CRT from 900 to about 100

- Substantially refactored and simplified the most convoluted parts of the CRT
  - Startup and termination
  - I/O libraries (notably scanf and printf; _read and _write)

Converting from C to “Modern” C++

- Using which language should the C Standard Library be implemented?
  - C++, obviously!
  - (And some hand-written assembly.)

- C++ has language features that *can* make code simpler than equivalent code written in C
  - Templates are much simpler than macro-heavy, multiply-compiled “generic” C code
  - Use of RAII makes resource management code much, *much* simpler

- We can’t use *all* of C++
  - No exceptions or exception handling
  - No operator new/delete
  - No polymorphic classes (classes with virtual functions)

Before We Begin…

- The source code presented in these slides is *based on* the CRT source code
  - The CRT sources are shipped as part of the Visual C++ SDK and Windows SDK
    - c:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\crt\src
    - c:\Program Files (x86)\Windows Kits\10\Source\[version]\ucrt
  - (Consult the SDK licenses for details about how the sources may be used.)
  - For brevity, some snippets are simplified and some things are renamed or reformatted

- We may discuss some implementation details from these source files
  - Do not assume that these implementation details will always be that way
  - Implementation details are subject to change at any time

Simplifying with RAII

```
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;
```

```
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)) {
```

Creating Something Like unique_ptr<T>

- Ideally, we'd just use unique_ptr<T>.
- But, we can't use unique_ptr<T>,
  - Initially, this was because of *layering*: The CRT is *logically* "below" the STL
  - But we also need to be able to get at the internal pointer
    - Like the GetAddressOf and ReleaseAndGetAddressOf for ATL's CComPtr<T> and WRL's ComPtr<T>
    - For use with APIs like:

```
errno_t get_buffer(
    _Out_ char**  buffer_pointer
    );
```

```
char* buffer{};
get_buffer(&buffer, &size);
```

```
unique_ptr<char> buffer{};
get_buffer(&buffer, &size);
```

- So we created our own unique_ptr-like type,
  - __crt_unique_heap_ptr<T, Deleter>
  - …and switched to use this for most internal memory management.

Creating Something Like unique_ptr<T>

- We also wanted to make use of malloc within the CRT *safer*
  - Lots of “unsafe” calls like `char* buffer = (wchar_t*)malloc(n * sizeof(wchar_t))`
  - We’d like to make this “safer” in two ways:
    - Avoid having to name the type twice (and make the multiplication explicit)
    - Make use of __crt_unique_heap_ptr<T> “automatic” (like make_unique returns a unique_ptr)

- So we did the most “Modern C++” thing we could think of.

- We created some macros:

```
#define __crt_calloc_t(t, n) (__crt_unique_heap_ptr<t>(static_cast<t*>(calloc((n),  sizeof(t)))))
#define __crt_malloc_t(t, n) (__crt_unique_heap_ptr<t>(static_cast<t*>(malloc((n) * sizeof(t)))))
```

Creating Something Like unique_ptr<T>

- We created a total of three RAII containers for use in the CRT:
  - __crt_unique_heap_ptr<T> (for use with malloc'ed pointers)
  - __crt_unique_stack_ptr<T> (for use with malloca'ed pointers)
  - __crt_unique_handle_t<Traits> (for use with "handles" like the various Win32 HANDLE types)

- These three were sufficient for almost all local resource management code in the CRT

- These are used almost everywhere in the CRT codebase now
  - There are a few exceptions; some for good reasons, others just because the code wasn't "updated"

Dealing with __try/__finally

```
int f()
{
    int result = 0;

    __crt_lock(__crt_locale_lock);
    __try
    {
        char* buffer = static_cast<char*>(malloc(1024 * 1024));
        result = compute_result_using_buffer(buffer);
        free(buffer);
    }
    __finally
    {
        __crt_unlock(__crt_locale_lock);
    }

    return result;
}
```

```
int f()
{
    int result = 0;

    __crt_lock(__crt_locale_lock);
    __try
    {
        char* buffer = static_cast<char*>(malloc(1024 * 1024));
        result = compute_result_using_buffer(buffer);
        free(buffer);
    }
    __finally
    {
        __crt_unlock(__crt_locale_lock);
    }

    return result;
}
```

```
int f()
{
    int result = 0;

    __crt_lock(__crt_locale_lock);
    __try
    {
        __crt_unique_ptr<char> buffer(static_cast<char*>(malloc(1024 * 1024)));
        result = compute_result_using_buffer(buffer.get());

    }
    __finally
    {
        __crt_unlock(__crt_locale_lock);
    }

    return result;
}
```

```
int f()
{
    int result = 0;

    __crt_lock(__crt_locale_lock);
    __try
    {
        __crt_unique_ptr<char> buffer(static_cast<char*>(malloc(1024 * 1024)));
        result = compute_result_using_buffer(buffer.get());

    }
    __finally
    {
        __crt_unlock(__crt_locale_lock);
    }

    return result;
}
```

```
int f()
{


    __crt_lock(__crt_locale_lock);
    __try
    {
        __crt_unique_ptr<char> buffer(static_cast<char*>(malloc(1024 * 1024)));
        return   compute_result_using_buffer(buffer.get());

    }
    __finally
    {
        __crt_unlock(__crt_locale_lock);
    }


}
```

```
int f()
{


    __crt_lock(__crt_locale_lock);
    __try
    {
        __crt_unique_ptr<char> buffer(static_cast<char*>(malloc(1024 * 1024)));
        return   compute_result_using_buffer(buffer.get());

    }
    __finally
    {
        __crt_unlock(__crt_locale_lock);
    }


}
```

```
int f()
{

    __crt_unique_lock lock(__crt_locale_lock);


    __crt_unique_ptr<char> buffer(static_cast<char*>(malloc(1024 * 1024)));
    return   compute_result_using_buffer(buffer.get());









}
```

```
int f()
{
    int result = 0;

    __crt_lock(__crt_locale_lock);
    __try
    {
        char* buffer = static_cast<char*>(malloc(1024 * 1024));
        result = compute_result_using_buffer(buffer);
        free(buffer);
    }
    __finally
    {
        __crt_unlock(__crt_locale_lock);
    }

    return result;
}
```

```
int f()
{
    return __crt_lock_and_call(__crt_locale_lock, [&]
    {
        __crt_unique_ptr<char> buffer(static_cast<char*>(malloc(1024 * 1024)));
        return compute_result_using_buffer(buffer.get());
    });
}
```

```
template <typename Callable>
auto __crt_lock_and_call(__crt_lock_id const lock, Callable&& f) -> decltype(Callable())
{
    decltype(Callable()) result{};

    __crt_lock(lock);
    __try
    {
        result = f();
    }
    __finally
    {
        __crt_unlock(lock);
    }

    return result;
}
```

```
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 ...
    });
}
```

Constifying Everything

Const Correctness

- Const correctness says that…
  - if a function has a pointer or reference type parameter
  - *and* the function does not modify the pointed-to or referred-to object,
  - *then* the pointer or reference should be const-qualified
- But this should be considered the *bare minimum*

- The common approach to const is to
  - make APIs const correct and then
  - only add const wherever else it is needed

- This is backwards: We should…
  - make everything that can be const, const
  - refactor code where required when doing so enables us to make more things const

```
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;
}
```

```
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;
}
```

```
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;
}
```

```
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;
}
```

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

    // Etc.
}
```

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

    // Etc.
}
```

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

    // Etc.
}
```

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

Reducing Use of the Preprocessor

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

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

```
#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
    )
{
    // ...
}
```

```
#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,
#else  /* POSITIONAL_PARAMETERS */
#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
    )
{
    // ...
}
```

```
#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);
}
```

```
template <typename Character> class console_output_adapter;
template <typename Character> class stream_output_adapter;
template <typename Character> class string_output_adapter;

template <typename Character, typename OutputAdapter>
class standard_base;

template <typename Character, typename OutputAdapter>
class format_validation_base;

template <typename Character, typename OutputAdapter>
class positional_parameter_base;

template <typename Character, typename OutputAdapter, typename ProcessorBase>
class output_processor : private ProcessorBase { /* ... */ };

template <template <typename, typename> class ProcessorBase, typename Character>
static int common_vfprintf(
    unsigned __int64 options,
    FILE*            stream,
    Character const* format,
    _locale_t        locale,
    va_list          arglist
    );
```

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 we try to keep code within regions simple

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

```
#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
```

Changes for Ensuring Binary Compatibility

Encapsulating Implementation Details

- The CRT headers used to define many internal implementation details inside of the public headers
  - This made it possible for people to take dependencies on these internal implementation details
  - And so, it also made it impossible for us to alter those details without breaking compatibility

Encapsulating Implementation Details

```
typedef struct
{
    char*           _ptr;
    char*           _base;
    int             _cnt;
    long            _flags;
    long            _file;
    int             _charbuf;
    int             _bufsiz;
    char*           _tmpfname;

} FILE;
```

Encapsulating Implementation Details

```
typedef struct
{
    char*            _ptr;
    char*            _base;
    int              _cnt;
    long             _flags;
    long             _file;
    int              _charbuf;
    int              _bufsiz;
    char*            _tmpfname;
    CRITICAL_SECTION _lock;
} _FILEX;
```

Encapsulating Implementation Details

```
typedef struct
{
    void* _Placeholder;
} FILE;
```

Encapsulating Implementation Details

```
FILE* __iob_func(void);

#define stdin  (__iob_func()[0])
#define stdout (__iob_func()[1])
#define stderr (__iob_func()[2])
```

Encapsulating Implementation Details

```
FILE* __acrt_iob_func(unsigned int stream_id);

#define stdin  (__acrt_iob_func(0))
#define stdout (__acrt_iob_func(1))
#define stderr (__acrt_iob_func(2))
```

Eliminating C++ and Data Exports

- C++ and data exports from a DLL are problematic for maintaining stability:
  - C++ name mangling is not stable across major versions of the Visual C++ toolset
  - Data exports directly expose internal implementation details to external callers

- We removed all C++ exports and, where required, replaced them with new C exports
  - For some former exports like the operators new and delete, we made them always-statically-linked

- We removed all data exports and, where we had to keep something similar, we used macro/function pairs
  - Old:

```
extern char* _pgmptr;
```

  - New:

```
char** __p__pgmptr(void);
#define _pgmptr (*__p__pgmptr())
```

Eliminating C++ and Data Exports

```
class _CRTIMP exception
{
public:
    explicit exception(char const* _Message);
    virtual ~exception();
private:
    char const* _What;
    bool        _DoFree;
};
```

```
??0exception@std@@QAE@ABQBD@Z
??0exception@std@@QAE@ABQBDH@Z
??0exception@std@@QAE@ABV01@@Z
??0exception@std@@QAE@XZ
??1exception@std@@UAE@XZ
??4exception@std@@QAEAAV01@ABV01@@Z
??_7exception@std@@6B@
?what@exception@std@@UBEPBDXZ
```

Eliminating C++ and Data Exports

```
struct __std_exception_data
{
    char const* _What;
    bool        _DoFree;
};

void __std_exception_copy(
    __std_exception_data const* _From,
    __std_exception_data*       _To
    );

void __std_exception_destroy(
    __std_exception_data* _Data
    );
```

Eliminating C++ and Data Exports

```
class exception
{
public:
    explicit exception(char const* const _Message)
        : _Data()
    {
        __std_exception_data _InitData = { _Message, true };
        __std_exception_copy(&_InitData, &_Data);
    }

    virtual ~exception() { __std_exception_destroy(&_Data); }
private:
    __std_exception_data _Data;
};
```

C99 and C11 Conformance

- C conformance was generally *pretty good*, but there were some known issues

- We implemented many missing C99 functions in Visual C++ 2013
  - The new C99 floating point and complex math library functions
  - The functions that handle “long long” (e.g. we had strtol and _strtoui64, but not strtoll)

- But some things were too scary to consider implementing in Visual C++ 2013
  - The snprintf and vsnprintf functions
  - The new printf length modifiers like ‘j’ and ‘z’
  - Support for hexadecimal floating point literal parsing

C99 and C11 Conformance

- For the initial release of the Universal CRT, we
  - Implemented the remaining C99 library features
  - Implemented selected C11 features that were either…
    - …easy to implement, or
    - …known to require breaking changes
  - Fixed most known C99 library conformance issues
  - Licensed a commercial conformance test suite to verify conformance

- Why?
  - By making these changes now, we reduced the likelihood of needing to make breaking changes later

- There's one exception to the conformance fixes.

Behavioral Compatibility for printf/scanf

- There are many behavioral differences between
  - the old Microsoft printf/scanf implementations and
  - what is required by the C Standard

- Examples of changed behavior include:
  - Infinity and NaN formatting (e.g., “inf” instead of “1.#INF”; “nan” instead of “1.#QNAN”)
  - ‘F’ used to be a length modifier (for FAR pointers); in C it is a format specifier, %F
  - %e used to print exponents with three digits (“1.5e+003”); it now uses only two if the exponent fits (“1.5e+03”)
  - Many invalid format strings are now rejected (e.g. “%lhlhlhlhlld“)

- We wanted to be able to support the legacy behavior for some of these things
  - But also support the standards-conforming behavior…
  - …and enable each DLL to choose what behavior it wants.

Behavioral Compatibility for printf/scanf

```
int __stdio_common_vfprintf(
    unsigned __int64 _Options,
    FILE*            _Stream,
    char const*      _Format,
    _locale_t        _Locale,
    va_list          _ArgList
    );
```

Behavioral Compatibility for printf/scanf

```
int __stdio_common_vfprintf(
    unsigned __int64 _Options,
    FILE*            _Stream,
    char const*      _Format,
    _locale_t        _Locale,
    va_list          _ArgList
    );
```

Behavioral Compatibility for printf/scanf

```
inline unsigned __int64* __local_stdio_printf_options(void)
{
    static unsigned __int64 _OptionsStorage;
    return &_OptionsStorage;
}

#define _CRT_INTERNAL_LOCAL_PRINTF_OPTIONS (*__local_stdio_printf_options())
```

Behavioral Compatibility for printf/scanf

```
inline int printf(
    char const* const _Format,
    ...)
{
    int _Result;
    va_list _ArgList;
    __crt_va_start(_ArgList, _Format);
    _Result = __stdio_common_vfprintf(
        _CRT_INTERNAL_LOCAL_PRINTF_OPTIONS,
        stdout, _Format, NULL, _ArgList);
    __crt_va_end(_ArgList);
    return _Result;
}
```

Pop Quiz: What is the type of *name*?

```
wprintf(L"Hello, %s!", name);
```

C99: *name* must be a char const*

- %s always takes a char const*

Visual C++ implementation: *name* must be a wchar_t const*

- %s takes a char const* for the narrow printf functions
- %s takes a wchar_t const* for the wide printf functions
- %s takes a _TCHAR const* for the _tprintf functions

References

- Blog articles from the Visual C++ Team Blog:
  - “The Great C Runtime (CRT) Refactoring”
    - https://blogs.msdn.microsoft.com/vcblog/2014/06/10/the-great-c-runtime-crt-refactoring/
  - “C Runtime (CRT) Features, Fixes, and Breaking Changes in Visual Studio 14 CTP1”
    https://blogs.msdn.microsoft.com/vcblog/2014/06/18/c-runtime-crt-features-fixes-and-breaking-changes-in-visual-studio-14-ctp1/
  - “Introducing the Universal CRT”
    - https://blogs.msdn.microsoft.com/vcblog/2015/03/03/introducing-the-universal-crt/
  - “Exception Boundaries”
    - https://blogs.msdn.microsoft.com/vcblog/2014/01/16/exception-boundaries/
- Windows 10 SDK:
  - https://developer.microsoft.com/it-it/windows/downloads/windows-10-sdk