C++ Seasoning
Sean Parent  |  Principal Scientist
3 Goals for Better Code
No Raw Loops
What is a Raw Loop?

- A *raw loop* is any loop inside a function where the function serves purpose larger than the algorithm implemented by the loop.
What is a Raw Loop?

```cpp
void PanelBar::RepositionExpandedPanels(Panel* fixed_panel) {
    CHECK(fixed_panel);

    // First, find the index of the fixed panel.
    int fixed_index = GetPanelIndex(expanded_panels_, *fixed_panel);
    CHECK_LT(fixed_index, expanded_panels_.size());

    // Next, check if the panel has moved to the other side of another panel.
    const int center_x = fixed_panel->cur_panel_center();
    for (size_t i = 0; i < expanded_panels_.size(); ++i) {
        Panel* panel = expanded_panels_[i].get();
        if (center_x <= panel->cur_panel_center() ||
            i == expanded_panels_.size() - 1) {
            if (panel != fixed_panel) {
                // If it has, then we reorder the panels.
                ref_ptr<Panel> ref = expanded_panels_[fixed_index];
                expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
                if (i < expanded_panels_.size()) {
                    expanded_panels_.insert(expanded_panels_.begin() + i, ref);
                } else {
                    expanded_panels_.push_back(ref);
                }
            } else {
                break;
            }
        }
    }
}
```

// Find the total width of the panels to the left of the fixed panel.
int total_width = 0;
int fixed_index = -1;
```
What is a Raw Loop?

```cpp
void PanelBar::RepositionExpandedPanels(Panel* fixed_panel) {
    CHECK(fixed_panel);

    // First, find the index of the fixed panel.
    int fixed_index = GetPanelIndex(expanded_panels_, *fixed_panel);
    CHECK_LT(fixed_index, expanded_panels_.size());

    // Next, check if the panel has moved to the other side of another panel.
    const int center_x = fixed_panel->cur_panel_left;
    for (size_t i = 0; i < expanded_panels_.size(); ++i) {
        Panel* panel = expanded_panels_[i].get();
        if (center_x <= panel->cur_panel_center() ||
            i == expanded_panels_.size() - 1) {
            if (panel != fixed_panel) {
                // If it has, then we reorder the panels.
                ref_ptr<Panel> ref = expanded_panels_[fixed_index];
                expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
                if (i < expanded_panels_.size()) {
                    expanded_panels_.insert(expanded_panels_.begin() + i, ref);
                } else {
                    expanded_panels_.push_back(ref);
                }
            }
        } else {
            break;
        }
    }

    // Find the total width of the panels to the left of the fixed panel.
    int total_width = 0;
    fixed_index = -1;
    for (int i = 0; i < fixed_index; ++i) {
        total_width += panel_width
    }

    // Find the total width of the panels to the right of the fixed panel.
    total_width += panel_width

    // Move the panel over.
    expanded_panels_[fixed_index].push_back(panel_width);
    total_width -= panel_width;
    fixed_index = new_fixed_index;
}
```
What is a Raw Loop?

```cpp
void PanelBar::RepositionExpandedPanels(Panel* fixed_panel) {
    CHECK(fixed_panel);

    // First, find the index of the fixed panel.
    int fixed_index = GetPanelIndex(expanded_panels_, *fixed_panel);
    CHECK_LT(fixed_index, expanded_panels_.size());

    // Next, check if the panel has moved to the other side of another panel.
    const int center_x = fixed_panel->cur_panel_center();
    for (size_t i = 0; i < expanded_panels_.size(); ++i) {
        Panel* panel = expanded_panels_[i].get();
        if (center_x <= panel->cur_panel_center() ||
            i == expanded_panels_.size() - 1) {
            if (panel != fixed_panel) {
                // If it has, then we reorder the panels.
                ref_ptr<Panel> ref = expanded_panels_[fixed_index];
                expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
                if (i < expanded_panels_.size()) {
                    expanded_panels_.insert(expanded_panels_.begin() + i, ref);
                } else {
                    expanded_panels_.push_back(ref);
                }
            }
            break;
        }
    }

    // Find the total width of the panels to the left of the fixed panel.
    int total_width = 0;
    fixed_index = -1;
```
What is a Raw Loop?

```cpp
void PanelBar::RepositionExpandedPanels(Panel* fixed_panel) {
    CHECK(fixed_panel);

    // First, find the index of the fixed panel.
    int fixed_index = GetPanelIndex(expanded_panels_, *fixed_panel);
    CHECK_LT(fixed_index, expanded_panels_.size());

    // Next, check if the panel has moved to the other side of another panel.
    const int center_x = fixed_panel->cur_panel_center();
    for (size_t i = 0; i < expanded_panels_.size(); ++i) {
        Panel* panel = expanded_panels_[i].get();
        if (center_x <= panel->cur_panel_center() ||
            i == expanded_panels_.size() - 1) {
            if (panel != fixed_panel) {
                // If it has, then we reorder the panels.
                ref_ptr<Panel> ref = expanded_panels_[fixed_index];
                expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
                if (i < expanded_panels_.size()) {
                    expanded_panels_.insert(expanded_panels_.begin() + i, ref);
                } else {
                    expanded_panels_.push_back(ref);
                }
            }
            break;
        }
    }

    // Find the total width of the panels to the left of the fixed panel.
    int total_width = 0;
    fixed_index = -1;
```
What is a Raw Loop?

```cpp
tvoid PanelBar::RepositionExpandedPanels(Panel* fixed_panel) {
  CHECK(fixed_panel);

  // First, find the index of the fixed panel.
  CHECK_LT(fixed_index, expanded_panels_.size());

  // Next, check if the panel has moved to the other side of another panel.
  const int center_x = fixed_panel->cur_panel_center();
  for (size_t i = 0; i < expanded_panels_.size(); ++i) {
    Panel* panel = expanded_panels_[i].get();
    if (center_x <= panel->cur_panel_center()) ||
      i == expanded_panels_.size() - 1) {
      if (panel != fixed_panel) {
        // If it has, then we reorder the panels.
        ref_ptr<Panel> ref = expanded_panels_[fixed_index];
        expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
        if (i < expanded_panels_.size()) {
          expanded_panels_.insert(expanded_panels_.begin() + i, ref);
        } else {
          expanded_panels_.push_back(ref);
        }
        break;
      }
    }
  }

  // Find the total width of the panels to the left of the fixed panel.
  int total_width = 0;
  fixed_index = -1;
```
What is a Raw Loop?

```cpp
expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
if (i < expanded_panels_.size()) {
    expanded_panels_.insert(expanded_panels_.begin() + i, ref);
} else {
    expanded_panels_.push_back(ref);
}
break;
}

// Find the total width of the panels to the left of the fixed panel.
int total_width = 0;
fixed_index = -1;
for (int i = 0; i < static_cast<int>(expanded_panels_.size()); ++i) {
    Panel* panel = expanded_panels_[i].get();
    if (panel == fixed_panel) {
        fixed_index = i;
        break;
    }
    total_width += panel->panel_width();
}
CHECK_NE(fixed_index, -1);
int new_fixed_index = fixed_index;

// Move panels over to the right of the fixed panel until all of the ones
// on the left will fit.
int avail_width = max(fixed_panel->cur_panel_left() - kBarPadding, 0);
while (total_width > avail_width) {
    new_fixed_index--;
    CHECK_GE(new_fixed_index, 0);
    total_width -= expanded_panels_[new_fixed_index]->panel_width();
```
What is a Raw Loop?

for (int i = 0; i < static_cast<int>(expanded_panels_.size()); ++i) {
    Panel* panel = expanded_panels_[i].get();
    if (panel == fixed_panel) {
        fixed_index = i;
        break;
    }
    total_width += panel->panel_width();
}
CHECK_NE(fixed_index, -1);
int new_fixed_index = fixed_index;

// Move panels over to the right of the fixed panel until all of the ones
// on the left will fit.
int avail_width = max(fixed_panel->cur_panel_left() - kBarPadding, 0);
while (total_width > avail_width) {
    new_fixed_index--;
    CHECK_GE(new_fixed_index, 0);
    total_width -= expanded_panels_[new_fixed_index]->panel_width();
}

// Reorder the fixed panel if its index changed.
if (new_fixed_index != fixed_index) {
    Panels::iterator it = expanded_panels_.begin() + fixed_index;
    ref_ptr<Panel> ref = *it;
    expanded_panels_.erase(it);
    expanded_panels_.insert(expanded_panels_.begin() + new_fixed_index, ref);
    fixed_index = new_fixed_index;
}

// Now find the width of the panels to the right, and move them to the
// left as needed.
total_width = 0;
for (Panels::iterator it = expanded_panels_.begin() + fixed_index + 1;
     it != expanded_panels_.end(); ++it) {
What is a Raw Loop?

// Move panels over to the right of the fixed panel until all of the ones
// on the left will fit.
for (Panel::iterator it = expanded_panels_.begin() + fixed_index + 1; it != expanded_panels_.end(); ++it) {
    total_width += (*it)->panel_width();
}

// Now find the width of the panels to the right, and move them to the
// left as needed.
total_width = 0;
for (Panel::iterator it = expanded_panels_.begin() + fixed_index + 1; it != expanded_panels_.end(); ++it) {
    total_width += (*it)->panel_width();
}

avail_width = max(wm_->width() - (fixed_panel->cur_right() + kBarPadding), 0);
while (total_width > avail_width) {
    new_fixed_index++;
    CHECK_LT(new_fixed_index, expanded_panels_.size());
    total_width -= expanded_panels_[new_fixed_index]->panel_width();
}

// Do the reordering again.
if (new_fixed_index != fixed_index) {
    Panel::iterator it = expanded_panels_.begin() + fixed_index;
    ref_ptr<Panel> ref = *it;
    expanded_panels_.erase(it);
    expanded_panels_.insert(expanded_panels_.begin() + new_fixed_index, ref);
    fixed_index = new_fixed_index;
}

// Reorder the fixed panel if its index changed.
if (new_fixed_index != fixed_index) {
    Panels::iterator it = expanded_panels_.begin() + fixed_index;
    ref_ptr<Panel> ref = *it;
    expanded_panels_.erase(it);
    expanded_panels_.insert(expanded_panels_.begin() + new_fixed_index, ref);
    fixed_index = new_fixed_index;
}
while (total_width > avail_width) {
    // Find the width of the panels to the right, and move them to the
    // right so they don't overlap.
    ref_ptr<Panel> ref = *it;
    expanded_panels_.erase(it);
    expanded_panels_.insert(expanded_panels_.begin() + new_fixed_index, ref);
    fixed_index = new_fixed_index;
}

// Finally, push panels to the left and the right so they don't overlap.
int boundary = expanded_panels_[fixed_index]->cur_panel_left() - kBarPadding;
for (Panel* panel = it->get();
     // Start at the panel to the left of 'new_fixed_index'.
     expanded_panels_.rbegin() +
     (expanded_panels_.size() - new_fixed_index);
     it != expanded_panels_.rend(); ++it) {
    Panel* panel = it->get();
    if (panel->cur_right() > boundary) {
        panel->Move(boundary, kAnimMs);
    }
    else if (panel->cur_panel_left() < 0) {
        panel->Move(min(boundary, panel->panel_width() + kBarPadding), kAnimMs);
    }
    boundary = panel->cur_panel_left() - kBarPadding;
}
boundary = expanded_panels_[fixed_index]->cur_right() + kBarPadding;
for (Panel* panel = it->get();
     // Finally, push panels to the left and the right so they don't overlap.
     expanded_panels_.begin() + new_fixed_index + 1;
     panel == fixed_panel) {
    fixed_index = -1;
    break
}
if (new_fixed_index != fixed_index) {
    Panels::iterator it = expanded_panels_.begin() + fixed_index;
    ref_ptr<Panel> ref = *it;
    expanded_panels_.erase(it);
    expanded_panels_.insert(expanded_panels_.begin() + new_fixed_index, ref);
    fixed_index = new_fixed_index;
}
} // Do the reordering again.
int boundary = expanded_panels_[fixed_index]->cur_panel_left() - kBarPadding;
for (Panel::reverse_iterator it =
     (expanded_panels_.rbegin() +
      (expanded_panels_.size() - new_fixed_index);
      it != expanded_panels_.rend(); ++it) {
    Panel* panel = it->get();
    if (panel->cur_right() > boundary) {
      panel->Move(boundary, kAnimMs);
    } else if (panel->cur_panel_left() < 0) {
      panel->Move(min(boundary, panel->panel_width() + kBarPadding), kAnimMs);
    }
    boundary = panel->cur_panel_left() - kBarPadding;
  }
  boundary = expanded_panels_[fixed_index]->cur_right() + kBarPadding;
for (Panel::iterator it = expanded_panels_.begin() + new_fixed_index + 1;
     it != expanded_panels_.end(); ++it) {
  Panel* panel = it->get();
  if (panel->cur_panel_left() < boundary) {
    panel->Move(boundary + panel->panel_width(), kAnimMs);
  } else if (panel->cur_right() > wm_->width()) {
    panel->Move(max(boundary + panel->panel_width(),
    wm_->width() - kBarPadding), kAnimMs);
  }
  boundary = panel->cur_right() + kBarPadding;
}
Why No Raw Loops?

- Difficult to reason about and difficult to prove post conditions
- Error prone and likely to fail under non-obvious conditions
- Introduce non-obvious performance problems
- Complicates reasoning about the surrounding code
Alternatives to Raw Loops
Alternatives to Raw Loops

- Use an existing algorithm
Alternatives to Raw Loops

- Use an existing algorithm
  - Prefer standard algorithms if available
Alternatives to Raw Loops

- Use an existing algorithm
  - Prefer standard algorithms if available
- Implement a known algorithm as a general function
Alternatives to Raw Loops

- Use an existing algorithm
  - Prefer standard algorithms if available
- Implement a known algorithm as a general function
  - Contribute it to a library
Alternatives to Raw Loops

- Use an existing algorithm
  - Prefer standard algorithms if available
- Implement a known algorithm as a general function
  - Contribute it to a library
    - Preferably open source
Alternatives to Raw Loops

- Use an existing algorithm
  - Prefer standard algorithms if available
- Implement a known algorithm as a general function
  - Contribute it to a library
    - Preferably open source
- Invent a new algorithm
Alternatives to Raw Loops

- Use an existing algorithm
  - Prefer standard algorithms if available
- Implement a known algorithm as a general function
  - Contribute it to a library
    - Preferably open source
- Invent a new algorithm
  - Write a paper
Alternatives to Raw Loops

- Use an existing algorithm
  - Prefer standard algorithms if available
- Implement a known algorithm as a general function
  - Contribute it to a library
    - Preferably open source
- Invent a new algorithm
  - Write a paper
  - Give talks
Alternatives to Raw Loops

- Use an existing algorithm
  - Prefer standard algorithms if available
- Implement a known algorithm as a general function
  - Contribute it to a library
    - Preferably open source
- Invent a new algorithm
  - Write a paper
  - Give talks
  - Become famous!
Alternatives to Raw Loops

- Use an existing algorithm
  - Prefer standard algorithms if available

- Implement a known algorithm as a general function
  - Contribute it to a library
    - Preferably open source

- Invent a new algorithm
  - Write a paper
  - Give talks
  - Become famous!
No Raw Loops
If you want to improve the code quality in your organization, replace all of your coding guidelines with one goal:
If you want to improve the code quality in your organization, replace all of your coding guidelines with one goal:

No Raw Loops
Two Beautiful Examples
No Raw Loops
No Raw Loops
No Raw Loops

rotate(f, l, p);
rotate(f, l, p);
No Raw Loops
No Raw Loops
rotate(p, f, l);
rotate(p, f, l);
No Raw Loops

```c
if (p < f) rotate(p, f, l);
if (l < p) rotate(f, l, p);
```
No Raw Loops

```c
if (p < f) rotate(p, f, l);
if (l < p) rotate(f, l, p);
```
No Raw Loops

```c
if (p < f) rotate(p, f, l);
if (l < p) rotate(f, l, p);
```
No Raw Loops

if (p < f) rotate(p, f, l);
if (l < p) rotate(f, l, p);
No Raw Loops

```c
if (p < f) return { p, rotate(p, f, l) };
if (l < p) return { rotate(f, l, p), p };
```
if (p < f) return { p, rotate(p, f, l) };
if (l < p) return { rotate(f, l, p), p };

C++11
if (p < f) return \{ p, rotate(p, f, l) \};;
if (l < p) return \{ rotate(f, l, p), p \};
if (p < f) return { p, rotate(p, f, l) };  
if (l < p) return { rotate(f, l, p), p };  
return { f, l };
template <typename I> // I models RandomAccessIterator
auto slide(I f, I l, I p) -> pair<I, I>
{
    if (p < f) return { p, rotate(p, f, l) };
    if (l < p) return { rotate(f, l, p), p };
    return { f, l };
}
No Raw Loops
No Raw Loops
stable_partition(p, l, s)
stable_partition(p, l, s)
stable_partition(f, p, not1(s))
stable_partition(f, p, not1(s))
stable_partition(f, p, not1(s))
stable_partition(p, l, s)
No Raw Loops

stable_partition(f, p, not1(s))
stable_partition(p, l, s)
stable_partition(f, p, not1(s))
stable_partition(p, l, s)
return { stable_partition(f, p, not1(s)),
          stable_partition(p, l, s) };
No Raw Loops

template <typename I, // I models BidirectionalIterator
typeone 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) };}

```cpp
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) };}
```

```cpp
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) };}
```
No Raw Loops

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) };
}
What about that messy loop?

```
// Next, check if the panel has moved to the other side of another panel.
for (size_t i = 0; i < expanded_panels_.size(); ++i) {
    Panel* panel = expanded_panels_[i].get();
    if (center_x <= panel->cur_panel_center() ||
        i == expanded_panels_.size() - 1) {
        if (panel != fixed_panel) {
            // If it has, then we reorder the panels.
            ref_ptr<Panel> ref = expanded_panels_[fixed_index];
            expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
            if (i < expanded_panels_.size()) {
                expanded_panels_.insert(expanded_panels_.begin() + i, ref);
            } else {
                expanded_panels_.push_back(ref);
            }
        }
        break;
    }
}
What about that messy loop?

// Next, check if the panel has moved to the other side of another panel.

for (size_t i = 0; i < expanded_panels_.size(); ++i) {
  Panel* panel = expanded_panels_[i].get();
  if (center_x <= panel->cur_panel_center() ||
      i == expanded_panels_.size() - 1) {
    if (panel != fixed_panel) {
      // If it has, then we reorder the panels.
      ref_ptr<Panel> ref = expanded_panels_[fixed_index];
      expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
      if (i < expanded_panels_.size()) {
        expanded_panels_.insert(expanded_panels_.begin() + i, ref);
      } else {
        expanded_panels_.push_back(ref);
      }
    }
    break;
  }
}
What about that bad loop?

```cpp
// Next, check if the panel has moved to the other side of another panel.

for (size_t i = 0; i < expanded_panels_.size(); ++i) {
  Panel* panel = expanded_panels_[i].get();
  if (center_x <= panel->cur_panel_center() ||
      i == expanded_panels_.size() - 1) {
    if (panel != fixed_panel) {
      // If it has, then we reorder the panels.
      ref_ptr<Panel> ref = expanded_panels_[fixed_index];
      expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
      expanded_panels_.insert(expanded_panels_.begin() + i, ref);
    }
    break;
  }
}
```
// Next, check if the panel has moved to the other side of another panel.

for (size_t i = 0; i < expanded_panels_.size(); ++i) {
    Panel* panel = expanded_panels_[i].get();

    if (center_x <= panel->cur_panel_center() ||
        i == expanded_panels_.size() - 1) {
        if (panel != fixed_panel) {
            // If it has, then we reorder the panels.
            ref_ptr<Panel> ref = expanded_panels_[fixed_index];
            expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
            expanded_panels_.insert(expanded_panels_.begin() + i, ref);
        }
        break;
    }
}
What about that bad loop?

// Next, check if the panel has moved to the other side of another panel.

for (size_t i = 0; i < expanded_panels_.size(); ++i) {
    Panel* panel = expanded_panels_[i].get();
    if (center_x <= panel->cur_panel_center() ||
        i == expanded_panels_.size() - 1) {
        break;
    }
}

// Fix this code – panel is the panel found above.

if (panel != fixed_panel) {
    // If it has, then we reorder the panels.
    ref_ptr<Panel> ref = expanded_panels_[fixed_index];
    expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
    expanded_panels_.insert(expanded_panels_.begin() + i, ref);
}
What about that bad loop?

// Next, check if the panel has moved to the other side of another panel.

for (size_t i = 0; i < expanded_panels_.size(); ++i) {
    Panel* panel = expanded_panels_[i].get();
    if (center_x <= panel->cur_panel_center() ||
        i == expanded_panels_.size() - 1) {
        break;
    }
}

// Fix this code - panel is the panel found above.

if (panel != fixed_panel) {
    // If it has, then we reorder the panels.
    ref_ptr<Panel> ref = expanded_panels_[fixed_index];
    expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
    expanded_panels_.insert(expanded_panels_.begin() + i, ref);
}
What about that bad loop?

// Next, check if the panel has moved to the other side of another panel.

for (size_t i = 0; i < expanded_panels_.size(); ++i) {
    Panel* panel = expanded_panels_[i].get();
    if (center_x <= panel->cur_panel_center()) break;
}

// Fix this code – panel is the panel found above.

if (panel != fixed_panel) {
    // If it has, then we reorder the panels.
    ref_ptr<Panel> ref = expanded_panels_[fixed_index];
    expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
    expanded_panels_.insert(expanded_panels_.begin() + i, ref);
}
What about that bad loop?

// Next, check if the panel has moved to the other side of another panel.

for (size_t i = 0; i < expanded_panels_.size(); ++i) {
    Panel* panel = expanded_panels_[i].get();
    if (center_x <= panel->cur_panel_center()) break;
}

// Fix this code - panel is the panel found above.

if (panel != fixed_panel) {
    // If it has, then we reorder the panels.
    ref_ptr<Panel> ref = expanded_panels_[fixed_index];
    expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
    expanded_panels_.insert(expanded_panels_.begin() + i, ref);
}
What about that bad loop?

// Next, check if the panel has moved to the other side of another panel.

auto p = find_if(begin(expanded_panels_), end(expanded_panels_),
     [&](const ref_ptr<Panel>& e){ return center_x <= e->cur_panel_center(); });

// Fix this code - panel is the panel found above.

if (panel != fixed_panel) {
    // If it has, then we reorder the panels.
    ref_ptr<Panel> ref = expanded_panels_[fixed_index];
    expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
    expanded_panels_.insert(expanded_panels_.begin() + i, ref);
}
// Next, check if the panel has moved to the other side of another panel.

auto p = find_if(begin(expanded_panels_), end(expanded_panels_),
    [&](const ref_ptr<Panel>& e){ return center_x <= e->cur_panel_center(); });

// Fix this code - panel is the panel found above.

if (panel != fixed_panel) {
    // If it has, then we reorder the panels.
    ref_ptr<Panel> ref = expanded_panels_[fixed_index];
    expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
    expanded_panels_.insert(expanded_panels_.begin() + i, ref);
}
What about that bad loop?

// Next, check if the panel has moved to the other side of another panel.

auto p = find_if(begin(expanded_panels_), end(expanded_panels_),
    [&](const ref_ptr<Panel>& e){ return center_x <= e->cur_panel_center(); });

// Fix this code – panel is the panel found above.

if (panel != fixed_panel) {
    // If it has, then we reorder the panels.
    ref_ptr<Panel> ref = expanded_panels_[fixed_index];
    expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
    expanded_panels_.insert(expanded_panels_.begin() + i, ref);
}
What about that bad loop?

// Next, check if the panel has moved to the other side of another panel.

auto p = find_if(begin(expanded_panels_), end(expanded_panels_),
                 [&](const ref_ptr<Panel>& e){ return center_x <= e->cur_panel_center(); });

// Fix this code – panel is the panel found above.
if (panel != fixed_panel) {
  // If it has, then we reorder the panels.
  auto f = begin(expanded_panels_) + fixed_index;
  rotate(p, f, f + 1);
}
What about that bad loop?

// Next, check if the panel has moved to the other side of another panel.

auto p = find_if(begin(expanded_panels_), end(expanded_panels_),
    [&](const ref_ptr<Panel>& e){ return center_x <= e->cur_panel_center(); });

// Fix this code – panel is the panel found above.

if (panel != fixed_panel) {
    // If it has, then we reorder the panels.
    auto f = begin(expanded_panels_) + fixed_index;
    rotate(p, f, f + 1);
}
What about that bad loop?

// Next, check if the panel has moved to the other side of another panel.
auto p = find_if(begin(expanded_panels_), end(expanded_panels_),
    [&](const ref_ptr<Panel>& e){ return center_x <= e->cur_panel_center(); });

// If it has, then we reorder the panels.
auto f = begin(expanded_panels_) + fixed_index;
rotate(p, f, f + 1);
What about that bad loop?

// Next, check if the panel has moved to the other side of another panel.

auto p = find_if(begin(expanded_panels_), end(expanded_panels_),
    [&](const ref_ptr<Panel>& e){ return center_x <= e->cur_panel_center(); });

// If it has, then we reorder the panels.
auto f = begin(expanded_panels_) + fixed_index;
rotate(p, f, f + 1);
What about that bad loop?

// Next, check if the panel has moved to the left side of another panel.
auto p = find_if(begin(expanded_panels_), end(expanded_panels_),
    [&](const ref_ptr<Panel>& e) {
        return center_x <= e->cur_panel_center(); });

// If it has, then we reorder the panels.
auto f = begin(expanded_panels_) + fixed_index;
rotate(p, f, f + 1);
What about that bad loop?

// Next, check if the panel has moved to the left side of another panel.

```cpp
auto p = find_if(begin(expanded_panels_), end(expanded_panels_),
    [&](const ref_ptr<Panel>& e){ return center_x <= e->cur_panel_center(); });
```

// If it has, then we reorder the panels.
auto f = begin(expanded_panels_) + fixed_index;
rotate(p, f, f + 1);
What about that bad loop?

// Next, check if the panel has moved to the left side of another panel.
auto f = begin(expanded_panels_) + fixed_index;

auto p = lower_bound(begin(expanded_panels_), f, center_x,
    [](const ref_ptr<Panel>& e, int x){ return e->cur_panel_center() < x; });

// If it has, then we reorder the panels.
rotate(p, f, f + 1);
What about that bad loop?

// Next, check if the panel has moved to the left side of another panel.

auto f = begin(expanded_panels_) + fixed_index;

auto p = lower_bound(begin(expanded_panels_), f, center_x,
    [](const ref_ptr<Panel>& e, int x){ return e->cur_panel_center() < x; });

// If it has, then we reorder the panels.
rotate(p, f, f + 1);

- This is 1/2 of a slide() that only supports a single element being selected
// Next, check if the panel has moved to the left side of another panel.
auto f = begin(expanded_panels_) + fixed_index;

auto p = lower_bound(begin(expanded_panels_), f, center_x,
    [](const ref_ptr<Panel>& e, int x){ return e->cur_panel_center() < x; });

// If it has, then we reorder the panels.
rotate(p, f, f + 1);

- This is 1/2 of a slide() that only supports a single element being selected
  - The other rotate() is the erase()/insert() further down in the function
What about that bad loop?

```cpp
auto f = begin(expanded_panels_) + fixed_index;

auto p = lower_bound(begin(expanded_panels_), f, center_x,
    [](const ref_ptr<Panel>& e, int x){ return e->cur_panel_center() < x; });

// Next, check if the panel has moved to the left side of another panel.
// If it has, then we reorder the panels.
rotate(p, f, f + 1);
```

- This is 1/2 of a `slide()` that only supports a single element being selected
- The other `rotate()` is the `erase()/insert()` further down in the function
- None of the special cases were necessary
What about that bad loop?

// Next, check if the panel has moved to the left side of another panel.
auto f = begin(expanded_panels_) + fixed_index;

auto p = lower_bound(begin(expanded_panels_), f, center_x,
[] (const ref_ptr<Panel>& e, int x){ return e->cur_panel_center() < x; });

// If it has, then we reorder the panels.
rotate(p, f, f + 1);

- This is 1/2 of a slide() that only supports a single element being selected
  - The other rotate() is the erase()/insert() further down in the function
- None of the special cases were necessary
- This code is considerably more efficient
What about that bad loop?

```cpp
// Next, check if the panel has moved to the left side of another panel.
auto f = begin(expanded_panels_) + fixed_index;

auto p = lower_bound(begin(expanded_panels_), f, center_x,
    [](const ref_ptr<Panel>& e, int x){ return e->cur_panel_center() < x; });

// If it has, then we reorder the panels.
rotate(p, f, f + 1);
```

- This is 1/2 of a `slide()` that only supports a single element being selected
  - The other `rotate()` is the `erase()/insert()` further down in the function
- None of the special cases were necessary
- This code is considerably more efficient
- Now we can have the conversation about supporting multiple selections and disjoint selections!
Seasoning
Seasoning

- Use a range library (Boost or ASL)
Seasoning

- Use a range library (Boost or ASL)
  
  ```
  auto p = find(begin(a), end(a), x);
  ```
Seasoning

- Use a range library (Boost or ASL)
  ```
  auto p = find(begin(a), end(a), x);
  auto p = find(a, x);
  ```
Seasoning

- Use a range library (Boost or ASL)
  ```cpp
  auto p = find(begin(a), end(a), x);
  auto p = find(a, x);
  ```

- Have many variants of simple, common algorithms such as find() and copy()
Seasoning

- Use a range library (Boost or ASL)
  ```cpp
  auto p = find(begin(a), end(a), x);
  auto p = find(a, x);
  ```

- Have many variants of simple, common algorithms such as find() and copy()

- Look for interface symmetry
Seasoning

- Use a range library (Boost or ASL)
  
  ```
  auto p = find(begin(a), end(a), x);
  auto p = find(a, x);
  ```

- Have many variants of simple, common algorithms such as find() and copy()

- Look for interface symmetry
  
  ```
  sort(a, [](const employee& x, const employee& y){ return x.last < y.last; });
  ```
Seasoning

- Use a range library (Boost or ASL)
  
  ```cpp
  auto p = find(begin(a), end(a), x);
  auto p = find(a, x);
  ```

- Have many variants of simple, common algorithms such as find() and copy()

- Look for interface symmetry

  ```cpp
  sort(a, [] (const employee& x, const employee& y) { return x.last < y.last; });
  auto p = lower_bound(a, "Parent", [] (const employee& x, const string& y) { return x.last < y; });
  ```
Seasoning

- Use a range library (Boost or ASL)
  ```
  auto p = find(begin(a), end(a), x);
  auto p = find(a, x);
  ```

- Have many variants of simple, common algorithms such as `find()` and `copy()`

- Look for interface symmetry
  ```
  sort(a, [](const employee& x, const employee& y){ return x.last < y.last; });
  auto p = lower_bound(a, "Parent", [](const employee& x, const string& y){ return x.last < y; });
  sort(a, less(), &employee::last);
  ```
Seasoning

- Use a range library (Boost or ASL)
  
  ```cpp
  auto p = find(begin(a), end(a), x);
  auto p = find(a, x);
  ```

- Have many variants of simple, common algorithms such as find() and copy()

- Look for interface symmetry
  
  ```cpp
  sort(a, [](const employee& x, const employee& y){ return x.last < y.last; });
  auto p = lower_bound(a, "Parent", [](const employee& x, const string& y){ return x.last < y; });

  sort(a, less(), &employee::last);
  auto p = lower_bound(a, "Parent", less(), &employee::last);
  ```
Seasoning
• Range based for loops for for-each and simple transforms

```c++
for (const auto& e: r) f(e);
for (auto& e: r) e = f(e);
```
Seasoning

- Range based for loops for for-each and simple transforms

```cpp
for (const auto& e: r) f(e);
for (auto& e: r) e = f(e);
```
Range based for loops for for-each and simple transforms

```cpp
for (const auto& e : r) f(e);
for (auto& e : r) e = f(e);
```

Use `const auto&` for for-each and `auto&` for transforms
Seasoning

- Range based for loops for for-each and simple transforms

```cpp
for (const auto& e: r) f(e);
for (auto& e: r) e = f(e);
```

- Use const auto& for for-each and auto& for transforms
- Keep the body **short**
Range based for loops for for-each and simple transforms

```cpp
for (const auto& e: r) f(e);
for (auto& e: r) e = f(e);
```

- Use `const auto&` for for-each and `auto&` for transforms
- Keep the body short
  - A general guideline is no longer than composition of two functions with an operator

```cpp
for (const auto& e: r) f(g(e));
for (const auto& e: r) { f(e); g(e); };
for (auto& e: r) e = f(e) + g(e);
```
Range based for loops for for-each and simple transforms

```cpp
for (const auto& e: r) f(e);
for (auto& e: r) e = f(e);
```

- Use const auto& for for-each and auto& for transforms

- Keep the body **short**
  - A general guideline is no longer than composition of two functions with an operator

```cpp
for (const auto& e: r) f(g(e));
for (const auto& e: r) { f(e); g(e); }
for (auto& e: r) e = f(e) + g(e);
```

- If the body is longer, factor it out and give it a name
Seasoning
Seasoning

- Use lambdas for predicates, comparisons, and projections, but keep them **short**
Seasoning

- Use lambdas for predicates, comparisons, and projections, but keep them **short**

- Use function objects with template member function to simulate polymorphic lambda

```cpp
struct last_name {
    using result_type = const string&;

    template <typename T>
    const string& operator()(const T& x) const { return x.last; }
};

// ...

auto p = lower_bound(a, "Parent", less(), last_name());
```
No Raw Synchronization Primitives
What are raw synchronization primitives?

- Synchronization primitives are basic constructs such as:
  - Mutex
  - Atomic
  - Semaphore
  - Memory Fence
You Will Likely Get It Wrong
template <typename T>

class bad_cow {

    struct object_t {
        explicit object_t(const T& x) : data_m(x) { ++count_m; }
        atomic<int> count_m;
        T data_m;
    };

    object_t* object_m;

public:

    explicit bad_cow(const T& x) : object_m(new object_t(x)) { }
    ~bad_cow() { if (0 == --object_m->count_m) delete object_m; }
    bad_cow(const bad_cow& x) : object_m(x.object_m) { ++object_m->count_m; }
    bad_cow& operator=(const T& x) {
        if (object_m->count_m == 1) object_m->data_m = x;
        else {
            object_t* tmp = new object_t(x);
            --object_m->count_m;
            object_m = tmp;
        }
        return *this;
    }
};
template <typename T>
class bad_cow {
    struct object_t {
        explicit object_t(const T& x) : data_m(x) { ++count_m; }
        atomic<int> count_m;
        T data_m;
    };

    public:
    explicit bad_cow(const T& x) : object_m(new object_t(x)) { }
    ~bad_cow() { if (0 == --object_m->count_m) delete object_m; }
    bad_cow(const bad_cow& x) : object_m(x.object_m) { ++object_m->count_m; }

    bad_cow& operator=(const T& x) {
        if (object_m->count_m == 1) object_m->data_m = x;
        else {
            object_t* tmp = new object_t(x);
            --object_m->count_m;
            object_m = tmp;
        }
        return *this;
    }
};

Problems with Locks
Problems with Locks

```cpp
template <typename T>
class bad_cow {
    struct object_t {
        explicit object_t(const T& x) : data_m(x) { ++count_m; }
        atomic<int> count_m;
        T data_m;
    };
    object_t* object_m;

    explicit bad_cow(const T& x) : object_m(new object_t(x)) { }
    ~bad_cow() { if (0 == --object_m->count_m) delete object_m; }
    bad_cow(const bad_cow& x) : object_m(x.object_m) { ++object_m->count_m; }
    bad_cow& operator=(const T& x) {
        if (object_m->count_m == 1) object_m->data_m = x;
        else {
            object_t* tmp = new object_t(x);
            --object_m->count_m;
            object_m = tmp;
        }
        return *this;
    }
};
```

- There is a subtle race condition here:
  - if count != 1 then the bad_cow could also be owned by another thread(s)
  - if the other thread(s) releases the bad_cow between these two atomic operations
  - then our count will fall to zero and we will leak the object
template <typename T>
class bad_cow {
    struct object_t {
        explicit object_t(const T& x) : data_m(x) { ++count_m; }
        atomic<int> count_m;
        T data_m;
    };
    object_t* object_m;
public:
    explicit bad_cow(const T& x) : object_m(new object_t(x)) { }
    ~bad_cow() { if (0 == --object_m->count_m) delete object_m; }
    bad_cow(const bad_cow& x) : object_m(x.object_m) { ++object_m->count_m; }
    bad_cow& operator=(const T& x) {
        if (object_m->count_m == 1) object_m->data_m = x;
        else {
            object_t* tmp = new object_t(x);
            if (0 == --object_m->count_m) delete object_m;
            object_m = tmp;
        }
        return *this;
    }
};
Why No Raw Synchronization Primitives?
Why No Raw Synchronization Primitives?
Why No Raw Synchronization Primitives?

- thread
- STOP
- Object
- GO
- thread

© 2013 Adobe Systems Incorporated. All Rights Reserved.
Why No Raw Synchronization Primitives?
Amdahl's Law

![Graph showing the performance of processors vs. number of processors using Amdahl's Law](image)
Minimize Locks

STOP
Minimize Locks
No Raw Synchronization Primitives
No Raw Synchronization Primitives

Task
No Raw Synchronization Primitives
No Raw Synchronization Primitives
No Raw Synchronization Primitives
No Raw Synchronization Primitives

- Task
- Object
- ...
No Raw Synchronization Primitives

Task

Object

...
Tasks

- A task is a unit of work (a function) which is executed asynchronously
  - Tasks are scheduled on a thread pool to optimize machine utilization

- The arguments to the task and the task results are convenient places to communicate with other tasks
  - Any function can be “packaged” into such a task
Unfortunately, we don’t yet have a standard async task model

- std::async() is currently defined to be based on threads
  - This may change in C++14 and Visual C++ 2012 already implements std::async() as a task model

- Windows - Window Thread Pool and PPL
- Apple - Grand Central Dispatch (libdispatch)
  - Open sourced, runs on Linux and Android
- Intel TBB - many platform
namespace adobe {

template<typename F, typename ...Args>
auto async(F&& f, Args&&... args)
     -> std::future<typename std::result_of<F (Args...)>::type>
{
    using result_type = typename std::result_of<F (Args...)>::type;
    using packaged_type = std::packaged_task<result_type ()>;

    auto p = new packaged_type(std::forward<F>(f), std::forward<Args>(args)...);
    auto result = p->get_future();

    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0),
    p, [](){
        packaged_type* f = static_cast<packaged_type*>(p);
        (*f)();
        delete f;
    });

    return result;
}

} // namespace adobe
No Raw Synchronization Primitives

Task

Args
No Raw Synchronization Primitives

```
Task

future

Args

Task
```
No Raw Synchronization Primitives

Diagram:

- Task
  - future
  - ...

- Args
  - Task
  - ...

No Raw Synchronization Primitives

Diagram:
- Task
  - future
  - ...
  - future.get()

- Args
  - Task
  - ...

Code snippet:
```
Task
future.get()
```
No Raw Synchronization Primitives

future

STOP

Task

...
No Raw Synchronization Primitives

Task

future

...

gfuture.get()

Result
Task Systems

- Blocking on std::future.get() has two problems
  - One thread resource is consumed, increasing contention
  - Any subsequent non-dependent calculations on the task are also blocked

- Unfortunately, C++11 doesn’t have dependent tasks
  - GCD has serialized queues and groups
  - PPL has chained tasks
  - TBB has flow graphs
- All are able to specify dependent tasks, including joins
std::list can be used in a pinch to create thread safe data structures with splice

```cpp
template <typename T>
class concurrent_queue {
  mutex mutex_;  // mutex
  list<T> q_;     // thread safe queue

public:
  void enqueue(T x) {
    list<T> tmp;
    tmp.push_back(move(x));
    {  // lock
      lock_guard<mutex> lock(mutex);
      q_.splice(end(q_), tmp);
    }

    // ...
  }
};
```
Seasoning

- `std::packaged_task` can be used to marshall results, including exceptions, from tasks
  - `std::packaged_task` is also useful to safely bridge C++ code with exceptions to C code
  - see prior `async()` implementation for an example
No Raw Pointers
What is a Raw Pointer?
What is a Raw Pointer?

- A pointer to an object with implied ownership and reference semantics
What is a Raw Pointer?

- A pointer to an object with implied ownership and reference semantics
  - \( T^* p = \text{new } T \)
What is a Raw Pointer?

- A pointer to an object with implied ownership and reference semantics
  - \( T^* \ p = \text{new } T \)
  - `unique_ptr<T>`
What is a Raw Pointer?

- A pointer to an object with implied ownership and reference semantics
  - T* p = new T
  - unique_ptr<T>
  - shared_ptr<T>
Why pointers (heap allocations)?

- Runtime variable size
  - Runtime polymorphic
  - Container
- Satisfy complexity or stability requirements within a container (list vs. vector)
- Shared storage for asynchronous communication (future, message queue, ...)
- Optimization to copy
  - Copy deferral (copy-on-write)
  - Immutable item
  - Transform Copy to Move [???]
- To separate implementation from interface (PIMPL)
Why Pointers
For containers we've moved from intrusive to non-intrusive (STL) containers
Why Pointers

- For containers we've moved from intrusive to non-intrusive (STL) containers
  - Except for hierarchies - but containment hierarchies or non-intrusive hierarchies are both viable options
Why Pointers

- For containers we've moved from intrusive to non-intrusive (STL) containers
  - Except for hierarchies - but containment hierarchies or non-intrusive hierarchies are both viable options
- PIMPL and copy optimizations are trivially wrapped
Why Pointers

- For containers we've moved from intrusive to non-intrusive (STL) containers
  - Except for hierarchies - but containment hierarchies or non-intrusive hierarchies are both viable options
- PIMPL and copy optimizations are trivially wrapped
- See previous section regarding shared storage for asynchronous operations
Why Pointers

- For containers we've moved from intrusive to non-intrusive (STL) containers
  - Except for hierarchies - but containment hierarchies or non-intrusive hierarchies are both viable options
- PIMPL and copy optimizations are trivially wrapped
- See previous section regarding shared storage for asynchronous operations

- Runtime polymorphism
using object_t = int;

void draw(const object_t& x, ostream& out, size_t position)
{ out << string(position, ' ') << x << endl; }

using document_t = vector<object_t>;

void draw(const document_t& x, ostream& out, size_t position)
{
    out << string(position, ' ') << "<document>" << endl;
    for (const auto& e : x) draw(e, out, position + 2);
    out << string(position, ' ') << "</document>" << endl;
}
```cpp
int main()
{
    document_t document;

document.emplace_back(0);
document.emplace_back(1);
document.emplace_back(2);
document.emplace_back(3);

draw(document, cout, 0);
}
```
```cpp
int main()
{
    document_t document;
    document.emplace_back(0);
    document.emplace_back(1);
    document.emplace_back(2);
    document.emplace_back(3);
    draw(document, cout, 0);
}

<document>
  0
  1
  2
  3
</document>
```
Polymorphism

- What happens if we want the document to hold any drawable object?
```cpp
class object_t {
    public:
        virtual ~object_t() {}
        virtual void draw(ostream&, size_t) const = 0;
    };

using document_t = vector<shared_ptr<object_t>>;

void draw(const document_t& x, ostream& out, size_t position)
{
    out << string(position, ' ') << "<document>" << endl;
    for (const auto& e : x) e->draw(out, position + 2);
    out << string(position, ' ') << "</document>" << endl;
}
```
class my_class_t : public object_t
{
    public:
        void draw(ostream& out, size_t position) const
        { out << string(position, ' ') << "my_class_t" << endl; }
        /* ... */
};

int main()
{
    document_t document;

    document.emplace_back(new my_class_t());

    draw(document, cout, 0);
}
class my_class_t : public object_t
{
    public:
        void draw(ostream& out, size_t position) const
        { out << string(position, ' ') << "my_class_t" << endl; }
    /* ... */
};

int main()
{
    document_t document;

    document.emplace_back(new my_class_t());

    draw(document, cout, 0);
}

<document>
  my_class_t
</document>
class my_class_t : public object_t
{
    public:
    void draw(ostream& out, size_t position) const
    { out << string(position, ' ') << "my_class_t" << endl; }
    /* ... */
};

int main()
{
    document_t document;
    document.emplace_back(new my_class_t());
    draw(document, cout, 0);
}
Deep problem

- Changed semantics of copy, assignment, and equality of my document
  - leads to incidental data structures
  - thread safety concerns
We define an operation in terms of the operation’s semantics:

“Assignment is a procedure taking two objects of the same type that makes the first object equal to the second without modifying the second.”
shared_ptr<T>  

T  

shared_ptr<T>
Semantics & Syntax

```
shared_ptr<T>

T

shared_ptr<T>
```
Considered as individual types, assignment and copy hold their regular semantic meanings.

However, this fails to account for the relationships (the arrows) which form an incidental data-structure. You cannot operate on T through one of the shared pointers without considering the effect on the other shared pointer.
shared_ptr<T>  

T  

shared_ptr<T>
If we extend our notion of object type to include the directly related part then we have intersecting objects which will interfere with each other.
When we consider the whole, the standard syntax for copy and assignment no longer have their regular semantics.

- This structure is still copyable and assignable but these operations must be done through other means.

- The shared structure also breaks our ability to reason locally about the code.
When we consider the whole, the standard syntax for copy and assignment no longer have their regular semantics.

- This structure is still copyable and assignable but these operations must be done through other means.

- The shared structure also breaks our ability to reason locally about the code.

A shared pointer is as good as a global variable.
template <typename T>
void draw(const T& x, ostream& out, size_t position)
{ out << string(position, ' ') << x << endl; }
template <typename T>
void draw(const T& x, ostream& out, size_t position)
{ out << string(position, ' ') << x << endl; }

class object_t {
    public:
        template <typename T>
        object_t(T x) : self_(make_shared<model<T>>(move(x))) { }

        friend void draw(const object_t& x, ostream& out, size_t position)
        { x.self_ ->draw_(out, position); }

    private:
        struct concept_t {
            virtual ~concept_t() = default;
            virtual void draw_(ostream&, size_t) const = 0;
        };
        template <typename T>
        struct model : concept_t {
            model(T x) : data_(move(x)) { }
            void draw_(ostream& out, size_t position) const
            { draw(data_, out, position); }

            T data_;}
};
void draw(const T& x, ostream& out, size_t position)
{ out << string(position, ' ') << x << endl; }

class object_t {
public:
  template <typename T>
  object_t(T x) : self_(make_shared<model<T>>(move(x))) { }

  friend void draw(const object_t& x, ostream& out, size_t position)
  { x.self_->draw_(out, position); }

private:
  struct concept_t {
    virtual ~concept_t() = default;
    virtual void draw_(ostream&, size_t) const = 0;
  };
  template <typename T>
  struct model : concept_t {
    model(T x) : data_(move(x)) { }
    void draw_(ostream& out, size_t position) const
    { draw(data_, out, position); }
    T data_;  
  };

```cpp
{ out << string(position, ' ') << x << endl; }

class object_t {
public:
    template <typename T>
    object_t(T x) : self_(make_shared<model<T>>(move(x))) { }

    friend void draw(const object_t& x, ostream& out, size_t position)
    { x.self_->draw_(out, position); }

private:
    struct concept_t {
        virtual ~concept_t() = default;
        virtual void draw_(ostream&, size_t) const = 0;
    };
    template <typename T>
    struct model : concept_t {
        model(T x) : data_(move(x)) { }
        void draw_(ostream& out, size_t position) const
        { draw(data_, out, position); }

        T data_;  
    };

    shared_ptr<concept_t> self_;  
};
```
class object_t {
public:
    template <typename T>
    object_t(T x) : self_(make_shared<model<T>>(move(x))) { }

    friend void draw(const object_t& x, ostream& out, size_t position)
    { x.self_->draw_(out, position); }

private:
    struct concept_t {
        virtual ~concept_t() = default;
        virtual void draw_(ostream&, size_t) const = 0;
    };
    template <typename T>
    struct model : concept_t {
        model(T x) : data_(move(x)) { }
        void draw_(ostream& out, size_t position) const
        { draw(data_, out, position); }

        T data_;
    };
    shared_ptr<const concept_t> self_; }
```cpp
class object_t {
public:
    template <typename T>
    object_t(T x) : self_(make_shared<model<T>>(move(x))) { }

    friend void draw(const object_t& x, ostream& out, size_t position)
    { x.self_->draw_(out, position); }

private:
    struct concept_t {
        virtual ~concept_t() = default;
        virtual void draw_(ostream&, size_t) const = 0;
    };

    template <typename T>
    struct model : concept_t {
        model(T x) : data_(move(x)) { }
        void draw_(ostream& out, size_t position) const
        { draw(data_, out, position); }

        T data_;
    };

    shared_ptr<const concept_t> self_;  
};
```
```cpp
class object_t {
    public:
        template<typename T>
        object_t(T x) : self_(make_shared<model<T>>(move(x))) { }

        friend void draw(const object_t& x, ostream& out, size_t position)
        { x.self_']->draw_(out, position); }

    private:
        struct concept_t {
            virtual ~concept_t() = default;
            virtual void draw_(ostream&, size_t) const = 0;
        };

        template<typename T>
        struct model : concept_t {
            model(T x) : data_(move(x)) { }
            void draw_(ostream& out, size_t position) const
            { draw(data_, out, position); }

                T data_; 
        };

        shared_ptr<const concept_t> self_; 
};
```
class object_t {
    public:
        template <typename T>
        object_t(T x) : self_(make_shared<model<T>>(move(x))) { }

        friend void draw(const object_t& x, ostream& out, size_t position)
        { x.self_->draw_(out, position); }

    private:
        struct concept_t {
            virtual ~concept_t() = default;
            virtual void draw_(ostream&, size_t) const = 0;
        };

        template <typename T>
        struct model : concept_t {
            model(T x) : data_(move(x)) { }
            void draw_(ostream& out, size_t position) const
            { draw(data_, out, position); }

            T data_;}

        shared_ptr<const concept_t> self_;};
class object_t {
    public:
    template <typename T>
    object_t(T x) : self_(make_shared<model<T>>(move(x))) {} 

    friend void draw(const object_t& x, ostream& out, size_t position)
    { x.self_->draw_(out, position); }

    private:
    struct concept_t {
        virtual ~concept_t() = default;
        virtual void draw_(ostream&, size_t) const = 0;
    };
    template <typename T>
    struct model : concept_t {
        model(T x) : data_(move(x)) {} 
        void draw_(ostream& out, size_t position) const
        { draw(data_, out, position); }
        T data_; 
    };
    
    shared_ptr<const concept_t> self_; 
};
public:
  template<typename T>
  object_t(T x) : self_(make_shared<model<T>>(move(x))) { }

  friend void draw(const object_t& x, ostream& out, size_t position)
  { x.self_->draw_(out, position); }

private:
  struct concept_t {
    virtual ~concept_t() = default;
    virtual void draw_(ostream&, size_t) const = 0;
  };
  template<typename T>
  struct model : concept_t {
    model(T x) : data_(move(x)) { }
    void draw_(ostream& out, size_t position) const
    { draw(data_, out, position); }
    T data_;  
  };

  shared_ptr<const concept_t> self_; 

using document_t = vector<object_t>;
template <typename T>
object_t(T x) : self_(make_shared<model<T>>(move(x))) { }

friend void draw(const object_t& x, ostream& out, size_t position)
{x.self_->draw_(out, position); }

private:
    struct concept_t {
        virtual ~concept_t() = default;
        virtual void draw_(ostream&, size_t) const = 0;
    };
    template <typename T>
    struct model : concept_t {
        model(T x) : data_(move(x)) { }
        void draw_(ostream& out, size_t position) const
        { draw(data_, out, position); }

        T data_;  
    };

    shared_ptr<const concept_t> self_; 
};

using document_t = vector<object_t> ;
friend void draw(const object_t& x, ostream& out, size_t position)
{ x.self_->draw_(out, position); }

private:
    struct concept_t {
        virtual ~concept_t() = default;
        virtual void draw_(ostream&, size_t) const = 0;
    };
    template <typename T>
    struct model : concept_t {
        model(T x) : data_(move(x)) { }
        void draw_(ostream& out, size_t position) const
        { draw(data_, out, position); }

        T data_;}

    shared_ptr<const concept_t> self_;}

using document_t = vector<object_t>;

void draw(const document_t& x, ostream& out, size_t position)
{
friend void draw(const object_t& x, ostream& out, size_t position)
{
    x.self_->draw_(out, position);
}

private:
    struct concept_t {
        virtual ~concept_t() = default;
        virtual void draw_(ostream&, size_t) const = 0;
    };
    template <typename T>
    struct model : concept_t {
        model(T x) : data_(move(x)) { }
        void draw_(ostream& out, size_t position) const
        {
            draw(data_, out, position);
        }
        T data_;  
    };

    shared_ptr<const concept_t> self_;  
};

using document_t = vector<object_t>;

void draw(const document_t& x, ostream& out, size_t position)
{
    out << string(position, ' ') << "<document>" << endl;
}
```cpp
{ x.self_->draw_(out, position); }

private:
    struct concept_t {
        virtual ~concept_t() = default;
        virtual void draw_(ostream&, size_t) const = 0;
    };

    template <typename T>
    struct model : concept_t {
        model(T x) : data_(move(x)) {}  
        void draw_(ostream& out, size_t position) const
        { draw(data_, out, position); }

        T data_
    }

    shared_ptr<const concept_t> self_
};

using document_t = vector<object_t>;

void draw(const document_t& x, ostream& out, size_t position)
{
    out << string(position, ' ') << "<document>" << endl;
    for (auto& e : x) draw(e, out, position + 2);
```

private:
struct concept_t {
    virtual ~concept_t() = default;
    virtual void draw_(ostream&, size_t) const = 0;
};
template <typename T>
struct model : concept_t {
    model(T x) : data_(move(x)) { }
    void draw_(ostream& out, size_t position) const
    { draw(data_, out, position); }
    T data_
};

shared_ptr<const concept_t> self_
};

using document_t = vector<object_t>;

void draw(const document_t& x, ostream& out, size_t position)
{
    out << string(position, ' ') << "<document>" << endl;
    for (auto& e : x) draw(e, out, position + 2);
    out << string(position, ' ') << "</document>" << endl;
}
private:
  struct concept_t {
    virtual ~concept_t() = default;
    virtual void draw_(ostream&, size_t) const = 0;
  };
  template <typename T>
  struct model : concept_t {
    model(T x) : data_(move(x)) { }
    void draw_(ostream& out, size_t position) const
    { draw(data_, out, position); }
    T data_;  
  };

  shared_ptr<const concept_t> self_; 
};

using document_t = vector<object_t>;

void draw(const document_t& x, ostream& out, size_t position)
{
  out << string(position, ' ') << "<document>" << endl;
  for (auto& e : x) draw(e, out, position + 2);
  out << string(position, ' ') << "</document>" << endl;
}
```cpp
class my_class_t {
    /* ... */
};

void draw(const my_class_t&, ostream& out, size_t position)
{ out << string(position, ' ') << "my_class_t" << endl; }

int main()
{
    document_t document;

    document.emplace_back(my_class_t());

    draw(document, cout, 0);
}
```
class my_class_t {
    /* ... */
};

void draw(const my_class_t&, ostream& out, size_t position)
{ out << string(position, ' ') << "my_class_t" << endl; }

int main()
{
    document_t document;

    document.emplace_back(my_class_t());

    draw(document, cout, 0);
}
class my_class_t {
  /* ... */
};

void draw(const my_class_t&, ostream& out, size_t position)
{ out << string(position, ' ') << "my_class_t" << endl; }

int main()
{
  document_t document;

  document.emplace_back(my_class_t());
  document.emplace_back(string("Hello World"));

  draw(document, cout, 0);
}
```cpp
class my_class_t {
    /* ... */
};

void draw(const my_class_t&, ostream& out, size_t position)
{ out << string(position, ' ') << "my_class_t" << endl; }

int main()
{
    document_t document;

    document.emplace_back(my_class_t());
    document.emplace_back(string("Hello World!"));

    draw(document, cout, 0);
}

<document>
  my_class_t
  Hello World!
</document>
```cpp
class my_class_t {
    /* ... */
};

void draw(const my_class_t&, ostream& out, size_t position)
{ out << string(position, ' ') << "my_class_t" << endl; }

int main()
{
    document_t document;

document.emplace_back(my_class_t());
document.emplace_back(string("Hello World"));
document.emplace_back(document);

draw(document, cout, 0);
}
```
# Client Library

```cpp
class my_class_t {
   /* ... */
};

void draw(const my_class_t&, ostream& out, size_t position)
{ out << string(position, ' ') << "my_class_t" << endl; }

int main()
{
    document_t document;
    document.emplace_back(my_class_t());
    document.emplace_back(string("Hello World!"));
    document.emplace_back(document);

    draw(document, cout, 0);
}
```

```xml
<document>
 my_class_t
 Hello World!
</document>
```
```cpp
class my_class_t {
    /* ... */
};

void draw(const my_class_t&, ostream& out, size_t position) {
    out << string(position, ' ') << "my_class_t" << endl; }

int main()
{
    document_t document;
    document.emplace_back(my_class_t());
    document.emplace_back(string("Hello World!"));

    auto saving = async([&]() {
        this_thread::sleep_for(chrono::seconds(3));
        cout << "-- save --" << endl;
        draw(document, cout, 0);
    });
    document.emplace_back(document);
    draw(document, cout, 0);
    saving.get();
}
```
class my_class_t {
  /* ... */
};

void draw(const my_class_t&, ostream& out, size_t position)
{ out << string(position, ' ') << "my_class_t" << endl; }

int main()
{
  document_t document;
  document.emplace_back(my_class_t());
  document.emplace_back(string("Hello World!"));

  -- save --
  draw(document, cout, 0);
  document.emplace_back(document);
  draw(document, cout, 0);
  saving.get();
}
Seasoning

- Using `make_shared<>` to create `shared_ptrs` eliminates an extra heap allocation

  ```cpp
  template <typename T> // T models Drawable
  object_t(T x) : self_(make_shared<model<T>>(move(x)))
  {
  }
  ```

- Pass `sink` arguments by value and move into place
Goals Recap

- No Raw Loops
- No Raw Syntonization Primitives
- No Raw Pointers
Locality of Reasoning

- Easier to reason about
- Composable
- General
- Correct
- Efficient
```cpp
template <typename I, // I models BidirectionalIterator
type_name S> // S models UnaryPredicate
auto gather(I f, I l, I p, S s) -> pair<I, I>
{
    using value_type = typename iterator_traits<I>::value_type;

    return { stable_partition(f, p, [&](const value_type& x){ return !s(x); }),
             stable_partition(p, l, s) };
}
```

- `not1` is not lambda friendly because of the `argument_type` requirement
- With C++ 14 we should be able to express this with a `const auto&` argument
  - Perhaps with a fixed `not1` or `!bind`
- The `BidirectionalIterator` requirement should be weakened to `ForwardIterator`
  - See SGI STL for an implementation
- The `gather()` function was developed with Marshall Clow and is in Boost
template<typename I, // I models BidirectionalIterator
typename S> // S models UnaryPredicate
auto gather(I f, I l, I p, S s) -> pair<I, I>
{
  using value_type = typename iterator_traits<I>::value_type;

  return { stable_partition(f, p, [&](const value_type& x){ return !s(x); }),
           stable_partition(p, l, s) };
}

- not1 is not lambda friendly because of the argument_type requirement
- With C++ 14 we should be able to express this with a const auto& argument
  - Perhaps with a fixed not1 or !bind
- The BidirectionalIterator requirement should be weakened to ForwardIterator
  - See SGI STL for an implementation
- The gather() function was developed with Marshall Clow and is in Boost
Stable Partition
Stable Partition

\[ \text{Stable Partition} \]

\[ \text{Diagram of Stable Partition} \]

- \( f \) -
- \( m \) -
- \( l \) -
Stable Partition
Stable Partition

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

stable_partition(f, m, p)
stable_partition(m, l, p)
Stable Partition

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

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));
}
// For illustration only

class group {
public:
    template <typename F>
    void async(F&& f) {
        auto then = then_;
        thread(bind([then](F& f){ f(); }, std::forward<F>(f))).detach();
    }

    template <typename F>
    void then(F&& f) {
        then_->f_ = forward<F>(f);
        then_.reset();
    }

private:
    struct packaged {
        ~packaged() { thread(bind(move(f_))).detach(); }
        function<void()> f_; 
    };

    shared_ptr<packaged> then_ = make_shared<packaged>();
};
int main()
{
    group g;
    g.async([]() {
        this_thread::sleep_for(chrono::seconds(2));
        cout << "task 1" << endl;
    });

    g.async([]() {
        this_thread::sleep_for(chrono::seconds(1));
        cout << "task 2" << endl;
    });

    g.then([](){
        cout << "done!" << endl;
    });

    this_thread::sleep_for(chrono::seconds(10));
}
```cpp
int main()
{
    group g;
    g.async([]() {
        this_thread::sleep_for(chrono::seconds(2));
        cout << "task 1" << endl;
    });

    g.async([]() {
        this_thread::sleep_for(chrono::seconds(1));
        cout << "task 2" << endl;
    });

    g.then([](){
        cout << "done!" << endl;
    });

    cout << endl;
    this_thread::sleep_for(chrono::seconds(10));
}
```
// For illustration only

class group {
public:
  template <typename F, typename ...Args>
  auto async(F&& f, Args&&... args)
  -> future<typename result_of<F (Args...)>::type>
  {
    using result_type = typename std::result_of<F (Args...)>::type;
    using packaged_type = std::packaged_task<
      result_type ()>;

    auto p = packaged_type(forward<F>(f), forward<Args>(args)...);
    auto result = p.get_future();

    auto then = then_
    thread(bind([then](packaged_type& p){ p(); }, move(p))).detach();

    return result;
  }

  template <typename F, typename ... Args>
  auto then(F&& f, Args&&... args)
  -> future<typename result_of<F (Args...)>::type>
  {
    using result_type = typename std::result_of<F (Args...)>::type;
    using packaged_type = std::packaged_task<
      result_type ()>;
  

```cpp
class group {
public:
    template <typename F, typename ...Args>
    auto async(F&& f, Args&&... args)
        -> future<typename result_of<F (Args...)>::type>
    {
        using result_type = typename std::result_of<F (Args...)>::type;
        using packaged_type = std::packaged_task<result_type>();
        auto p = packaged_type(forward<F>(f), forward<Args>(args)...);
        auto result = p.get_future();
        auto then = then_;
        thread(bind([then](packaged_type& p){ p(); }, move(p))).detach();
        return result;
    }

template <typename F, typename ...Args>
auto then(F&& f, Args&&... args)
    -> future<typename result_of<F (Args...)>::type>
    {
        using result_type = typename std::result_of<F (Args...)>::type;
        using packaged_type = std::packaged_task<result_type>();
    }
};
```
public:
  template <typename F, typename ...Args>
  auto async(F&& f, Args&&... args)
    -> future<typename result_of<F (Args...)>::type>
  {
    using result_type = typename std::result_of<F (Args...)>::type;
    using packaged_type = std::packaged_task<result_type ()>;
    auto p = packaged_type(forward<F>(f), forward<Args>(args)...);
    auto result = p.get_future();
    auto then = then_;
    thread(bind([then](packaged_type& p){ p(); }, move(p))).detach();
    return result;
  }

  template <typename F, typename ...Args>
  auto then(F&& f, Args&&... args)
    -> future<typename result_of<F (Args...)>::type>
  {
    using result_type = typename std::result_of<F (Args...)>::type;
    using packaged_type = std::packaged_task<result_type ()>;
    auto p = packaged_type(forward<F>(f), forward<Args>(args)...);
template <typename F, typename ...Args>
auto async(F&& f, Args&&... args)
-> future<typename result_of<F (Args...)>::type>
{
    using result_type = typename std::result_of<F (Args...)>::type;
    using packaged_type = std::packaged_task<result_type ()>;

    auto p = packaged_type(forward<F>(f), forward<Args>(args)...);
    auto result = p.get_future();

    auto then = then_;
    thread(bind([then](packaged_type& p){ p(); }, move(p))).detach();

    return result;
}

template <typename F, typename ...Args>
auto then(F&& f, Args&&... args)
-> future<typename result_of<F (Args...)>::type>
{
    using result_type = typename std::result_of<F (Args...)>::type;
    using packaged_type = std::packaged_task<result_type ()>;

    auto p = packaged_type(forward<F>(f), forward<Args>(args)...);
    auto result = p.get_future();
auto async(F&& f, Args&&... args)
-> future<typename result_of<F (Args...)>::type>
{
    using result_type = typename std::result_of<F (Args...)>::type;
    using packaged_type = std::packaged_task<result_type ()>;

    auto p = packaged_type(forward<F>(f), forward<Args>(args)...);
    auto result = p.get_future();

    auto then = then_;
    thread(bind([then](packaged_type& p){ p(); }, move(p))).detach();

    return result;
}

template <typename F, typename ...Args>
auto then(F&& f, Args&&... args)
-> future<typename result_of<F (Args...)>::type>
{
    using result_type = typename std::result_of<F (Args...)>::type;
    using packaged_type = std::packaged_task<result_type ()>;

    auto p = packaged_type(forward<F>(f), forward<Args>(args)...);
    auto result = p.get_future();
-> future<typename result_of<F (Args...)>::type>
{
    using result_type = typename std::result_of<F (Args...)>::type;
    using packaged_type = std::packaged_task<result_type> ();

    auto p = packaged_type(forward<F>(f), forward<Args>(args)...);
    auto result = p.get_future();

    auto then = then_;
    thread(bind([then](packaged_type& p){ p(); }, move(p))).detach();

    return result;
}

template <typename F, typename ...Args>
auto then(F&& f, Args&&... args)
    -> future<typename result_of<F (Args...)>::type>
{
    using result_type = typename std::result_of<F (Args...)>::type;
    using packaged_type = std::packaged_task<result_type> ();

    auto p = packaged_type(forward<F>(f), forward<Args>(args)...);
    auto result = p.get_future();

    then_ -> reset(new packaged<packaged_type>(move(p)));
}
```cpp
{  
  using result_type = typename std::result_of<F (Args...)>::type;
  using packaged_type = std::packaged_task<result_type ()>;

  auto p = packaged_type(forward<F>(f), forward<Args>(args)...);
  auto result = p.get_future();

  auto then = then_;  
  thread(bind([&then](packaged_type & p){ p(); }, move(p))).detach();

  return result;
}

template <typename F, typename ...Args>
auto then(F&& f, Args&&... args) -> future<typename result_of<F (Args...)>::type>
{
  using result_type = typename std::result_of<F (Args...)>::type;
  using packaged_type = std::packaged_task<result_type ()>;

  auto p = packaged_type(forward<F>(f), forward<Args>(args)...);
  auto result = p.get_future();

  then_ -> reset(new packaged<packaged_type>(move(p)));
  then_ = nullptr;
}
```cpp
using result_type = typename std::result_of<F (Args...)>::type;
using packaged_type = std::packaged_task<result_type ()>;

auto p = packaged_type(forward<F>(f), forward<Args>(args)...);
auto result = p.get_future();

auto then = then_;
thread(bind([&then](packaged_type& p){ p(); }, move(p))).detach();

return result;
```

```cpp
template <typename F, typename ...Args>
auto then(F&& f, Args&&... args) -> future<typename result_of<F (Args...)>::type>
{
    using result_type = typename std::result_of<F (Args...)>::type;
    using packaged_type = std::packaged_task<result_type ()>;

    auto p = packaged_type(forward<F>(f), forward<Args>(args)...);
    auto result = p.get_future();

    then_ = reset(new packaged<packaged_type>(move(p)));
    then_ = nullptr;
}
using packaged_type = std::packaged_task<result_type ()>;

auto p = packaged_type(forward<F>(f), forward<Args>(args)...);
auto result = p.get_future();

auto then = then_;

thread(bind([&then](packaged_type& p) { p(); }, move(p))).detach();

return result;

}
auto p = packaged_type(forward<F>(f), forward<Args>(args)...);
auto result = p.get_future();

auto then = then_;
thread(bind([&then](packaged_type& p){ p(); }, move(p))).detach();

return result;
}

template <typename F, typename ...Args>
auto then(F&& f, Args&&... args)
  -> future<typename result_of<F (Args...)>::type>
{
  using result_type = typename std::result_of<F (Args...)>::type;
  using packaged_type = std::packaged_task<result_type>();

  auto p = packaged_type(forward<F>(f), forward<Args>(args)...);
auto result = p.get_future();

  then_ = reset(new packaged<packaged_type>(move(p)));
  then_ = nullptr;

  return result;
}
auto p = packaged_type(forward<F>(f), forward<Args>(args)...);
auto result = p.get_future();

auto then = then_
thread(bind([&then](packaged_type& p){ p(); }, move(p))).detach();

return result;

}
auto result = p.get_future();

auto then = then_;
thread(bind([&then](packaged_type& p){ p(); }, move(p))).detach();

return result;

}  

 template<typename F, typename ...Args>
auto then(F&& f, Args&&... args) -> future<typename result_of<F (Args...)>::type> 
{
    using result_type = typename std::result_of<F (Args...)>::type;
    using packaged_type = std::packaged_task<result_type>();

    auto p = packaged_type(forward<F>(f), forward<Args>(args)...);
    auto result = p.get_future();
    then_->reset(new packaged<packaged_type>(move(p)));
    then_ = nullptr;

    return result;

}  

private:
auto then = then_;
thread(bind([then](){ p(); }, move(p))).detach();

return result;
}

template<
type F, typename ...Args>
auto then(F&& f, Args&&... args)
-> future<typename result_of<F (Args...)>::type>
{
  using result_type = typename std::result_of<F (Args...)>::type;
  using packaged_type = std::packaged_task<result_type ()>;

  auto p = packaged_type(forward<F>(f), forward<Args>(args)...);
  auto result = p.get_future();

  then_ = nullptr;

  return result;
}

private:
  struct any_packaged {

auto then = then_;
thread(bind([&then](packaged_type& p){ p(); }, move(p))).detach();

return result;
}

template<typename F, typename ...Args>
auto then(F&& f, Args&&... args)
    -> future<typename result_of<F (Args...)>::type>
{
    using result_type = typename std::result_of<F (Args...)>::type;
    using packaged_type = std::packaged_task<result_type>();

    auto p = packaged_type(forward<F>(f), forward<Args>(args)...);
    auto result = p.get_future();

    then_ = nullptr;

    return result;
}

private:
    struct any_packaged {
        virtual ~any_packaged() = default;
thread(bind([then](packaged_type& p){ p(); }, move(p))).detach();

return result;
}

template <typename F, typename ...Args>
auto then(F&& f, Args&&... args)
-> future<typename result_of<F (Args...)>::type>
{
    using result_type = typename std::result_of<F (Args...)>::type;
    using packaged_type = std::packaged_task<result_type>();

    auto p = packaged_type(forward<F>(f), forward<Args>(args)...);
    auto result = p.get_future();

    then_ = nullptr;

    return result;
}

private:
    struct any_packaged {
        virtual ~any_packaged() = default;
    };
return result;
}

template <typename F, typename ...Args>
auto then(F&& f, Args&&... args)
    -> future<typename result_of<F (Args...)>::type>
{
    using result_type = typename std::result_of<F (Args...)>::type;
    using packaged_type = std::packaged_task<result_type>();

    auto p = packaged_type(forward<F>(f), forward<Args>(args)...);
    auto result = p.get_future();

    then_->reset(new packaged<packaged_type>(move(p)));
    then_ = nullptr;

    return result;
}

private:
    struct any_packaged {
        virtual ~any_packaged() = default;
    };
return result;
}

template <typename F, typename ...Args>
auto then(F&& f, Args&&... args)
  -> future<typename result_of<F (Args...)>::type>
{
  using result_type = typename std::result_of<F (Args...)>::type;
  using packaged_type = std::packaged_task<result_type ()>;

  auto p = packaged_type(forward<F>(f), forward<Args>(args)...);
  auto result = p.get_future();

  then_ = reset(new packaged<packaged_type>(move(p)));

  then_ = nullptr;

  return result;
}

private:
struct any_packaged {
  virtual ~any_packaged() = default;
};

template <typename P>
template <typename F, typename ...Args>
auto then(F&& f, Args&&... args)
    -> future<typename result_of<F (Args...)>::type>
{
    using result_type = typename std::result_of<F (Args...)>::type;
    using packaged_type = std::packaged_task<result_type ()>;

    auto p = packaged_type(forward<F>(f), forward<Args>(args)...);
    auto result = p.get_future();

    then_->reset(new packaged<packaged_type>(move(p)));
    then_ = nullptr;

    return result;
}

private:
    struct any_packaged {
        virtual ~any_packaged() = default;
    };

    template <typename P>
    struct packaged : any_packaged {

template <typename F, typename ...Args>
auto then(F&& f, Args&&... args)
    -> future<typename result_of<F (Args...)>::type>
{
    using result_type = typename std::result_of<F (Args...)>::type;
    using packaged_type = std::packaged_task<result_type>();

    auto p = packaged_type(forward<F>(f), forward<Args>(args)...);
    auto result = p.get_future();

    then_->reset(new packaged<packaged_type>(move(p)));
    then_ = nullptr;

    return result;
}

private:
struct any_packaged {
    virtual ~any_packaged() = default;
};

template <typename P>
struct packaged : any_packaged {
    packaged(P&& f) : f_(move(f)) { }
};
```cpp
template<typename F, typename ...Args>
auto then(F&& f, Args&&... args)
    -> future<typename result_of<F (Args...)>::type>
{
    using result_type = typename std::result_of<F (Args...)>::type;
    using packaged_type = std::packaged_task<result_type>();

    auto p = packaged_type(forward<F>(f), forward<Args>(args)...);
    auto result = p.get_future();

    then_ = reset(new packaged<packaged_type>(move(p)));
    then_ = nullptr;

    return result;
}

private:
    struct any_packaged {
        virtual ~any_packaged() = default;
    };

    template<typename P>
    struct packaged : any_packaged {
        packaged(P&& f) : f_(move(f)) {} 
        ~packaged() { thread(bind(move(f_))).detach(); }
    }
```
auto then(F&& f, Args&&... args)
-> future<typename result_of<F (Args...)>::type>
{
    using result_type = typename std::result_of<F (Args...)>::type;
    using packaged_type = std::packaged_task<result_type>();

    auto p = packaged_type(forward<F>(f), forward<Args>(args)...);
    auto result = p.get_future();

    then_ = new packaged<packaged_type>(move(p));
    then_ = nullptr;

    return result;
}

private:
struct any_packaged {
    virtual ~any_packaged() = default;
};

template <typename P>
struct packaged : any_packaged {
    packaged(P&& f) : f_(move(f)) { }
    ~packaged() { thread(bind(move(f_))).detach(); }
-> future<typename result_of<F (Args...)>::type>
{
    using result_type = typename std::result_of<F (Args...)>::type;
    using packaged_type = std::packaged_task<result_type>();

    auto p = packaged_type(forward<F>(f), forward<Args>(args)...);
    auto result = p.get_future();

    then_ -> reset(new packaged<packaged_type>(move(p)));
    then_ = nullptr;

    return result;
}

private:
struct any_packaged {
    virtual ~any_packaged() = default;
};

template <typename P>
struct packaged : any_packaged {
    packaged(P&& f) : f_(move(f)) { }
    ~packaged() { thread(bind(move(f_))).detach(); }  
    P f_;
```cpp
{  
    using result_type = typename std::result_of<F (Args...)>::type;  
    using packaged_type = std::packaged_task<result_type>();  
    auto p = packaged_type(forward<F>(f), forward<Args>(args)...);  
    auto result = p.get_future();  
    then_->reset(new packaged<packaged_type>(move(p)));  
    then_ = nullptr;  
    return result;  
}

private:  
    struct any_packaged {  
        virtual ~any_packaged() = default;  
    };

    template<typename P>  
    struct packaged : any_packaged {  
        packaged(P&& f) : f_(move(f)) { }  
        ~packaged() { thread(bind(move(f_))).detach(); }  
        P f_;  
    };
```
using result_type = typename std::result_of<F (Args...)>::type;
using packaged_type = std::packaged_task<result_type>();

auto p = packaged_type(forward<F>(f), forward<Args>(args)...);
auto result = p.get_future();
then_ = nullptr;
return result;
}

private:
struct any_packaged {
    virtual ~any_packaged() = default;
};

template <typename P>
struct packaged : any_packaged {
    packaged(P&& f) : f_(move(f)) { }
    ~packaged() { thread(bind(move(f_))).detach(); }
    P f_;
};
```cpp
using packaged_type = std::packaged_task<result_type>();

auto p = packaged_type(forward<F>(f), forward<Args>(args)...);
auto result = p.get_future();

then_ -> reset(new packaged<packaged_type>(move(p)));
then_ = nullptr;

return result;
}

private:
struct any_packaged {
    virtual ~any_packaged() = default;
};

template <typename P>
struct packaged : any_packaged {
    packaged(P&& f) : f_(move(f)) {}
    ~packaged() { thread(bind(move(f_))).detach(); }
    P f_; 
};

shared_ptr<unique_ptr<any_packaged>> then_ = make_shared<unique_ptr<any_packaged>>();
```
auto p = packaged_type(forward<F>(f), forward<Args>(args)...);
auto result = p.get_future();

then_—>reset(new packaged_type<packaged_type>(move(p)));
then_ = nullptr;

return result;

private:
struct any_packaged {
    virtual ~any_packaged() = default;
};

template <typename P>
struct packaged : any_packaged {
    packaged(P&& f) : f_(move(f)) { }
    ~packaged() { thread(bind(move(f_))).detach(); } 
    P f_; 
};

shared_ptr<unique_ptr<any_packaged>> then_ = make_shared<unique_ptr<any_packaged>>();
```cpp
int main()
{
    group g;

    auto x = g.async([]() {
        this_thread::sleep_for(chrono::seconds(2));
        cout << "task 1" << endl;
        return 10;
    });

    auto y = g.async([]() {
        this_thread::sleep_for(chrono::seconds(1));
        cout << "task 2" << endl;
        return 5;
    });

    auto r = g.then(bind([](future<int>& x, future<int>& y) {
        cout << "done:" << (x.get() + y.get()) << endl;
    }, move(x), move(y)));

    r.get();
}
```
```cpp
int main()
{
    group g;

    auto x = g.async([]() {
        this_thread::sleep_for(chrono::seconds(2));
        cout << "task 1" << endl;
        return 10;
    });

    auto y = g.async([]() {
        this_thread::sleep_for(chrono::seconds(1));
        cout << "task 2" << endl;
        return 5;
    });

    auto r = g.then(bind([](future<int>& x, future<int>& y) {
        cout << "done:" << (x.get() + y.get()) << endl;
        move(y));
    }, move(x), move(y));
}
```