Generic Programming and the STL

CS 211

Problem: finding a vector's maximum element

A simple fixed-size vector struct:

```
struct Int_vec
{
    int*   data;
    size_t size;
};
```

Solution: max_int_vec

```
// Finds the index of the maximum element in vec.
//  - If vec is empty returns 0.
//  - If the maximum element repeats, returns the index of the
//    first occurrence.
size_t max_int_vec(Int_vec const& vec)
{
    size_t best = 0;

    for (size_t i = 1; i < vec.size; ++i)
        if (vec.data[best] < vec.data[i])
            best = i;

    return best;
}
```

Testing max_int_vec

```
TEST_CASE("max_int_vec")
{
    int data[] = { 2, 0, 5, 3, 9, 5, 1 };
    Int_vec v{data, 7};

    CHECK( max_int_vec(v) == 4 );
}
```

Problem: finding a linked list’s maximum element

A simple linked list:

```
struct Int_node
{
    int                         data;
    std::shared_ptr<Int_node> next;
};
```

Problem: finding a linked list's maximum element

A simple linked list:

```
struct Int_node
{
    int                             data;
    std::shared_ptr<Int_node> next;
};

using Int_list = std::shared_ptr<Int_node>;
```

Problem: finding a linked list’s maximum element

A simple linked list:

```
struct Int_node
{
    int                       data;
    std::shared_ptr<Int_node> next;
};

using Int_list = std::shared_ptr<Int_node>;

Int_list cons(int data, Int_list next)
{
    return std::make_shared<Int_node>({data, next});
}
```

Solution: max_int_list

```c
// Finds the link to the node containing the
// maximum element.
//  - If empty, returns the null pointer.
//  - If the maximum repeats, returns the first occurrence.
Int_list max_int_list(Int_list lst)
{
    Int_list best = lst;

    for (Int_list i = lst; i; i = i->next)
        if (best->data < i->data)
            best = i;

    return best;
}
```

Testing `max_int_list`

```
TEST_CASE("max_int_list")
{
    Int_list exp =
        cons(9, cons(5, cons(1, nullptr)));
    Int_list lst =
        cons(2, cons(0, cons(5, cons(3, exp))));

    CHECK( max_int_list(lst) == exp );
}
```

Making our code more general

To make our code more general (and thus more reusable):

- Make the data structures generic over the element types
- Make the algorithm generic over the data structures

Generic fixed-size vector

```
template <typename T>
struct Vec
{
    T*     data;
    size_t size;
};
```

Generic max_vec

```
template <typename T>
size_t max_vec(Vec<T> const& vec)
{
    size_t best = 0;

    for (size_t i = 1; i < vec.size; ++i)
        if (vec.data[best] < vec.data[i])
            best = i;

    return best;
}
```

Generic linked list

```
template <typename T>
struct Node
{
    T                        data;
    std::shared_ptr<Node<T>> next;
};
```

Generic linked list

```
template <typename T>
struct Node
{
    T                         data;
    std::shared_ptr<Node<T>> next;
};

template <typename T>
using List = std::shared_ptr<Node<T>>;

template <typename T>
List<T> cons(T const& data, List<T> next)
{
    return std::make_shared<Node<T>>({data, next});
}
```

Generic max_list

```
template <typename T>
List<T> max_list(List<T> const& lst)
{
    List<T> best = lst;

    for (List<T> i = lst; i; i = i->next)
        if (best->data < i->data)
            best = i;

    return best;
}
```

Introducing the Standard Template Library

- Includes containers like `std::vector<T>`, `std::list<T>` (a doubly-linked list), and more
- Containers have *iterators* for traversing them
- An iterator is like a pointer to one element of a container

Vector iterators

- Like pointers to vector elements

Vector iterators

- Like pointers to vector elements
- `v.begin()` returns an iterator to the beginning of `v`

Vector iterators

- Like pointers to vector elements
- `v.begin()` returns an iterator to the beginning of `v`
- `v.end()` returns an iterator to one past the end of `v`

Vector iterators

- Like pointers to vector elements
- `v.begin()` returns an iterator to the beginning of `v`
- `v.end()` returns an iterator to one past the end of `v`
- `*i` dereferences an iterator `i`

Vector iterators

- Like pointers to vector elements
- `v.begin()` returns an iterator to the beginning of `v`
- `v.end()` returns an iterator to one past the end of `v`
- `*i` dereferences an iterator `i`
- `++i` advances an iterator `i` to the next element

Vector iterators

- Like pointers to vector elements
- `v.begin()` returns an iterator to the beginning of `v`
- `v.end()` returns an iterator to one past the end of `v`
- `*i` dereferences an iterator `i`
- `++i` advances an iterator `i` to the next element
- (and more...)

List iterators

- Like pointers to list elements

List iterators

- Like pointers to list elements
- `lst.begin()` returns an iterator to the beginning of `lst`

List iterators

- Like pointers to list elements
- `lst.begin()` returns an iterator to the beginning of `lst`
- `lst.end()` returns an iterator to one past the end of `lst`

List iterators

- Like pointers to list elements
- `lst.begin()` returns an iterator to the beginning of `lst`
- `lst.end()` returns an iterator to one past the end of `lst`
- `*i` dereferences an iterator `i`

List iterators

- Like pointers to list elements
- `lst.begin()` returns an iterator to the beginning of `lst`
- `lst.end()` returns an iterator to one past the end of `lst`
- `*i` dereferences an iterator `i`
- `++i` advances an iterator `i` to the next element

List iterators

- Like pointers to list elements
- `lst.begin()` returns an iterator to the beginning of `lst`
- `lst.end()` returns an iterator to one past the end of `lst`
- `*i` dereferences an iterator `i`
- `++i` advances an iterator `i` to the next element
- (and more...)

max_vec using std::vector iterators

```
#include <vector>

using iter = typename std::vector<int>::iterator;

iter max_vec(std::vector<int>& vec)
{
    iter best  = vec.begin();

    for (iter i = vec.begin(); i != vec.end(); ++i)
        if (*best < *i)
            best = i;

    return best;
}
```

max_vec using auto

```
#include <vector>

typename std::vector<int>::iterator
max_vec(std::vector<int>& vec)
{
    auto best  = vec.begin();

    for (auto i = vec.begin(); i != vec.end(); ++i)
        if (*best < *i)
            best = i;

    return best;
}
```

max_list using std::list iterators

```
#include <list>

typename std::list<int>::iterator
max_list(std::list<int>& lst)
{
    auto best  = lst.begin();

    for (auto i = lst.begin(); i != lst.end(); ++i)
        if (*best < *i)
            best = i;

    return best;
}
```

Making the algorithm generic

`max_vec` and `max_list` are the same! except for the iterator type

Making the algorithm generic

`max_vec` and `max_list` are the same! except for the iterator type

We can use a template to abstract over the iterator type

Making the algorithm generic

`max_vec` and `max_list` are the same! except for the iterator type

We can use a template to abstract over the iterator type

We'll make the function take an iterator range to search through

Generic maximum element algorithm

```
template <typename Fwd_iter>
Fwd_iter max_gen(Fwd_iter start, Fwd_iter limit)
{
    Fwd_iter best = start;

    for (Fwd_iter i = start; i != limit; ++i)
        if (*best < *i)
            best = i;

    return best;
}
```

`max_generic` is very generic

It doesn't care about:

- the shape of the data structure
- the element type of the data structure
- whether the iterator is const or not

`max_generic` is very generic

It doesn't care about:

- the shape of the data structure
- the element type of the data structure
- whether the iterator is const or not

What it does care about:

- `Fwd_iter` is copyable (`best = i`), pre-incrementable (`++i`), and dereferenceable (`*i`)
- The results of dereferencing `Fwd_iter` are comparable with `operator<`

Using max_generic

```
TEST_CASE("max_gen(vector<int>)")
{
    std::vector<int> vec{ 2, 0, 5, 3, 9, 5, 1 };
    auto exp = vec.begin() + 4;
    CHECK( max_gen(vec.begin(), vec.end()) == exp );
}

TEST_CASE("max_gen(list<double>)")
{
    std::list<double> lst{ 2, 0, 5, 3, 9, 5, 1 };
    auto exp = lst.begin();
    advance(exp, 4);
    CHECK( max_gen(lst.begin(), lst.end()) == exp );
}
```

It's in `<algorithm>`

```
#include <algorithm>

TEST_CASE("max_element(vector<int>)")
{
    std::vector<int> v{ 2, 0, 5, 3, 9, 5, 1 };
    auto i = v.begin() + 4;
    CHECK(std::max_element(v.begin(), v.end()) == i);
}

TEST_CASE("max_element(list<double>)")
{
    std::list<double> w{ 2, 0, 5, 3, 9, 5, 1 };
    auto i = w.begin();
    advance(i, 4);
    CHECK(std::max_element(w.begin(), w.end()) == i);
}
```

STL algorithms

The STL `<algorithm>` header contains many algorithms:
`http://en.cppreference.com/w/cpp/algorithm`

STL algorithms

The STL `<algorithm>` header contains many algorithms:
`http://en.cppreference.com/w/cpp/algorithm`

Let's try using it for counting...

Counting occurrences

```
#include <algorithm>

using namespace std;

const vector<int> v{ 2, 0, 5, 3, 9, 5, 1 };

TEST_CASE("count")
{
    CHECK( count(v.begin(), v.end(), 4) == 0 );
    CHECK( count(v.begin(), v.end(), 3) == 1 );
    CHECK( count(v.begin(), v.end(), 5) == 2 );
}
```

Counting with a predicate

```
bool lt6(int x) { return x < 6; }

const vector<int> v{ 2, 0, 5, 3, 9, 5, 1 };

TEST_CASE("count_if(lt6)")
{
    CHECK( count_if(v.begin(), v.end(), lt6) == 6 );
}
```

Counting with a function object

```
struct Less_than
{
    int value;

    bool operator()(int x) const
    {
        return x < value;
    }
};
```

Counting with a function object

```
struct Less_than
{
    int value;

    bool operator()(int x) const
    {
        return x < value;
    }
};

TEST_CASE("Less_than")
{
    Less_than lt{5};
    CHECK( lt(4) );
    CHECK_FALSE( lt(5) );
}
```

Counting with a function object

```
struct Less_than
{
    int value;

    bool operator()(int x) const
        { return x < value; }
};

const vector<int> v{ 2, 0, 5, 3, 9, 5, 1 };

CHECK( count_if(v.begin(), v.end(), Less_than{6})
           == 6 );
CHECK( count_if(v.begin(), v.end(), Less_than{5})
           == 4 );
```

Constructing a function object using `std::bind`

```
using namespace std;
using namespace std::placeholders;

const vector<int> v{ 2, 0, 5, 3, 9, 5, 1 };

CHECK( count_if(v.begin(), v.end(),
                bind(less<int>(), _1, 6))
           == 6 );
```

The slickest way: lambda

```
const vector<int> v{ 2, 0, 5, 3, 9, 5, 1 };

CHECK( count_if(v.begin(), v.end(),
                [](auto x) { return x < 6; })
           == 6 );
```

The slickest way: lambda

```
const vector<int> v{ 2, 0, 5, 3, 9, 5, 1 };

CHECK( count_if(v.begin(), v.end(),
                []( auto x) { return x < 6; })
           == 6 );

int y = 5;
CHECK( count_if(v.begin(), v.end(),
                [&](auto x) { return x < y; })
           == 4 );
```

The slickest way: lambda

```
const vector<int> v{ 2, 0, 5, 3, 9, 5, 1 };

CHECK( count_if(v.begin(), v.end(),
                [](auto x) { return x < 6; })
           == 6 );

int y = 5;
CHECK( count_if(v.begin(), v.end(),
                [&](auto x) { return x < y; })
           == 4 );

int z = 4;
CHECK( count_if(v.begin(), v.end(),
                [=](auto x) { return x < z; })
           == 4 );
```