Better Code
Sean Parent | Principal Scientist
Better Code

- Regular Types
  - Goal: No Incomplete Types
- Algorithms
  - Goal: No Raw Loops
- Data Structures
  - Goal: No Incidental Data Structures
- Runtime Polymorphism
  - Goal: No Raw Pointers
- Concurrency
  - Goal: No Raw Synchronization Primitives

“There are rules!”

– The Big Lebowski
template <class ForwardIterator, class T, class Compare>
ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
    const T& value, Compare comp)
{
    auto n = distance(first, last);

    while (n != 0) {
        auto h = n / 2;
        auto m = next(first, h);

        if (comp(*m, value)) {
            first = next(m);
            n -= h + 1;
        } else { n = h; }
    }

    return first;
}
Undo Typing
Can't Repeat
Cut
Copy
Paste
Paste Special...
Paste and Match Formatting
Clear
Select All
Find
Links...
Start Dictation
Emoji & Symbols
Good Code

Good code is *correct*
Good Code

Good code is *correct*

Consistent; without contradiction
void print_string(const char* s) {
    while (*s != '\0') {
        cout << *s++;
    }
}

int main() {
    print_string(nullptr);
}
Simple Bug

```c
void print_string(const char* s) {
    while (*s != '\0') {
        cout << *s++;
    }
}

int main() {
    print_string(nullptr);
}
```
void print_string(const char* s) {
    while (*s != '\0') {
        cout << *s++;
    }
}

int main() {
    print_string(nullptr);
}
Simple Bug

```c
void print_string(const char* s) {
    while (*s != '\0') {
        cout << *s++;
    }
}

int main() {
    print_string(nullptr); // FORCE CRASH!
}
```
Subtle defects
Subtle defects

Consistency requires context
Subtle defects

Consistency requires context

template<class T> const T& min(const T& a, const T& b);
Returns: The smaller value.
Remarks: Returns the first argument when the arguments are equivalent.
Subtle defects

Consistency requires context

```
template<class T> const T& min(const T& a, const T& b);
Returns: The smaller value.
Remarks: Returns the first argument when the arguments are equivalent.
```

```
template<class T> const T& max(const T& a, const T& b);
Returns: The larger value.
Remarks: Returns the first argument when the arguments are equivalent.
```
Subtle defects
Subtle defects

```cpp
template<typename T>
const T& clamp(const T& a, const T& lo, const T& hi)
{
    return min(max(lo, a), hi);
}
```
Subtle defects

template<typename T>
const T& clamp(const T& a, const T& lo, const T& hi)
{
    return min(max(lo, a), hi);
}

template<typename T, typename Compare>
const T& clamp(const T& a, const T& lo, const T& hi, Compare comp)
{
    return min(max(lo, a, comp), hi, comp);
}
Subtle defects
Subtle defects

```cpp
int main() {
    using pair = pair<int, string>;

    pair a = { 1, "OK" };  

    pair lo = { 1, "FAIL: LO" };  
    pair hi = { 2, "FAIL: HI" };  

    a = clamp(a, lo, hi, [](const auto& a, const auto& b) {
        return a.first < b.first;
    });

    cout << a.second << endl;
}
```
```cpp
int main() {
    using pair = pair<int, string>;

    pair a = { 1, "OK" };  

    pair lo = { 1, "FAIL: LO" };  
    pair hi = { 2, "FAIL: HI" };  

    a = clamp(a, lo, hi, [](const auto& a, const auto& b) {
        return a.first < b.first;
    });  

    cout << a.second << endl;
};

FAIL: LO
```
Subtle defects
Subtle defects

template<typename T>
const T& clamp(const T& a, const T& lo, const T& hi)
{
    return min(max(a, lo), hi);
}
Subtle defects

```cpp
template<typename T>
const T& clamp(const T& a, const T& lo, const T& hi)
{
    return min(max(a, lo), hi);
}

template<typename T, typename Compare>
const T& clamp(const T& a, const T& lo, const T& hi, Compare comp)
{
    return min(max(a, lo, comp), hi, comp);
}
```
Subtle defects
Subtle defects

\[\text{template} <\text{class } T> \text{ const } T & \text{ min}(\text{const } T & a, \text{ const } T & b);\]
Returns: The smaller value.
Remarks: Returns the first argument when the arguments are equivalent.

\[\text{template} <\text{class } T> \text{ const } T & \text{ max}(\text{const } T & a, \text{ const } T & b);\]
Returns: The larger value.
Remarks: Returns the second argument when the arguments are equivalent.
Subtle defects

```cpp
//template<class T> const T& min(const T& a, const T& b);
Returns: The smaller value.
Remarks: Returns the first argument when the arguments are equivalent.

//template<class T> const T& max(const T& a, const T& b);
Returns: The larger value.
Remarks: Returns the second argument when the arguments are equivalent.

//template <class T> const T& max(const T& a, const T& b, const T& c);
Returns: The larger value.
Remarks: ???
```
Rules are Contentious
Rules are Contentious

“Names should not be associated with semantics because everybody has their own hidden assumptions about what semantics are, and they clash, causing comprehension problems without knowing why. This is why it's valuable to write code to reflect what code is actually doing, rather than what code ‘means’: it’s hard to have conceptual clashes about what code actually does.”

– Craig Silverstein, Google
“There is no spoon.”

– The Matrix
How can nothing be something?
How can nothing be something?

```c
int x;
```
How can nothing be something?

```c
int x;
// indeterminate value
```
How can nothing be something?

```c
int x;
// indeterminate value

int x = 1 / 0;
```
How can nothing be something?

```c
int x;
// indeterminate value

int x = 1 / 0;
// undefined behavior
```
How can nothing be something?

```c
int x;
   // indeterminate value

int x = 1 / 0;
   // undefined behavior

double x = 1.0 / 0.0;
```
How can nothing be something?

```c
int x;
// indeterminate value

int x = 1 / 0;
// undefined behavior

double x = 1.0 / 0.0;
// inf
```
How can nothing be something?

```c
int x;
  // indeterminate value

int x = 1 / 0;
  // undefined behavior

double x = 1.0 / 0.0;
  // inf

double x = 0.0 / 0.0;
```
How can nothing be something?

```c
int x;
// indeterminate value

int x = 1 / 0;
// undefined behavior

double x = 1.0 / 0.0;
// inf

double x = 0.0 / 0.0;
// NaN
```
How can nothing be something?

```c
int x;
    // indeterminate value

int x = 1 / 0;
    // undefined behavior

double x = 1.0 / 0.0;
    // inf

double x = 0.0 / 0.0;
    // NaN

struct empty { };
```
How can nothing be something?

```c
int x;
// indeterminate value

int x = 1 / 0;
// undefined behavior

double x = 1.0 / 0.0;
// inf

double x = 0.0 / 0.0;
// NaN

struct empty {
};
// sizeof(empty) == 1
```
How can nothing be something?
How can nothing be something?

```c
int a[0];
```
How can nothing be something?

```c
int a[0];
// Error
```
How can nothing be something?

```c
int a[0];
// Error
// but common extension
```
How can nothing be something?

```c
int a[0];
// Error
// but common extension
using empty = int[0];
```
How can nothing be something?

```c
int a[0];
// Error
// but common extension
using empty = int[0];
// sizeof(empty) == 0
empty a[2];
```
How can nothing be something?

```c
int a[0];
// Error
// but common extension
using empty = int[0];
// sizeof(empty) == 0
empty a[2];
// &a[0] == &a[1]
```
How can nothing be something?

```c
int a[0];
// Error
// but common extension
using empty = int[0];
// sizeof(empty) == 0
empty a[2];
// &a[0] == &a[1]

void f() { return void(); }
```
How can nothing be something?

```cpp
int a[0];  // Error
// but common extension
using empty = int[0];
// sizeof(empty) == 0
empty a[2];  // &a[0] == &a[1]

void f() { return void(); }  // OK
```
How can nothing be something?

```c
int a[0];
// Error
// but common extension
using empty = int[0];
// sizeof(empty) == 0
empty a[2];
// &a[0] == &a[1]

void f() { return void(); }
// OK

void x = f();
```
How can nothing be something?

```c
int a[0];
// Error
// but common extension
using empty = int[0];
// sizeof(empty) == 0
empty a[2];
// &a[0] == &a[1]

void f() { return void(); }
// OK

void x = f();
// Error
```
How can nothing be something?

```c
int a[0];
// Error
// but common extension
using empty = int[0];
// sizeof(empty) == 0
empty a[2];
// &a[0] == &a[1]

void f() { return void(); }
// OK

void x = f();
// Error
// but void* is a pointer to anything...
```
How can nothing be something?
How can nothing be something?

```c++
std::vector<int> x = { 1, 2, 3 };
try {
    x.insert(x.begin(), 0);
} catch (...) {
    std::cout << x.size() << std::endl;
}
```
How can nothing be something?

```cpp
std::vector<int> x = { 1, 2, 3 };
try {
    x.insert(x.begin(), 0);
} catch (...) {
    std::cout << x.size() << std::endl;
}
// Basic Exception Guarantee:
// Valid but unspecified
```
How can nothing be something?

```cpp
std::vector<int> x = { 1, 2, 3 };
try {
    x.insert(x.begin(), 0);
} catch (...) {
    std::cout << x.size() << std::endl;
}
// Basic Exception Guarantee:
// Valid but unspecified

std::vector<int> y = std::move(x);
```
How can nothing be something?

```cpp
std::vector<int> x = { 1, 2, 3 };
try {
    x.insert(x.begin(), 0);
} catch (...) {
    std::cout << x.size() << std::endl;
}
// Basic Exception Guarantee:
// Valid but unspecified

std::vector<int> y = std::move(x);
// Moved from object, x, is valid but unspecified
```
Good Code
Good Code

Good code is *correct*

Consistent; without contradiction
Good Code

Good code is *correct*

  Consistent; without contradiction

Good code has *meaning*
Good Code

Good code is *correct*
Consistent; without contradiction

Good code has *meaning*
Correspondence to an entity; specified, defined
Categories of nothing
Categories of nothing

Absence of *something*

0, Ø, [p, p), void
Categories of nothing

Absence of *something*
0, Ø, [p, p), void

Absence of *meaning*
NaN, undefined, indeterminate
How can nothing be something?
How can nothing be something?

int x;
How can nothing be something?

```c
int x;
// Partially formed; assign value or destruct
```
How can nothing be something?

```c
int x;
// Partially formed; assign value or destruct
int x = 1 / 0;
```
How can nothing be something?

```c
int x;
// Partially formed; assign value or destruct

int x = 1 / 0;
// undefined behavior; reading from meaningless value
```
How can nothing be something?

```c
int x;
// Partially formed; assign value or destruct

int x = 1 / 0;
// undefined behavior; reading from meaningless value

double x = 1.0 / 0.0;
```
How can nothing be something?

```c
int x;
// Partially formed; assign value or destruct

int x = 1 / 0;
// undefined behavior; reading from meaningless value

double x = 1.0 / 0.0;
// inf; OK, approximation for underflow
```
How can nothing be something?

```c
int x;
// Partially formed; assign value or destruct

int x = 1 / 0;
// undefined behavior; reading from meaningless value

double x = 1.0 / 0.0;
// inf; OK, approximation for underflow

double x = 0.0 / 0.0;
```
How can nothing be something?

```c
int x;
// Partially formed; assign value or destruct

int x = 1 / 0;
// undefined behavior; reading from meaningless value

double x = 1.0 / 0.0;
// inf; OK, approximation for underflow

double x = 0.0 / 0.0;
// NaN; OK, though undefined behavior would also be
```
How can nothing be something?

```c
int x;
    // Partially formed; assign value or destruct

int x = 1 / 0;
    // undefined behavior; reading from meaningless value

double x = 1.0 / 0.0;
    // inf; OK, approximation for underflow

double x = 0.0 / 0.0;
    // NaN; OK, though undefined behavior would also be
```
How can nothing be something?

```c
int x;
// Partially formed; assign value or destruct

int x = 1 / 0;
// undefined behavior; reading from meaningless value

double x = 1.0 / 0.0;
// inf; OK, approximation for underflow

double x = 0.0 / 0.0;
// NaN; OK, though undefined behavior would also be

struct empty : void { };
```
How can nothing be something?

```c
int x;
// Partially formed; assign value or destruct

int x = 1 / 0;
// undefined behavior; reading from meaningless value

double x = 1.0 / 0.0;
// inf; OK, approximation for underflow

double x = 0.0 / 0.0;
// NaN; OK, though undefined behavior would also be

struct empty : void { }
// sizeof(empty) == 0;
```
How can nothing be something?
How can nothing be something?

```c
int a[0];
```
How can nothing be something?

```c
int a[0];
// OK
```
How can nothing be something?

```cpp
int a[0];
// OK
using empty = int[0];
```
How can nothing be something?

```cpp
int a[0];
// OK
using empty = int[0];
// sizeof(empty) == 0
empty a[2];
```
How can nothing be something?

```c
int a[0];
// OK
using empty = int[0];
// sizeof(empty) == 0
empty a[2];
// &a[0] == &a[1]
```
How can nothing be something?

```c
int a[0]; // OK
using empty = int[0]; // sizeof(empty) == 0
empty a[2]; // &a[0] == &a[1]

void f() { return void(); }
```
How can nothing be something?

```c
int a[0];
// OK
using empty = int[0];
// sizeof(empty) == 0
empty a[2];
// &a[0] == &a[1]

void f() { return void(); }
// OK
```
How can nothing be something?

```c
int a[0];
// OK
using empty = int[0];
// sizeof(empty) == 0
empty a[2];
// &a[0] == &a[1]

void f() { return void(); }  // OK
void x = f();
```
How can nothing be something?

```cpp
int a[0];
// OK
using empty = int[0];
// sizeof(empty) == 0
empty a[2];
// &a[0] == &a[1]

void f() { return void(); }  // OK

void x = f();
// OK
// void* is OK
```
How can nothing be something?
How can nothing be something?

```cpp
std::vector<int> x = {1, 2, 3};
try {
    x.insert(x.begin(), 0);
} catch (...) {
    std::cout << x.size() << std::endl;
}
```
How can nothing be something?

```cpp
std::vector<int> x = { 1, 2, 3 };
try {
    x.insert(x.begin(), 0);
} catch (...) {
    std::cout << x.size() << std::endl;
}
// Basic Exception Guarantee:
// Partially formed object. Reading is undefined behavior
```
How can nothing be something?

```cpp
std::vector<int> x = { 1, 2, 3 };
try {
    x.insert(x.begin(), 0);
} catch (...) {
    std::cout << x.size() << std::endl;
}
// Basic Exception Guarantee:
// Partially formed object. Reading is undefined behavior

std::vector<int> y = std::move(x);
```
How can nothing be something?

```cpp
std::vector<int> x = { 1, 2, 3 }; 
try {
    x.insert(x.begin(), 0);
} catch (...) {
    std::cout << x.size() << std::endl;
}
// Basic Exception Guarantee:
// Partially formed object. Reading is undefined behavior

std::vector<int> y = std::move(x);
// Moved from object, x, is partially formed
```
“What’s in the box?”

- Seven
The Permutation Paradox
The Permutation Paradox
The Permutation Paradox
The Permutation Paradox

\[ \text{nothing} \Rightarrow \text{unsafe} \]
The Permutation Paradox

nothing $\Rightarrow$ unsafe

something $\Rightarrow$ inefficient
The Permutation Paradox
The Permutation Paradox

“There is a duality between transformations and the corresponding actions: An action is definable in terms of a transformation and vice versa:”
The Permutation Paradox

“There is a duality between transformations and the corresponding actions: An action is definable in terms of a transformation and vice versa:

```c
void a(T& x) { x = f(x); } // action from transformation

and

T f(T x) { a(x); return x; } // transformation from action
```
The Permutation Paradox

“There is a duality between transformations and the corresponding actions: An action is definable in terms of a transformation and vice versa:

```c
void a(T& x) { x = f(x); } // action from transformation
```

and

```c
T f(T x) { a(x); return x; } // transformation from action
```

Despite this duality, independent implementations are sometimes more efficient, in which case both action and transformation need to be provided.”

– *Elements of Programming* (section 2.5)
“It’s not that I’m lazy, it’s that I just don’t care.”

– Office Space
Good Code
Good Code

Good code is *correct*
   Consistent; without contradiction

Good code has *meaning*
   Correspondence to an entity; specified, defined
Good Code

Good code is *correct*
   Consistent; without contradiction

Good code has *meaning*
   Correspondence to an entity; specified, defined

Good code is *efficient*
Good Code

Good code is *correct*

Consistent; without contradiction

Good code has *meaning*

Correspondence to an entity; specified, defined

Good code is *efficient*

Maximum effect with minimum resources
Efficiency
Efficiency

Choice of data structures and algorithms

Choice of what to optimize for
Efficiency

A B C D E F G
Efficiency

G F E D C B A
Efficiency
template <class ForwardIterator>
void reverse(ForwardIterator f, ForwardIterator l) {
    auto n = distance(f, l);

    if (n == 0 || n == 1) return;

    auto m = next(f, n / 2);

distance(f, m);
    reverse(f, m);
    reverse(m, l);
    rotate(f, m, l);
}
template <class ForwardIterator>
void reverse(ForwardIterator f, ForwardIterator l) {
    auto n = distance(f, l);

    if (n == 0 || n == 1) return;

    auto m = next(f, n / 2);

    reverse(f, m);
    reverse(m, l);
    rotate(f, m, l);
}

O(n log n)
Efficiency

G  F  E  D  C  B  A
Simple Word Model

Hello World!
Simple Word Model

- Current Document
- Selection
  - Provides a range; an empty range denotes a location
More Complex Word Model

- Need to be able to set the selection in “constant” time
  - This would imply a vector data structure
- Also need constant time insert and erase
  - This would imply a list data structure

- Solution: a more complex data structure such as a rope
“I don’t smoke, I don’t drink... I recycle...”

- 50/50
Good Code

Good code is *correct*
  Consistent; without contradiction

Good code has *meaning*
  Correspondence to an entity; specified, defined

Good code is *efficient*
  Maximum effect with minimum resources

Good code is *reusable*
  Applicable to multiple problems; general in purpose
Reusable
Reusable

Concrete but of general use, i.e. numeric algorithms, utf conversions, ...

Generic when algorithm is useful with different models
Sometimes faster to convert one model to another

Runtime dispatched when types not known at compile time
Reusable
Reusable

Minimize client dependencies and intrusive requirements

Separate data structures from algorithms
Reusable
template <class T, class InputIterator, class OutputIterator>
OutputIterator copy_utf(InputIterator first, InputIterator last, OutputIterator result);

const char str[] = u8"Hello World!";
vector<uint16_t> out;
copy_utf<uint16_t>(begin(str), end(str), back_inserter(out));
“You mean we're in the future.”
– Back to the Future Part II
Why Status Quo Will Fail
Why Status Quo Will Fail

“I’ve assigned this problem [binary search] in courses at Bell Labs and IBM. Professional programmers had a couple of hours to convert the description into a programming language of their choice; a high-level pseudo code was fine... Ninety percent of the programmers found bugs in their programs (and I wasn’t always convinced of the correctness of the code in which no bugs were found).”

– Jon Bentley, Programming Pearls, 1986
int* lower_bound(int* first, int* last, int value) {
    while (first != last) {
        int* middle = first + (last - first) / 2;
        if (*middle < value) first = middle + 1;
        else last = middle;
    }
    return first;
}
Signs of Hope

Elements of Programming

Concepts aren’t dead yet in C++
Increased interest in new languages and formalisms
Renewed interest in Communication Sequential Processes
Renewed interest in Functional Programming ideas
Rise of Reactive Programming & Functional Reactive Programming
Work Continues
Work Continues

Generating Reactive Programs for Graphical User Interfaces from Multi-way Dataflow Constraint Systems, GPCE 2015, Gabriel Foust, Jaakko Järvi, Sean Parent

One Way To Select Many, ECOOP 2016, Jaakko Järvi, Sean Parent

https://github.com/stlab
Write Better Code