Object-Oriented Programming

# Generalizing Functions

Michael L. Collard, Ph.D.

Department of Computer Science, The University of Akron

# Generalizing Functions

- The term *generalization* is overloaded

- For now, referring to making functions applicable to more situations

- Very related to *concerns*

# Notice

*The following examples are, in many cases, **not** good design examples*

# Extremely Limited Sort ☹

```
void sort() {

    std::vector<int> v{ 5, 3, 1, 2, 4 };

    // sort v
    for (int i = 0; i < v.size(); ++i) {

        // locate smallest from v[i]..v[v.size() - 1]
        int smallPos = i;
        for (int j = i; j < v.size(); ++j) {
            if (v[j] < v[smallPos])
                smallPos = j;
        }

        // vector is now sorted from v[0]...v[i]
        std::swap(v[smallPos], v[i]);
    }
}
```

- **Concern**

The vector to sort

- **Assumption**

Sort is for a hard-coded vector

# Good Sort Interface 🙂

```
void sort(std::vector<int>& v) {

    // sort v
    for (int i = 0; i < v.size(); ++i) {

        // locate smallest from v[i]..v[v.size() - 1]
        int smallPos = i;
        for (int j = i; j < v.size(); ++j) {
            if (v[j] < v[smallPos])
                smallPos = j;
        }

        // vector is now sorted from v[0]...v[i]
        std::swap(v[smallPos], v[i]);
    }
}
```

- **Concern**

  The vector to sort

- **Generalization**

  Replace hard-coded vector with a parameter

  *parameterize* the vector

# Good Sort Interface Scenario😊

```cpp
std::vector<int> responses{3, 2, 1, 4, 5};

// sort responses
sort(responses);
```

# Good Sort Interface Scenario😕

```cpp
std::vector<int> responses{3, 2, 1, 4, 5};

// sort prefix of responses

// copy prefix of responses
std::vector<int> prefix{ responses.begin(), responses.begin() + 3 };

// sort prefix
sort(prefix);

// overwrite responses with new prefix
std::copy(prefix.begin(), prefix.end, responses.begin());
```

- **Concern**

  The vector to sort

- **Assumption**

  Sort is for the entire vector

# Better Sort Interface 😊

```
void sort(std::vector<int>::iterator begin,
          std::vector<int>::iterator end) {

    // sort [begin, end)
    for (auto p = begin; p != end; ++p) {

        // locate smallest from [p, end)
        auto smallPos = p;
        for (auto q = p; q != end; ++q) {
            if (*q < *smallPos)
                smallPos = q;
        }

        // swap the smallest from [p, end) to p
        std::swap(*smallPos, *p);

        // vector is now sorted from [begin, p)
        assert(std::is_sorted(begin, p));
    }
    assert(std::is_sorted(begin, end));
}
```

- **Concern**

  The vector to sort

- **Generalization**

  Replace complete vector with *iterators*

  *iterators* can refer to the entire vector or a subrange of the vector

  More flexible parameterization

- The C++ standard algorithm library takes *iterators*, not complete *containers*

# Better Sort Interface Scenario 😊

```cpp
std::vector<int> responses{3, 2, 1, 4, 5};

// sort responses
sort(responses.begin(), responses.end());
```

```cpp
std::vector<int> responses{3, 2, 1, 4, 5};

// sort prefix of responses
sort(responses.begin(), responses.begin() + 3);
```

# Better Sort Interface Scenario☹

```
std::vector<int> responses{3, 2, 1, 4, 5};

// sort responses in descending order
sort(/* ??? */);
```

# Solution for Descending Sort: Flag Parameter☹

```
void sort(std::vector<int>::iterator begin,
          std::vector<int>::iterator end,
          bool ascending);
```

# Solution for Descending Sort: Separate Functions🙂

```
void sortAscending(std::vector<int>::iterator begin,
                   std::vector<int>::iterator end);

void sortDescending(std::vector<int>::iterator begin,
                    std::vector<int>::iterator end);
```

# Solution to Descending Sort: Pass Comparison😊

```
void sort(std::vector<int>::iterator begin,
          std::vector<int>::iterator end,
          bool(*compare)(int, int)) {

    // sort [begin, end) using compare
    for (auto p = begin; p != end; ++p) {

        // locate smallest from [p, end)
        auto smallPos = p;
        for (auto q = p; q != end; ++q) {
            if (compare(*q, *smallPos))
                smallPos = q;
        }

        // swap the smallest from [p, end) to p
        std::swap(*smallPos, *p);

        // vector is now sorted from [begin, p)
        assert(std::is_sorted(begin, p));
    }
    assert(std::is_sorted(begin, end));
}
```

# Pass Comparison C++ Template

```cpp
template <typename Compare>
void sort(std::vector<int>::iterator begin,
          std::vector<int>::iterator end,
          Compare compare) {

    // sort v
    for (auto p = begin; p < end; ++p) {

        // locate smallest from [p, end)
        auto smallPos = p;
        for (auto q = p; q < end; ++q) {
            if (compare(*q, *smallPos))
                smallPos = q;
        }

        // swap the smallest from [p, end) to p
        std::swap(*smallPos, *p);

        // vector is now sorted from [begin, p)
        assert(std::is_sorted(begin, p));
    }
    assert(std::is_sorted(begin, end));
}
```

# Extremely Limited Binary Search ☹

```
std::vector<int>::const_iterator search(std::vector<int>::const_iterator begin,
                                        std::vector<int>::const_iterator end) {

    auto currentEnd = end;
    while (std::distance(begin, currentEnd) > 0) {

        // middle
        const auto middle = std::next(begin, std::distance(begin, currentEnd) / 2);

        if (5 < *middle) {
            // 5 in left half
            currentEnd = middle;
        } else if (5 > *middle) {
            // 5 in right half
            begin = middle + 1;
        } else {
            // 5 found
            return middle;
        }
    }

    // 5 not found
    return end;
}
```

# Better Binary Search 🙂

```cpp
std::vector<int>::const_iterator search(std::vector<int>::const_iterator begin,
                                        std::vector<int>::const_iterator end,
                                        int value) {
    auto currentEnd = end;
    auto currentBegin = begin;
    while (std::distance(currentBegin, currentEnd) > 0) {

        // middle
        const auto middle = std::next(currentBegin, std::distance(currentBegin, currentEnd) / 2);

        if (value < *middle) {
            // value in left half
            currentEnd = middle;
        } else if (value > *middle) {
            // value in right half
            currentBegin = middle + 1;
        } else {
            // value found
            return middle;
        }
    }

    // value not found
    return end;
}
```

## Generalizing a Function

- Change from assuming using all of a container, e.g., `std::vector<int>& v`, to iterators

- Change a literal value (which is an assumption) into a parameter

# Better Sort Interface 😊 (Revisited)

```
void sort(std::vector<int>::iterator begin,
          std::vector<int>::iterator end) {

    // sort [begin, end)
    for (auto p = begin; p != end; ++p) {

        // locate smallest from [p, end)
        auto smallPos = p;
        for (auto q = p; q != end; ++q) {
            if (*q < *smallPos)
                smallPos = q;
        }

        // swap the smallest from [p, end) to p
        std::swap(*smallPos, *p);

        // vector is now sorted from [begin, p)
        assert(std::is_sorted(begin, p));
    }
    assert(std::is_sorted(begin, end));
}
```

# Better Sort Interface Scenario

```
std::vector<int> responses{3, 2, 1, 4, 5};

// sort every other value starting with first

// cope the even elements out of the responses
std::vector<int> even;
for (auto p = responses.begin(); p != responses.end(); ++2)
    even.push_back(*p);

// sort even elements
sort(even.begin(), even.end());

// overwrite even elements in responses with sorted even elements
auto peven = even.begin();
for (auto p = responses.begin(); p != responses.end(); ++2) {
    *p = *peven;
    ++peven;
}
```

- Sort only the responses in the even positions
- Call this a *stride* of 2
- The C++ Standard Library does not generally provide stride versions
- C++23 has the `std::ranges::stride_view()`. We will cover this more with `std::ranges`.
- Many parallel, numerical computing, and scientific computing libraries have the option to use a stride

# Sort with Stride 😇

```
void sort(std::vector<int>::iterator begin,
          std::vector<int>::iterator end,
          int stride) {

    // sort v
    for (auto p = begin; p < end; p += stride) {

        // locate smallest from [p, end)
        auto smallPos = p;
        for (auto q = p; q < end; q += stride) {
            if (*q < *smallPos)
                smallPos = q;
        }

        // swap the smallest from [p, end) to p
        std::swap(*smallPos, *p);

        // vector is now sorted from [begin, p)
        assert(std::is_sorted(begin, p));
    }
    assert(std::is_sorted(begin, end));
}
```

# Stride vs. No Stride 😊

```
void sort(std::vector<int>::iterator begin,
          std::vector<int>::iterator end,
          int stride) {

    // sort v
    for (auto p = begin; p < end; p += stride) {

        // locate smallest from [p, end)
        auto smallPos = p;
        for (auto q = p; q < end; q += stride) {
            if (*q < *smallPos)
                smallPos = q;
        }

        // swap the smallest from [p, end) to p
        std::swap(*smallPos, *p);

        // vector is now sorted from [begin, p)
        assert(std::is_sorted(begin, p));
    }
    assert(std::is_sorted(begin, end));
}
```

```
void sort(std::vector<int>::iterator begin,
          std::vector<int>::iterator end) {

    // sort [begin, end)
    for (auto p = begin; p != end; ++p) {

        // locate smallest from [p, end)
        auto smallPos = p;
        for (auto q = p; q != end; ++q) {
            if (*q < *smallPos)
                smallPos = q;
        }

        // swap the smallest from [p, end) to p
        std::swap(*smallPos, *p);

        // vector is now sorted from [begin, p)
        assert(std::is_sorted(begin, p));
    }
    assert(std::is_sorted(begin, end));
}
```

# Reuse 😇

```cpp
void sort(std::vector<int>& v) {
    sort(v.begin(), v.end(), 1);
}

void sort(std::vector<int>::iterator begin, std::vector<int>::iterator end) {
    sort(begin, end, 1);
}
```

## Questions to Ask

- *Are we assuming a constant/literal that could change?*

- *Are we assuming it is applied to the entire data set?*

## Even More Assumptions

- Assumption: Container is `std::vector`
- Assumption: Container elements are `int`
- Assumption: Sorting in ascending order, i.e., smallest to largest
- Solution: Generalize *types* using *templates*
- Solution: Generalize comparison using *templates*

# Viewpoint of Concerns

- Why should the search() care about a fixed value? Only cares about the value type `int`

- Why should the sort() assume we want to sort the entire container?

- Why should the sort() assume we want a stride of 1?