Algorithms and Analysis

Lesson 7: *Iterate*

Array iteration, iterators

Outline

Iterators

- One common task you want to do on a collection of objects is to iterate through each component

- If we have a standardised method for all collections then it is much easier to remember what to do

- But we can also write code that works for any collection that follows this pattern

- This pattern is known as the **iterator pattern**

- The pattern was first developed in C++, but is commonly used in many other languages

Iterating Over C Arrays

- In C we would typically use a for-loop to iterate over an array

```
int n = 10;                               // size of array
int* begin = malloc(n*sizeof(10));   // malloc returns beginning of arr
int* end = begin + n;                     // address past end of array

int sum = 0;
for(int* pt = begin; pt != end; pt++) {
  sum += *pt;                             // need to dereference pointer
}
```

- Ugly, but efficient

Outline

1. Iterators

2. **The C++ Iterator Pattern**

3. Linked-List Iterators

4. Generic Programming

C++ Iterator Pattern

- The C++ iterator pattern says for every `container<T>` we create a nested class called

  `container::iterator`

  which acts as a pointer (for arrays this could just be a pointer to the array) ▌

- The class should implement

  ⋆ a derefenecing operator `T` **operator**⋆`()` ▌
  ⋆ an increment operator **operator**++`()` ▌
  ⋆ a not equal function
  bool operator!=`(`**const** `ITER&,` **const** `ITER&)`
  where `ITER` is `container::iterator` ▌

A Beginning and an Ending

- In addition the container should have two methods

  - ⋆ `begin()`
  - ⋆ `end()`

  that return iterators representing the first element and an iterator representing one position past the last element▮

- Wow! That seems awfully complicated▮

- Don't panic!▮We can hack this▮

Minimal Iterator

```cpp
template <typename>
class Container<T> {
private:

  class iterator {    // this is a nested class
  public:
    iterator() {...}             // constructor
    iterator operator++() {...} // increment
    T& operator*() {...}         // dereference
    friend bool operator!=(const iterator&, const iterator&){
      // code to determine inequality
    }
  }

public:

  iterator begin() {...}          // return begin iter
  iterator end() {...}            // return end iter

}
```

Array-based iterators

- For array based containers such as vector we don't actually need to create an iterator class as we can just use the normal pointer

```
template <typename T>
class Array {
private:
  T *data;
  unsigned length;
  unsigned capacity;

public:
  ...
  typedef T* iterator;          // iterator is pseudonym for T*
  iterator begin() {return data;}
  iterator end() {return data+length;}
};
```

- That's all we need

Using Array Iterators

```
main() {

  Array<string> elements(4): {"earth", "water", "wind", "fire"};

  for(Array:iterator it=elements.begin(); it!=elements.end(); ++it) {
    cout << *it << endl;
  }

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

  for(string& element: elements) {  // range-based loop
    cout << element << endl;
  }
}
```
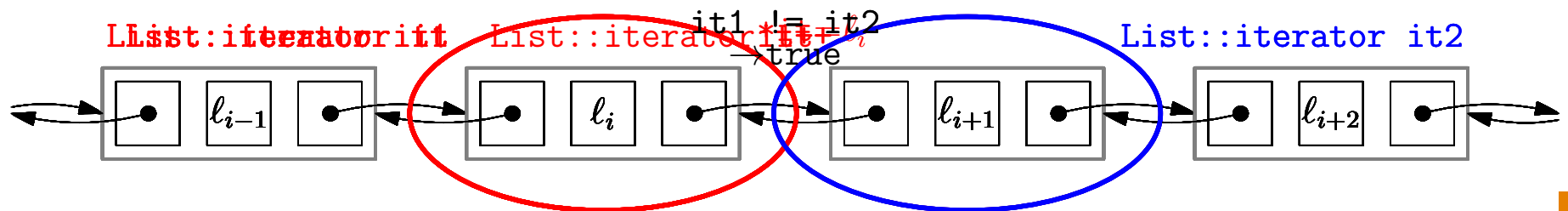
Outline

1. Iterators

2. The C++ Iterator Pattern

3. **Linked-List Iterators**

4. Generic Programming

Linked-List Iterators

- Linked-lists are not array based

- To use the iterator we need to implement the iterator class

- The object instantiated from the class should represent the
  position we are in the linked list

Linked-List

```cpp
template <typename T>
class MyList {      // My linked list class
private:
  struct Node{      // A simple node nested class
    Node(T value, Node *node): value(value), next(node) {}
    T value;
    Node *next;
  };

  struct iterator {                    // An iterator class
    Node* entry;                       // Holds node pointer
    iterator(Node* pt): entry(pt) {}          // constructor
    T& operator*() {return entry->value;}  // dereferencing
    const T& operator*() const {return entry->value;}
    iterator operator++() {  // next entry
      entry = entry->next;
      return entry;
    }
    bool operator!=(const iterator& other) const {
      return entry != other.entry;
    }
  };
```

Linked-List

```cpp
template <typename T>
class MyList {
private:
  struct Node{...}

  struct iterator {...}

  Node* head;            // head of linked list
  unsigned no_elements;

public:
  MyList(): head(nullptr), no_elements(0) {}
  void add(T value) {...}

  iterator begin() {return iterator(head);}
  iterator end() {return nullptr;}
}
```

Increment Operators

- C++ has a pre-increment opeator `++a` and a post-increment operator `a++`

- The pre-increment operator increments a and returns the incremented version, e.g.

  ```
  T& operator++() {++count; return *this;}  // defines ++a
  ```

- The post-increment operator copies a increments it and returns the copy, e.g.

  ```
  T operator++(int) {T b=a; ++count; return b;} // defines a++
  ```

- The **int** argument is not used, but tells the compiler which increment is which

- We might want to implement `it++`

Const Iterators

- C++ uses the compiler to test whether functions change their argument or not

```
func1(Class obj)        // obj is copied so will only modify copy
func2(Class& obj)       // passed by reference, might change obj
func3(const Class& Obj)     // will not change obj
```

- `func3` will only call methods of Obj that are const

```
class Class {
  void method() const;   // won't change the object
  void change();         // might change the object
}
```

- We want to declare a const iterator so need to define

```
const T& operator*() const   // const derefencing operator
```

Bidirectional Iterators

- For the linked list we have implemented a **forward iterator**▐

- This is the only iterator possible for a singly linked list▐

- For a doubly linked list we can implement a **bidirectional iterator**▐

- This requires us to implement the decrement operators

```
T&  operator--();        // implements --obj
T   operator--(int);     //implements obj--
```

▐

- There also exist **random-access iterators** that implements methods including

```
T&  operator[int i]    // returns i'th element
    operator+=[int i]    // move forward i places
```

▐

Outline

1. Iterators

2. The C++ Iterator Pattern

3. Linked-List Iterators

4. **Generic Programming**

Range-Based For Loop

- C++ allows you to iterate over collections elegantly

```
Collection<string> collection;

for (string& element: collection) {
  print(element);   // or whatever function you want
}
```

- This is syntatic sugar! The compiler just replaces this with

```
for(auto& it=collection.begin(); it!=collection.end(); ++it) {
  print(*it);
}
```

- This works for any class that has an iterator

- **auto** just works out the correct type

- By being pretty it reduces bugs in code

Generic Algorithms

- Iterators allow us to write generic functions▮

- E.g. summing elements

```
template <typename Iter, typename T>
T accum(Iter it, Iter end, T init) {
  for(; it != end; ++it)
    init += *it;
    return init;
}
```

- This will sum many collections

```
int array[20];
vector<double> v[5];
set<int> s;

cout << " array sum = " << accum(&array[0], &array[20], 0) << endl;
cout << "vector sum = " << accum(v.begin(), v.end(), 0.0) << endl;
cout << "   set sum = " << accum(s.begin(), s.end(), 0) << endl;
```