C++ in the Trenches

David Brownell
db@DavidBrownell.com

Northwest C++ Users Group
October 19th, 2005

Background

- Founded Wise Riddles Software (www.WiseRiddles.com) in December 2003
  - Wise Riddles focuses on custom software development, services, mentoring, and training
  - In June 2005 released Audiomatic (www.WiseRiddles.com/Audiomatic), a voice-activated macro application

More Background

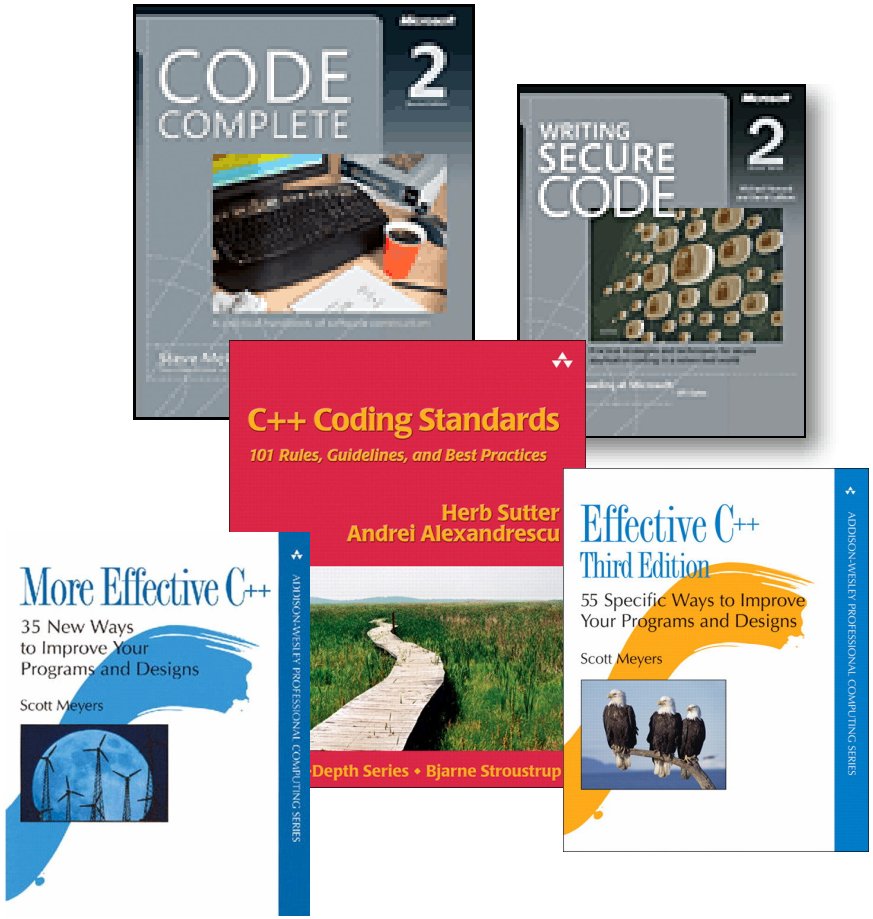
- Started programming in 1986 (BASIC on Apple II GS)
- Started programming with C++ in 1995
- Started programming professionally in 1996

Even More Background

- Shipped 8 commercial applications in C++
- Designer/Architect on 5 of those applications
- Development lead on 5 of those applications

Books to Read

- Code Complete, Second Edition
  Steve McConnell
  ISBN: 0735619670
- Writing Secure Code, Second Edition
  Michael Howard
  ISBN: 0735617228
- C++ Coding Standards
  Herb Sutter and Andrei Alexandrescu
  ISBN: 0321113586
- Effective C++, Third Edition
  Scott Meyers
  ISBN: 0321334876
- More Effective C++
  Scott Meyers
  ISBN: 020163371X



Design/Coding Goals

- Write clear code
- Write concise code
- Write accurate code
- Write secure code
- Write code that is easy to change
- Write code that is difficult to use incorrectly
- Ease transfer of ownership
- Learn from my mistakes

Magic Seven, Plus or Minus Two

- “For memory, a chunk or information is loosely defined as, precisely, one of those items that the immediate memory can hold up to seven of.”
  - George A. Miller, *The Psychological Review*, 1956

- Human brain is capable of simultaneously “juggling” between 5 and 9 items
- When new information is encountered, an item must be discarded or, with effort, committed to longer term memory
- http://www.well.com/user/smalin/miller.html

Magic Seven, Plus or Minus Two

- Code should minimize the number of items occupied in short-term memory
- Guides every aspect of my coding process
- Motivation for many of the techniques I describe in this presentation
- Unfortunately, hard to teach this to new programmers

Roadmap

- Partnering with the Compiler
- ASSERT, VERIFY, and ENFORCE
- Handling Errors without Error
- Final Design Tidbits

Partnering with the Compiler

- The Compiler:
  - Doesn’t get tired
  - Doesn’t feel pressure
  - Doesn’t understand stress
  - Produces predictable results
  - Makes less mistakes
- If the compiler can figure something out, let it!

Partnering with the Compiler

- Initial Design:

```
unsigned char buffer[1024];
// More code
memcpy(buffer, ptr, 1024);
```

- Change:

```
unsigned char buffer[128];
// More code
memcpy(buffer, ptr, 1024);
```

- Modification:

```
#define ELEMENT_COUNT(array) sizeof(array) / sizeof(*array)

unsigned char buffer[128];
// More code
memcpy(buffer, ptr, ELEMENT_COUNT(buffer));
```

Strive to make changes in only one place – let the compiler figure out what it can!

Partnering with the Compiler

- Initial Design:

```
static unsigned long const \
            NUMBER = 100;
// Some Code
double d = static_cast<double>(100) \
            / NUMBER;
```

- Change:

```
static unsigned long const \
            NUMBER = 0;
// Some Code
double d = static_cast<double>(100) \
            / NUMBER;
```

- Modification:

```
static unsigned long const  NUMBER = 100;
// Some Code
BOOST_STATIC_ASSERT(NUMBER != 0);
double d = static_cast<double>(100) / NUMBER;
```

Strive to find errors at compile time rather than runtime!

Partnering with the Compiler

- Tips:
  - When taking a break, type one or two sentences that capture your current though process

```
void AReallyHairyFunction(int i) {
        // Some Code
        I know that the input is valid, something must be wrong in
        this function.  Figure this out tomorrow.
        // More Code
}
```

  - Generates an error that serves as a mnemonic for what you were doing when you left

ASSERT, VERIFY, and ENFORCE

- ASSERT is macro used to ensure that reality matches the programmer's original intent
- ASSERTs:
  - Provide immediate feedback that logic is incorrect
  - Do not add overhead to Release builds
  - Clearly and concisely communicate developer's original intent
- ASSERTs should become second-nature in your coding process

ASSERT, VERIFY, and ENFORCE

- Example of ASSERT macro:

```
#if (defined DEBUG || defined _DEBUG)
        #define ASSERT(stmt) \
                if(stmt == false) \
                { LogError("Assertion failed: " #stmt); Exit(); }
#else
        #define ASSERT(stmt)
#endif
```

ASSERT, VERIFY, and ENFORCE

- Initial Design:

```
void MyFunction(char *szString,
                int positive_int)
{ /* Do something */ }

 MyFunction("foo", 1);
```

- Change:

```
void MyFunction(char *szString,
                int positive_int)
{ /* Do something */ }

MyFunction(0, -3);
```

- Modification:

```
void MyFunction(char *szString, int positive_int) {
        ASSERT(szString);  ASSERT(*szString != 0);  ASSERT(i >= 0);
/* Do something */ }

MyFunction("foo", -3);
```

Strive to ASSERT every assumption, even those that seem obvious. While it may seem like overkill to you, the next developer will thank you for it.

ASSERT, VERIFY, and ENFORCE

- ASSERTs are not a replacement for handling errors
- ASSERTs aren't all that valuable when you are initially writing code, but…
  - Become valuable as you change code
  - Become even more valuable as ownership of the code changes

ASSERT, VERIFY, and ENFORCE

- VERIFY is similar to ASSERT, but the verified code remains in Release builds
- Useful for checking the return status of functions or methods that never fail

ASSERT, VERIFY, and ENFORCE

- Example of VERIFY macro:

```
#if (defined DEBUG || defined _DEBUG)
        #define VERIFY(stmt) ASSERT(stmt)
#else
        #define VERIFY(stmt) stmt
#endif
```

ASSERT, VERIFY, and ENFORCE

- Initial Design:

```
bool ReturnsTrue(void) {
        /* Do something */
        return(true) }

ReturnsTrue();
```

- Change:

```
bool ReturnsTrue(void) {
        if(OnceInABlueMoon())
                return(false);
        return(true);
}

ReturnsTrue();
```

- Modification:

```
VERIFY(ReturnsTrue());
```

Strive to use VERIFYs to ensure methods succeed, but…

ASSERT, VERIFY, and ENFORCE

- Use VERIFYs sparingly
  - If a method always returns a successful error code, should it return an error code at all?
  - If a method returns a failure error code, shouldn't it be handled?
- ENFORCE should be used in place of VERIFYs

ASSERT, VERIFY, and ENFORCE

- ENFORCE is a macro that is compiled in both Debug and Release builds
- Throws exception when encountering a failure
- Should be used in situations where functions should never fail, but occasionally do
  - This is the reality of our programming ecosystem

ASSERT, VERIFY, and ENFORCE

- Example of ENFORCE macro:

```
#define ENFORCE(stmt) \
        if(stmt == false) \
                throw std::runtime_error("ENFORCE failed");
```

ASSERT, VERIFY, and ENFORCE

- Initial Design:

```
bool ReturnsTrue(void) {
        /* Do something */
        return(true)
}

ReturnsTrue();
```

- Change:

```
bool ReturnsTrue(void) {
        if(OnceInABlueMoon())
                return(false);
        return(true);
}

ReturnsTrue();
```

- Modification:

```
ENFORCE(ReturnsTrue());
```

Strive to use ENFORCEs for all methods that should always succeed

ASSERT, VERIFY, and ENFORCE

- Tips:
  - ASSERT all assumptions
    - Incoming variables for private and protected methods
    - Unsigned integer operations
    - Program flow
  - Use VERIFY sparingly, if at all
  - All functions return values should be checked
    - ENFORCE the function if recovery is not possible

Handling Errors without Error

- Two primary error handling strategies
  - Error Codes
  - Exception Handling
- Choose one strategy up front
  - Error handling is one of the most important (and influential) design decisions you will make
- Stick with your strategy
  - Mental models are difficult with multiple strategies

Handling Errors without Error

- Consider:

```
unsigned char * ptr = new unsigned char; // throw std::bad_alloc
```

- Options:

```
unsigned char * ptr;
try { ptr = new unsigned char; }
catch(std::bad_alloc const &) { return(0); }
```

```
unsigned char * ptr;
ptr = new (nothrow) unsigned char;
if(ptr ==) return(0);
```

- Requires rigid coding standard that is easy to get wrong
- If you are using C++, your error handling strategy has been chosen for you

Handling Errors without Error

- Contractual basis for exception handling:
  - The ***basic*** guarantee:
    - The invariants of the component are preserved and no resources are leaked
  - The ***strong*** guarantee:
    - The operation has either completed successfully or thrown an exception, leaving the program state exactly as it was before the operation started
  - The ***no-throw*** guarantee:
    - The operation will not throw an exception
  - http://www.boost.org/more/generic_exception_safety.html

Handling Errors without Error

- ***Basic*** guarantee is just good programming
- ***No-throw*** guarantee is good for edge cases
  - Destructors
  - Main thread loops
  - Etc
- ***Strong*** guarantee is the most significant, and requires the most work

Handling Errors without Error

- Initial Design:

```
void Start(void) {
        InitInternalState();
        InitCommunications();
        HandshakeWithPeer();
}

void Stop(void) {
        DisconnectPeer();
        TerminateCommunications();
        DestroyInternalState();
}
```

- Change:

```
void InitCommunications(void) {
        // Some Code
        throw \
          std::runtime_error \
          (“Unable to InitCommunications”);
        // More code
}
```

Handling Errors without Error

- Modification:

```
typedef enum StartState { DEFAULT_STATE = 0, INIT_INTERNAL_STATE,
                          INIT_COMMUNICATIONS, HANDSHAKE };

void Start(void) {
        StartState  state_completed = DEFAULT_STATE;
        try {
                InitInteranlState();
                state_completed = INIT_INTERNAL_STATE;
                InitCommunications();
                state_completed = INIT_COMMUNICATIONS;
                HandshakePeer();
                state_completed = HANDSHAKE;
        }
        catch(…) {
                if(state >= HANDSHAKE) DisconnectPeer();
                if(state >= INIT_COMMUNICATIONS) TerminateCommunications();
                if(state >= INIT_INTERNAL_STATE) DestroyInternalState();

                // Communicate error to parent
        }
}
```

Handling Errors without Error

- ScopeGuard makes this process much cleaner
- ScopeGuard:
  - Creates function calls that are executed at the end of the current scope
  - Function arguments are bound to function
  - Is perfect for non-object clean up duties
- http://www.cuj.com/documents/s=8000/cujcexp1812alexandr/alexandr.htm

Handling Errors without Error

- Modification:

```
void Start(void) {
        InitInternalState();
        ScopeGuard                  destroy_internal = \
                                    MakeGuard(&DestroyInternalState, this);

        InitCommunications();
        ScopeGuard                  terminate_communications = \
                                    MakeGuard(&TerminateCommunications, this);

        HandshakePeer();
        ScopeGuard                  disconnect_peer = \
                                    MakeGuard(&DisconnectPeer, this);


        // More code

        // If here, things worked as expected
        disconnect_peer.release();
        terminate_communications.release();
        destroy_internal.release();
}
```

Strive to make every method support the strong exception guarantee. This requires a change in thought, but soon becomes second nature.

Handling Errors without Error

- Initial Design:

```
class MyModule {
        void Method1(void) { throw MyException(); }
        void Method2(void) { throw MyException(); }
        void Method3(void) { throw MyException(); }
}

MyModule m;

try {
        m.Method1();
        m.Method2();
        m.Method3();
}
catch(MyException const &ex) { /* Some Code */ }
```

Handling Errors without Error

- Change:

```
class MyModule {
        void Method1(void) { throw MyException(__FILE__, __LINE__); }
        void Method2(void) { throw MyException(__FILE__, __LINE__); }
        void Method3(void) { throw MyException(__FILE__, __LINE__); }
}

MyModule m;

try {
        m.Method1();
        m.Method2();
        m.Method3();

}
catch(MyException const &ex) {
        std::cerr << "MyException at " << ex.file << ", " << ex.line;
}
```

Handling Errors without Error

- Modification:

```
class MyModule {
        void Method1(void) { THROW_EXCEPTION(MyException()); }
        void Method2(void) { THROW_EXCEPTION(MyException()); }
        void Method3(void) { THROW_EXCEPTION(MyException()); }
}

MyModule m;

try {
        m.Method1();
        m.Method2();
        m.Method3();
}
catch(MyException const &ex) {
        std::cerr << “MyException at “ << ex.file << “, “ << ex.line;
}
```

Strive to include contextual information with exceptions that communicate what, where, and when an error happened.

Handling Errors without Error

```
struct FileLineException {
        char *          pszFile;
        int             iLine;
};

#define THROW_EXCEPTION(ex) ThrowException(ex, __FILE__, __LINE__)

template <typename ExceptionT>
void ThrowException(ExceptionT ex, char *pszFile, int iLine) {
        ex.pszFile = pszFile;
        ex.iLine = iLine;
        throw static_cast<ExceptionT &>(ex);
}
```

Handling Errors without Error

- Initial Design:

```
MyModule m;

try { m.Method1(); } catch(MyException const &ex) { std::cerr << … } catch(…) {}
try { m.Method2(); } catch(MyException const &ex) { std::cerr << … } catch(…) {}
try { m.Method3(); } catch(MyException const &ex) { std::cerr << … } catch(…) {}
```

- Change:

```
void HandleMyException(MyException const &ex) { std::cerr << … }

MyModule m;

try { m.Method1(); } catch(MyException const &ex) { HandleMyException(ex); } catch(…) {}
try { m.Method2(); } catch(MyException const &ex) { HandleMyException(ex); } catch(…) {}
try { m.Method3(); } catch(MyException const &ex) { HandleMyException(ex); } catch(…) {}
```

Handling Errors without Error

- Modification

```
void HandleException(void) {
        try {
                throw; // rethrow existing exception
        } catch(MyException const &ex) {
                std::cerr << ….
        } catch(MyOtherException const &ex) {
                std::cerr << …
        } catch(…) {
                std::cerr << ….
        }
}

MyModule m;

try { m.Method1(); } catch(…) { HandleException(); }
try { m.Method2(); } catch(…) { HandleException(); }
try { m.Method3(); } catch(…) { HandleException(); }
```

Strive to place error handling in one place

Handling Errors without Error

- All exceptions should ultimately be children of std::exception
- All library exceptions should ultimately be children of a common parent
- Always catch everything on thread boundaries
- Include enough information with an exception to reliably infer “why” given “where”, “what”, and “when”

Final Design Tidbits

- Prefer C++ constructs over platform specific techniques
  - C++ is the common denominator for developers working on your project
  - Makes the code easier to port
  - Cleans up design
- Prefer quality public libraries over home-grown solutions
  - More developers will be familiar with the code/terminology
  - Makes the code easier to port
- Beware of GUI, COM, Database, <your framework here> in design
  - Paradigms get mixed
  - Frameworks have a nasty habit of creeping into other areas of the code
- Embrace an easy unit test framework
  - Easy to learn
  - Easy to use
  - Easy to compile
  - Easy to run

Final Design Tidbits

- Embrace smart pointers / RAII Techniques
  - Resource Acquisition Is Initialization (RAII) is one of the greatest strengths of C++
- Use boost::noncopyable
- Use documentation macros
  - Doxygen is a great source code documentation tool
    - www.doxygen.org
- Maintain public/protected/private ordering in class declarations
- Learn from your mistakes!

Questions / Comments?

David Brownell
db@DavidBrownell.com