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?

Definition: A data structure is a format for organizing and storing data.
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.
[ This slide intentionally left void ]
0100
0100

[-8..7]
hash( ) != hash( )
Memory Space
Memory Space
Whole-Part Relationships and Composite Objects

- Connected
- Noncircular
- Logically Disjoint
- Owning

Standard Containers are Composite Objects

Elements of Programming, Chapter 12
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

- RAM behaves much like a disk drive

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

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(p, l, s)
stable_partition(f, p, not1(s))
stable_partition(f, p, not1(s))
stable_partition(p, l, s)
Gather

\[
\text{stable\_partition}(f, \ p, \ \text{not1}(s)) \\
\text{stable\_partition}(p, \ l, \ s)
\]
return { stable_partition(f, p, not1(s)),
        stable_partition(p, l, s) };
template <typename I, // I models BidirectionalIterator
typename S> // S models UnaryPredicate
auto gather(I f, I l, I p, S s) -> pair<I, I> 
{
    return { stable_partition(f, p, not1(s)),
             stable_partition(p, l, s) };
}
template <typename I, // I models BidirectionalIterator
typename S> // S models UnaryPredicate
auto gather(I f, I l, I p, S s) -> pair<I, I>
{
    return { stable_partition(f, p, not1(s)),
             stable_partition(p, l, s) };
}
Stable Partition
Stable Partition

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

\[
\text{return } \text{rotate(stable\_partition}(f, m, p), \\
m, \\
\text{stable\_partition}(m, l, p));
\]
Stable Partition

\[
\text{return } \text{rotate(stable\_partition}(f, m, p), m, \text{stable\_partition}(m, l, p));
\]

\[
\text{if } (n == 1) \text{return } f + p(*f);
\]
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));
}
Stable Partition

```cpp
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));
}
```

```cpp
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));
}
```
```cpp
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;

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

<table>
<thead>
<tr>
<th>3</th>
<th>8</th>
<th>1</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>11</td>
<td>15</td>
<td>7</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>14</td>
<td>9</td>
</tr>
<tr>
<td>15</td>
<td>5</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>13</td>
<td>12</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>15</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>16</td>
<td>16</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Minimize Work
Minimize Work
Minimize Work
Minimize Work

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

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

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

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

```
  2  f  ≤ *sf
  1
  3
  4
  5  sf
  6
  7
 14
 12
15
 9
16
10
13
 8
11  l
  ≥ *sf
```
Minimize Work

\[
\text{nth\_element}(f, sf, l);
\]
Minimize Work

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

```
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);
```
```
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);
}
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?

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
Incidental Data Structures

- Delegates

- Message handlers

- Any pointer or reference stored in an object which refers to another object which is not a part
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);
panel->Children()->Add(element);
panel2->Children()->Add(element);
panel->Children()->Add(panel);
Hierarchies

forest

A

B C D

E
Hierarchies
Hierarchies

list
A
B
Hierarchies

forest

A

B

C

D

E
Hierarchies
Hierarchies
Hierarchies

begin()  
forest

dead()  

A

E

B
C
D

begin()  
end()
Hierarchies

leading

trailing
Hierarchies
Hierarchies

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");
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");

auto r = depth_range(f);
for (auto f = begin(r), l = end(r); f != l; ++f) {
    cout << string(f.depth() * 4, ' ') << (f.edge() ? "<" : "</") << *f << "\n";
}
```
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

- Slides and code from talk:

- Forest library:
  - https://github.com/stlab/adobe_source_libraries
No incidental data structures

Composite Types

Better Code