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 Inheritance
- Concurrency
  - Goal: No Raw Synchronization Primitives

http://sean-parent.stlab.cc/papers-and-presentations
The Knowledge
“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;
}
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

```cpp
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); // FORCE CRASH!
}
Subtle defects
Subtle defects

Consistency requires context
Subtle defects

Consistency requires context

```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.
Subtle defects

Consistency requires context

```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.
```

```cpp
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

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
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;
}
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;  
}

FAIL: LO
```
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

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

```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.
```

```cpp
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.
```
Subtle defects

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

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

\texttt{template <\texttt{class } T> \ const \ T& \ \texttt{max}(\texttt{const } T& \ a, \ \texttt{const } T& \ b, \ \texttt{const } T& \ c);} \newline
Returns: The larger value. \newline
Remarks: ???

© 2016 Adobe Systems Incorporated. All Rights Reserved.
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?

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?

```
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?

```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
```
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?

```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:
// 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 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?

```c
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?

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

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

```cpp
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
```
“That makes you wonder. Take chicken, for example.”

– Matrix
Specification
Specification

- `clone_ptr<T>` is like `std::unique_ptr<T>` but with two additional operations, copy and assignment that copy the object pointed to.

- Example implementation of new operations:
  ```cpp
  clone_ptr(const clone_ptr& x) : _ptr(new T(*x)) {}
  clone_ptr& operator=(const clone_ptr& x) { return *this = clone_ptr(x); }
  ```
Specification

- `clone_ptr<T>` is like `std::unique_ptr<T>` but with two additional operations, copy and assignment that copy the object pointed to.

- Example implementation of new operations:
  ```cpp
  clone_ptr(const clone_ptr& x) : _ptr(new T(*x)) { }
  clone_ptr& operator=(const clone_ptr& x) { return *this = clone_ptr(x); }
  ```

- copy-assignment written in terms of copy and move-assignment
What is copy?

- *Copying* an object creates a new object which is equal-to and logically disjoint from the original.

\[
T \ a = b; \ \Rightarrow \ a == b;
\]
\[
T \ a = b; \ \text{modify}(b); \ \Rightarrow \ a != b;
\]
“copy” of clone_ptr

\[
\text{clone_ptr<T> } a = b; \Rightarrow a \neq b;
\]

- “Copying” a clone pointer creates an object that is not equal to the original
- Contradiction

- Defining a copy-constructor that doesn't copy is dangerous
  - The compiler may elide copies
  - Programmers will assume they are substitutable
 Specification: Amendment 1

- Two `clone_ptr`s are considered equal if the value they point to is equal. Because we don’t want to require that the pointed to types are equal, `operator==()` and `operator!=()` are not implemented. i.e.:

  ```cpp
  clone_ptr<T> a = b; ⇒ a == b;
  ```

  However, `==` is not implemented.
What is a pointer?

- A *pointer* is an object that refers to another object via a dereference operation. Two pointers are equal if they refer to the same instance of an object.

\[ a == b; \Rightarrow \&*a == \&*b; \]
“equality” of clone_ptr

clone_ptr<T> a = b; ⇒ a == b;

- Because clone_ptr is a pointer this would imply:

  assert(&*a == &*b);

- But that is false - contradiction.
Specification: Amendment 2

- Because `clone_ptr<>` is not a pointer it is to be renamed `indirect<>`. 
What is a const?

- `const` is a type qualifier. An object accessed through a `const` reference may not be modified.

  ```
  const T a = b; read(a); → a == b;
  modify(a); is not allowed
  ```
"const" of indirect

const indirect<T> a = b; read(a); ⊨ a == b;

- Because const does not propagate (from unique_ptr):

  void read(const indirect<T>& x) {
    modify(*x);
  }

- Contradiction!
Specification: Amendment 3

• Because copy of remote part implies const propagation, get(), operator*() and operator->() must be overloaded:

```cpp
T* get();
const T* get() const;

T& operator*();
const T& operator*() const;

T* operator->();
const T* operator->() const;
```
Alternative Specification:
Alternative Specification:

- `clone_ptr<T>` is like `std::unique_ptr<T>` but with one additional operation, `clone()` that works by copying the object pointed to.

- Example implementation of clone operation:

  ```cpp
  clone_ptr clone() const { return make_clone<T>(**this); }
  ```
“What’s in the box?”
– Seven
The Permutation Paradox
The Permutation Paradox
The Permutation Paradox
The Permutation Paradox

nothing $\Rightarrow$ 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

```c
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)
This section borrowed from Andrei Alexandrescu
Purity

- Text book purity requires tail-recursion

This section borrowed from Andrei Alexandrescu
Purity

- Text book purity requires tail-recursion

```cpp
// If C++ had tail recursion

int helper(int n, int result) {
    return n <= 1 ? result : helper(n - 1, n * result);
}

int factorial(int n) {
    return helper(n, 1);
}
```

This section borrowed from Andrei Alexandrescu
Purity

\[ n! = \prod_{k=1}^{n} k. \]
Purity

- In math, factorial is defined as iteration

\[ n! = \prod_{k=1}^{n} k. \]
Purity

- In math, factorial is defined as iteration

\[ n! = \prod_{k=1}^{n} k. \]

```java
int factorial(int n) {
    int result = 1;
    for (int i = 2; i <= n; ++i) {
        result *= i;
    }
    return result;
}
```
Purity
Purity

- Pure functions always return the same result for the same arguments
- No reading and writing of global variables (global constants are okay)
- No calling of impure functions
- Local transient state, inside the function, may be modified
- Anything reachable from the arguments may be modified
Purity

- Pure functions always return the same result for the same arguments
- No reading and writing of global variables (global constants are okay)
- No calling of impure functions
- Local transient state, inside the function, may be modified
- Anything reachable from the arguments may be modified
  - Action to Function Transformation
Purity

- Pure functions always return the same result for the same arguments
- No reading and writing of global variables (global constants are okay)
- No calling of impure functions
- Local transient state, inside the function, may be modified
- Anything reachable from the arguments may be modified
  - Action to Function Transformation
  - `std::sort` is pure
“It’s not that I’m lazy, it’s that I just don’t care.”

– Office Space
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
Efficiency

G F E D C B A
template <classForwardIterator, class N>
auto reverse_n(ForwardIterator f, N n) {
    if (n < 2) return next(f, n);
    auto h = n / 2;
    auto m1 = reverse_n(f, h);
    auto m2 = next(m1, n % 2);
    auto l = reverse_n(m2, h);
    swap_ranges(f, m1, m2);
    return l;
}

template <class ForwardIterator>
void reverse(ForwardIterator f, ForwardIterator l) {
    reverse_n(f, distance(f, l));
}

\(O(n \log n)\)  

*Elements of Programming*, 10.3
Efficiency
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
What is an efficient type?
What is an efficient type?

- A type is *complete* if the set of provided basis operations allow us to construct and operate on any valid, representable value
- A type is *efficient* if the set of basis operations allow for any valid operation to be performed in the most efficient way possible for the chosen representation
What is an efficient type?

- A type is *complete* if the set of provided basis operations allow us to construct and operate on any valid, representable value.
- A type is *efficient* if the set of basis operations allow for any valid operation to be performed in the most efficient way possible for the chosen representation.

- By simply making all data members public, you provide, by definition, an efficient basis.
- However, you may fail to protect the invariants of the type, making the approach unsafe.

- `std::move` is both unsafe and inefficient.
“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
Why Status Quo Will Fail

```c
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

http://sean-parent.stlab.cc/papers-and-presentations
https://github.com/stlab
Write Better Code