Better Code: Data Structures
Sean Parent | Principal Scientist
Better Code

- Regular Types
  - Goal: Implement Complete and Efficient 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
- ...
Better Code

- Regular Types
  - Goal: Implement Complete and Efficient 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
- ...
Goal: No incidental data structures
What is an *incidental* data structure?
What is a data structure?
What is a data structure?

Definition: A data structure is a format for organizing and storing data.
What is a structure?
What is a structure?

Definition: A structure on a set consists of additional entities that, in some manner, relate to the set, endowing the collection with meaning or significance.
[−8..7]

0100

0011
0100  0011

4 > 3
hash( ) != hash( )
hash() != hash()
Memory Space

0100 0011
Memory Space

0100

0100

0011
Memory Space
Memory Space

0100

0011
Memory Space
Memory Space
Memory Space
Memory Space

0011 + 0100 = 0111

Memory Space
Whole-Part Relationships and Composite Objects

Elements of Programming, Chapter 12
Whole-Part Relationships and Composite Objects

- Connected

Elements of Programming, Chapter 12
Whole-Part Relationships and Composite Objects

- Connected
- Noncircular
Whole-Part Relationships and Composite Objects

- Connected
- Noncircular

Elements of Programming, Chapter 12
Whole-Part Relationships and Composite Objects

- Connected
- Noncircular
- Logically Disjoint

Elements of Programming, Chapter 12
Whole-Part Relationships and Composite Objects

- Connected
- Noncircular
- Logically Disjoint
Whole-Part Relationships and Composite Objects

- Connected
- Noncircular
- Logically Disjoint
- Owning

Elements of Programming, Chapter 12
Whole-Part Relationships and Composite Objects

- Connected
- Noncircular
- Logically Disjoint
- Owning

Elements of Programming, Chapter 12
Whole-Part Relationships and Composite Objects

- Connected
- Noncircular
- Logically Disjoint
- Owning

Elements of Programming, Chapter 12
Whole-Part Relationships and Composite Objects

- Connected
- Noncircular
- Logically Disjoint
- Owning

- Standard Containers are Composite Objects
What is a data structure?
What is a data structure?

Definition: A structure utilizing value, physical, and representational relationships to encode semantic relationships on a collection of objects.
What is a data structure?

Definition: A structure utilizing value, physical, and representational relationships to encode semantic relationships on a collection of objects.

The choice of encoding can make a dramatic difference on the performance of operations.
Data Structure Performance

- Hierarchical Memory Structure
  - Register Access 0.1 ns
  - L1 Cache 0.5 ns
  - L2 Cache 7.0 ns
  - Memory 100.0 ns

3GHz processor, from Chandler Carruth talk - Credit to Jeff Dean
Data Structure Performance

- Hierarchical Memory Structure
  - Register Access 0.1 ns
  - L1 Cache 0.5 ns
  - L2 Cache 7.0 ns
  - Memory 100.0 ns

- RAM behaves much like a disk drive

3GHz processor, from Chandler Carruth talk - Credit to Jeff Dean
Data Structure Performance

- Hierarchical Memory Structure
  - Register Access 0.1 ns
  - L1 Cache 0.5 ns
  - L2 Cache 7.0 ns
  - Memory 100.0 ns

- RAM behaves much like a disk drive

\[ \log_2 1,000,000,000,000,000 = 40 \]

3GHz processor, from Chandler Carruth talk - Credit to Jeff Dean
Data Structure Performance

- Locality matters - use arrays or vector
  - Parallel Arrays
  - Static Lookup Tables
  - Closed Hash Maps
  - Algorithms
Example: Parallel Array & Algorithms
Stable Partition
Stable Partition
Stable Partition
Stable Partition
Stable Partition
Stable Partition

\texttt{stable\_partition(f, m, p)}

\texttt{stable\_partition(m, l, p)}
Stable Partition

\[
\text{stable_partition}(f, m, p) \\
\text{stable_partition}(m, l, p)
\]
Stable Partition

stable_partition(f, m, p)
stable_partition(m, l, p)
rotate(stable_partition(f, m, p),
    m,
    stable_partition(m, l, p));
rotate(stable_partition(f, m, p),
  m,
  stable_partition(m, l, p));
```
return rotate(stable_partition(f, m, p),
    m,
    stable_partition(m, l, p));
```
Stable Partition

```c
return rotate(stable_partition(f, m, p), m, stable_partition(m, l, p));
```
if (n == 1) return f + p(*f);

return rotate(stable_partition(f, m, p),
              m,
              stable_partition(m, l, p));
Stable Partition

```
if (n == 1) return f + p(*f);

return rotate(stable_partition(f, m, p), m, stable_partition(m, l, p));
```
template <typename I,
          typename P>
auto stable_partition(I f, I l, P p) -> I
{
    auto n = l - f;
    if (n == 0) return f;
    if (n == 1) return f + p(*f);

    auto m = f + (n / 2);

    return rotate(stable_partition(f, m, p),
                  m,
                  stable_partition(m, l, p));
}

© 2013 Adobe Systems Incorporated. All Rights Reserved.
template<typename I, typename P>
auto stable_partition(I f, I l, P p) -> I
{
    auto n = l - f;
    if (n == 0) return f;
    if (n == 1) return f + p(*f);
    auto m = f + (n / 2);
    return rotate(stable_partition(f, m, p),
                  m,
                  stable_partition(m, l, p));
}
template <typename I, 
        typename P>
auto stable_partition(I f, I l, P p) -> I
{
  auto n = l - f;
  if (n == 0) return f;
  if (n == 1) return f + p(*f);
  auto m = f + (n / 2);

  return rotate(stable_partition(f, m, p),
                m,
                stable_partition(m, l, p));
}
template<typename I, 
typename P>
auto stable_partition(I f, I l, P p) -> I
{
    auto n = l - f;
    if (n == 0) return f;
    if (n == 1) return f + p(*f);
    auto m = f + (n / 2);
    return rotate(stable_partition(f, m, p), 
                  m, 
                  stable_partition(m, l, p));
}
template <typename I, 
    typename P>
auto stable_partition_position(I f, I l, P p) -> I 
{
    auto n = l - f;
    if (n == 0) return f;
    if (n == 1) return f + p(f);
    auto m = f + (n / 2);

    return rotate(stable_partition_position(f, m, p), 
        m, 
        stable_partition_position(m, l, p));
}
int a[] = { 1, 2, 3, 4, 5, 5, 4, 3, 2, 1 };  
bool b[] = { 0, 1, 0, 1, 0, 0, 1, 0, 1, 0 };  

auto p = stable_partition_position(begin(a), end(a), [&](auto i) {
    return *begin(b) + (i - begin(a));
});

for (auto f = begin(a), l = p; f != l; ++f) cout << *f << " ";
for (auto f = p, l = end(a); f != l; ++f) cout << *f << " ";
cout << endl;
```
int a[] = { 1, 2, 3, 4, 5, 5, 4, 3, 2, 1 };
bool b[] = { 0, 1, 0, 1, 0, 0, 1, 0, 1, 0 };

auto p = stable_partition_position(begin(a), end(a), [&](auto i) {
    return *begin(b) + (i - begin(a));
});

for (auto f = begin(a), l = p; f != l; ++f)
    cout << *f << " ";
cout << "^ ";
for (auto f = p, l = end(a); f != l; ++f)
    cout << *f << " ";
cout << endl;
```

```
2 4 4 2 ^ 1 3 5 5 3 1
```
Example: Algorithms & Minimal Work
Minimize Work

```
   4
  13 12
   7
  9
  5 15
 14
  2
 11
  6
 16
 10
  1
  8
  3
```

sf-

sl-
Minimize Work

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16

sf -

sl -
Minimize Work

```
X X X X X
  6 7 8 9 X
```

sf - sl -
Minimize Work
Minimize Work

```
  4
  13
  12
   7
   9
   5
  15
  14
   2
  11
   6
  16
  10
   1
   8
   3
```

\( f \) - \( s_f \) - \( l \) -
Minimize Work

```c
nth_element(f, sf, l);
```
Minimize Work

```cpp
nth_element(f, sf, l);
```
nth_element(f, sf, l);
Minimize Work

nth_element(f, sf, l);

\[ f \leq *sf \]
Minimize Work

```
nth_element(f, sf, l);
```

```
2 1 3 4 5 6 sf
```

```
7 14 12 15 9 16 ≥ *sf
```

```
10 13 8 11 f ≤ *sf
```

```
l
```
Minimize Work

```
nth_element(f, sf, l);
```

```
2          1          3          4          5
2          1          3          4          5
2          1          3          4          5
f          sf         sl         l
```
Minimize Work

```c
nth_element(f, sf, l);
++sf;
```
Minimize Work

```cpp
nth_element(f, sf, l);
++sf;
```
Minimize Work

```cpp
nth_element(f, sf, l);
++sf;
partial_sort(sf, sl, l);
```
Minimize Work

```cpp
nth_element(f, sf, l);
++sf;
partial_sort(sf, sl, l);
```
Minimize Work

```cpp
nth_element(f, sf, l);
++sf;
partial_sort(sf, sl, l);
```
if (sf == sl) return;

nth_element(f, sf, l);
++sf;

partial_sort(sf, sl, l);
if (sf == sl) return;
if (sf != f) {
    nth_element(f, sf, l);
    ++sf;
}
partial_sort(sf, sl, l);
template <typename I> // I models RandomAccessIterator
void sort_subrange(I f, I l, I sf, I sl) {
    if (sf == sl) return;
    if (sf != f) {
        nth_element(f, sf, l);
        ++sf;
    }
    partial_sort(sf, sl, l);
}
Minimize Work

```c
sort_subrange(f, l, sf, sl);
```
sort_subrange(f, l, sf, sl);
sort_subrange(f, l, sf, sl);
sort_subrange(f, l, sf, sl);
partial_sort(sl, nl, l);
sort_subrange(f, l, sf, sl);
partial_sort(sl, nl, l);
What is an *incidental* data structure?
What is an *incidental* data structure?

Definition: An incidental data structure is a data structure that occurs within a system when there is no object representing the structure as a whole.
What is an *incidental* data structure?

Definition: An incidental data structure is a data structure that occurs within a system when there is no object representing the structure as a whole.

Structures formed in the absence of a whole/part relationship
Why no incidental data structures?

- They cause ambiguities and break our ability to reason about code locally
• Delegates
Incidental Data Structures

- Delegates

- Message handlers
Incidental Data Structures

- Self-referential interface
Self-referential interface

```cpp
class UIElement { }

class UIElementCollection {
  public:
    void Add(shared_ptr<UIElement>);
};

class Panel : public UIElement {
  public:
    shared_ptr<UIElementCollection> Children() const;
};
```
Self-referential interface

class UIElement {
};

class UIElementCollection {
    public:
        void Add(shared_ptr<UIElement>);
};

class Panel : public UIElement {
    public:
        shared_ptr<UIElementCollection> Children() const;
};

Panel->Children()->Add(element);
- Self-referential interface

```cpp
class UIElement { }

class UIElementCollection {
  public:
    void Add(shared_ptr<UIElement>);
};

class Panel : public UIElement {
  public:
    shared_ptr<UIElementCollection> Children() const;
};

panel->Children()->Add(element);
panel->Children()->Add(element);
```
• Self-referential interface

```cpp
class UIElement { };

class UIElementCollection {
  public:
    void Add(shared_ptr<UIElement>);
}

class Panel : public UIElement {
  public:
    shared_ptr<UIElementCollection> Children() const;
};

panel->Children()->Add(element);
panel->Children()->Add(element);
panel2->Children()->Add(element);
```

![Diagram of the code structure]
Self-referential interface

```cpp
class UIElement {}

class UIElementCollection {
  public:
    void Add(shared_ptr<UIElement>);
};

class Panel : public UIElement {
  public:
    shared_ptr<UIElementCollection> Children() const;
};

panel->Children()->Add(element);
panel->Children()->Add(element);
panel2->Children()->Add(element);
panel->Children()->Add(panel);
```
Self-referential interface

```cpp
class UIElement { }

class UIElementCollection {
    public:
        void Add(shared_ptr<UIElement>);
    

class Panel : public UIElement {
    public:
        shared_ptr<UIElementCollection> Children() const;
    
panel->Children()->Add(element);
panel->Children()->Add(element);
panel2->Children()->Add(element);
panel->Children()->Add(panel);
```
Hierarchies

forest

A
  B
  C
  D

E
Hierarchies
Hierarchies
Hierarchies

list

A

B

© 2013 Adobe Systems Incorporated. All Rights Reserved.
Hierarchies

A

B C D

E

forest
Hierarchies
Hierarchies
Hierarchies
Hierarchies

```
begin()
forest
end()
```

```
begin()
A
E
end()
```

```
begin()
B
C
D
end()
```
Hierarchies
Hierarchies

```
forest
begin()
A
B
C
D
```

```
E
end()
```
Hierarchies

```cpp
forest<string> f;

f.insert(end(f), "A");
f.insert(end(f), "E");

auto a = trailing_of(begin(f));
f.insert(a, "B");
f.insert(a, "C");
f.insert(a, "D");
```

![Hierarchical diagram showing a forest of strings with nodes A, B, C, D, and E connected in a hierarchical structure.]
Conclusions

- Understand the structures created by relationships
- Encapsulate structure invariants in composite types
- Learn to use the tools at your disposal
  - And how to create new ones
No incidental data structures
No incidental data structures

Composite Types
No incidental data structures

Composite Types

Better Code