UCLA Samueli Computer Science

CS32: Introduction to Computer Science II

Discussion Week 7

Yichao (Joey)

May 15, 2020

Credit: Carey Nachenberg, Junheng Hao

Outline Today

- Template
- STL Iterator

Announcements

UCLA Samueli Computer Science

- Project 3 is due 11:00 PM Friday, May 22.
- Midterm 2 is scheduled May 19 (Tue. 6:00 pm to 7:00 pm).

  open book, open notes, no electronic devices,

  emphasizing **stacks and queues, inheritance and polymorphism, and recursion** (not templates, not big-O).

Template

Motivation: More generic class

- Think about the `Pair` class. The class should not work only with integers. That is we want a **“generic” `Pair` class**.
- `Pair<int> p1; Pair<char> p2;`

```
class Pair {
    public:
       Pair();
       Pair(int firstValue,
            int secondValue);
       void setFirst(int newValue);
       void setSecond(int newValue);
       int getFirst() const;
       int getSecond() const;
    private:
       int m_first;
       int m_second;
};
```

```
template<typename T>
class Pair {
    public:
       Pair();
       Pair(T firstValue,
            T secondValue);
       void setFirst(T newValue);
       void setSecond(T newValue);
       T getFirst() const;
       T getSecond() const;
    private:
       T m_first;
       T m_second;
};
```

Template

Multi-type template

- What if we need pair with different types?
- Change your template class: `Pair<int, string> p1;`

```
template<typename T>
class Pair {
    public:
       Pair();
       Pair(T firstValue,
            T secondValue);
       void setFirst(T newValue);
       void setSecond(T newValue);
       T getFirst() const;
       T getSecond() const;
    private:
       T m_first;
       T m_second;
};
```

```
template<typename T, U>
class Pair {
    public:
       Pair();
       Pair(T firstValue,
            U secondValue);
       void setFirst(T newValue);
       void setSecond(U newValue);
       T getFirst() const;
       U getSecond() const;
    private:
       T m_first;
       U m_second;
};
```

Template

Change member functions in template classes

- Member function should also be edited in template class as well.

```cpp
void Pair::setFirst(int newValue)
{
  M_first = newValue;
}
```

```cpp
template<typename T>
void Pair<T>::setFirst(T newValue)
{
  M_first = newValue;
}
```

Template

Template Specialization

- What if we want a template class with certain data type to **have its own exclusive behaviors**? For example, in `Pair` class we only allow `Pair<char>` has `uppercase()` and `lowercase()` function but not for `Pair<int>`.

```
template<>
class Pair<char> {
    public:
       Pair();
       Pair(char firstValue,
            char secondValue);
       void setFirst(char newValue);
       void setSecond(char newValue);
       char getFirst() const;
       char getSecond() const;
       void uppercase();
    private:
       char m_first;
       char m_second;
};
```

```
Pair<int> p1;
Pair<char> p2;

p1.uppercase(); //error
p2.uppercase(); //correct
```

Template

Template Specialization

```
// template specialization
#include <iostream>
using namespace std;

// class template:
template <class T>
class mycontainer {
    T element;
  public:
    mycontainer (T arg)
{element=arg;}
    T increase () { return
++element;}
};
```

```
// class template
specialization:
template <>
class mycontainer <char> {
    char element;
  public:
    mycontainer (char arg)
{element=arg;}
    char uppercase ()
    {
      if
((element>='a')&&(element<='z'))
      element+='A'-'a';
      return element;
    }
};
```

```
int main () {
  mycontainer<int> myint (7);
  mycontainer<char> mychar
('j');
  cout << myint.increase() <<
endl;
  cout << mychar.uppercase() <<
endl;
  return 0;
}
```

```
8
J
```

Template

Const references as parameters

- When you are not changing the values of the parameters, make them **`const`** references to avoid potential computational cost. (Pass by value for ADTs are slow.)

```
template<typename T>
T minimum(const T& a, const T& b)
{
  if (a < b)
    return a;
  else
    return b;
}
```

Template

Some notes

- Generic comparisons:
  - `bool operator>=(const ItemType& a, const ItemType& b)`
- Use the template data type (e.g. `T`) to define the type of at least one formal parameter.
- Add the prefix `template <typename T>` before the class definition itself and before each function definition outside the class.
- Place the postfix `<T>` Between the class name and the `::` in all function definition.

```
template <typename T>
class Foo
{
  public:
    void setVal(T a);
    void printVal(void);
  private:
    T m_a;
};
```

```
template <typename T>
void Foo<T>::setVal(T a)
{
  m_a = a;
}
template <typename T>
void Foo<T>::printVal(void)
{
  cout << m_a << "\n";
}
```

STL: Standard Template Library

Easy and efficient implementation

- A collection of pre-written, tested classes provided by C++.
- All built using templates (adaptive with many data types).
- Provide useful data structures
  - `vector(array), set, list, map, stack, queue`
- Standard functions:
  - Common ones: `.size(), .empty()`
  - For a container that is neither stack or queue: `.insert(), .erase(), swap(), .clear()`
  - For list or vector: `.push_back(), .pop_back()`
  - For set or map: `.find(), .count()`
  - More on stacks and queues…

STL: Standard Template Library

Notes on `vector` and `list`

- You may only use brackets to access existing items in `vector`. **Keep the current size vector in mind** especially after `push_back()` and `pop_back()`.
- **You cannot access list element by brackets.**
- Choose between vector and list:
  - `vectors` are based on **dynamic arrays** placed in contiguous storage. Fast on access but slow on insertion/deletion.
  - `lists` are the opposite (**linked list**). It offers fast insertion/deletion, but slow access to middle elements.

STL: Standard Template Library

Notes on `size and capacity`

- Question: Size and capacity of a vector?

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
  vector<int> myVec;
  // insert only one item
  myVec.push_back(999);
  cout << "size:" << myVec.size() << endl;
  cout << "capacity:" << myVec.capacity() << endl;
  // insert 100 items
  for (int i=0; i<100; i++){ myVec.push_back(i); }
  cout << "size:" << myVec.size() << endl;
  cout << "capacity:" << myVec.capacity() << endl;
  cout << "max size:" << myVec.max_size() << endl;
  return 0;
}
```

```
size: ?
capacity: ?

size: ?
capacity: ?

max size: ?
```

STL: Standard Template Library

Notes on `size` and `capacity`

- Question: Size and capacity of a vector?

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
  vector<int> myVec;
  // insert only one item
  myVec.push_back(999);
  cout << "size:" << myVec.size() << endl;
  cout << "capacity:" << myVec.capacity() << endl;
  // insert 100 items
  for (int i=0; i<100; i++){ myVec.push_back(i); }
  cout << "size:" << myVec.size() << endl;
  cout << "capacity:" << myVec.capacity() << endl;

  return 0;
}
```

Size is not allowed to differ between multiple compilers. The size of a vector is the number of elements that it contains, which is directly controlled by how many elements you put into the vector.

Capacity is the amount of space that the vector is currently using. This is always equal to or larger than the size.

STL: Standard Template Library

Notes on `size` and `capacity`

- Question: Size and capacity of a vector?

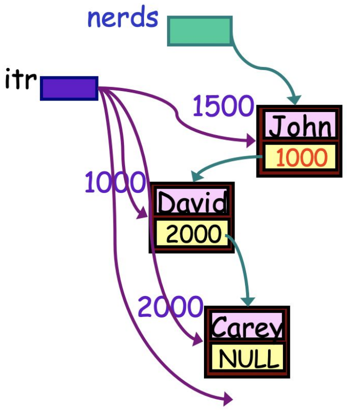
```cpp
#include <iostream>
#include <vector>
using namespace std;
int main() {
  vector<int> myVec;
  // insert only one item
  myVec.push_back(999);
  cout << "size:" << myVec.size() << endl;
  cout << "capacity:" << myVec.capacity() << endl;
  // insert 100 items
  for (int i=0; i<100; i++){ myVec.push_back(i); }
  cout << "size:" << myVec.size() << endl;
  cout << "capacity:" << myVec.capacity() << endl;

  return 0;
}
```

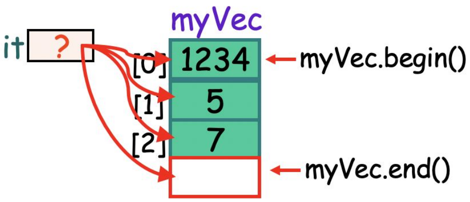
```
→ On my computer:
size:1
capacity:1
size:101
capacity:128
```

STL: Standard Template Library

Implementation example: Iterators

- STL Iterators: Use `.begin()` and `.end()`
  - `.begin()` : return an iterator that points to the first element.
  - `.end()`: return an iterator that points to the ***past-the-last*** element.
- A container as a `const` reference cannot use regular iterator but **need to use const iterator**. Example: `list<string>::const_iterator it;`
- Examples






```
void main()
{
  vector<int>   myVec;
  myVec.push_back(1234);
  myVec.push_back(5);
  myVec.push_back(7);
  vector<int>::iterator it;
  it = myVec.begin();
  while (  it != myVec.end() ){
    cout << (*it);
    it++;
  }
}
```

STL: Standard Template Library

Warning: using iterators for changing vector

- It could be dangerous to **use iterator to traverse a vector when we have performed insertion/deletion**.
- Safe solution: **Reinitialize iterators** of a vector whenever its size has been changed.

```
// Guess what is the output?
int main ()
{
  vector<int> v{1,2};
  v.push_back(3);
  v.push_back(4);
  v.push_back(5);
  vector<int>::iterator b = v.begin();
  vector<int>::iterator e = v.end();
  for (int i = 6; i < 100; i++) { v.push_back(i); }
  while (b != e) {
    cout << *b++ << endl;
  }
}
```

On my computer:
0 0 3 4 5

STL

Iterator invalidation

<table>
<tr><th rowspan="2">Category</th><th rowspan="2">Container</th><th colspan="2">After insertion, are...</th><th colspan="2">After erasure, are...</th><th rowspan="2">Conditionally</th></tr>
<tr><th>iterators valid?</th><th>references valid?</th><th>iterators valid?</th><th>references valid?</th></tr>
<tr><td rowspan="8">Sequence containers</td><td><code>array</code></td><td colspan="2">N/A</td><td colspan="2">N/A</td><td></td></tr>
<tr><td rowspan="3"><code>vector</code></td><td colspan="2">No</td><td colspan="2">N/A</td><td>Insertion changed capacity</td></tr>
<tr><td colspan="2">Yes</td><td colspan="2">Yes</td><td>Before modified element(s)</td></tr>
<tr><td colspan="2">No</td><td colspan="2">No</td><td>At or after modified element(s)</td></tr>
<tr><td rowspan="2"><code>deque</code></td><td rowspan="2">No</td><td>Yes</td><td colspan="2">Yes, except erased element(s)</td><td>Modified first or last element</td></tr>
<tr><td>No</td><td colspan="2">No</td><td>Modified middle only</td></tr>
<tr><td><code>list</code></td><td colspan="2">Yes</td><td colspan="2">Yes, except erased element(s)</td><td></td></tr>
<tr><td><code>forward_list</code></td><td colspan="2">Yes</td><td colspan="2">Yes, except erased element(s)</td><td></td></tr>
<tr><td>Associative containers</td><td><code>set</code>
<code>multiset</code>
<code>map</code>
<code>multimap</code></td><td colspan="2">Yes</td><td colspan="2">Yes, except erased element(s)</td><td></td></tr>
<tr><td rowspan="2">Unordered associative containers</td><td rowspan="2"><code>unordered_set</code>
<code>unordered_multiset</code>
<code>unordered_map</code>
<code>unordered_multimap</code></td><td>No</td><td rowspan="2">Yes</td><td colspan="2">N/A</td><td>Insertion caused rehash</td></tr>
<tr><td>Yes</td><td colspan="2">Yes, except erased element(s)</td><td>No rehash</td></tr>
</table>

STL

Vector vs List

vector:

- **Contiguous memory.**
- Pre-allocates space for future elements, so extra space required beyond what's necessary for the elements themselves.
- Each element only requires the space for the element type itself (no extra pointers).
- **Can re-allocate memory for the entire vector any time that you add an element.**
- Insertions at the end are constant, amortized time, but insertions elsewhere are a costly O(n).
- Erasures at the end of the vector are constant time, but for the rest it's O(n).
- You can randomly access its elements.
- **Iterators are invalidated if you add or remove elements to or from the vector.**
- You can easily get at the underlying array if you need an array of the elements.

list:

- **Non-contiguous memory.**
- No pre-allocated memory. The memory overhead for the list itself is constant.
- Each element requires extra space for the node which holds the element, including pointers to the next and previous elements in the list.
- **Never has to re-allocate memory for the whole list just because you add an element.**
- Insertions and erasures are cheap no matter where in the list they occur.
- It's cheap to combine lists with splicing.
- You cannot randomly access elements, so getting at a particular element in the list can be expensive.
- **Iterators remain valid even when you add or remove elements from the list.**
- If you need an array of the elements, you'll have to create a new one and add them all to it, since there is no underlying array.

STL

Iterator erase

erase() return:
An iterator pointing to the element that followed the last element erased by the function call.

```
std::list<int> mylist;
std::list<int>::iterator it1,it2;

for (int i=1; i<10; ++i) mylist.push_back(i*10);
it1 = it2 = mylist.begin();
advance (it2,6);
++it1;

it1 = mylist.erase (it1);
it2 = mylist.erase (it2);
++it1;
--it2;
mylist.erase (it1,it2);

std::cout << "mylist contains:";
for (it1=mylist.begin(); it1!=mylist.end(); ++it1)
  std::cout << ' ' << *it1;
std::cout << '\n';
```

```
// 10 20 30 40 50 60 70 80 90
// ^*
// ^                 *
//    ^              *
// 10 30 40 50 60 70 80 90
//    ^           *
// 10 30 40 50 60 80 90
//    ^           *
//       ^        *
//       ^     *
// 10 30 60 80 90
//       ^
```

STL

removeOdds

```cpp
int a[8] = { 2, 8, 5, 6, 7, 3, 4, 1 };
vector<int> li(a, a+8);

void removeOdds(vector<int>& li)
{
    vector<int>::iterator it = li.begin();
    while (it != li.end())
        if (*it % 2 == 1) {
            it = li.erase(it);
            cout<<*it<<endl;
    }
        else
            it++;
}
```

```cpp
int a[8] = { 2, 8, 5, 6, 7, 3, 4, 1 };
list<int> li(a, a+8);

void removeOdds(list<int>& li)
{
    list<int>::iterator it = li.begin();
    while (it != li.end())
        if (*it % 2 == 1) {
            it = li.erase(it);
            cout<<*it<<endl;
    }
        else
            it++;
}
```

erase() return:
An iterator pointing to the element that followed the last element erased by the function call.

STL
removeBad

```
int a[8] = { 85, 80, 30, 70, 20, 15, 90, 10 };
list<Movie*> li;
for (int k = 0; k < 8; k++)
    li.push_back(new Movie(a[k]));
void removeBad(list<Movie*>& li)
{
     auto it=li.begin();
     while (it!=li.end())
       if ((*it)->rating() < 50) {
         delete *it;
         it = li.erase(it);
     }
       else
         it++;
}
```

Standard Template Library

How to use STL? No need to recite all of them!

- Remember the basic provided libraries (such as size, etc)
- Check http://www.cplusplus.com/reference/stl/ for more details if needed.

STL: Standard Template Library

Some more topics

- More STL examples, such as `map`, `set`, `etc.`
- More STL algorithms, such as `find()`, `sort()`, `etc.`

* Smart Pointer

A good tool in modern C++

UCLA Samueli Computer Science

* Smart Pointer

A good tool in modern C++

UCLA Samueli Computer Science

- A smart pointer is an **abstract data type** that simulates a pointer while providing added features, such as **automatic memory management or bounds checking**.
- C++ libraries provide implementations of smart pointers in the form of **`unique_ptr`**, **`shared_ptr`** and **`weak_ptr`**
- Trade-off by using smart pointers: may increase memory usage (for example in `list`)
- More info: [Smart pointer tutorial]

```
// normal pointers
void UseNormalPointer{
  MyClass *ptr = new MyClass();
  ptr->doSomething();
}
// We must delete ptr to avoid memory leak!
```

```
// smart pointers, defined in std
void UseSmartPointer{
  unique_ptr<MyClass> ptr(new MyClass());
  ptr->doSomething();
}
// ptr is deleted automatically here!
// unique_ptr:encapsulated pointer as only data member
```

* Smart Pointer

`Unique_ptr, shared_ptr and weak_ptr`

UCLA Samueli Computer Science

- **What is a smart pointer?**
  It's a type whose values can be used like pointers, but which provides the additional feature of automatic memory management: When a smart pointer is no longer in use, the memory it points to is deallocated

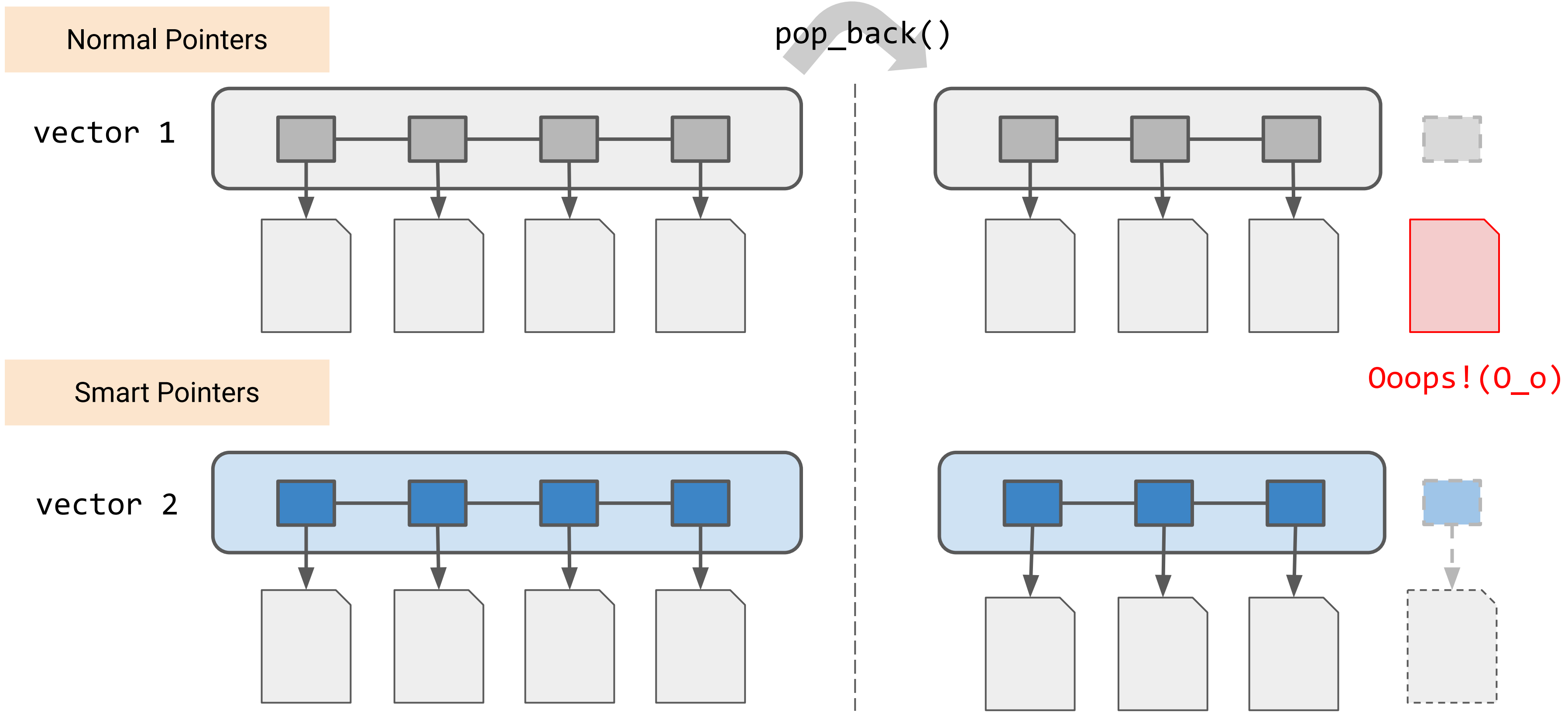
- **When should I use one?**
  In code which **involves tracking the ownership of a piece of memory,** allocating or de-allocating; the smart pointer often saves you the need to do these things explicitly.

- **But which smart pointer should I use in which of those cases?**
  - Use `std::unique_ptr` when you **don't intend to hold multiple references to the same object**. For example, use it for a pointer to memory which gets allocated on entering some scope and de-allocated on exiting the scope.
  - Use `std::shared_ptr` when you **do want to refer to your object from multiple places** - and do not want your object to be de-allocated until all these references are themselves gone.
  - Use `std::weak_ptr` when you **do want to refer to your object from multiple places** - for those references for which it's ok to ignore and deallocate (so they'll just note the object is gone when you try to dereference).

Pointers vs Smart Pointers

UCLA Samueli Computer Science

Example: Container of pointers



Group Exercises: Worksheet

- Exercise problems from **Worksheet #7** (see “LA worksheet” tab in CS32 website). Answers will be posted next week.
- Questions for today:
  - Code Output
  - Debugging
  - removeAllZeros
  - deleteOddSumLists
  - Implement Templated Vector Class

Group Exercises: Worksheet Prob. #1

UCLA Samueli Computer Science

```
template <class T>
void foo(T input) {
  cout << "Inside the main template foo(): " <<input<< endl;
}

template<>
void foo(int input) {
  cout << "Specialized template for int: " << input << endl;
}

int main() {
  foo<char>('A');
  foo<int>(19);
  foo<double>(19.97);
}
```

What does the code output?

Group Exercises: Worksheet Prob. #1

UCLA Samueli Computer Science

```
template <class T>
void foo(T input) {
  cout << "Inside the main template foo(): " <<input<< endl;
}

template<>
void foo(int input) {
  cout << "Specialized template for int: " << input << endl;
}

int main() {
  foo<char>('A');
  foo<int>(19);
  foo<double>(19.97);
}
```

Output:

```
Inside the main template foo(): A
Specialized template for int: 19
Inside the main template foo(): 19.97
```

Group Exercises: Worksheet Prob. #2

UCLA Samueli Computer Science

```
class Potato {
  public:
    Potato(int in_size) : size(in_size) { }
    int getSize() const { return size; }
  private:
    int size;
};
int main() {
  set<Potato> potatoes;
  Potato p1(3);
  Potato p2(4);
  Potato p3(5);
  potatoes.insert(p1);
  potatoes.insert(p2);
  potatoes.insert(p3);
  set<Potato>::iterator it = potatoes.begin();
  while (it != potatoes.end()) {
    potatoes.erase(it);
    it++;
  }
  for (it = potatoes.begin(); it != potatoes.end(); it++) {
    cout << it.getSize() << endl;
  }
}
```

Find the 3 compilation and runtime errors.

Group Exercises: Worksheet Prob. #2

UCLA Samueli Computer Science

```
class Potato {
  public:
    Potato(int in_size) : size(in_size) { }
    int getSize() const { return size; }
  private:
    int size;
};
int main() {
  set<Potato> potatoes;                                        // 1
  Potato p1(3);
  Potato p2(4);
  Potato p3(5);
  potatoes.insert(p1);
  potatoes.insert(p2);
  potatoes.insert(p3);
  set<Potato>::iterator it = potatoes.begin();
  while (it != potatoes.end()) {
    potatoes.erase(it);                                        // 2
    it++;
  }
  for (it = potatoes.begin(); it != potatoes.end(); it++) {
    cout << it.getSize() << endl;                              // 3
  }
}
```

1. set uses the comparison operator, therefore we need to define one for Potato class
2. `erase()` returns the iterator to the next value in the data structure, and we have to assume the the iterator becomes invalid
3. an iterator behaves like a pointer; we must use `*it.` or `it->`

Group Exercises: Worksheet Prob. #3

UCLA Samueli Computer Science

Create a function that takes a **container of integers** and **removes all zeros** while **preserving the ordering** of all the elements. Do the operation in place, which means **do not create a new container**. Make sure to have the correct #include commands.

```
void removeAllZeroes(vector<int>& x);
```

Group Exercises: Worksheet Prob. #3

UCLA Samueli Computer Science

Remember to how to remove elements from an STL container:

```
it = x.erase(it);
```

```
void removeAllZeroes(vector<int>& x){
  while (it != x.end()) {
    if (*it == 0) it = x.erase(it);
    else it++;
  }
}
```

Group Exercises: Worksheet Prob. #5

You are given an STL `set<list<int>*>`. In other words, you have a set of pointers, and each pointer points to a list of ints. Consider the sum of a list to be the result of adding up all elements in the list. If a list is empty, treat its sum as zero.

Write a function that **removes the lists with odd sums** from the set. The lists with odd sums should be **deleted from memory** and their **pointers should be removed** from the set. This function should **return the number of lists that are removed**. You may assume that none of the pointers is null.

```
int deleteOddSumLists(set<list<int>*>& s);
```

Group Exercises: Worksheet Prob. #5

UCLA Samueli Computer Science

```
int deleteOddSumLists(set<list<int>*>& s) {
   int numDeleted = 0;
   set<list<int>*>::iterator set_it = s.begin();
   while (set_it != s.end())
   {
      int sum = 0;
      list<int>::iterator list_it = (*set_it)->begin();
      list<int>::iterator list_end = (*set_it)->end();
      while (list_it != list_end)
      {
         sum += *list_it;
         list_it++;
      }
      if (sum % 2 == 1)
      {
         delete *set_it;
         set_it = s.erase(set_it);
         numDeleted++;
      }
      else set_it++;
   }
   return numDeleted;
}
```

Keep track of what is an iterator and what is a pointer.

Use the correct function calls with proper syntax.

Group Exercises: Worksheet Prob. #6

UCLA Samueli Computer Science

Implement a vector class *Vector* that can be used with any data type using templates. Use a dynamically allocated array to store the data. Implement only the *push_back()* function, default constructor, and destructor.

```
template <typename T>
class Vector {
  public:
    Vector();
    ~Vector();
    void push_back(const T& item);
  private:
    int m_capacity;                                     // Total capacity of the vector -- doubles each time
    int m_size;                                         // The number of elements in the array
    T* m_buffer;                                        // Underlying dynamic array
};
```

Group Exercises: Worksheet Prob. #6

UCLA Samueli Computer Science

Constructor and Destructor:

```
template <typename T>
Vector<T>::Vector()
: m_capacity(0), m_size(0), m_buffer(nullptr)
{}

template <typename T>
Vector<T>::~Vector() {
      delete[] m_buffer;
}
```

Group Exercises: Worksheet Prob. #6

Pseudocode:

- If capacity is reached
    - Create new array with double capacity
    - Copy values into new array
    - Delete old buffer
    - Assign new buffer
- Else
    - Assign value at end
    - Increment size

```
template <typename T>
void Vector<T>::push_back(const T& item) {
  if (m_size == m_capacity)
  {
    if (m_capacity == 0)
      m_capacity = 1;
    else
      m_capacity *= 2;

    T* newBuffer = new T[m_capacity];
    for(int i = 0; i < m_size; i++) {
      newBuffer[i] = m_buffer[i];
    }

    delete [] m_buffer;

    m_buffer = newBuffer;
  }

  m_buffer[m_size] = item;
  m_size++;
}
```