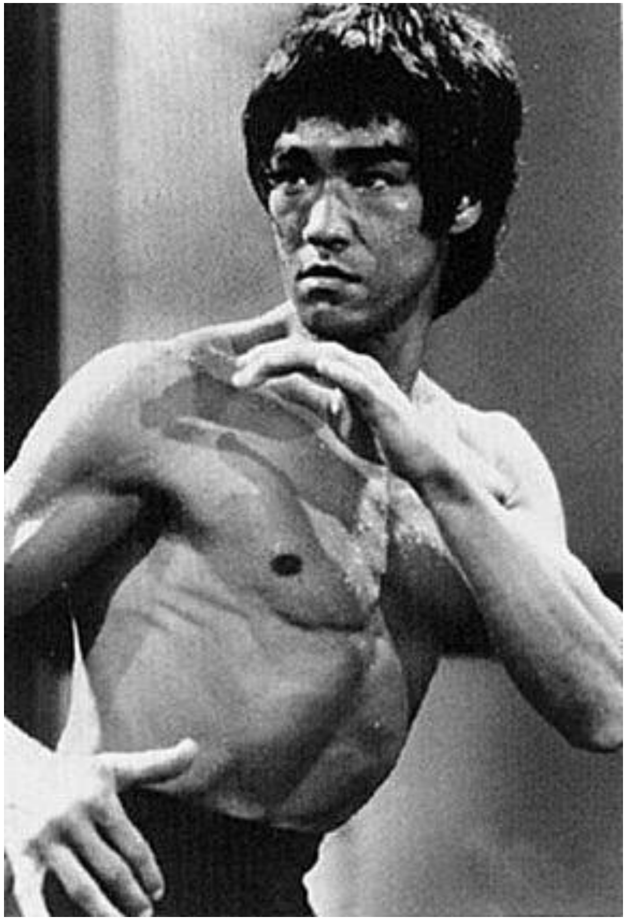
Intro. to Functional Programming in C++

C++Now 2014

David Sankel

Why should you pay attention to this talk?



the style of no style

Brief Functional Programming History

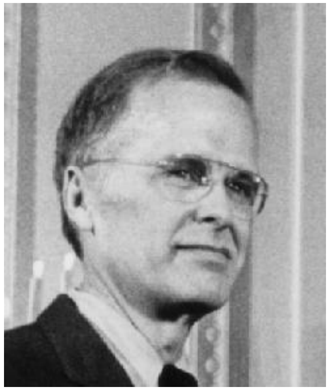
Lambda Calculus. Alonzo Church 1930's
- Mathematical abstraction
- Attempt at foundation of mathematics

The Next 700 Programming Languages. Peter Landin 1966.
- Theoretical Programming Language
- Sugaring of lambda calculus

"Can Programming Be Liberated From the von Neumann Style?". John Backus 1977.
- Algebra of programs
- Precursor of purity
- Popularized research into functional programming

Alonzo Church

Peter Landin



John Backus

What is Functional Programming?

Math applied to programming.

- Languages
- Semantics
- Style

Functional Programming Effects

- Simplification of Complex domains.
- Strong insights from mathematical study.
- Inherent Composability.
- Power from generalities.

Purity

purity: free from what vitiates, weakens, or pollutes

```
int f(int);
```

Purity

purity: free from what vitiates, weakens, or pollutes

int f(int);

- referentially transparent. For every x, f(x) returns the same value.
- No observable side-effects.

Pure Functions

- Directly map with mathematical functions (+)
- Easy to reason about.

- Also, pure values.

Pure Lists

```
template< typename T >
struct list {
private:
//...
};

// Constructors
template <typename T> list<T> empty();
template <typename T> list<T> addToFront( T t, list<T> );

// Access
template <typename T>
bool isEmpty(list<T>);

template <typename T>
T front(list<T>);

template <typename T>
list<T> rest(list<T>);
```

Map

```
template <typename T, typename U>
list<U> map(function<U(T)> f,
            list<T> list) {
  if (isEmpty(list))
    return empty<U>();
  else
    return addToFront(
        f(front(list)),
        map(f, rest(list)));
}
```

Map

```
template <typename F, typename T>
list<
    typename std::result_of<F(T)>::type>
map(F f, list<T> list) {
  typedef typename std::result_of<
      F(T)>::type U;
  if (isEmpty(list))
    return empty<U>();
  else
    return addToFront(
        f(front(list)),
        map(f, rest(list)));
}
```

Functions aren't special

```
const int i = 6;

const function<int(int)> f = [](int i) {
  return i + 1;
};

const function<int(int)> g = foo(i);

int j = bar(f);
```

Higher Order Functions

- A function which has either a function as an argument or a function as a result type.

Fold

```
template <typename T, typename U>
U fold(function<U(T, U)> f, U u,
       list<T> list) {
  /*?*/
}
```

Fold

```
template <typename T, typename U>
U fold(function<U(T, U)> f, U u,
       list<T> list) {
  if (isEmpty(list))
    /*?*/;
  else
    /*?*/;
}
```

Fold

```
template <typename T, typename U>
U fold(function<U(T, U)> f, U u,
       list<T> list) {
  if (isEmpty(list))
    return u;
  else
    /*?*/;
}
```

Fold

```
template <typename T, typename U>
U fold(function<U(T, U)> f, U u,
       list<T> list) {
  if (isEmpty(list))
    return u;
  else
    return f(/*?*/,/*?*/);
}
```

Fold

```
template <typename T, typename U>
U fold(function<U(T, U)> f, U u,
       list<T> list) {
  if (isEmpty(list))
    return u;
  else
    return f(front(list), /*?*/);
}
```

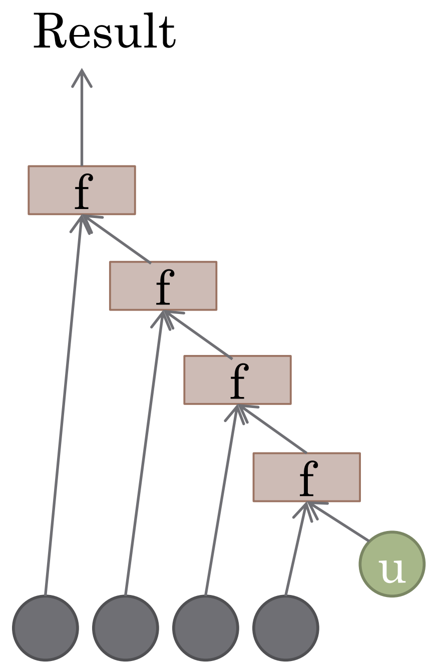
Fold

```
template <typename T, typename U>
U fold(function<U(T, U)> f, U u,
       list<T> list) {
  if (isEmpty(list))
    return u;
  else
    return f(front(list),
             fold(f, u,rest(list)));
}
```

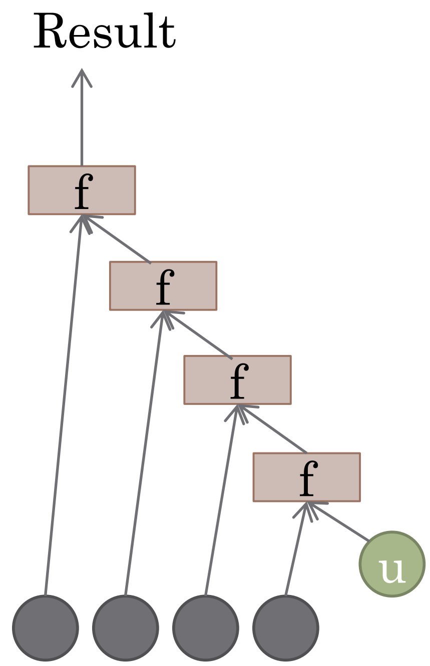
Fold

```
template <typename F, typename U>
U fold(F f, U u, list<T> list) {
  if (isEmpty(list))
    return u;
  else
    return f(front(list),
             fold(f, u, rest(list)));
}
```

Fold, what is it?

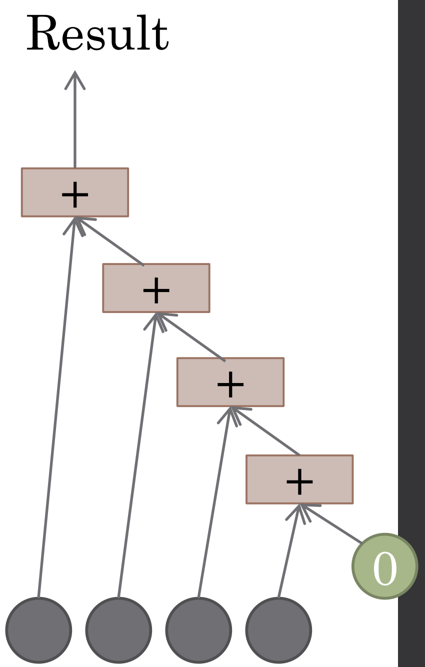


How is it useful?



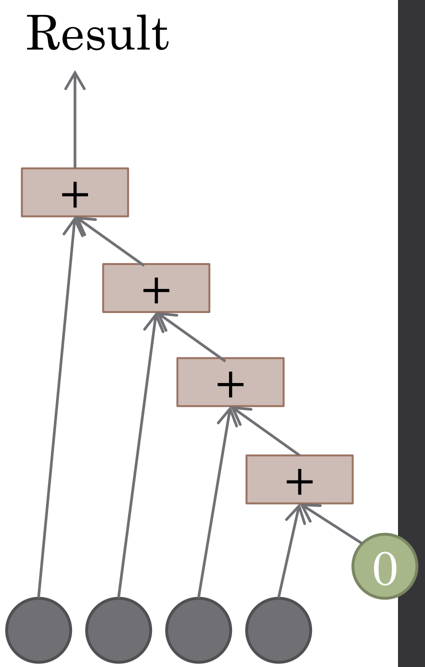
How is it useful?

```
int sum(list<int> intList) {
  return fold([](int i, int j) {
                   return i + j;
                 },
                 0, intList);
}
```



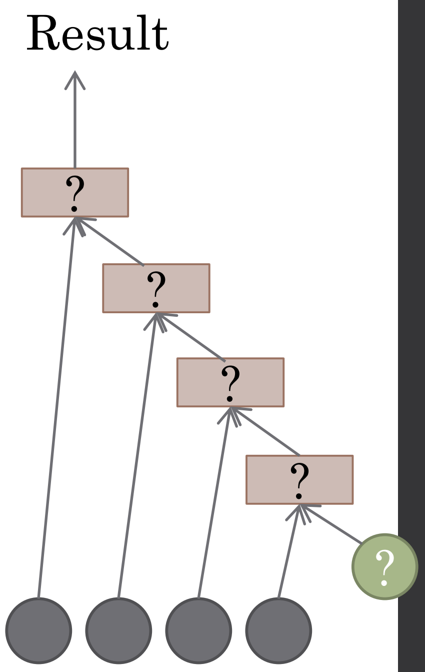
How is it useful?

```
template <typename T>
T sum(list<T> summableList) {
  return fold([](T i,
                 T j) { return i + j; },
              T(0), summableList);
}
```



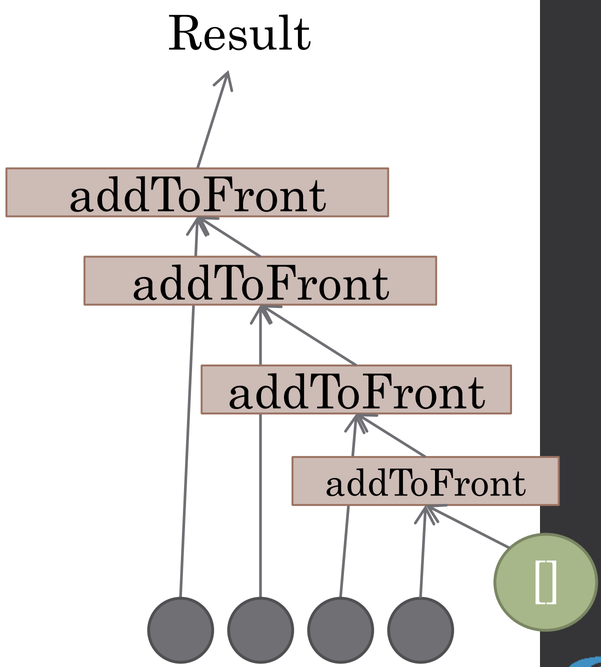
What is this thing?

```
template <typename T>
list<T> thing(list<T> value) {
  return fold(addToFront<T>, empty<T>(),
              value);
}
```



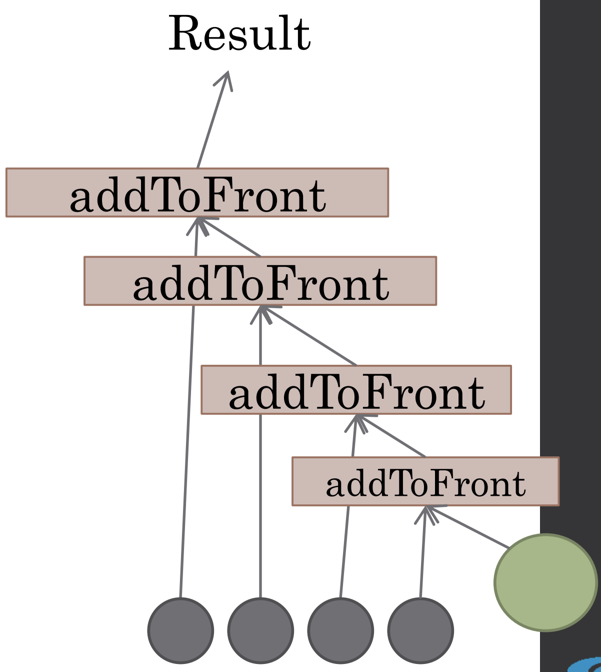
What is this thing?

```
template <typename T>
list<T> identity(list<T> value) {
  return fold(addToFront<T>, empty<T>(),
              value);
}
```



Append

```
template <typename T>
T append(list<T> value,
           list<T> anotherValue) {
  return fold(addToFront<T>,
              anotherValue, value);
}
```



Map Revisited

```
template <typename F, typename T>
list<
    typename std::result_of<F(T)>::type>
map(F f, list<T> l) {
  typedef typename std::result_of<
      F(T)>::type U;
  return fold([f](T t, list<U> l) {
                return addToFront(f(t),
                                  l);
              },
              empty<U>(), l);
}
```

Map Revisited

```
template <typename F, typename T>
list<
    typename std::result_of<F(T)>::type>
map(F f, list<T> l) {
  typedef typename std::result_of<
      F(T)>::type U;
  return fold([f](T t, list<U> l) {
                  return addToFront(f(t),
                                    l);
              },
              empty<U>(), l);
}
```

map = λ(f,l). fold(λ(t,l). addToFront(f t, l), empty, l)

Algebraic Data Types

- Mathematical fundamentals of base types.
- Two types, 1 and 0
- Two ops, $\oplus$ and $\otimes$, to compose them

Product

Given types ‘a’ and ‘b’, the product of ‘a’ and ‘b’ (a $\otimes$ b) is a type whose values have an ‘a’ and a ‘b’.

Product

Given types 'a' and 'b', the product of 'a' and 'b' (a $\otimes$ b) is a type whose values have an 'a' and a 'b'.

Several ways to implement in C++.

```
pair<A,B>

tuple<A,B>

struct AB {

  A a;

  B b;

};
```

Product

Is this an implementation of A $\otimes$ B?

```
struct AB {
  unique_ptr<A> a;
  unique_ptr<B> b;
};
```

0

0 is the type with no values.

How would we implement it?

How would we implement 0?

```
struct Zero {
  Zero() = delete;
};
```

What can we say about this pure function?

```
Zero f(int);
```

What can we say about this pure function?

```
Zero f(int i) {
  return f( i + 1 );
}
```

What can we say about this pure function?

```
Zero f(int);
```

$$\bot$$

Bottom

$\perp$

- Every type has this value.
- Values of type 'unsigned': ⊥, 0, 1, …

1

- 1 (Unit) is a type with one value.
- How would we implement this?

1

- 1 (Unit) is a type with one value.
- How would we implement this?

```
struct Unit{};
```

$\oplus$ (Sum/Or)

- A $\oplus$ B is a type whose values are either a value of type 'A' or a value of type 'B'.
- How would we implement this?

$$A \oplus B$$

```
union AOrB {
  A a;
  B b;
};
```

$$A \oplus B$$

```
struct AOrB {
  bool hasA;
  union {
    A a;
    B b;
  } contents;
};
```

$$A \oplus B$$

```
struct AOrB {
  bool hasA;
  A a;
  B b;
};
```

$A \oplus B$

```
struct AOrB {
  bool hasA;
  A a;
  B b;
};
```

Nope!

$A \oplus B$

```
struct AOrB {
  bool hasA();
  // Return the embedded 'A' object. The
  // behavior is undefined unless 'hasA()'.
  A getA();
  // Return the embedded 'B' object. The
  // behavior is undefined unless 'hasA()'.
  B getB();
  // post: 'hasA()'.
  void setA(A);
  // post: '!hasA()'.
  void setB(B);
private:
  //...
};
```

$A \oplus B$

```
struct AOrB {
protected:
  virtual void dummy(){}
};

struct AOrBWithA : AOrB {
  A a;
};

struct AOrBWithB : AOrB {
  B b;
};
```

$$A \oplus B$$

```
boost::variant<A,B>
```

Back to the pure list

0, 1, $\oplus$, $\otimes$, A, List<A>

empty:

addToFront:

Back to the pure list

0, 1, $\oplus$, $\otimes$, A, List<A>

empty: 1

addToFront:

Back to the pure list

0, 1, $\oplus$, $\otimes$, A, List<A>

empty: 1

addToFront: A $\otimes$ List<A>

Back to the pure list

$0, 1, \oplus, \otimes, A, \text{List<A>}$

$\text{List<A>} = 1 \oplus (A \otimes \text{List<A>})$

Magic Function 1

U = 1

```
template< typename T >
T magicUnit( U u, T t ) {
  return t;
}
```

Magic Function $\otimes$

std::pair<A,B> = A $\otimes$ B

```
template< typename T >
T magicProduct(
    std::pair<A,B> pair,
    function<T (A,B)> f ) {
  return f( pair.first, pair.second );
}
```

Magic Function ⊕

$AOrB = A \oplus B$

```
template< typename T >
T magicSum(
    AOrB aOrB,
    function<T (A)> fa,
    function<T (B)> fb) {
  return aOrB.isA()
    ? fa( aOrB.getA() )
    : fb( aOrB.getB() );
}
```

Fold, a magic function

$\text{List<A>} = 1 \oplus (\text{A} \otimes \text{List<A>})$

```
template <typename T, typename U>
U fold(function<U(T, U)> f, U u,
       list<T> list) {
  if (isEmpty(list))
    return u;
  else
    return f(front(list),
             fold(f, u,rest(list)));
}
```

Functional Approach

- Math → Implementation
- Implementation → Math

Functions

$$A \rightarrow B$$

- Functions can be data structures too.

What is this?

$\text{Foo} = 1 \rightarrow (\text{Int} \otimes \text{Foo})$

What is this?

$\text{Foo} = 1 \rightarrow (\text{Int} \otimes \text{Foo})$

```
typedef std::function<
  std::pair<int,Foo> (Unit) > Foo;
```

What is this?

$\mathrm{Foo} = 1 \rightarrow (\mathrm{Int} \otimes \mathrm{Foo})$

```
typedef std::function<
  std::pair<int,Foo> (Unit) > Foo;
```

What is this?

$\text{Foo} = 1 \rightarrow (\text{Int} \otimes \text{Foo})$

```
struct Foo {
  std::function< std::pair<int,Foo> (Unit) >
    function;
};
```

What is this?

$\text{Foo} = 1 \to (\text{Int} \otimes \text{Foo})$

```
struct Foo

  : std::function< std::pair<int,Foo> (Unit) >

{

  template< typename F >

  Foo( F && f )

    : std::function< std::pair<int,Foo> (Unit) >(std::forward<F>(f))

  {}

};
```

What is this?

$\text{Foo} = 1 \rightarrow (\text{Int} \otimes \text{Foo})$

```
struct Foo
  : std::function< std::pair<int,Foo> () >
{
  template< typename F >
  Foo( F && f )
    : std::function< std::pair<int,Foo> ()
>( std::forward<F>( f ) )
  {}
};
```

Remove unit argument.

What is this?

$\mathrm{Foo} = 1 \rightarrow (\mathrm{Int} \otimes \mathrm{Foo})$

```
Foo foo = [](){ return std::make_pair(1, foo); };
```

What is this?

$\mathrm{Foo} = 1 \rightarrow (\mathrm{Int} \otimes \mathrm{Foo})$

```
Foo foo = [](){ return std::make_pair(1, foo); };
```

`foo().first` $\rightsquigarrow 1$

`foo().second().first` $\rightsquigarrow 1$

`foo().second().second().first` $\rightsquigarrow 1$

IntStream

$\text{IntStream} = 1 \to (\text{Int} \otimes \text{Foo})$

```
IntStream always1 = [](){ return std::make_pair(1, always1); };
```

`always1().first` $\rightsquigarrow 1$

`always1().second().first` $\rightsquigarrow 1$

`always1().second().second().first` $\rightsquigarrow 1$

IntStream

$\text{IntStream} = 1 \to (\text{Int} \otimes \text{Foo})$

```
std::pair<int,IntStream> naturalsFrom( int i ) {
  return std::make_pair(
    i,
    std::bind( naturalsFrom, i+1 ) );
}

IntStream naturals = [](){ return naturalsFrom( 0 ); };
```

`naturals().first` $\rightsquigarrow 0$

`naturals().second().first` $\rightsquigarrow 1$

`naturals().second().second().first` $\rightsquigarrow 2$

Other Strange Things?

$\text{Foo}<A> = \text{Int} \to (A \otimes \text{Foo}<A>)$

Other Strange Things?

$\text{Foo}<\text{A}> = \text{Int} \to (\text{A} \otimes \text{Foo}<\text{A}>)$

Practical Application

$$\text{Source}<A> = 1 \rightarrow (1 \oplus (A \otimes \text{Source}<A>))$$

$$\text{Transformer}<A,B> = (1 \oplus (A \otimes \text{Source}<A>)) \rightarrow$$

$$(1 \oplus (B \otimes \text{Source}<B>))$$

$$\text{Sink}<A> = (1 \oplus (A \otimes \text{Source}<A>)) \rightarrow \text{IO}$$

Practical Application

$\text{Source<A>} = 1 \rightarrow (1 \oplus (A \otimes \text{Source<A>}))$

$\text{Transformer<A,B>} = (1 \oplus (A \otimes \text{Source<A>})) \rightarrow$

$\qquad (1 \oplus (B \otimes \text{Source<B>}))$

$\text{Sink<A>} = (1 \oplus (A \otimes \text{Source<A>})) \rightarrow \text{IO}$

```
template< typename A, typename B >
Source<std::pair<A,B>> zipSources( Source<A>, Source<B> );

template< typename A, typename B, typename C >
Transformer<A,C> mergeTrans( Transformer<A,B>, Transformer<B,C> );

template< typename A, typename B >
Source<B> transSrc( Source<A>, Transformer<A,B> );

template< typename A, typename B >
Sink<A> transSink( Transformer<A,B>, Sink<B> );
```

Denotative Design

- Discover the math
- Derive the implementation


NAME
BEDROOM #
LOUNGE
DINING
CONTENTS
NAME
BEDROOM #
LOUNGE
DINING
CONTENTS
KITCHEN
BATHROOM
TRANSIT-PAC

Functional Programming

Further Learning:

- Denotational Semantics: A Methodology for Language Development. David Schmidt
- The Intellectual Ascent to Agda. C++Now 2013 Talk.
- The Journal of Functional Programming. Cambridge University Press.
- Modern Functional Programming in C++. BoostCon 2010 Paper.
- The Haskell Community. Haskell.org
- Category Theory for Computing Science. Barr & Wells

Digression...

```
template< typename T >
class List {
//...
public:
  // This can break invariants of this class. Caller’s
  // responsibility to restore them.
  void unsafeSetLink( ListIterator, ListIterator );
};
```

Digression...

```
template< typename T >
class List {
//...
public:
  // This can break inva[illegible]ants of thi[illegible]class.[illegible]ler's
  // responsibility to re[illegible]re them
  void[illegible]SetLink( List[illegible]erator, [illegible]tIterato[illegible]
};
```

No!

Digression...

```
template< typename T >
class ListFragments {
//...
public:
  void setLink( ListIterator, ListIterator );
};

template< typename T >
class List {
//...
public:
  // Set this list to the empty list. Return a
  // ‘ListFragments’ object consisting of a single list
  // corresponding to the previous value of this list.
  ListFragments extractFragments( );

  // Set this list to the single list in the specified
  // ‘listFragments’ structure. Behavior undefined unless
  // ‘listFragments’ consists of a single list.
  void setToFragments( ListFragments listFragments );
};
```

Digression…

```
template< typename T >
class ListFragments {
//...
public:
  void setLink( ListIterator, ListIterator );
};

template< typename T >
class List {
//...
public:
  // Set this list to the empty list. Return a
  // ‘ListFragments’ object consisting of a single list
  // corresponding to the previous value of this list.
  ListFragments extractFragments( );

  // Set this list to the single list in the specified
  // ‘listFragments’ structure. Behavior undefined unless
  // ‘listFragments’ consists of a single list.
  void setToFragments( ListFragments listFragments );
};
```

Seun Parent approved!

Functional Programming

Further Learning:

- Denotational Semantics: A Methodology for Language Development. David Schmidt
- The Intellectual Ascent to Agda. C++Now 2013 Talk.
- The Journal of Functional Programming. Cambridge University Press.
- Modern Functional Programming in C++. BoostCon 2010 Paper.
- The Haskell Community. Haskell.org
- Category Theory for Computing Science. Barr & Wells