The Standard Template Library

EECS 211

Winter 2018

Problem: finding the maximum element of a vector

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 first occurrence.
size_t max_int_vec(const Int_vec& 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( 4 == max_int_vec(v) );
}
```

Problem: finding the maximum element of a linked list

A simple linked list:

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

Problem: finding the maximum element of a linked list

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 the maximum element of a linked list

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)
{
        Int_list result = std::make_shared<Int_node>();
        result->data = data; result->next = next;
        return result;
}
```

Solution: `max_int_list`

```
// Finds the link to the node containing the maximum element.
//  - If the list is 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 != nullptr; i = i->next) {
        if (best->data < i->data) best = i;
    }

    return best;
}
```

Testing `max_int_list`

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

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

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(const Vec<T>& 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(const T& data, List<T> next)
{
    List<T> result = std::make_shared<Node<T>>();
    result->data = data; result->next = next;
    return result;
}
```

Generic `max_list`

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

    for (List<T> i = lst; i != nullptr; 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 vector_int_iter = typename std::vector<int>::iterator;

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

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

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

It's in <algorithm>

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

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

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> vec{ 2, 0, 5, 3, 9, 5, 1 };

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

Counting with a predicate

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

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

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

Counting with a function object

```
class Less_than
{
        int value_;
public:
        Less_than(int value) : value_(value) { }
        bool operator()(int x) const { return x < value_; }
};
```

Counting with a function object

```
class Less_than
{
        int value_;
public:
        Less_than(int value) : value_(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

```
class Less_than
{
    int value_;
public:
    Less_than(int value) : value_(value) { }
    bool operator()(int x) const { return x < value_; }
};

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

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

Constructing a function object using `std::bind`

```
using namespace std::placeholders;

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

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

The slickest way: lambda

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

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

The slickest way: lambda

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

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

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

The slickest way: lambda

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

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

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

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

Next: tokenize using STL and iterators (in CLion)