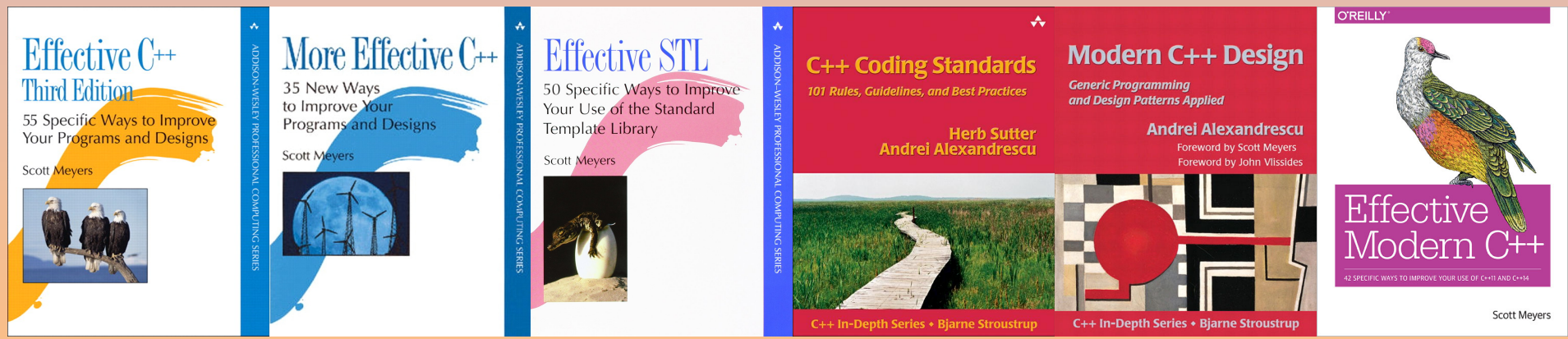
C++ Coding Standards

Akim Demaille akim@lrde.epita.fr

April, 29th 2015

(2016-11-16 09:53:54 +0100 121bea3)



Outline

Design Preprocessor Language Libraries

This Talk is not About...

- A Given Coding Style
- Introducing C++
- Design Patterns
- Advanced Programming Techniques
- Testing
- Documenting
- Linking and ABI issues

Part I

Design

Organizational and Policy Issues

Design a Guide for Developers

- Coding Style
- Tools
- Procedures
- Antipatterns
- Readings
- Publish updates
- Give rules a name

> *Good programmers use their brains, but good guidelines save us having to think out every case.*
>
> — Francis Glassborow

> *Actually, it's more of a guideline than a rule.*
>
> — Captain Barbossa

Design a Guide for Developers

- Coding Style
- Tools
- Procedures
- Antipatterns
- Readings
- Publish updates
- Give rules a name

> *Good programmers use their brains, but good guidelines save us having to think out every case.*
>
> — Francis Glassborow

> *Actually, it's more of a guideline than a rule.*
>
> — Captain Barbossa

Design a Guide for Developers

- Coding Style
- Tools
- Procedures
- Antipatterns
- Readings
- Publish updates
- Give rules a name

> *Good programmers use their brains, but good guidelines save us having to think out every case.*
>
> — Francis Glassborow

> *Actually, it's more of a guideline than a rule.*
>
> — Captain Barbossa

Design a Guide for Developers

- Coding Style
- Tools
- Procedures
- Antipatterns
- Readings
- Publish updates
- Give rules a name

> *Good programmers use their brains, but good guidelines save us having to think out every case.*
>
> — Francis Glassborow

> *Actually, it's more of a guideline than a rule.*
>
> — Captain Barbossa

Design a Guide for Developers

- Coding Style
- Tools
- Procedures
- Antipatterns
- Readings
- Publish updates
- Give rules a name

> *Good programmers use their brains, but good guidelines save us having to think out every case.*
>
> — Francis Glassborow

> *Actually, it's more of a guideline than a rule.*
>
> — Captain Barbossa

Warnings

- Enable warnings, many of them [ccs1]
- At least '`-Wall -Wextra`' for GCC
- Consider using '`-Werror`' on your machine
- Avoid clutter in the build logs
- Learn about `_GLIBCXX_DEBUG`, `_FORTIFY_SOURCES`, `_GLIBCXX_PROFILE`, etc.

Tests

- Write tests and documentation
- Consider writing them *first*

> *Let us change our traditional attitude to the construction of programs. Instead of imagining that our main task is to instruct a computer what to to, let us concentrate rather on explaining to human beings what we want a computer to do.*
>
> — Donald Knuth

Tests

- Write tests and documentation
- Consider writing them *first*

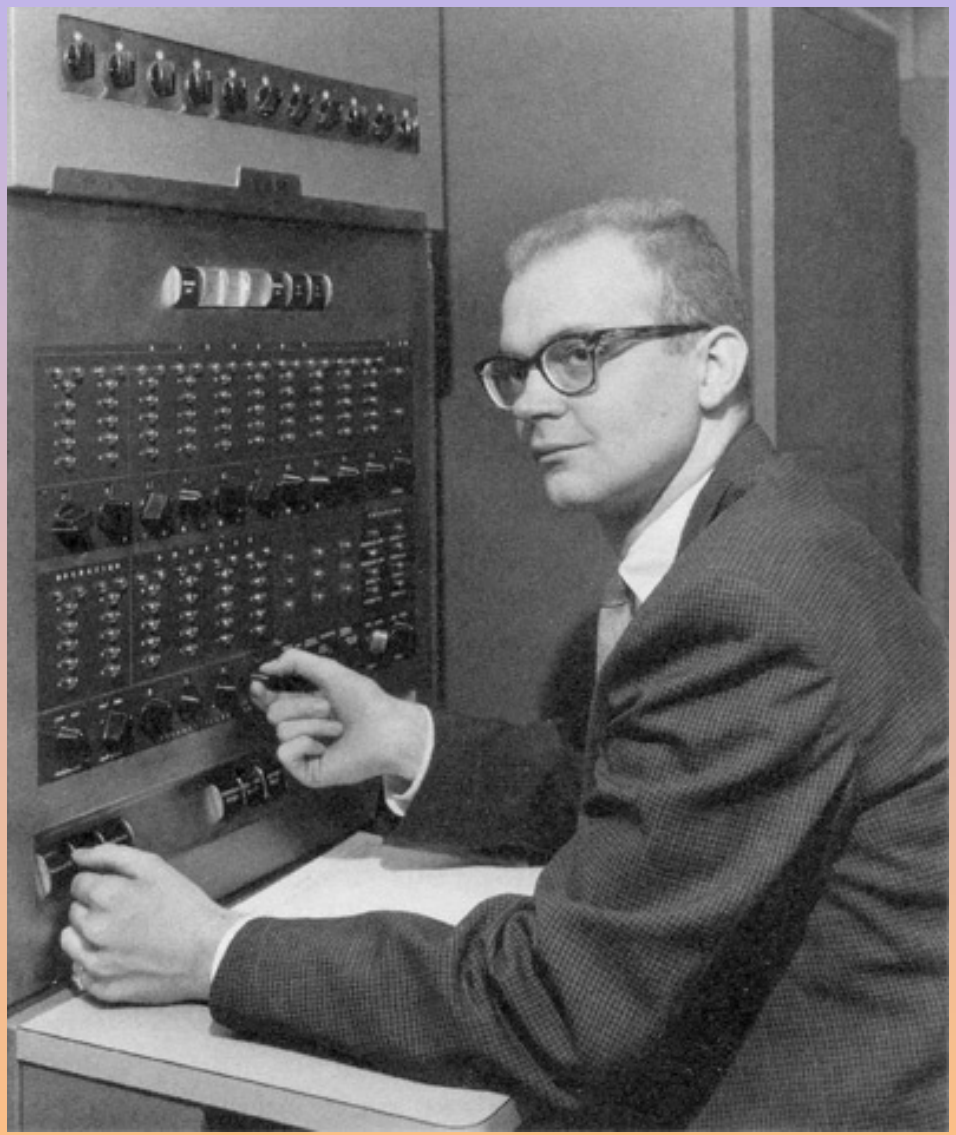
> *Let us change our traditional attitude to the construction of programs. Instead of imagining that our main task is to instruct a computer what to to, let us concentrate rather on explaining to human beings what we want a computer to do.*
>
> — Donald Knuth

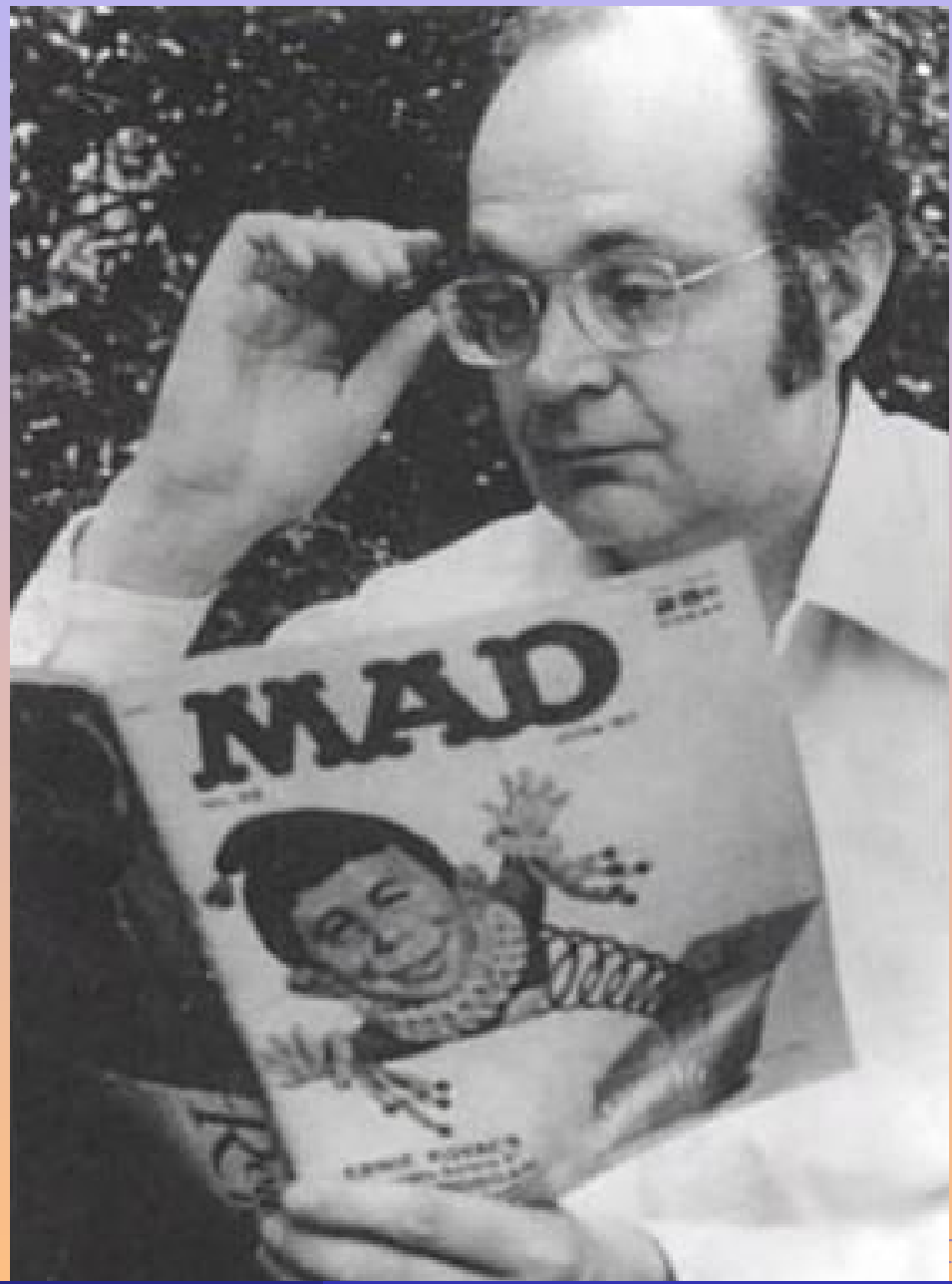
Tests

- Write tests and documentation
- Consider writing them *first*

> *Let us change our traditional attitude to the construction of programs. Instead of imagining that our main task is to instruct a computer what to to, let us concentrate rather on explaining to human beings what we want a computer to do.*
>
> — Donald Knuth

Donald Knuth

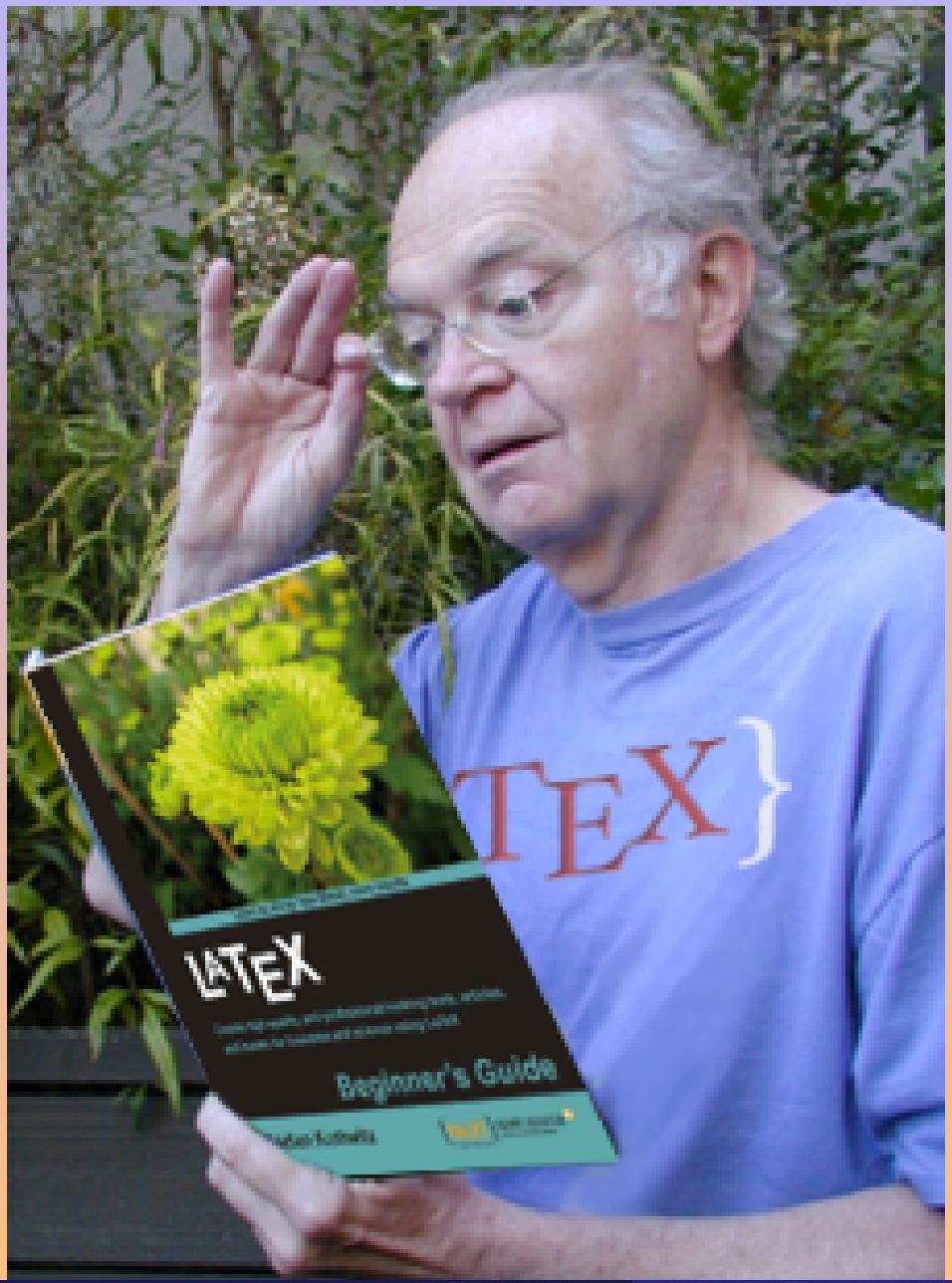




Organizational and Policy Issues

Figure: The Electronic Coach (circa 1959)

Donald Knuth



Commit

- Make commits as small as possible
- Check, then push

Comment Where Appropriate

- Why is this line weird?
  Comment, *not* commit message
- Why this code is weird
  A '`README.txt`' or a special header, and point to it
- Why was this change made?
  Commit message, with details (URL, Ticket, logs of the error)
- Why write this way?
  Guideline
- Treat tests the same way
  Give many details

Comment Where Appropriate

- Why is this line weird?
  Comment, *not* commit message
- Why this code is weird
  A '`README.txt`' or a special header, and point to it
- Why was this change made?
  Commit message, with details (URL, Ticket, logs of the error)
- Why write this way?
  Guideline
- Treat tests the same way
  Give many details

Comment Where Appropriate

- Why is this line weird?
  Comment, *not* commit message
- Why this code is weird
  A '`README.txt`' or a special header, and point to it
- Why was this change made?
  Commit message, with details (URL, Ticket, logs of the error)
- Why write this way?
  Guideline
- Treat tests the same way
  Give many details

Comment Where Appropriate

- Why is this line weird?
  Comment, *not* commit message
- Why this code is weird
  A '`README.txt`' or a special header, and point to it
- Why was this change made?
  Commit message, with details (URL, Ticket, logs of the error)
- Why write this way?
  Guideline
- Treat tests the same way
  Give many details

Comment Where Appropriate

- Why is this line weird?
  Comment, *not* commit message
- Why this code is weird
  A '`README.txt`' or a special header, and point to it
- Why was this change made?
  Commit message, with details (URL, Ticket, logs of the error)
- Why write this way?
  Guideline
- Treat tests the same way
  Give many details

Run Fixit Sessions

- 'FIXME:' killers
- Code migration
- Tool migration
- Valgrind, static analysis tools
- Voluntary basis
- Organize team events
- Bounded in time
- Consider giving away small gifts

Design Style

Correctness, simplicity, and clarity first [ccs6]

> *The cheapest, fastest and most reliable components of a computer system are those that aren't there.*
>
> — Gordon Bell

> *Those missing components are also the most accurate (they never make mistakes), the most secure (they can't be broken into), and the easiest to design, document, test and maintain. The importance of a simple design can't be overemphasized.*
>
> — Jon Bentley

> *Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.*
>
> — Brian W. Kernighan

Correctness, simplicity, and clarity first [ccs6]

> *The cheapest, fastest and most reliable components of a computer system are those that aren't there.*
>
> — Gordon Bell

> *Those missing components are also the most accurate (they never make mistakes), the most secure (they can't be broken into), and the easiest to design, document, test and maintain. The importance of a simple design can't be overemphasized.*
>
> — Jon Bentley

> *Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.*
>
> — Brian W. Kernighan

Correctness, simplicity, and clarity first [ccs6]

> *The cheapest, fastest and most reliable components of a computer system are those that aren't there.*
>
> — Gordon Bell

> *Those missing components are also the most accurate (they never make mistakes), the most secure (they can't be broken into), and the easiest to design, document, test and maintain. The importance of a simple design can't be overemphasized.*
>
> — Jon Bentley

> *Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.*
>
> — Brian W. Kernighan

Don't optimize prematurely
[ccs8]

> *Premature optimization is the root of all evil.*
>
> — Donald Knuth

- Think about scale issues early
- Think about your users (especially when writing libraries)

Don't optimize prematurely
[ccs8]

> *Premature optimization is the root of all evil.*
>
> — Donald Knuth [dubiously quoting Hoare]

- Think about scale issues early
- Think about your users (especially when writing libraries)

Don't optimize prematurely

[ccs8]

> *We should forget about small efficiencies, say about 97% of the time: Premature optimization is the root of all evil.*
>
> — Donald Knuth [dubiously quoting Hoare]

- Think about scale issues early
- Think about your users (especially when writing libraries)

Don't optimize prematurely

[ccs8]

> *We should forget about small efficiencies, say about 97% of the time: Premature optimization is the root of all evil.*
>
> — Donald Knuth [dubiously quoting Hoare]

- Think about scale issues early
- Think about your users (especially when writing libraries)

Beware of the NIH Virus

Not Invented Here

- STL is great
- Boost is rich, portable, well documented, mature
- Audit your dependencies
- Consider contributing

Don't pessimize prematurely
[ccs9]

> *On the other hand, we cannot ignore efficiency.*
>
> — Jon Bentley

- Prefer prefix increment/decrement
- Use the right container
- Avoid pass-by-value where useless
- Consider `const shared_ptr<foo>&`

Pay Attention to $O(f(n))$
[ccs7]

- Don't sweat for small local optimizations, yet...
- Can the code need to scale?
- On what dimension should it scale?
- Chose your algorithms wisely (think of $O(n)$)
- Look at the state of the art (Boost, etc.)

Minimize Global and Shared Data
[ccs10]

Global data

- often prevents reuse
- complicates recursion and/or concurrency
- compromises testing

Shared data

- increases coupling
  Bad for maintenance
- increases contention in concurrent execution
  Prefer communication to data sharing

Hide your Private Parts
[ccs11]

- Expose as little as possible
  Less corelation, fewer recompilations
- Don't expose your members [ccs41]
  Less unexpected interface to maintain, easier consistency
- Don't give away your internals [ccs42]
  Don't provide uncontrolled write access
- Consider the Pimpl idiom [ccs43]

```
class Map
{
  ...
private:
   struct Impl;
   shared_ptr<Impl> pimpl_;
};
```

Avoid Multipurpose Entities

> *Simplicity is prerequisite for reliability.*
>
> — Edsger W. Dijkstra

- Each class, function, variable, library, commit, etc. should have a single purpose
- Thanks to ADL, the interface of a class includes free-standing functions
- `std::basic_string`
  - 103 member functions
  - 71 of them could be nonmember nonfriends
  - many duplicate '`<algorithms>`'
  - some miss in '`<algorithms>`'

> *An API that isn't comprehensible isn't usable.*
>
> — James Gosling

Avoid Multipurpose Entities

> *Simplicity is prerequisite for reliability.*
>
> — Edsger W. Dijkstra

- Each class, function, variable, library, commit, etc. should have a single purpose
- Thanks to ADL, the interface of a class includes free-standing functions
- `std::basic_string`
  - 103 member functions
  - 71 of them could be nonmember nonfriends
  - many duplicate '`<algorithms>`'
  - some miss in '`<algorithms>`'

> *An API that isn't comprehensible isn't usable.*
>
> — James Gosling

Avoid Multipurpose Entities

> *Simplicity is prerequisite for reliability.*
>
> — Edsger W. Dijkstra

- Each class, function, variable, library, commit, etc. should have a single purpose
- Thanks to ADL, the interface of a class includes free-standing functions
- `std::basic_string`
  - 103 member functions
  - 71 of them could be nonmember nonfriends
  - many duplicate '`<algorithms>`'
  - some miss in '`<algorithms>`'

> *An API that isn't comprehensible isn't usable.*
>
> — James Gosling

Avoid Multipurpose Entities

> *Simplicity is prerequisite for reliability.*
>
> — Edsger W. Dijkstra

- Each class, function, variable, library, commit, etc. should have a single purpose
- Thanks to ADL, the interface of a class includes free-standing functions
- `std::basic_string`
  - 103 member functions
  - 71 of them could be nonmember nonfriends
  - many duplicate '`<algorithms>`'
  - some miss in '`<algorithms>`'

> *An API that isn't comprehensible isn't usable.*
>
> — James Gosling

Avoid Multipurpose Entities

> *Simplicity is prerequisite for reliability.*
>
> — Edsger W. Dijkstra

- Each class, function, variable, library, commit, etc. should have a single purpose
- Thanks to ADL, the interface of a class includes free-standing functions
- `std::basic_string`
  - 103 member functions
  - 71 of them could be nonmember nonfriends
  - many duplicate '`<algorithms>`'
  - some miss in '`<algorithms>`'

> *An API that isn't comprehensible isn't usable.*
>
> — James Gosling

Resources are Owned by Objects

- Programmers often manage resources improperly
- Especially in the presence of errors
- malloc/free, new/delete, fopen/fclose, fdopen/fdclose, popen/pclose, lock/unlock, opendir/closedir, openzipper/closezipper
- C++ guarantees the calls to the destructors

RAII — Resource Acquisition Is Initialization

shared_ptr, stream, fdescriptor, lock, dir

Resources are Owned by Objects

- Programmers often manage resources improperly
- Especially in the presence of errors
- malloc/free, new/delete, fopen/fclose, fdopen/fdclose, popen/pclose, lock/unlock, opendir/closedir, openzipper/closezipper
- C++ guarantees the calls to the destructors

RAII — Resource Acquisition Is Initialization

shared_ptr, stream, fdescriptor, lock, dir

Resources are Owned by Objects

- Programmers often manage resources improperly
- Especially in the presence of errors
- malloc/free, new/delete, fopen/fclose, fdopen/fdclose, popen/pclose, lock/unlock, opendir/closedir, openzipper/closezipper
- C++ guarantees the calls to the destructors

RAII — Resource Acquisition Is Initialization

shared_ptr, stream, fdescriptor, lock, dir

Resources are Owned by Objects

- Programmers often manage resources improperly
- Especially in the presence of errors
- malloc/free, new/delete, fopen/fclose, fdopen/fdclose, popen/pclose, lock/unlock, opendir/closedir, openzipper/closezipper
- C++ guarantees the calls to the destructors

RAII — Resource Acquisition Is Initialization

shared_ptr, stream, fdescriptor, lock, dir

Resources are Owned by Objects

- Programmers often manage resources improperly
- Especially in the presence of errors
- malloc/free, new/delete, fopen/fclose, fdopen/fdclose, popen/pclose, lock/unlock, opendir/closedir, openzipper/closezipper
- C++ guarantees the calls to the destructors

RAII — Resource Acquisition Is Initialization

shared_ptr, stream, fdescriptor, lock, dir

Resources are Owned by Objects

Locks

```
void
UServer::schedule(const AsyncJob& j)
{
  BlockLock lock(async_jobs_lock_);
  async_jobs_.push_back(j);
}
```

```
class UServer
{
  Lockable async_jobs_lock_;
  ...
};
```

```
class BlockLock
{
public:
  BlockLock(Lockable& l)
    : lockable_(l)
  {
    lockable_.lock();
  }

  ~BlockLock()
  {
    lockable_.unlock();
  }

private:
  Lockable& lockable_;
};
```

Resources are Owned by Objects

Locks

```
void
UServer::schedule(const AsyncJob& j)
{
  BlockLock lock(async_jobs_lock_);
  async_jobs_.push_back(j);
}
```

```
class UServer
{
  Lockable async_jobs_lock_;
  ...
};
```

```
class BlockLock
{
public:
  BlockLock(Lockable& l)
    : lockable_(l)
  {
    lockable_.lock();
  }

  ~BlockLock()
  {
    lockable_.unlock();
  }

private:
  Lockable& lockable_;
};
```

Resources are Owned by Objects

Locks

```
void
UServer::schedule(const AsyncJob& j)
{
  BlockLock lock(async_jobs_lock_);
  async_jobs_.push_back(j);
}
```

```
class UServer
{
  Lockable async_jobs_lock_;
  ...
};
```

```
class BlockLock
{
public:
  BlockLock(Lockable& l)
    : lockable_(l)
  {
    lockable_.lock();
  }

  ~BlockLock()
  {
    lockable_.unlock();
  }

private:
  Lockable& lockable_;
};
```

Resources are Owned by Objects

C++ 11 [Wikipedia, 2012b]

```cpp
#include <fstream>
#include <iostream>
#include <mutex>
#include <stdexcept>
#include <string>

void write_to_file(const std::string& message)
{
  static std::mutex mutex;  // Protect file access.

  std::lock_guard<std::mutex> lock(mutex);  // Lock file access.
  std::ofstream file("example.txt");
  if (!file.is_open())
    throw std::runtime_error("unable to open file");
  file << message << '\n';

  // File closed 1st, mutex unlocked 2nd.
}
```

Fight Duplication, By All Means, Flight Duplication

> *Copy and paste is a design error.*
>
> — David Parnas

- Often increases the code base
- Increases the object size
- Sign of a missing abstraction
- Duplicates thinking
- An exponential process
- Hinders maintenance
- Complicates code coverage
- Spreads the bugs
- Code metastasis

Fight Duplication, By All Means, Flight Duplication

> *Copy and paste is a design error.*
>
> — David Parnas

- Often increases the code base
- Increases the object size
- Sign of a missing abstraction
- Duplicates thinking
- An exponential process
- Hinders maintenance
- Complicates code coverage
- Spreads the bugs
- Code metastasis

Fight Duplication, By All Means, Flight Duplication

> *Copy and paste is a design error.*
>
> — David Parnas

- Often increases the code base
- Increases the object size
- Sign of a missing abstraction
- Duplicates thinking
- An exponential process
- Hinders maintenance
- Complicates code coverage
- Spreads the bugs
- Code metastasis

Fight Duplication, By All Means, Flight Duplication

> *Copy and paste is a design error.*
>
> — David Parnas

- Often increases the code base
- Increases the object size
- Sign of a missing abstraction
- Duplicates thinking
- An exponential process
- Hinders maintenance
- Complicates code coverage
- Spreads the bugs
- Code metastasis

Fight Duplication at All Levels

- Inlined functions are free
- Use macros *if needed*
- Factor common needs in a small module
- Help migration to this module
- Tests are part of the project
- Factor tests too!
- Likewise for documentation
- Likewise for builds
- Likewise for your gestures
- Mechanize!
- Be cautious not to be too clever

Fight Duplication at All Levels

- Inlined functions are free
- Use macros *if needed*
- Factor common needs in a small module
- Help migration to this module
- Tests are part of the project
- Factor tests too!
- Likewise for documentation
- Likewise for builds
- Likewise for your gestures
- Mechanize!
- Be cautious not to be too clever

Fight Duplication at All Levels

- Inlined functions are free
- Use macros *if needed*
- Factor common needs in a small module
- Help migration to this module
- Tests are part of the project
- Factor tests too!
- Likewise for documentation
- Likewise for builds
- Likewise for your gestures
- Mechanize!
- Be cautious not to be too clever

Fight Duplication at All Levels

- Inlined functions are free
- Use macros *if needed*
- Factor common needs in a small module
- Help migration to this module
- Tests are part of the project
- Factor tests too!
- Likewise for documentation
- Likewise for builds
- Likewise for your gestures
- Mechanize!
- Be cautious not to be too clever

Fight Duplication at All Levels

- Inlined functions are free
- Use macros *if needed*
- Factor common needs in a small module
- Help migration to this module
- Tests are part of the project
- Factor tests too!
- Likewise for documentation
- Likewise for builds
- Likewise for your gestures
- Mechanize!
- Be cautious not to be too clever

Part II

Preprocessor

Macros

Avoid Macros

- Dangerous
  - Lexical
  - No scoping
  - Incomprehensible errors
  - Poor feature set

Avoid Macros

- Dangerous
  - Lexical
  - No scoping
  - Incomprehensible errors
  - Poor feature set
- Yet, terribly useful...

Obfuscated Library Code

```
template<typename _Tp, typename _Alloc>
  void
  list<_Tp, _Alloc>::
  _M_fill_assign(size_type __n, const value_type& __val)
  {
    iterator __i = begin();
    for (; __i != end() && __n > 0; ++__i, --__n)
      *__i = __val;
    if (__n > 0)
      insert(end(), __n, __val);
    else
      erase(__i, end());
  }
```

Obfuscated Library Code

```
template<typename _Tp, typename _Alloc>
  void
  list<_Tp, _Alloc>::
  _M_default_append(size_type __n)
  {
    size_type __i = 0;
    __try
      {
        for (; __i < __n; ++__i)
          emplace_back();
      }
    __catch(...)
      {
        for (; __i; --__i)
          pop_back();
        __throw_exception_again;
      }
  }
```

Coding Style

- Use upper case only for macros
  Warn your users that they are playing risky business
- Indent with care
- CamelCase for macro arguments

Use Protection 1

```
// Day 1 in cpp.
#define SUB(X, Y) X-Y
```

Use Protection 1

```
// Day 1 in cpp.
#define SUB(X, Y) X-Y
```

```
int three = SUB(2, 1) * 3;
```

Use Protection 1

```
// Day 1 in cpp.
#define SUB(X, Y) X-Y
```

```
int three = SUB(2, 1) * 3;
```

```
int three = 2 - 1 * 3;
```

Use Protection 1

```
// Day 1 in cpp.
#define SUB(X, Y) X-Y
```

```
int three = SUB(2, 1) * 3;
```

```
int three = 2 - 1 * 3;
```

```
int two = SUB(1, SUB(3, 2));
```

Use Protection 1

```
// Day 1 in cpp.
#define SUB(X, Y) X-Y
```

```
int three = SUB(2, 1) * 3;
```

```
int three = 2 - 1 * 3;
```

```
int two = SUB(1, SUB(3, 2));
```

```
int two = 1 - 3 - 2;
```

Use Protection 1

```
// Day 1 in cpp.
#define SUB(X, Y) X-Y
```

```
int three = SUB(2, 1) * 3;
```

```
int three = 2 - 1 * 3;
```

```
int two = SUB(1, SUB(3, 2));
```

```
int two = 1 - 3 - 2;
```

```
int zero = SUB(1, -1);
```

Use Protection 1

```
// Day 1 in cpp.
#define SUB(X, Y) X-Y
```

```
int three = SUB(2, 1) * 3;
```

```
int three = 2 - 1 * 3;
```

```
int two = SUB(1, SUB(3, 2));
```

```
int two = 1 - 3 - 2;
```

```
int zero = SUB(1, -1);
```

```
int zero = 1 --1;
```

Use Protection 1

```
// Day 1 in cpp.
#define SUB(X, Y) X-Y
```

```
int three = SUB(2, 1) * 3;
```

```
int three = 2 - 1 * 3;
```

```
int two = SUB(1, SUB(3, 2));
```

```
int two = 1 - 3 - 2;
```

```
int zero = SUB(1, -1);
```

```
int zero = 1 --1;
```

```
// Day 2 in cpp.
#define SUB(X, Y) ((X) - (Y))
```

Use Protection 2

```
// Much better.
inline int sub(int x, int y) { return x - y; }
```

Use Protection 2

```
// Much better.
inline int sub(int x, int y) { return x - y; }
```

```
// Much much better.
template <typename T>
inline T sub(T x, T y) { return x - y; }
```

Use Protection 2

```
// Much better.
inline int sub(int x, int y) { return x - y; }
```

```
// Much much better.
template <typename T>
inline T sub(T x, T y) { return x - y; }
```

```
// Much much much better.
template <typename T, typename U>
inline T sub(T x, U y) { return x - y; }
```

Use Protection 2

```
// Much better.
inline int sub(int x, int y) { return x - y; }
```

```
// Much much better.
template <typename T>
inline T sub(T x, T y) { return x - y; }
```

```
// Much much much better.
template <typename T, typename U>
inline T sub(T x, U y) { return x - y; }
```

```
// Much much much much better.
template <typename T, typename U>
inline auto sub(T x, U y) -> decltype(x - y) { return x - y; }
```

Use Protection 2

```
// Much better.
inline int sub(int x, int y) { return x - y; }
```

```
// Much much better.
template <typename T>
inline T sub(T x, T y) { return x - y; }
```

```
// Much much much better.
template <typename T, typename U>
inline T sub(T x, U y) { return x - y; }
```

```
// Much much much much better.
template <typename T, typename U>
inline auto sub(T x, U y) -> decltype(x - y) { return x - y; }
```

```
// Much much much much much better.
template <typename T, typename U>
inline auto sub(T x, U y) { return x - y; }
```

Use the do-while idiom, 1

```
#define SWAP(X, Y) \
    t = X;         \
    X = Y;         \
    Y = x;
```

Use the do-while idiom, 1

```
#define SWAP(X, Y) \
    t = X;         \
    X = Y;         \
    Y = x;
```

```
if (y < x)
  SWAP(y, x);
```

Use the do-while idiom, 1

```
#define SWAP(X, Y) \
    t = X;         \
    X = Y;         \
    Y = x;
```

```
if (y < x)
  SWAP(y, x);
```

```
if (y < x)
  t = X;
X = Y;
Y = x;
```

Use the do-while idiom, 2

```
#define SWAP(X, Y) \
  {                \
    int x = X;     \
    X = Y;         \
    Y = x;         \
  }
```

Use the do-while idiom, 2

```
#define SWAP(X, Y) \
  {                \
    int x = X;     \
    X = Y;         \
    Y = x;         \
  }
```

```
if (y < x)
  SWAP(y, x);
else
  printf("done");
```

Use the do-while idiom, 2

```
#define SWAP(X, Y) \
  {                \
    int x = X;     \
    X = Y;         \
    Y = x;         \
  }
```

```
if (y < x)
  SWAP(y, x);
else
  printf("done");
```

```
if (y < x)
  {
    ...
  }
;
else
  printf("done");
```

```
error: 'else' without a previous 'if'
   else
   ^
```

Use the do-while idiom, 3

```
#define SWAP(X, Y) \
  do {             \
    int x = X;     \
    X = Y;         \
    Y = x;         \
  } while (false)
```

Use the do-while idiom, 3

```
#define SWAP(X, Y) \
  do {             \
    int x = X;     \
    X = Y;         \
    Y = x;         \
  } while (false)
```

```
if (y < x)
  SWAP(y, x);
else
  printf("done");
```

.

Use the do-while idiom, 3

```
#define SWAP(X, Y) \
  do {             \
    int x = X;     \
    X = Y;         \
    Y = x;         \
  } while (false)
```

```
if (y < x)
  SWAP(y, x);
else
  printf("done");
```

```
if (y < x)
  do {
    ...
  } while (false);
else
  printf("done");
```

Use the do-while idiom, 3

```
#define SWAP(X, Y) \
  do {             \
    int x = X;     \
    X = Y;         \
    Y = x;         \
  } while (false)
```

```
if (y < x)
  SWAP(y, x);
else
  printf("done");
```

```
if (y < x)
  do {
    int x = y;
    y = x;
    x = x;
  } while (false);
else
  printf("done");
```

Use the do-while idiom, 3

```
#define SWAP(X, Y) \
  do {             \
    int x = X;     \
    X = Y;         \
    Y = x;         \
  } while (false)
```

```
if (y < x)
  SWAP(y, x);
else
  printf("done");
```

```
if (y < x)
  do {
    int x = y;
    y = x;
    x = x;
  } while (false);
else
  printf("done");
```

See __COUNTER__.

Prefer enums to integers (or macros)
[ccs14]

- Enums have names in debuggers
- Their "deconstructor" ('switch') is checked by the compiler
- Never use 'default:' for a switch over an enum

```
switch (traffic_light->color())
{
case green:  pass(); break
case orange: cops ? slow_down() : accelerate(); break
case red:    stop(); break
case yellow:
case blue:
case pink:
  assert(!"invalid color (too drunk to drive?)");
}
```

Headers

Headers

- Document public API in headers
- Self contained
  No required pre-`#include`s
- Protected against multiple inclusions [ccs24]

```
#ifndef API_FILE_HH_
# define API_FILE_HH_

namespace api
{
  ...
}
#endif // !API_FILE_HH_
```

- Avoids clashes
- Faster
  (optimized by compilers)

- Using `#pragma once` seems to be portable enough

Header Inclusion

- As few as possible
- As many forward declarations as possible
  Fewer dependencies, faster compilations
- Maintain a '`fwd.hh`' file per directory
  Avoids duplication
- Use '`<iosfwd>`' instead of '`<iostream>`'
- On occasions, check for useless includes
- If there is no required order, sort
- There should not be required order
- '`foo.cc`' *starts* by including '`foo.hh`'
  Find missing includes

Header Inclusion

- As few as possible
- As many forward declarations as possible
  Fewer dependencies, faster compilations
- Maintain a '`fwd.hh`' file per directory
  Avoids duplication
- Use '`<iosfwd>`' instead of '`<iostream>`'
- On occasions, check for useless includes
- If there is no required order, sort
- There should not be required order
- '`foo.cc`' *starts* by including '`foo.hh`'
  Find missing includes

Header Inclusion

- As few as possible
- As many forward declarations as possible
  Fewer dependencies, faster compilations
- Maintain a '`fwd.hh`' file per directory
  Avoids duplication
- Use '`<iosfwd>`' instead of '`<iostream>`'
- On occasions, check for useless includes
- If there is no required order, sort
- There should not be required order
- '`foo.cc`' *starts* by including '`foo.hh`'
  Find missing includes

Header Inclusion

- As few as possible
- As many forward declarations as possible
  Fewer dependencies, faster compilations
- Maintain a '`fwd.hh`' file per directory
  Avoids duplication
- Use '`<iosfwd>`' instead of '`<iostream>`'
- On occasions, check for useless includes
- If there is no required order, sort
- There should not be required order
- '`foo.cc`' *starts* by including '`foo.hh`'
  Find missing includes

Header Inclusion

- As few as possible
- As many forward declarations as possible
  Fewer dependencies, faster compilations
- Maintain a '`fwd.hh`' file per directory
  Avoids duplication
- Use '`<iosfwd>`' instead of '`<iostream>`'
- On occasions, check for useless includes
- If there is no required order, sort
- There should not be required order
- '`foo.cc`' *starts* by including '`foo.hh`'
  Find missing includes

Going Further

Variadic Macros

Variadic macros are portable enough

```
# define RAISE(Message)                                    \
  runner::raise_primitive_error(Message)
```

Variadic Macros

Variadic macros are portable enough

```
# define RAISE(Message)                                  \
  runner::raise_primitive_error(Message)
```

```
# define FRAISE(...)                                     \
  RAISE(libport::format(__VA_ARGS__))
```

Boost.Preprocessor

```
class Interpreter : public runner::Runner
{
public:
  // 52 types of node.
  object::rObject visit(const ast::And* n);
  object::rObject visit(const ast::Assign* n);
  object::rObject visit(const ast::Assignment* n);
  object::rObject visit(const ast::Ast* n);
  object::rObject visit(const ast::At* n);
  // ...
  object::rObject visit(const ast::While* n);
```

Boost.Preprocessor

```
#define AST_NODES_SEQ                          \
  (And) (Assign) (Assignment) (Ast) (At) \
  ... (While)

#define AST_FOR_EACH_NODE(Macro)                    \
    BOOST_PP_SEQ_FOR_EACH(Macro, , AST_NODES_SEQ)
```

Boost.Preprocessor

```
#define AST_NODES_SEQ                          \
  (And) (Assign) (Assignment) (Ast) (At) \
  ... (While)

#define AST_FOR_EACH_NODE(Macro)                    \
    BOOST_PP_SEQ_FOR_EACH(Macro, , AST_NODES_SEQ)
```

```
class Interpreter : public runner::Runner
{
public:
#define VISIT(Macro, Data, Node)                \
  object::rObject visit(const ast::Node* n);
  AST_FOR_EACH_NODE(VISIT);
#undef VISIT
```

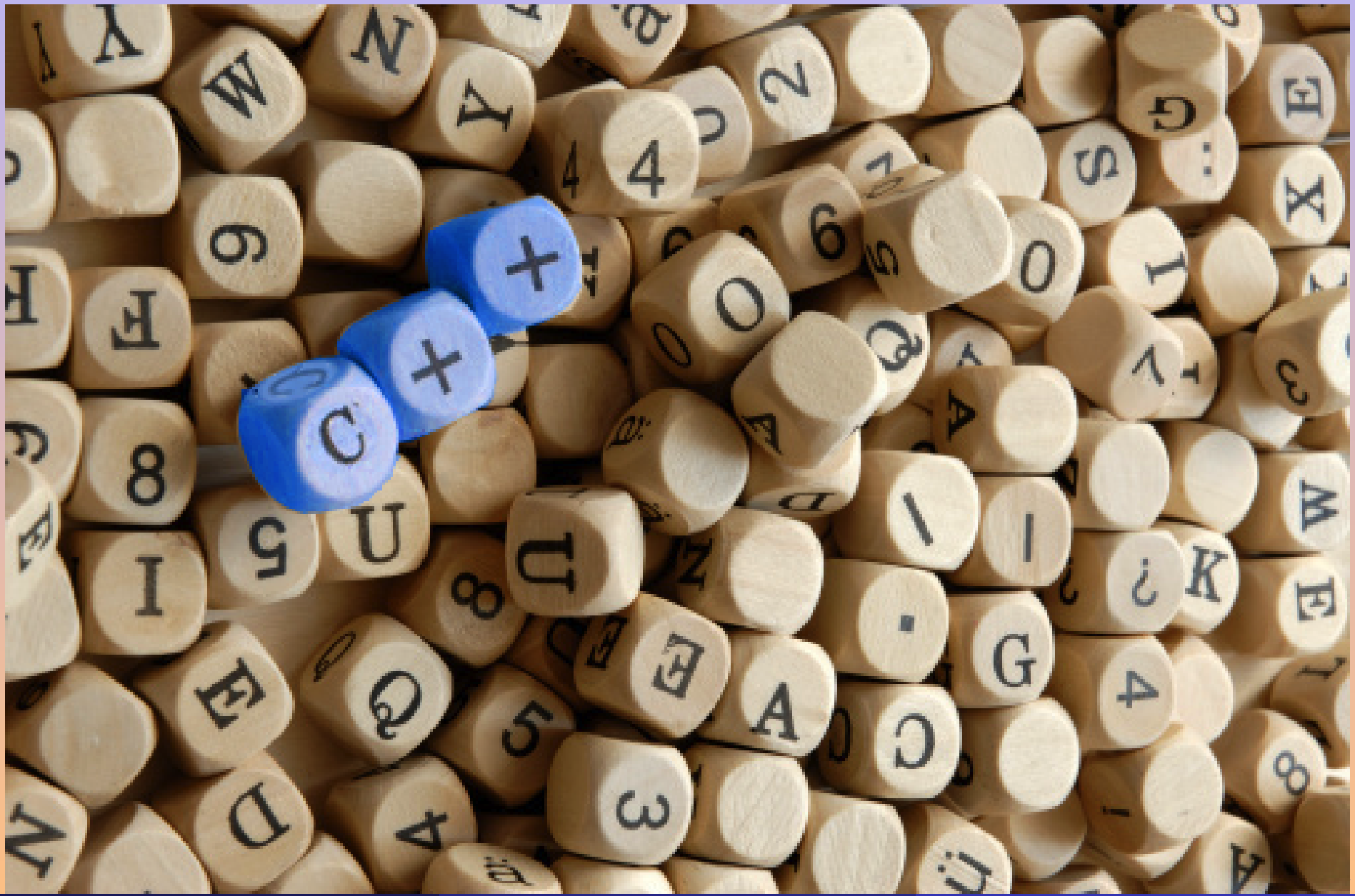
Even Further

- M4 is a nice tool
- There are plenty of tools out there
- Time for a small DSL?
- Check model-driven development

Part III

Language

C++



Coding Style

Write to be Read

> *Programs must be written for people to read, and only incidentally for machines to execute.*
>
> — Harold Abelson & Gerald Jay Sussman

> *Write programs for people first, computers second.*
>
> — Steve McConnell

> *Programming can be fun, so can cryptography;*
>
> — Kreitzberg & Shneiderman

Write to be Read

> *Programs must be written for people to read, and only incidentally for machines to execute.*
>
> — Harold Abelson & Gerald Jay Sussman

> *Write programs for people first, computers second.*
>
> — Steve McConnell

> *Programming can be fun, so can cryptography;*
>
> — Kreitzberg & Shneiderman

Write to be Read

> *Programs must be written for people to read,*
> *and only incidentally for machines to execute.*
>
> — Harold Abelson & Gerald Jay Sussman

> *Write programs for people first,*
> *computers second.*
>
> — Steve McConnell

> *Programming can be fun, so can cryptography;*
> *however they should not be combined.*
>
> — Kreitzberg & Shneiderman

A word of warning

- Coding style is always debatable
- The value is not in the choice
- But on *consistency*
- Avoid patois
- Follow (and enforce?) the rules
- Style changes are separate commits
- Specify the directory/file layout

Formatting

- Ban tabulations
- Kill trailing spaces
- Reject empty initial/file lines
- Stick to a single end-of-line convention
- Automate checks

Naming

- Use clear names for long lived ones
- Use short names for short lived entities
- Be consistent in your API

Beware of Namespace Pollution

- Never use names that start with '_' or contain '__'
  Reserved for the system
- Work in a some namespace
- Put details into a separate namespace
  Beware of Koenig look-up (aka Argument Dependent Look-up)
- Never use `using namespace` in a header

Prefer Explicit over Implicit

- Emphasize assignments at risky places

```
bool is_done = false;
char *filename = nullptr;
...
while ((filename = argz_next (argz, argz_len, filename)))
  if ((is_done = (*func) (filename, data2)))
    break;
```

- Prefer definitions to assignments

```
while (char *filename = argz_next (argz, argz_len, filename))
  if (bool is_done = (*func) (filename, data2))
    break;
```

Prefer Explicit over Implicit

Don't leave "empty" places

```
while ((*yyd++ = *yys++));
```

Prefer Explicit over Implicit

Don't leave "empty" places

```
while ((*yyd++ = *yys++));
```

```
while ((*yyd++ = *yys++))
  ;
```

Prefer Explicit over Implicit

Don't leave "empty" places

```
while ((*yyd++ = *yys++));
```

```
while ((*yyd++ = *yys++))
  ;
```

```
while ((*yyd++ = *yys++))
  /* nothing */;
```

Prefer Explicit over Implicit

Don't leave "empty" places

```
while ((*yyd++ = *yys++));
```

```
while ((*yyd++ = *yys++))
  ;
```

```
while ((*yyd++ = *yys++))
  /* nothing */;
```

```
while ((*yyd++ = *yys++))
  {}
```

Prefer Explicit over Implicit

Don't leave "empty" places

```
while ((*yyd++ = *yys++));
```

```
while ((*yyd++ = *yys++))
  ;
```

```
while ((*yyd++ = *yys++))
  /* nothing */;
```

```
while ((*yyd++ = *yys++))
  {}
```

```
while ((*yyd++ = *yys++))
  continue;
```

Prefer Explicit over Implicit

Don't leave "empty" places

In a Flex scanner:

```
<SC_C_COMMENT>{ /* Comments. */
  [^*/\n\r]+    |
  [*/]          continue;
}
```

Prefer Explicit over Implicit

Don't leave "empty" places

When you have no choice, leave a comment

```
exp.opt: /* empty */     { $$ = 0; }
       | exp             { std::swap($$, $1); }
```

```
for (jobs::iterator i = jobs_.begin(); i != jobs_.end(); /* nothing */)
  if (i->terminated())
    i = jobs_.erase(i);
  else
    ++i;
```

```
try { ... }
catch (...)
{
  // Yes, really, ignore.
}
```

Prefer Explicit over Implicit

Use `explicit` for your constructor [ccs40]

```
explicit Symbol(const std::string& s);
explicit Symbol(const char* s);
```

Prefer Explicit over Implicit

- When overriding a virtual function, repeat `virtual` [ccs38]

```
struct base
{
  virtual const char* name() const = 0;
};

struct derived: base
{
  virtual const char* name() const;
};
```

- If possible, use `override`

```
struct derived: base
{
  const char* name() const override;
};
```

Prefer Explicit over Implicit

- When overriding a virtual function, repeat `virtual` [ccs38]

```
struct base
{
  virtual const char* name() const = 0;
};

struct derived: base
{
  virtual const char* name() const;
};
```

- If possible, use `override`

```
struct derived: base
{
  const char* name() const override;
};
```

Prefer Explicit over Implicit

Exceptions

- Don't state the obvious

```
/// Constructor.
Foo();
/// Destructor.
~Foo();
```

- Avoid cascades of parentheses

```
if (((x < y1) && (y1 < z)) || ((x < y2) && (y2 < z)))
```

Prefer Explicit over Implicit

Exceptions

- Don't state the obvious

```
/// Constructor.
Foo();
/// Destructor.
~Foo();
```

- Avoid cascades of parentheses

```
if (((x < y1) && (y1 < z)) || ((x < y2) && (y2 < z)))
```

```
if (x < y1 && y1 < z || x < y2 && y2 < z)
```

Prefer Explicit over Implicit

Exceptions

- Don't state the obvious

```
/// Constructor.
Foo();
/// Destructor.
~Foo();
```

- Avoid cascades of parentheses

```
if (((x < y1) && (y1 < z)) || ((x < y2) && (y2 < z)))
```

```
if (x < y1 && y1 < z || x < y2 && y2 < z)
```

```
if (x < y1 && y1 < z
    || x < y2 && y2 < z)
```

Prefer Explicit over Implicit

Exceptions

- Don't state the obvious

```
/// Constructor.
Foo();
/// Destructor.
~Foo();
```

- Avoid cascades of parentheses

```
if (((x < y1) && (y1 < z)) || ((x < y2) && (y2 < z)))
```

```
if (x < y1 && y1 < z || x < y2 && y2 < z)
```

```
if (x < y1 && y1 < z
    || x < y2 && y2 < z)
```

```
if (y1 % range(x, z) || y2 % range(x, z))
```

Qualify as Much as Possible

- Use `const` proactively [ccs15]
  In function *declarations*, don't use `const` for passed-by-value arguments

```
int twice(int n);
int twice(const int n) { return 2 * n; }
```

- Use `override`
- Use `throw()` (C++11: `noexcept`)
- Use compiler attributes `__attribute__((const))`, `__attribute__((noreturn))`, `__attribute__((pure))`

Qualify as Much as Possible

- Use `const` proactively [ccs15]
  In function *declarations*, don't use `const` for passed-by-value arguments

```
int twice(int n);
int twice(const int n) { return 2 * n; }
```

- Use `override`
- Use `throw()` (C++11: `noexcept`)
- Use compiler attributes `__attribute__((const))`, `__attribute__((noreturn))`, `__attribute__((pure))`

Qualify as Much as Possible

- Use `const` proactively [ccs15]
  In function *declarations*, don't use `const` for passed-by-value arguments

```
int twice(int n);
int twice(const int n) { return 2 * n; }
```

- Use `override`
- Use `throw()` (C++11: `noexcept`)
- Use compiler attributes `__attribute__((const))`, `__attribute__((noreturn))`, `__attribute__((pure))`

Qualify as Much as Possible

- Use `const` proactively [ccs15]
  In function *declarations*, don't use `const` for passed-by-value arguments

```
int twice(int n);
int twice(const int n) { return 2 * n; }
```

- Use `override`
- Use `throw()` (C++11: `noexcept`)
- Use compiler attributes
  `__attribute__((const))`, `__attribute__((noreturn))`, `__attribute__((pure))`

Reduce Scopes, Define with Initial Value

[ccs18,19]

- Keep code contexts as small as possible
- Simplify the task of readers
- Help the compiler helping you
- Tend to a functional style, with a single assignment
- Use `const`!

Reduce Scopes, Define with Initial Value

```
{
  int d = 0;
  if (dir == south)
    d = 180;
  else if (dir == east)
    d = 90;
  else if (dir == west)
    d = 270;
  ...
}
```

Reduce Scopes, Define with Initial Value

```
{
  int d = 0;
  if (dir == south)
    d = 180;
  else if (dir == east)
    d = 90;
  else if (dir == west)
    d = 270;
  ...
}
```

```
inline
int angle(direction d)
{
  switch(d)
  {
  case north: return   0;
  case east:  return  90;
  case south: return 180;
  case west:  return 270;
  }
}
```

```
{
  const int d = angle(dir);
  ...
}
```

Reduce Scopes, Define with Initial Value

```
static void add(const std::string& key, libport::utime_t d)
{
  Values::iterator i = hash.find(key);
  if (i == hash.end())
  {
    Value& v = hash[key];
    v.sum = v.min = v.max = d;
    v.count = 1;
  }
  else
  {
    i->second.sum += d;
    i->second.count++;
    i->second.min = std::min(i->second.min, d);
    i->second.max = std::max(i->second.max, d);
  }
}
```

Reduce Scopes, Define with Initial Value

```
static void add(const std::string& key, libport::utime_t d)
{
  if (Value const* i = libport::find0(hash, key))
  {
    i->sum += d;
    i->count++;
    i->min = std::min(i->min, d);
    i->max = std::max(i->max, d);
  }
  else
  {
    Value& v = hash[key];
    v.sum = v.min = v.max = d;
    v.count = 1;
  }
}
```

Reduce Scopes, Define with Initial Value

- Prefer `static` variables to globals
- Introduce variables as late as possible
- Consider introducing scopes to shorten extent
- Avoid useless "optimizations"
  Don't extract an integer definition from a loop
- Yet, avoid useless pessimizations
  Try to reuse objects in loops

Reduce Scopes, Define with Initial Value

- Prefer `static` variables to globals
- Introduce variables as late as possible
- Consider introducing scopes to shorten extent
- Avoid useless "optimizations"
  Don't extract an integer definition from a loop
- Yet, avoid useless pessimizations
  Try to reuse objects in loops

Isolate Portability Issues

- Do not use `_MSC_VER` and other `HAVE_FOO` in the main code
- Define, and promote, an abstraction layer
- Promote the *standard* interface
- Consider providing transparent wrappers for standard headers

Isolate Portability Issues

```
# if defined __GNUC__
#  define ATTRIBUTE_MALLOC     __attribute__((malloc))
#  define ATTRIBUTE_NOINLINE   __attribute__((noinline))
#  define ATTRIBUTE_NORETURN   __attribute__((noreturn))
#  define ATTRIBUTE_NOTHROW    __attribute__((nothrow))
// ...
# elif defined _MSC_VER
#  define ATTRIBUTE_MALLOC
#  define ATTRIBUTE_NOINLINE   __declspec(noinline)
#  define ATTRIBUTE_NORETURN   __declspec(noreturn)
#  define ATTRIBUTE_NOTHROW    __declspec(nothrow)
// ...
# else // !__GNUC__ && !_MSC_VER, e.g., Swig.
#  define ATTRIBUTE_MALLOC
#  define ATTRIBUTE_NOINLINE
#  define ATTRIBUTE_NORETURN
#  define ATTRIBUTE_NOTHROW
# endif
```

Isolate Portability Issues

- 'warning-push.hh' (no guard, of course)

```
#ifdef _MSC_VER
# pragma warning(push)
// Lots of comments and URLs.
# pragma warning(disable:                          \
                  4003 4061                        \
                  4121 4127 4150                   \
                  4251 4275 4290                   \
                  4503 4512 4514 4571              \
                  4619 4625 4626 4628 4640 4668    \
                  4710 4711                        \
                  4800 4820)
#endif
```

- 'warning-pop.hh'

```
#ifdef _MSC_VER
# pragma warning(pop)
#endif
```

Functions and Operators

Know Argument Passing Options
[ccs25]

- Pass "small" entities by value
- Pass larger ones by `const&`
- Pass (`const`) pointers when transferring ownership
- Consider Boost.Optional as an alternative to pointers
- Consider Boost.Utility's `call_traits`
- Likewise for return values!
- Discover C++ 11's rvalue-references and move semantics

Operators

- Prefer prefix to postfix operators
- Define postfix from prefix
- Don't mess with `operator&&`, `operator||`, and `operator,`
- Refrain from defining conversion operators [ccs40]
- Beware of non `explicit` constructors
- Prefer overloading

Discover C++11's 'delete'

```
struct NoInt
{
  void f(double i);
  void f(int) = delete;
};
```

```
struct OnlyDouble
{
  void f(double d);
  template<typename T> void f(T) = delete;
};
```

Classes and Inheritance

Kinds of Classes [ccs32]

Value Classes

E.g., std::pair, std::vector.

- Public destructor
- Public copy constructor
- Public assignment with value semantics
- No virtual functions (including the destructor)
- To be used as a concrete class, not as a base class
- Usually on the stack or directly held member of another class

Kinds of Classes [ccs32]

Base Classes

Made for inheritance.

- Destructor is public-and-virtual or protected-and-nonvirtual
- Nonpublic copy constructor and assignment operator
- Establishes interfaces through virtual functions
- Usually on the heap
- Usually used via a (smart) pointer

Kinds of Classes [ccs32]

Traits Classes

Facts about types.

- Contains `typedef`s
- Contains static functions
- No modifiable state or virtuals
- Not usually instantiated (construction disabled)

Kinds of Classes [ccs32]

Policy Classes

Fragments (usually templated) of pluggable behavior.

- May or may not have state or virtual functions
- Not usually instantiated standalone, but only as a base or member

Kinds of Classes [ccs32]

Exception classes

Mix of value (`throw`) and reference semantics (`catch`).

- Public destructor
- Public no-fail constructors
  especially copy constructor
  throwing from an exception's copy constructor aborts
- Has virtual functions
- Often implements cloning and visitation
- Derives from `std::exception`

Public Inheritance is Design by Contract [ccs37,38]

Liskov substitution principle [Wikipedia, 2012a]

If S is a *subtype* of T, then objects of type T may be replaced with objects of type S without altering any of the desirable properties of that program.

- Respect at least the original contract
- You cannot accept less (preconditions)
- You may accept more
- You cannot guarantee less (post-conditions)
- You may guarantee more

Do not inherit (publicly) for reuse.

Nonpublic Inheritance Provides Mixins

- Fragments of plugable code
- Prefer composition
- Sometimes one needs to
  - access a protected member
  - override a virtual function
  - have a longer lifetime than another base subobject
  - use the IS-A relationship privately

Nonpublic Inheritance Provides Mixins

- Fragments of plugable code
- Prefer composition
- Sometimes one needs to
  - access a protected member
  - override a virtual function
  - have a longer lifetime than another base subobject
  - use the IS-A relationship privately

Nonpublic Inheritance Provides Mixins

- Fragments of plugable code
- Prefer composition
- Sometimes one needs to
  - access a protected member
  - override a virtual function
  - have a longer lifetime than another base subobject
  - use the IS-A relationship privately

Nonpublic Inheritance Provides Mixins

- Fragments of plugable code
- **Prefer composition**
- Sometimes one needs to
  - access a protected member
  - override a virtual function
  - have a longer lifetime than another base subobject
  - use the IS-A relationship privately

Nonpublic Inheritance Provides Mixins

- Fragments of plugable code
- **Prefer composition**
- Sometimes one needs to
  - access a protected member
  - override a virtual function
  - have a longer lifetime than another base subobject
  - use the IS-A relationship privately

Consider the Nonvirtual Interface (NVI) pattern [ccs39]

- Public virtual member function

  public an interface
  possibly enforces a contract

  virtual an implementation technique
- Consider making virtual functions nonpublic, and public functions nonvirtual
- Similarities with "Single entry, single exit (SESE)"

Consider the Nonvirtual Interface (NVI) pattern [ccs39]

- Public virtual member function

  public an interface
  possibly enforces a contract

  virtual an implementation technique
- Consider making virtual functions nonpublic, and public functions nonvirtual
- Similarities with "Single entry, single exit (SESE)"

Consider the Nonvirtual Interface (NVI) pattern [ccs39]

- Public virtual member function

  public an interface
  possibly enforces a contract

  virtual an implementation technique
- Consider making virtual functions nonpublic, and public functions nonvirtual
- Similarities with "Single entry, single exit (SESE)"

Consider Nonmember Functions

- Member functions are not the sole object interface
- Freestanding functions also participate (ADL)
- Nonfriend, or friend

Construction, Destruction, and Copying

Explicitly Disable the Unwanted Functions

Non-copyable class.

```
class T
{
  // ...
private:
  // Make T noncopyable.
  T(const T&);
  T& operator=(const T&);
  // Do not implement them.
};
```

```
class T : boost::noncopyable
{
  // ...
};
```

```
class T
{
  // ...
  T(const T&) = delete;
  T& operator=(const T&) = delete;
};
```

Explicitly Disable the Unwanted Functions

Non-copyable class.

```
class T
{
  // ...
private:
  // Make T noncopyable.
  T(const T&);
  T& operator=(const T&);
  // Do not implement them.
};
```

```
class T : boost::noncopyable
{
  // ...
};
```

```
class T
{
  // ...
  T(const T&) = delete;
  T& operator=(const T&) = delete;
};
```

Define and Initialize in the Same Order

[ccs47]

```
class Employee
{
  std::string email_, firstName_, lastName_;
public:
  Employee(const char* firstName, const char* lastName);
  ...
};
```

```
Employee::Employee(const char* fn, const char* ln)
  : firstName_(fn)
  , lastName_(ln)
  , email_(firstName_ + "." + lastName_ + "@narabedla.com")
{}
```

Destructors, Deallocation and Swap Never Fail
[ccs51]

- In C++, destructors *must not* throw
- Swap should never throw [ccs56]

```
T&
T::operator=(const T& other)
{
  T t(other);
  this->swap(t);
  return *this;
}
```

```
T&
T::operator=(T t)
{
  this->swap(t);
  return *this;
}
```

Namespaces and Modules

Namespaces and Modules

- Nonmember function interface of a type in the same namespace [ccs57]
- Details in another (sub)namespace
- Avoid allocating and deallocating memory in different modules [ccs60]
- No entities with linkage in a header file [ccs61]
- No exception propagation across module boundaries [ccs62]

Templates and Genericity

By Default, Be Generic [ccs67]

- Use base classes
- Be const-correct
- Use iterators, not indexes
- Use `!=` on iterator, not `<`
- Use `empty()`, not `size() == 0`
- When writing an algorithm, consider making it template

Error Handling and Exceptions

Learn the Lessons of Design by Contract

- State your pre- and post-conditions
- This is part of documentation
- Check them
- Use `assert`
- or some light-weight variation
- or some heavy-weight variation
- Prefer several asserts to a single one
- Leave the system in a consistent state [ccs71]

Learn the Lessons of Design by Contract

- State your pre- and post-conditions
- This is part of documentation
- Check them
- Use `assert`
- or some light-weight variation
- or some heavy-weight variation
- Prefer several asserts to a single one
- Leave the system in a consistent state [ccs71]

Learn the Lessons of Design by Contract

- State your pre- and post-conditions
- This is part of documentation
- Check them
- Use `assert`
- or some light-weight variation
- or some heavy-weight variation
- Prefer several asserts to a single one
- Leave the system in a consistent state [ccs71]

Prefer Exceptions to Return Codes

- Way too easy to forget about checking an `errno`
- Prefer exceptions
  - Cannot be ignored
  - Propagate
  - Cleaner code
  - Work for constructors and special operators
  - Clean up
- Distinguish exceptional cases from regular errors

  `std::find` does not throw

Prefer Exceptions to Return Codes

- Way too easy to forget about checking an `errno`
- Prefer exceptions
  - Cannot be ignored
  - Propagate
  - Cleaner code
  - Work for constructors and special operators
  - Clean up
- Distinguish exceptional cases from regular errors
  `std::find` does not throw

Prefer Exceptions to Return Codes

- Way too easy to forget about checking an `errno`
- Prefer exceptions
  - Cannot be ignored
  - Propagate
  - Cleaner code
  - Work for constructors and special operators
  - Clean up
- Distinguish exceptional cases from regular errors
  `std::find` does not throw

Prefer Exceptions to Return Codes

But...

"Exceptions are costly"

> "Dealing with exceptions will slow code down, and templates are used specifically to get the best possible performance."
>
> *A good implementation of C++ will not devote a single instruction cycle to dealing with exceptions until one is thrown, and then it can be handled at a speed comparable with that of calling a function.*
>
> — [Abrahams, 2001]

Prefer Exceptions to Return Codes

But. . .

"Exceptions are costly"

> "Dealing with exceptions will slow code down, and templates are used specifically to get the best possible performance."
>
> *A good implementation of C++ will not devote a single instruction cycle to dealing with exceptions until one is thrown, and then it can be handled at a speed comparable with that of calling a function.*
>
> — [Abrahams, 2001]

Prefer Exceptions to Return Codes

But. . .

"Exceptions are costly *in space*"

> *Exception handling overhead can be measured in the size of the executable binary, and varies with the capabilities of the underlying operating system and specific configuration of the C++ compiler. On recent hardware with GNU system software of the same age, the combined code and data size overhead for enabling exception handling is around 7%.*
>
> *Of course, if code size is of singular concern then using the appropriate optimizer setting with exception handling enabled (ie, '`-Os -fexceptions`') may save up to twice that, and preserve error checking.*
>
> — [Carlini et al., 2012, Chap. 3 — Using Exceptions]

Prefer Exceptions to Return Codes

But. . .

"Exceptions are costly *in space*"

> *Exception handling overhead can be measured in the size of the executable binary, and varies with the capabilities of the underlying operating system and specific configuration of the C++ compiler. On recent hardware with GNU system software of the same age, the combined code and data size overhead for enabling exception handling is around 7%.*
>
> *Of course, if code size is of singular concern then using the appropriate optimizer setting with exception handling enabled (ie, '`-Os -fexceptions`') may save up to twice that, and preserve error checking.*
>
> — [Carlini et al., 2012, Chap. 3 — Using Exceptions]

Prefer Exceptions to Return Codes

But really, no, thanks. . .

- Consider passing a mandatory argument for error handling

```
boost::system::error_code ec;
std::string hostname = boost::asio::ip::host_name(ec);
std::cerr << ec.value() << '\n';
```

- Consider the dual approach

```
try
{
  std::cout << boost::asio::ip::host_name() << '\n';
}
catch (const boost::system::system_error& e)
{
  boost::system::error_code ec = e.code();
  std::cerr << ec.value() << '\n';
}
```

Prefer Exceptions to Return Codes

But really, no, thanks. . .

- Consider passing a mandatory argument for error handling

```
boost::system::error_code ec;
std::string hostname = boost::asio::ip::host_name(ec);
std::cerr << ec.value() << '\n';
```

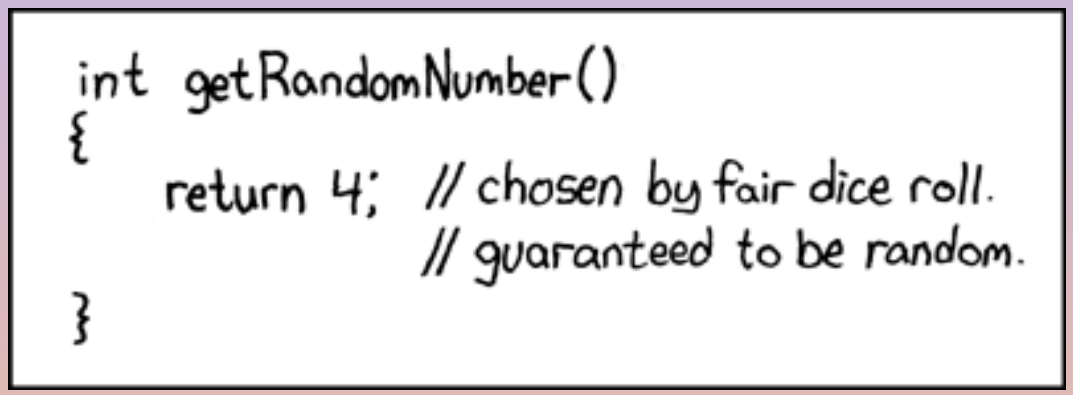
- Consider the dual approach

```
try
{
  std::cout << boost::asio::ip::host_name() << '\n';
}
catch (const boost::system::system_error& e)
{
  boost::system::error_code ec = e.code();
  std::cerr << ec.value() << '\n';
}
```

Prefer Exceptions to Return Codes

But really, no, thanks. . .

- Consider using the `warn_unused_result` attribute




```
int
main()
{
  getRandomNumber();
}
```

Prefer Exceptions to Return Codes

But really, no, thanks. . .

- Consider using the warn_unused_result attribute

```
__attribute__((warn_unused_result))
int getRandomNumber()
{
  return 4; // chosen by fair dice roll.
            // guaranteed to be random.
}
```

```
int
main()
{
  getRandomNumber();
}
```

```
foo.c: In function 'main':
foo.c:11:18: warning: ignoring return value of 'getRandomNumber',
                      declared with attribute warn_unused_result
                      [-Wunused-result]
   getRandomNumber();
                  ^
```

Exceptions

- Throw by value, catch by reference [ccs73]
- Avoid exception specifications [ccs75]
  Except `throw()` and `noexcept`
- Derive from standard exceptions (‘`<stdexcept>`’)
- Avoid inventing a large hierarchy of exception classes
- Check Boost.System (now part of C++11 ‘`<system_error>`’)

Type Safety

The Compiler is your Friend

> *If you lie to the compiler, it will get its revenge.*
>
> — Henry Spencer

> *Trying to outsmart a compiler defeats much of the purpose of using one.*
>
> — Kernighan & Plauger, The Elements of Programming Style

> *C++ tries to guard against Murphy, not Machiavelli.*
>
> — Damian Conway

The Compiler is your Friend

> *If you lie to the compiler, it will get its revenge.*
>
> — Henry Spencer

> *Trying to outsmart a compiler defeats much of the purpose of using one.*
>
> — Kernighan & Plauger, The Elements of Programming Style

> *C++ tries to guard against Murphy, not Machiavelli.*
>
> — Damian Conway

The Compiler is your Friend

> *If you lie to the compiler, it will get its revenge.*
>
> — Henry Spencer

> *Trying to outsmart a compiler defeats much of the purpose of using one.*
>
> — Kernighan & Plauger, The Elements of Programming Style

> *C++ tries to guard against Murphy, not Machiavelli.*
>
> — Damian Conway

Casts

- Avoid type switching, prefer polymorphism [ccs90]
- Don't use C casts
- Avoid `reinterpret_cast`

```
T1* p1 = ...;
T2* p2 = reinterpret_cast<T2*>(p1);
```

```
Tl* p1 = ... ;
void* pV = p1;
T2* p2 = static_cast<T2*>(pV);
```

- Prefer `dynamic_cast` on references rather than pointers
- Consider adding your own `checked_cast`
  bounces to `dynamic_cast` or `static_cast`
- Avoid casting `const` away
  `const_cast` is useful to *add* `const`

Part IV

Libraries

STL: Containers

Prefer Vectors

- One rarely needs lists
- Avoid arrays in C++
- Prefer `std::vector`
- Even when interfacing with C [ccs78]

```
// Wrong!
c_function(v.begin(), v.size());
// Good.  But, of course, do not store the address.
c_function(&*v.begin(), v.size());
c_function(&v.front(), v.size());
c_function(&v[0], v.size());
// For strings.
c_strfun(s.c_str());
c_strnfun(s.data(), s.size());
```

Save Time

- Book space, `reserve`
- Prefer `push_back` [ccs80]
- Better yet, `emplace_back`

Save Time

Consider adding sugar to STL

```
#include <vector>

namespace std
{
  template<typename Lhs, typename Alloc, typename Rhs>
  vector<Lhs, Alloc>&
  operator<<(vector<Lhs, Alloc>& c, const Rhs& v)
  {
    c.push_back(v);
    return c;
  }
}
```

```
std::vector<int> is;
is << 1 << 2 << 3;
```

```
std::vector<int> is{1, 2, 3};
```

```
auto is = std::vector<int>{1, 2, 3};
```

Save Time

Consider adding sugar to STL

```
#include <vector>

namespace std
{
  template<typename Lhs, typename Alloc, typename Rhs>
  vector<Lhs, Alloc>&
  operator<<(vector<Lhs, Alloc>& c, const Rhs& v)
  {
    c.push_back(v);
    return c;
  }
}
```

```
std::vector<int> is;
is << 1 << 2 << 3;
```

```
std::vector<int> is{1, 2, 3};
```

```
auto is = std::vector<int>{1, 2, 3};
```

Save Space

- Shrink a container

```
container<T>(c).swap(c);
```

- Empty a container

```
container<T>().swap(c);
```

- Remove members

```
c.erase(remove(c.begin(), c.end(), value), c.end());
```

```
is.erase(remove_if(begin(is), end(is),
                   [=] (int i) { return i % range(lo, hi); }),
         end(is));
```

STL: Algorithms

Use a Checked Library

- Use Valgrind (of course)
- Enable `_GLIBCXX_DEBUG`
  You might discover singular iterators
- Check `_FORTIFY_SOURCES`

Use a Checked Library

- Check _GLIBCXX_PROFILE

```
#include <vector>
int main() {
  std::vector<int> v;
  for (int k = 0; k < 1024; ++k) v.insert(v.begin(), k);
}
```

```
$ g++ -D_GLIBCXX_PROFILE foo.cc
$ ./a.out
$ cat libstdcxx-profile.txt
vector-to-list: improvement = 5: call stack = 0x804842c ...
    : advice = change std::vector to std::list
vector-size: improvement = 3: call stack = 0x804842c ...
    : advice = change initial container size from 0 to 1024
```

STL has More Than You Might Think

- Selection, partition algorithms [ccs85]
  `partition`, `stable_partition`, `nth_element`
- Several flavors of sort [ccs86]
  `partial_sort`, `partial_sort_copy`, `sort`, `stable_sort`
- Many search algorithms [ccs85]
  `find`, `find_if`, `count`, `count_if`, `binary_search`, `lower_bound`, `upper_bound`, `equal_range`

Algorithms

- Range operations are often faster [ccs81]
- Prefer algorithms to loops [ccs84]

```
deque<double>::iterator current = d.begin();
for (size_t i = 0; i < max; ++i)
{
  current = d.insert(current, data[i] + 41);
  ++current;
}
```

```
// Using Boost.Lambda.
transform(data, data + max,
          inserter(d, d.begin()), _1 + 41);
```

Algorithms

- Algorithms can be clearer [ccs81]

```
std::vector<int>::iterator i;
for (i = v.begin(); i != v.end(); ++i)
  if (x < *i && *i < y)
    break;
```

```
using boost::lambda::_1;
std::vector<int>::iterator i =
   find_if(v.begin(), v.end(), x < _1 && _1 < y);
```

```
auto i = boost::find_if(v,
                        [](int x) { return x < _1 && _1 < y; });
```

Boost

Boost

> *[...]one of the most highly regarded and expertly designed C++ library projects in the world.*
>
> — [Sutter and Alexandrescu, 2005]

> *Item 55: Familiarize yourself with Boost.*
>
> — [Meyers, 2005]

> *The obvious solution for most programmers is to use a library that provides an elegant and efficient platform independent to needed services. Examples are Boost*[...]
>
> — [Stroustrup, 2003]

A Meta Library

- Algorithms
- Broken Compiler Workarounds
- Concurrent Programming
- Containers
- Correctness and Testing
- Data Structures
- Domain Specific
- Function Objects and Higher-order Programming
- Generic Programming
- Image Processing
- Input/Output
- Inter-language Support
- Iterators
- Language Features Emulation
- Math and Numerics
- Memory
- Parsing
- Patterns and Idioms
- Preprocessor Metaprogramming
- Programming Interfaces
- State Machines
- String and Text Processing
- System
- Template Metaprogramming
- Miscellaneous

Boost 1.58.0: 131 libraries

Accumulators, Algorithm, Align, Any, Array, Asio, Assert, Assign, Atomic, Bimap, Bind, Call Traits, Chrono, Circular Buffer, Compatibility, Compressed Pair, Concept Check, Config, Container, Context, Conversion, Core, Coroutine, CRC, Date Time, Dynamic Bitset, Enable If, Endian, Exception, Filesystem, Flyweight, Foreach, Format, Function, Function Types, Functional, Functional/Factory, Functional/Forward, Functional/Hash, Functional/Overloaded Function, Fusion, Geometry, GIL, Graph, Heap, ICL, Identity Type, In Place Factory, Typed In Place Factory, Integer, Interprocess, Interval, Intrusive, IO State Savers, Iostreams, Iterator, Lambda, Lexical Cast, Local Function, Locale, Lockfree, Log, Math, Math Common Factor, Math Octonion, Math Quaternion, Math/Special Functions, Math/Statistical Distributions, Member Function, Meta State Machine, Min-Max, Move, MPI, MPL, Multi-Array, Multi-Index, Multiprecision, Numeric Conversion, Odeint, Operators, Optional, Parameter, Phoenix, Pointer Container, Polygon, Pool, Predef, Preprocessor, Program Options, Property Map, Property Tree, Proto, Python, Random, Range, Ratio, Rational, Ref, Regex, Result Of, Scope Exit, Serialization, Signals2, Smart Ptr, Sort, Spirit, Statechart, Static Assert, String Algo, Swap, System, Test, Thread, ThrowException, Timer, Tokenizer, Tribool, TTI, Tuple, Type Erasure, Type Index, Type Traits, Typeof, uBLAS, Units, Unordered, Utility, Uuid, Value Initialized, Variant, Wave, Xpressive

Boost 1.58.0: 131 libraries

Accumulators, Algorithm, Align, Any, Array, Asio, Assert, Assign, Atomic, Bimap, Bind, Call Traits, Chrono, Circular Buffer, Compatibility, Compressed Pair, Concept Check, Config, Container, Context, Conversion, Core, Coroutine, CRC, Date Time, Dynamic Bitset, Enable If, Endian, Exception, Filesystem, Flyweight, Foreach, Format, Function, Function Types, Functional, Functional/Factory, Functional/Forward, Functional/Hash, Functional/Overloaded Function, Fusion, Geometry, GIL, Graph, Heap, ICL, Identity Type, In Place Factory, Typed In Place Factory, Integer, Interprocess, Interval, Intrusive, IO State Savers, Iostreams, Iterator, Lambda, Lexical Cast, Local Function, Locale, Lockfree, Log, Math, Math Common Factor, Math Octonion, Math Quaternion, Math/Special Functions, Math/Statistical Distributions, Member Function, Meta State Machine, Min-Max, Move, MPI, MPL, Multi-Array, Multi-Index, Multiprecision, Numeric Conversion, Odeint, Operators, Optional, Parameter, Phoenix, Pointer Container, Polygon, Pool, Predef, Preprocessor, Program Options, Property Map, Property Tree, Proto, Python, Random, Range, Ratio, Rational, Ref, Regex, Result Of, Scope Exit, Serialization, Signals2, Smart Ptr, Sort, Spirit, Statechart, Static Assert, String Algo, Swap, System, Test, Thread, ThrowException, Timer, Tokenizer, Tribool, TTI, Tuple, Type Erasure, Type Index, Type Traits, Typeof, uBLAS, Units, Unordered, Utility, Uuid, Value Initialized, Variant, Wave, Xpressive

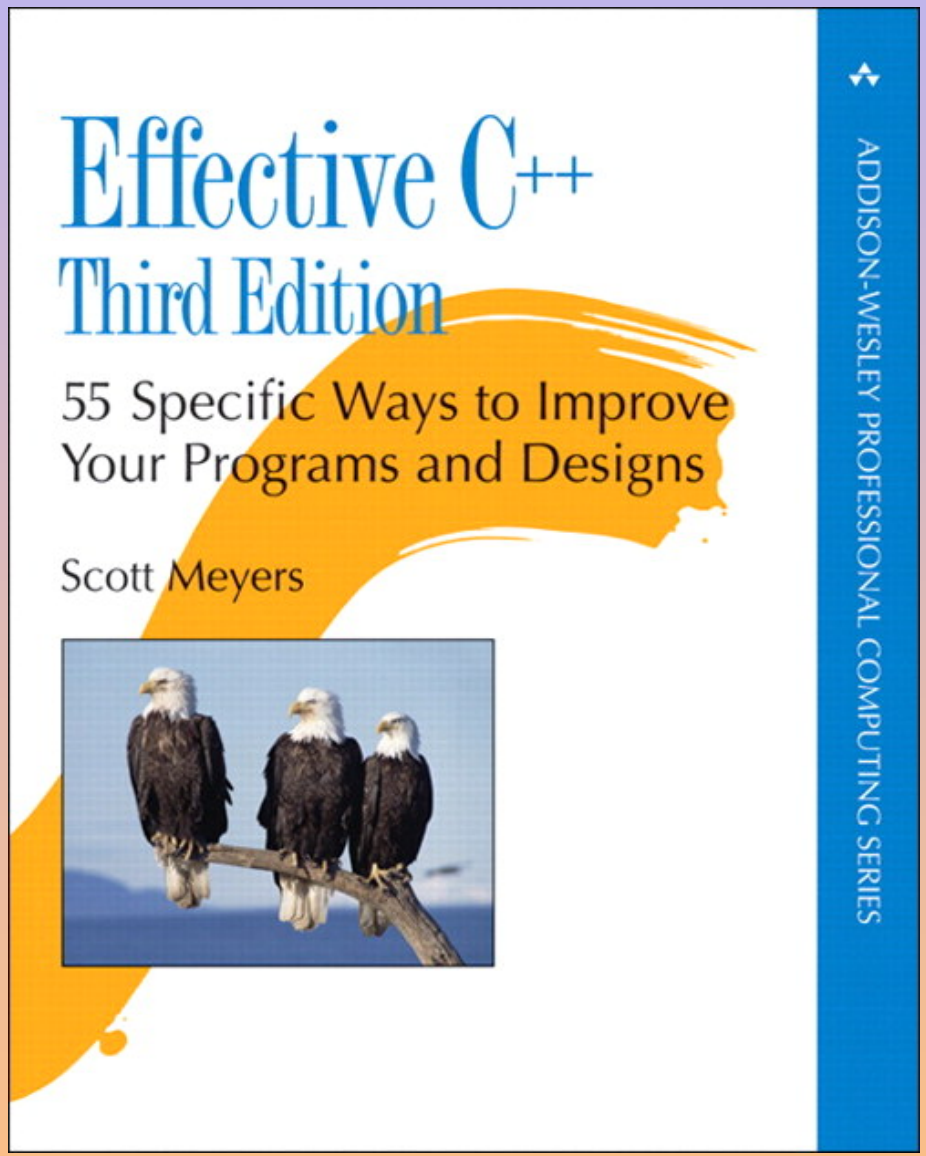
> *Once a new technology starts rolling, if you're not part of the steamroller, you're part of the road.*
>
> — Stewart Brand

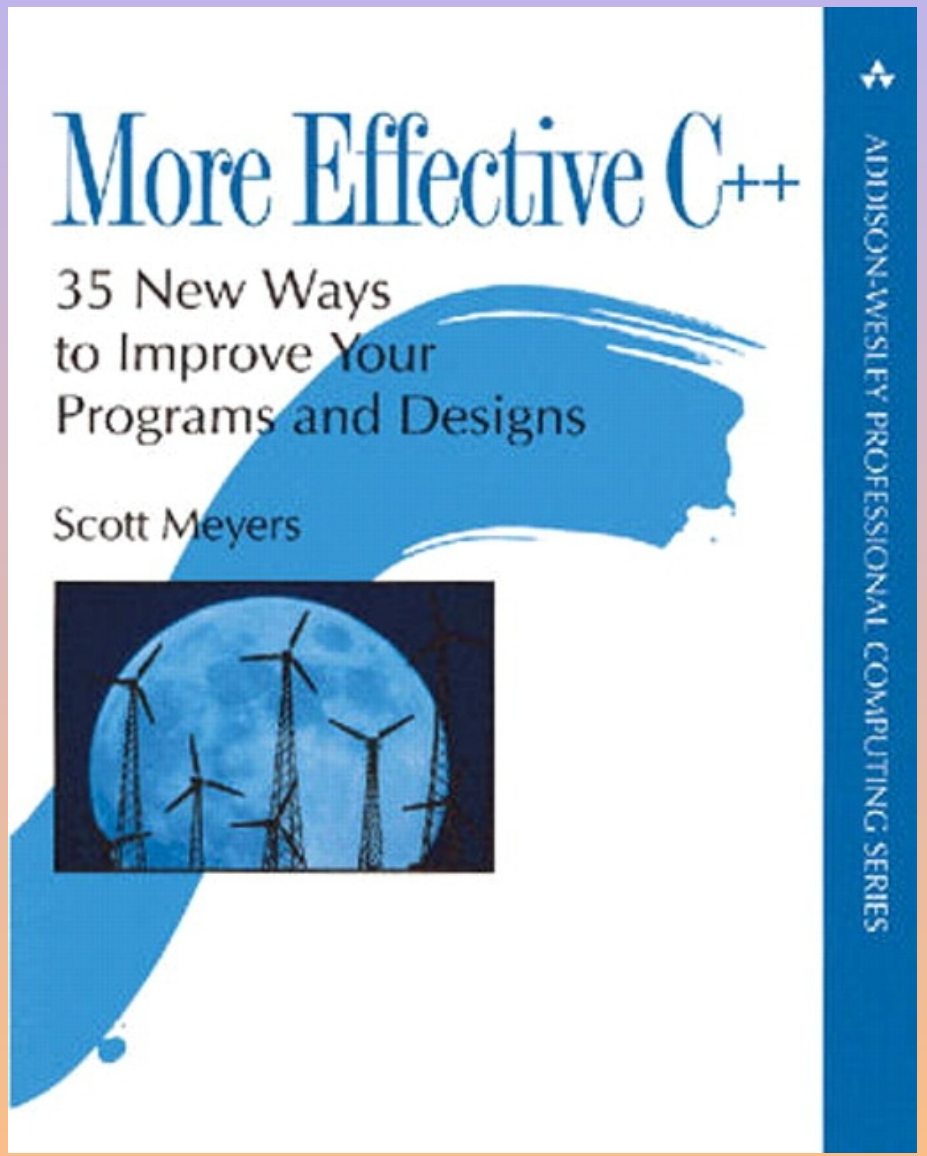
Part V

Appendix

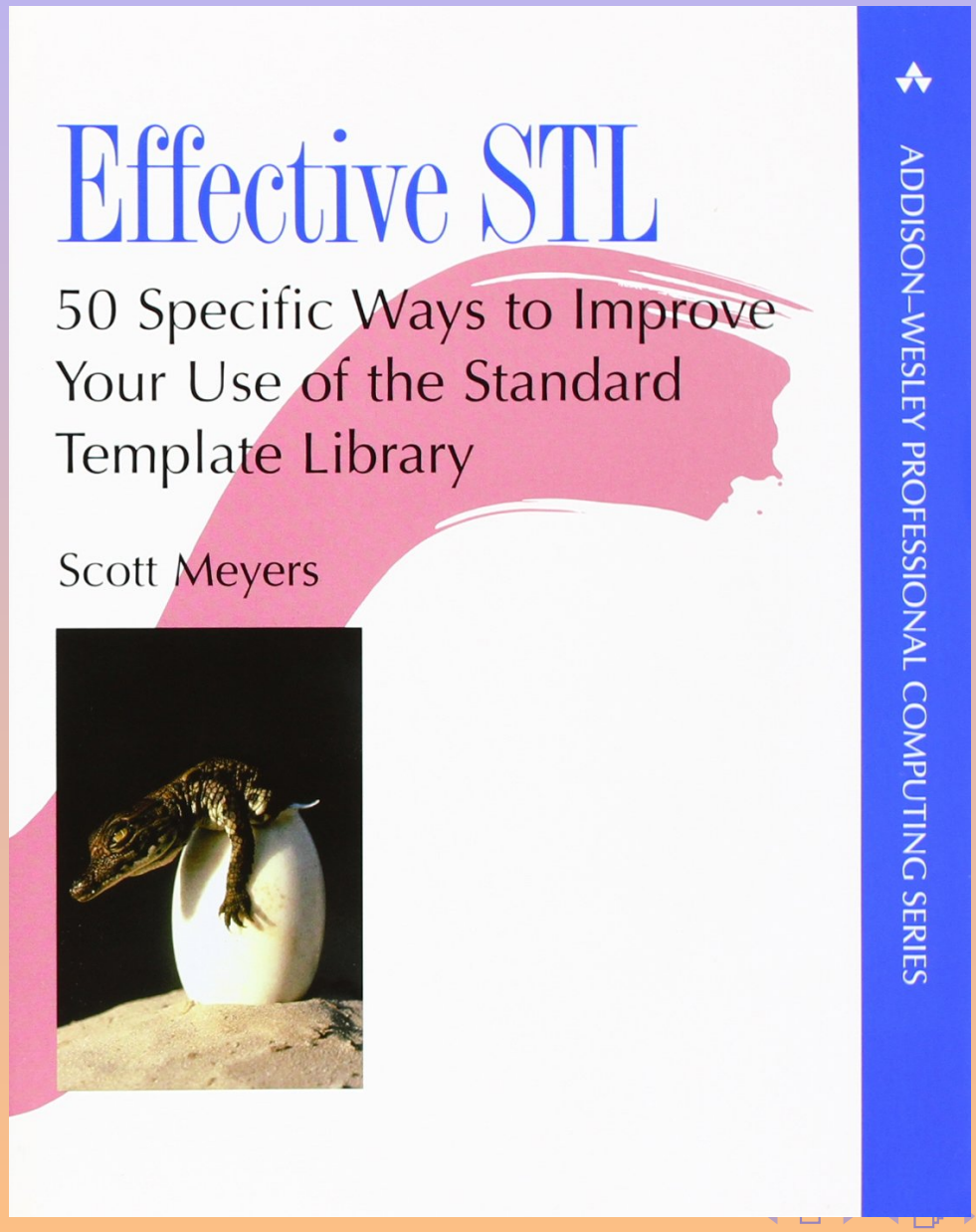
Further Readings

Further Readings

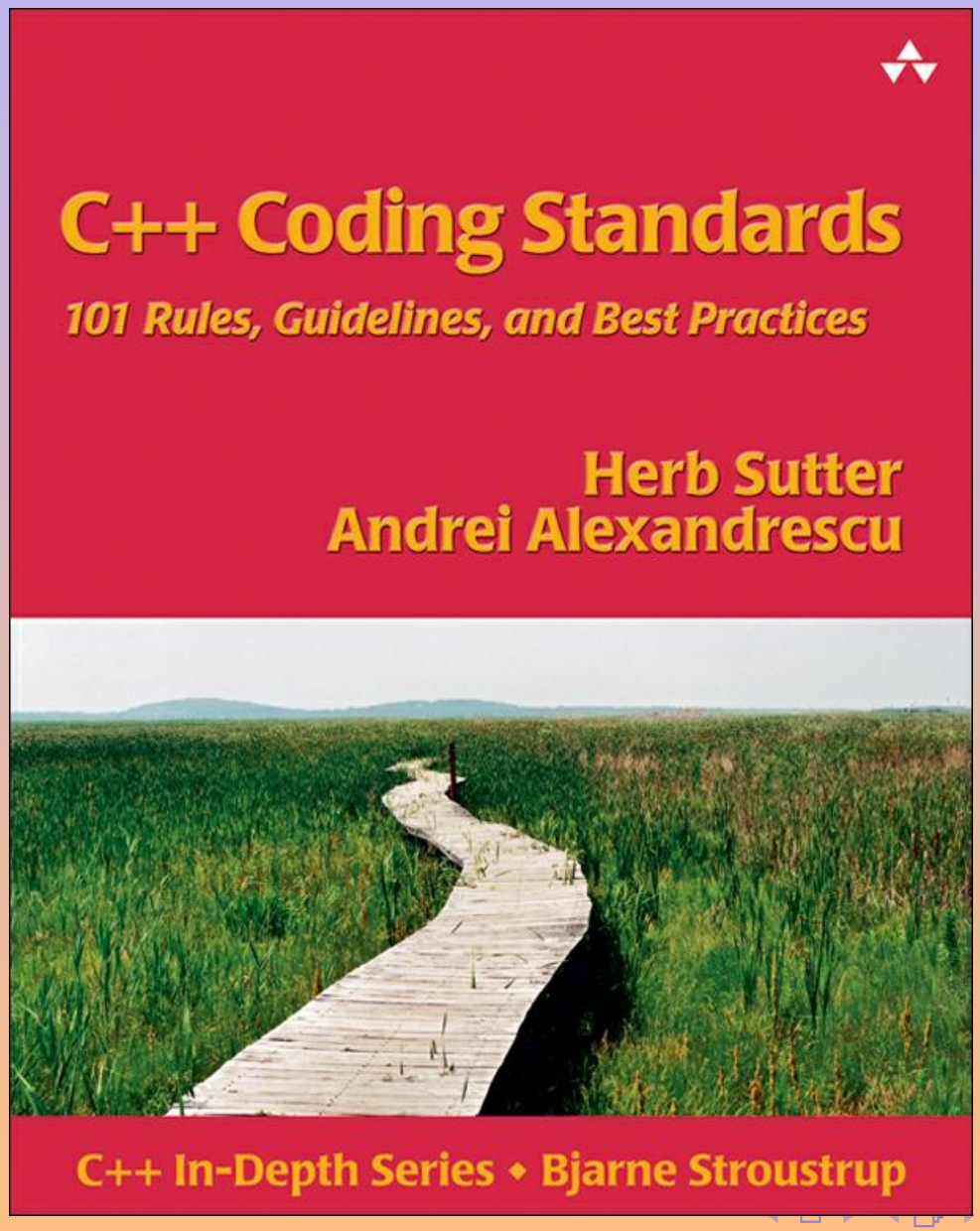




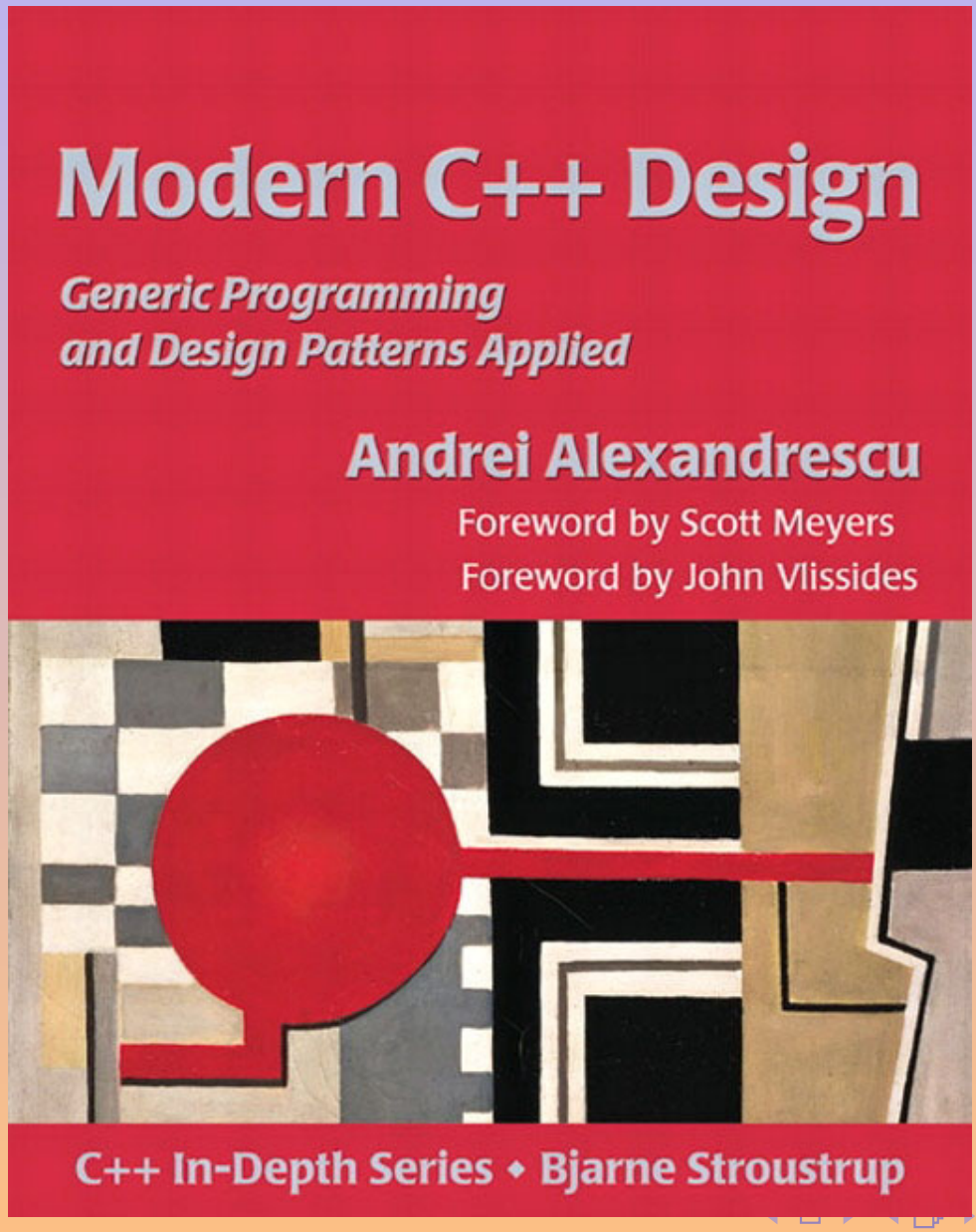
Further Readings



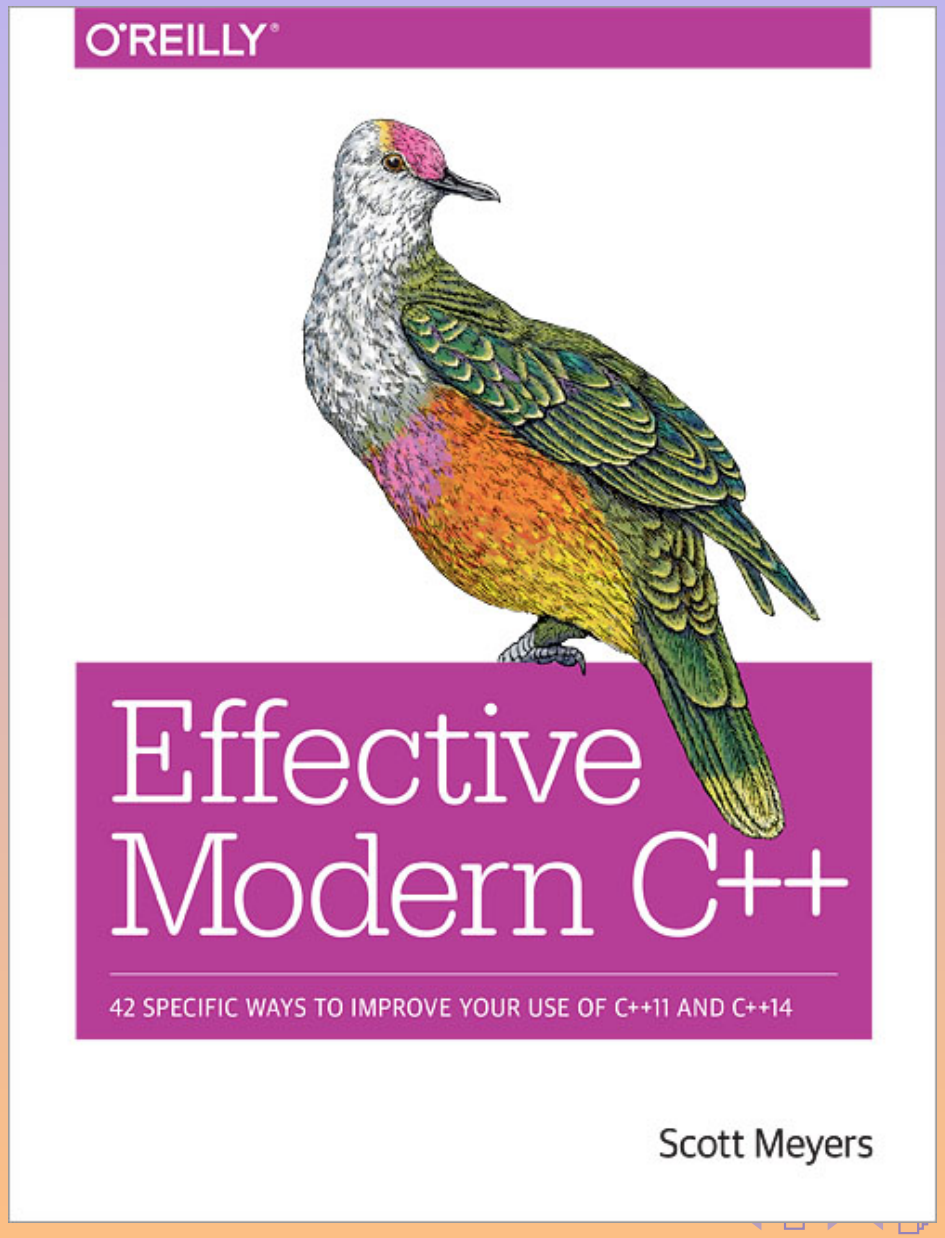
Further Readings



Further Readings



Further Readings



Bibliography

Bibliography I


Abrahams, D. (2001).
Exception-safety in generic components — lessons learned from specifying exception-safety for the C++ standard library.
http://www.boost.org/community/exception_safety.html.


Alexandrescu, A. (2001).
Modern C++ Design: Generic Programming and Design Patterns Applied.
Addison-Wesley.


Carlini, P., Edwards, P., Gregor, D., Kosnik, B., Matani, D., Merrill, J., Mitchell, M., Myers, N., Natter, F., Olsson, S., Rus, S., Singler, J., Tavory, A., and Wakely, J. (2012).
The GNU C++ library manual.
http://gcc.gnu.org/onlinedocs/libstdc++/manual.

Bibliography II


Meyers, S. (1996).
More Effective C++.
Addison-Wesley Professional.


Meyers, S. (2001).
Effective STL: 50 Specific Ways to Improve the Use of the Standard Template Library.
Addison-Wesley Professional Computing Series. Addison Wesley, Boston.


Meyers, S. (2005).
Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition).
Addison-Wesley Professional.

Bibliography III


Meyers, S. (2014).
Effective Modern C++.
O'Reilly.


Stroustrup, B. (2003).
Abstraction, libraries, and efficiency in C++.
Dr. Dobb's Journal China, 1(1).


Sutter, H. and Alexandrescu, A. (2005).
C++ Coding Standards: 101 Rules, Guidelines, And Best Practices.
The C++ In-Depth Series. Addison-Wesley.


Wikipedia (2012a).
Liskov substitution principle — Wikipedia, the free encyclopedia.
[Online; accessed 24-October-2012].

Bibliography IV


Wikipedia (2012b).
Resource acquisition is initialization — Wikipedia, the free encyclopedia.
[Online; accessed 24-October-2012].


Questions?
Design
Preprocessor
Language
Libraries