CSCI 104
Iterators
Mark Redekopp
David Kempe
C++11, 14, 17

• Most of what we have taught you in this class are language features that were part of C++ since the C++98 standard

• New, helpful features have been added in C++11, 14, and now 17 standards
  – Beware: compilers are often a bit slow to implement the standards so check the documentation and compiler version
  – You often must turn on special compile flags to tell the compiler to look for C++11 features, etc.
    • For g++ you would need to add: -std=c++11 or -std=c++0x

• Many of the features in the next C++11 are part of a 3rd party library called Boost
Plugging the leaks

SMART POINTERS
Pointers or Objects? Both!

- In C++, the dereference operator (*) should appear before...
  - A pointer to an object
  - An actual object
- "Good" answer is
  - A Pointer to an object
- "Technically correct" answer...
  - EITHER!!!!
- Due to operator overloading we can make an object behave as a pointer
  - Overload operator *, &, ->, ++, etc.

```cpp
class Thing
{
};

int main()
{
    Thing t1;
    Thing *ptr = &t1
    // Which is legal?
    *t1;
    *ptr;
}
```
A "Dumb" Pointer Class

- We can make a class operate like a pointer
- Use template parameter as the type of data the pointer will point to
- Keep an actual pointer as private data
- Overload operators
- This particular class doesn't really do anything useful
  - It just does what a normal pointer would do

```cpp
template <typename T>
class dumb_ptr
{
    private:
        T* p_;
    public:
        dumb_ptr(T* p) : p_(p) { }
        T& operator*() { return *p_; }
        T* operator->() { return p_; }
        dumb_ptr& operator++() // pre-inc
            { ++p_; return *this; }
};

int main()
{
    int data[10];
    dumb_ptr<int> ptr(data);

    for(int i=0; i < 10; i++) {
        cout << *ptr;  ++ptr;
    }
}```
A "Useful" Pointer Class

• I can add automatic memory deallocation so that when my local "unique_ptr" goes out of scope, it will automatically delete what it is pointing at.

```cpp
template <typename T>
class unique_ptr
{
    private:
        T* p_;   
    public:
        unique_ptr(T* p) : p_(p) {}
        ~unique_ptr() { delete p_; }
        T& operator*() { return *p_; }
        T* operator->() { return p_; }
        unique_ptr& operator++() // pre-inc
            { ++p_; return *this; }
};

int main()
{
    unique_ptr<Obj> ptr(new Obj);
    // ...
    ptr->all_words();
    // Do I need to delete Obj?
}
```
A "Useful" Pointer Class

• What happens when I make a copy?

• Can we make it impossible for anyone to make a copy of an object?
  – Remember C++ provides a default "shallow" copy constructor and assignment operator

```cpp
template <typename T>
class unique_ptr
{
  private:
    T* p_;
  public:
    unique_ptr(T* p) : p_(p) { }
    ~unique_ptr() { delete p_; }
    T& operator*() { return *p_; }
    T* operator->() { return p_; }
    unique_ptr& operator++() // pre-inc
      { ++p_; return *this; }
};

int main()
{
  unique_ptr<Obj> ptr(new Obj);
  unique_ptr<Obj> ptr2 = ptr;
  // ...
  ptr2->all_words();
  // Does anything bad happen here?
}
```
Hiding Functions

• Can we make it impossible for anyone to make a copy of an object?
  – Remember C++ provides a default "shallow" copy constructor and assignment operator
• Yes!!
  – Put the copy constructor and operator= declaration in the private section...now the implementations that the compiler provides will be private (not accessible)
• You can use this technique to hide "default constructors" or other functions

```cpp
template <typename T>
class unique_ptr
{
  private:
    T* p_
  public:
    unique_ptr(T* p) : p_(p) { }
    ~unique_ptr() { delete p_; }
    T& operator*() { return *p_; }
    T* operator->() { return p_; }
    unique_ptr& operator++() // pre-inc
      { ++p_; return *this; }
  private:
    unique_ptr(const UsefultPtr& n);
    unique_ptr& operator=(const UsefultPtr& n);
};

int main()
{
  unique_ptr<Obj> ptr(new Obj);
  unique_ptr<Obj> ptr2 = ptr;
  // Try to compile this?
}
```
A "shared" Pointer Class

• Could we write a pointer class where we can make copies that somehow "know" to only delete the underlying object when the last copy of the smart pointer dies?

• Basic idea
  – shared_ptr class will keep a count of how many copies are alive
  – shared_ptr destructor simply decrements this count
    • If count is 0, delete the object

```cpp
template <typename T>
class shared_ptr
{
    public:
        shared_ptr(T* p);
        ~shared_ptr();
        T& operator*();
        shared_ptr& operator++();
    }

shared_ptr<Obj> f1()
{
    shared_ptr<Obj> ptr(new Obj);
    cout << "In F1\n" << *ptr << endl;
    return ptr;
}

int main()
{
    shared_ptr<Obj> p2 = f1();
    cout << "Back in main\n" << *p2;
    cout << endl;
    return 0;
}
```
A "shared" Pointer Class

• Basic idea
  – shared_ptr class will keep a count of how many copies are alive
  – Constructors/copies increment this count
  – shared_ptr destructor simply decrements this count
    • If count is 0, delete the object

```cpp
int main()
{
    shared_ptr<Obj> p1(new Obj);
    doit(p1);
    return 0;
}

void doit(shared_ptr<Obj> p2)
{
    if(...){
        shared_ptr<Obj> p3 = p2;
    }
}
```
A "shared" Pointer Class

- Basic idea
  - `shared_ptr` class will keep a count of how many copies are alive
  - `shared_ptr` destructor simply decrements this count
    - If count is 0, delete the object

```cpp
int main()
{
    shared_ptr<Obj> p1(new Obj);
    doit(p1);
    return 0;
}

void doit(shared_ptr<Obj> p2)
{
    if(...){
        shared_ptr<Obj> p3 = p2;
    }
}
```
A "shared" Pointer Class

- Basic idea
  - shared_ptr class will keep a count of how many copies are alive
  - shared_ptr destructor simply decrements this count
    - If count is 0, delete the object

```c++
int main()
{
    shared_ptr<Obj> p1(new Obj);
    doit(p1);
    return 0;
}

void doit(shared_ptr<Obj> p2)
{
    if(...){
        shared_ptr<Obj> p3 = p2;
    }
}
```
A "shared" Pointer Class

• Basic idea
  – shared_ptr class will keep a count of how many copies are alive
  – shared_ptr destructor simply decrements this count
    • If count is 0, delete the object

```cpp
int main()
{
    shared_ptr<Obj> p1(new Obj);
    doit(p1);
    return 0;
}

void doit(shared_ptr<Obj> p2)
{
    if(...)
    {
        shared_ptr<Obj> p3 = p2;
    }
    // p3 dies
}
```
A "shared" Pointer Class

• Basic idea
  – shared_ptr class will keep a count of how many copies are alive
  – shared_ptr destructor simply decrements this count
    • If count is 0, delete the object

```
int main()
{
    shared_ptr<Obj> p1(new Obj);
    doit(p1);
    return 0;
}

void doit(shared_ptr<Obj> p2)
{
    if(...){
        shared_ptr<Obj> p3 = p2;
    } // p3 dies
} // p2 dies
```
A "shared" Pointer Class

- Basic idea
  - `shared_ptr` class will keep a count of how many copies are alive
  - `shared_ptr` destructor simply decrements this count
    - If count is 0, delete the object

```cpp
int main()
{
    shared_ptr<Obj> p1(new Obj);
    doit(p1);
    return 0;
} // p1 dies

void doit(shared_ptr<Obj> p2)
{
    if(...){
        shared_ptr<Obj> p3 = p2;
        // p3 dies
    }
    // p2 dies
}
```
C++ shared_ptr

- C++ std::shared_ptr / boost::shared_ptr
  - Boost is a best-in-class C++ library of code you can download and use with all kinds of useful classes
- Can only be used to point at dynamically allocated data (since it is going to call delete on the pointer when the reference count reaches 0)
- Compile in g++ using '-std=c++11' since this class is part of the new standard library version

```cpp
#include <memory>
#include "obj.h"
using namespace std;

shared_ptr<Obj> f1()
{
    shared_ptr<Obj> ptr(new Obj);
    // ...
    cout << "In F1\n" << *ptr << endl;
    return ptr;
}

int main()
{
    shared_ptr<Obj> p2 = f1();
    cout << "Back in main\n" << *p2;
    cout << endl;
    return 0;
}
```

```bash
$ g++ -std=c++11 shared_ptr1.cpp obj.cpp
```
C++ shared_ptr

- Using shared_ptr's you can put pointers into container objects (vectors, maps, etc) and not have to worry about iterating through and deleting them.
- When myvec goes out of scope, it deallocates what it is storing (shared_ptr's), but that causes the shared_ptr destructor to automatically delete the Objs.
- Think about your project homeworks...this might be (have been) nice.

```cpp
#include <memory>
#include <vector>
#include "obj.h"
using namespace std;

int main()
{
    vector<shared_ptr<Obj> > myvec;

    shared_ptr<Obj> p1(new Obj);
    myvec.push_back( p1 );

    shared_ptr<Obj> p2(new Obj);
    myvec.push_back( p2 );

    return 0;
    // myvec goes out of scope...
}
```

```
$ g++ -std=c++11 shared_ptr1.cpp obj.cpp
```
shared_ptr vs. unique_ptr

• Both will perform automatic deallocation
• Unique_ptr only allows one pointer to the object at a time
  – Copy constructor and assignment operator are hidden as private functions
  – Object is deleted when pointer goes out of scope
  – Does allow "move" operation
    • If interested read more about this on your own
    • C++11 defines "move" constructors (not just copy constructors) and "rvalue references" etc.
• Shared_ptr allow any number of copies of the pointer
  – Object is deleted when last pointer copy goes out of scope
• Note: Many languages like python, Java, C#, etc. all use this idea of reference counting and automatic deallocation (aka garbage collection) to remove the burden of memory management from the programmer
References

• [http://www.umich.edu/~eeecs381/handouts/C++11_smart_ptrs.pdf](http://www.umich.edu/~eeecs381/handouts/C++11_smart_ptrs.pdf)

• [http://stackoverflow.com/questions/3476938/example-to-use-shared-ptr](http://stackoverflow.com/questions/3476938/example-to-use-shared-ptr)
ITERATORS
Iteration

• Consider how you iterate over all the elements in a list
  – Use a for loop and get() or operator[]
• For an array list this is fine since each call to get() is O(1)
• For a linked list, calling get(i) requires taking i steps through the linked list
  – 0th call = 0 steps
  – 1st call = 1 step
  – 2nd call = 2 steps
  – 0+1+2+...+n-2+n-1 = O(n^2)
• You are repeating the work of walking the list...

```
ArrayList<int> mylist;
...
for(int i=0; i < mylist.size(); ++i)
{
    cout << mylist.get(i) << endl;
}
```

```
LinkedList<int> mylist;
...
for(int i=0; i < mylist.size(); ++i)
{
    cout << mylist.get(i) << endl;
}
```
Iteration: A Better Approach

- Solution: Don't use get(i)
- Use an **iterator**
  - Stores internal state variable (i.e. another pointer) that remembers where you are and allows taking steps efficiently
- Iterator tracks the internal location of each successive item
- Iterators provide the semantics of a pointer (they look, smell, and act like a pointer to the values in the list
- Assume
  - Mylist.begin() returns an "iterator" to the beginning item
  - Mylist.end() returns an iterator "one-beyond" the last item
  - ++it (preferrer) or it++ moves iterator on to the next value

```cpp
LinkedList<int> mylist;
...
iterator it = mylist.begin();
for(it = mylist.begin();
    it != mylist.end();
    ++it)
{
    cout << *it << endl;
}
```
Why Iterators

• Can be more efficient
  – Keep internal state variable for where you are in your iteration process so you do NOT have to traverse (re-walk) the whole list every time you want the next value

• Hides the underlying implementation details from the user
  – User doesn't have to know whether its an array or linked list behind the scene to know how to move to the next value
    • To take a step with a pointer in array: ++ptr
    • To take a step with a pointer in a linked list: ptr = ptr->next
  – For some of the data structures like a BST the underlying structure is more complex and to go to the next node in a BST is not a trivial task
More operator overloading...

DEFINING ITERATORS
A "Dumb" Pointer Class

- Operator*
  - Should return reference (T&) to item pointed at

- Operator->
  - Should return a pointer (T*) to item be referenced

- Operator++()
  - Preincrement
  - Should return reference to itself iterator& (i.e. return *this)

- Operator++(int)
  - Postincrement
  - Should return another iterator pointing to current item will updating itself to point at the next

- Operator== & !=

```cpp
template <typename T>
class DumbPtr
{
private:
  T* p_
;
public:
  DumbPtr(T* p) : p_(p) { }
  T& operator*() { return *p_; }
  T* operator->() { return p_; }
  DumbPtr& operator++() // pre-inc
  { ++p_; return *this; }
  DumbPtr operator++(int) // post-inc
  { DumbPtr x; x.p_ = p_; ++p_; return x; }
  bool operator==(const DumbPtr& rhs);
  { return p_ == rhs.p_; }
  bool operator!=(const DumbPtr& rhs);
  { return p_ != rhs.p_; }
};

int main()
{
  int data[10];
  DumbPtr<int> ptr(data);
  for(int i=0; i < 10; i++){
    cout << *ptr;  ++ptr;
  }
}
```
Pre- vs. Post-Increment

- Recall what makes a function signature unique is combination of name AND number/type of parameters
  - \texttt{int} \texttt{f1()} and \texttt{void} \texttt{f1()} are the same
  - \texttt{int} \texttt{f1(int)} and \texttt{void} \texttt{f1()} are unique

- When you write: \texttt{obj++} or \texttt{++obj} the name of the function will be the same: \texttt{operator++}

- To differentiate the designers of C++ arbitrarily said, we'll pass a dummy \texttt{int} to the \texttt{operator++()} for POST-increment

- So the prototypes look like this...
  - Preincrement: \texttt{iterator\& operator++();}
  - Postincrement: \texttt{iterator operator++(int)};
    - Prototype the 'int' argument, but ignore it...never use it...
    - It's just to differentiate pre- from post-increment
Pre- vs. Post-Increment

• Consider an expression like the following (a=1, b=5):
  – (a++ * b) + (a * ++b)
  – 1*5 + 2*6
  – Operator++ has higher precedence than multiply (*), so we do it first but the post increment means it should appear as if the old value of a is used
  – To achieve this, we could have the following kind of code:
    – a++ => { int x = a; a = a+1; return x; }
      • Make a copy of a (which we will use to evaluate the current expr.
      • Increment a so its ready to be used the next time
      • Return the copy of a that we made
  – Preincrement is much easier because we can update the value and then just use it
    – ++b => { b = b+1; return b; }
• Takeaway: Post-increment is "less efficient" because it causes a copy to be made
Exercise

• Add an iterator to the supplied linked list class
  – $ mkdir iter_ex
  – $ cd iter_ex
  – $ wget
    http://ee.usc.edu/~redekopp/cs104/iter.tar
  – $ tar xvf iter.tar
Building Our First Iterator

- Let's add an iterator to our Linked List class
  - Will be an object/class that holds some data that allows us to get an item in our list and move to the next item
  - How do you iterate over a linked list normally:
    - Item<T>* temp = head;
    - While(temp) temp = temp->next;
  - So my iterator object really just needs to model (contain) that 'temp' pointer
- Iterator needs following operators:
  - * 
  - -> 
  - ++ 
  - == / != 
  - < ??

```
template<typename T>
struct Item {
  T val;
  Item<T>* next;
};

template<typename T>
class LList {
public:
  LList(); // Constructor
  ~LList(); // Destructor

private:
  Item<T>* head_; 
};
```
Implementing Our First Iterator

- We store the Item<T> pointer to our current item/node during iteration
- We return the value in the Item when we dereference the iterator
- We update the pointer when we increment the iterator

```cpp
template<typename T>
class LList
{
    public:
        LList() { head_ = NULL; }

    class iterator {
        private:
            Item<T>* curr_;

        public:
            iterator& operator++();
            iterator operator++(int);
            T& operator*();
            T* operator->();
            bool operator!=(const iterator & other);
            bool operator==(const iterator & other);
    }

    private:
        Item<T>* head_;
        int size_;
};
```

Note: Though class iterator is defined inside LList<T>, it is completely separate and what's private to iterator can't be access by LList<T> and vice versa
Outfitting LList to Support Iterators

- `begin()` and `end()` should return a new iterator that points to the head or end of the list.
- But how should `begin()` and `end()` seed the iterator with the correct pointer?

```cpp
template<typename T> class LList {
public:
    LList() { head_ = NULL; }

    class iterator {
    private:
        Item<T>* curr_;
    public:
        iterator& operator++();
        iterator operator++(int);
        T& operator*();
        T* operator->();
        bool operator!=(const iterator & other);
        bool operator==(const iterator & other);
    };

    iterator begin() { ??? }
    iterator end() { ??? }

private:
    Item<T>* head_;
    int size_;
};
```
Outfitting LList to Support Iterators

- We could add a public constructor...
- But that's bad form, because then anybody outside the LList could create their own iterator pointing to what they want it to point to...
  - Only LList<T> should create iterators
  - So what to do??

```cpp
template<typename T>
class LList
{
    public:
    LList() { head_ = NULL; }

    class iterator {
        private:
            Item<T>* curr_;
        public:
            iterator(Item<T>* init) : curr_(init) {}
            iterator& operator++() ;
            iterator operator++(int);
            T& operator*();
            T* operator->();
            bool operator!=(const iterator & other);
            bool operator==(const iterator & other);
    };

    iterator begin()  { ???  }
    iterator end() { ???  }

    private:
        Item<T>* head_;
        int size_;  
};
```
Friends and Private Constructors

- Let's only have the iterator class grant access to its "trusted" friend: Llist
- Now LList<T> can access iterators private data and member functions
- And we can add a private constructor that only 'iterator' and 'LList<T>' can use
  - This prevents outsiders from creating iterators that point to what they choose
- Now begin() and end can create iterators via the private constructor & return them

```cpp
template<typename T>
class LList {
  public:
    LList() { head_ = NULL; }

    class iterator {
      private:
        Item<T>* curr_;
        iterator(Item<T>* init) : curr_(init) {} 
      public:
        friend class LList<T>;
        iterator(Item<T>* init);
        iterator& operator++();
        iterator operator++(int);
        T& operator*();
        T* operator->();
        bool operator!=(const iterator & other);
        bool operator==(const iterator & other);
    };

    iterator begin() { iterator it(head_);
      return it; }
    iterator end() { iterator it(NULL);
      return it; }

  private:
    Item<T>* head_; 
    int size_; 
};
```
Expanding to ArrayLists

• What internal state would an ArrayList iterator store?
• What would begin() stuff the iterator with?
• What would end() stuff the iterator with that would mean "1 beyond the end"?
Const Iterators

• If a LList<T> is passed in as a const argument, then begin() and end() will violate the const'ness because they aren't declared as const member functions
  – iterator begin() const;
  – iterator end() const;

• While we could change them, it would violate the idea that the List will stay const, because once someone has an iterator they really CAN change the List's contents

• Solution: Add a second iterator type: const_iterator

```cpp
template<typename T>
class LList
{
    public:
    LList() { head_ = NULL; }

    class iterator {
    };

    // non-const member functions
    iterator begin() { iterator it(head_); return it; }
    iterator end() { iterator it(NULL); return it; }

    private:
    Item<T>* head_
    int size_
};

void printMyList(const LList<int>& mylist)
{
    LList<int>::iterator it;
    for(it = mylist.begin(); // compile error
        it != mylist.end();
        ++it)
    {
        cout << *it << endl;
    }
}
The const_iterator type should return references and pointers to const T's

We should add an overloaded begin() and end() that are const member functions and return const_iterators

template<typename T>
class LList
{
    public:
        LList() { head_ = NULL; }

    class iterator {
        ...
    }

    iterator begin();
    iterator end();

    class const_iterator {
        private:
            Item<T>* curr_;
            const_iterator(Item<T>* init);
        public:
            friend class LList<T>;
            iterator& operator++() ;
            iterator operator++(int);
            T const & operator*();
            T const * operator->();
            bool operator!=(const iterator & other);
            bool operator==(const iterator & other);
        }

        const_iterator begin() const;
        const_iterator end() const;
    };
};
Const Iterators

• An updated example

```cpp
void printMyList(const LList<int>& mylist) {
    LList<int>::const_iterator it;
    for(it = mylist.begin(); // no more error
        it != mylist.end();
        ++it)
    { cout << *it << endl; }
}
```
!= vs <

- It's common idiom to have the loop condition use != over <
- Some iterators don't support '<' comparison
  - Why? Think about what we're comparing with our LLList<T>::iterator
  - We are comparing the pointer...Is the address of Item at location 1 guaranteed to be less-than the address of Item at location 2?

```cpp
void printMyList(const LLList<int>& mylist)
{
    LLList<int>::const_iterator it;

    for(it = mylist.begin(); it != mylist.end(); ++it)
    {  cout << *it << endl; }

    for(it = mylist.begin(); it < mylist.end(); ++it)
    {  cout << *it << endl; }
}
```
Kinds of Iterators

• This leads us to categorize iterators based on their capabilities (of the underlying data organization)

• Access type
  – Input iterators: Can only READ the value be pointed to
  – Output iterators: Can only WRITE the value be pointed to

• Movement/direction capabilities
  – Forward Iterator: Can only increment (go forward)
    • ++it
  – Bidirectional Iterators: Can go in either direction
    • ++it or --it
  – Random Access Iterators: Can jump beyond just next or previous
    • it + 4 or it – 2

• Which movement/direction capabilities can our LList<T>::iterator naturally support
Implicit Type Conversion

- Would the following if condition make sense?
  - No! If statements want Boolean variables
  - But you've done things like this before
    - Operator>> returns an ifstream
  - So how does ifstream do it?
    - With an "implicit type conversion operator overload"
    - Student::operator bool()
      - Code to specify how to convert a Student to a bool
    - Student::operator int()
      - Code to specify how to convert a Student to an int

```cpp
class Student {
    private:  int id; double gpa;
};
int main()
{
    Student s1;
    if(s1){  cout << "Hi" << endl; }
    return 0;
}

ifstream ifile(filename);
...
while( ifile >> x )
{
    ...  
}
```

```cpp
class Student {
    private:
        int id; double gpa;
    public:
        operator bool() { return gpa>= 2.0;}
        operator int() { return id; }
    private:
}

Student s1;
if(s1) // calls operator bool() and int x = s1; // calls operator int()
```
Iterators With Implicit Conversions

- Can use operator bool() for iterator

```cpp
template<typename T>
class LList
{
    public:
    LList() { head_ = NULL; }

    class iterator {
        private:
            Item<T>* curr_;
        public:
            operator bool()
            {
                return curr_ != NULL;
            }
    };
};

void printMyList(LList<int>& mylist) {
    LList<int>::iterator it = mylist.begin();
    while(it){
        cout << *it++ << endl;
    }
}
```
Finishing Up

• Iterators provide a nice abstraction between user and underlying data organization
  – Wait until we use trees and other data organizations
• Due to their saved internal state they can be more efficient than simpler approaches [like get(i)]