Motivation

- A C++ pointer uses the deference operator “*” to access the data it points to.
  - the “arrow” operator “->” is a combination of “*” and “.”
- In an array, we use the pre/post increment operators “++” to move to the next item.
  - in a linked list, we have to follow the next pointer
  - however, we can overload the “++” operators to follow the next pointers
- All the above operators can be overloaded in C++.
- This creates the possibilities of user-defined “fancy pointers,” or iterators.

Using Iterators to Create Generic Code

Consider the DFS routine for the adjacency list representation.

```cpp
void Diraph::recursive_dfs (int start, bool* marked)
{
    marked[start] = true;
    cout << start;

    ListNode* p
    for (p=neighbors_list[start]; p!=NULL; p=p->next)
        if (!marked[p->vertex_id])
            recursive_dfs (p->vertex_id, marked);
}
```

Obviously, the purpose of the for loop is to "visit" each neighbor of vertex start one by one.
With adjacency matrix representation, the for loop should be modified as follows.

```c
for (int i=0; i<vertex_count; i++)
    if (edges[start][i]==true)
        if (!marked[i])
            recursive_dfs (i, marked);
```

Again, the purpose of first two lines to visit every neighbor of vertex `start`.

**Question:** Can we create a ”generic” `recursive_dfs` that can work with both representations?

The answer to the question lies in a uniform way to visit all neighbors of a given nodes, regardless of the underlying representation.

---

**Using Iterators**

- In many graph algorithms, we need to visit all neighbors of a given node.
- By using class "NbIterator" (neighbor iterator), implementation given later, we have a generic implementation of DFS:

```c
NbIterator p;
for (p=nb_first(start); p!=nb_end(start); p++)
    if (!marked[*p])
        recursive_dfs (*p);
```
• Method `nb_first(u)` returns an iterator "pointing to" the first neighbor of vertex `u`;

• In the loop body, we use the deference operator to get access to the vertex ID of a neighbor.

• Both `p++` or `++p` can be used to move to the next neighbor.

• Method `nb_end(u)` returns something conceptually equivalent to "NULL," whose exact definition depends on the natural of the iterators.

---

**Neighbor Iterators for Adjacency Lists**

class NbIterator {
private:
    ListNode *_ptr;
public:
    NbIterator (ListNode*p=NULL) {_ptr = p;}
    ~NbIterator () {};
    NbIterator& operator++ () // prefix ++
    NbIterator  operator++ (int) // postfix ++
    // define deference operator
    int operator* () {return _ptr->vertex_id;}
};
NbIterator& NbIterator::operator++ ()
{
    _p = _p->next;
    return *this;
};

NbIterator NbIterator::operator (int)
{
    NbIterator tmp(_p);
    _p = _p->next;
    return tmp
}

Lastly, add the following two methods to the Digraph class:

NbIterator Digraph::nb_first (int i)
{
    return NbIterator(neighbors_list[i]);
};

NbIterator NbIterator::nb_end (int i)
{
    return NbIterator(NULL);
}
Neighbor Iterators for Adjacency Matrices

Recall that the adjacency table is a two dimension array:

```c++
bool edges[MAX_SIZE][MAX_SIZE];
```

class NbIterator {
private:
  bool* row_u;
  int row_width;
  int v; // current neighbor of vertex u
public:
  NbIterator (bool* row, int width, int i)
  {row_u=row; row_width=width; v=i;}
  ~NbIterator () {};
  NbIterator& operator++ ();
  NbIterator operator++ (int);
  int operator* ();
};
```

Add the following public methods to class Digraph:

```c++
template <int MAX_SIZE>
NbIterator Digraph<MAX_SIZE>::nb_first (int u) {
  for (int v=0; edges[u][v]==0 && v<vertex_count; v++);
  return NbIterator (edges+u, vertex_count, v);
}
```

```c++
template <int MAX_SIZE>
NeighborIterator nb_end (int u) {
  return NeighborIterator (edges+u, vertex_count, vertex_count);
}
```
template <int MAX_SIZE>
NbIterator& NbIterator::operator++ () // prefix ++
{

}

template <int MAX_SIZE>
int NbIterator::operator* () // deference
{

}

Postfix increment operator is left as an exercise.

Applications of Iterators

- Generic implementation of algorithms
  - consider the DFS implementations we discussed
- Provide a uniform way to go thru different data structures without revealing the internal working of the data structures.
  - in the following, we will show a uniform way to go thru a linked list and a hash table
- Allow users to access to a subset of data items in a data structure without revealing the internal working of the data structure.
Four Elements of a Data Structure Iterator

We need to define an iterator class and augment the data structure class.

1. Add a first() method to the data structure class to return an iterator that “points” to the first data item.

2. Add an end() method to the data structure class to return an iterator that serves the function of “NULL.”

3. Implement the deference operator in the iterator class to provide access to the current data item.

4. Implement the pre/post increment operators in the iterator class to move to the next data item.

Linked-List Iterators

```cpp
template<class T>
class ListNode {public: T _data; ListNode* _next; }

template<class T>
class ListIterator {
private:
    ListNode<T>* _ptr;
public:
    ListIterator () {_ptr = NULL;}
    ListIterator (ListIterator q) {_ptr = q._ptr;}
    ListIterator (ListNode<T> *q) {_ptr = q;}

    ListIterator& operator++ (); // prefix, ++p
    ListIterator operator++ (int); // postfix, p++
    T& operator* () {return _ptr->_data;}
};
```
The implementation of ListIterator is similar to that of NbIterator with adjacency list representation.

Also similar to the example of NbIterator, we add

```cpp
iterator first() {return iterator(list_head);}
iterator end() {return iterator(NULL);}
```

to the linked list class.

---

**Using List Iterators**

```cpp
int main () {
 LinkList<int> l;

 ... construct the list l ...

 // print list l
 LinkList<int>::iterator p;
 for (p = l.first(); p != l.end(); ++p)
     cout << *p << ', ';
     cout << '\n';
}
```
Consider Iterators for Chained Hash Tables

- Which pieces of information should an iterator contain?

- Let \( t \) be a chained hash table.
  What is \( t.\text{first}() \)?
  \( t.\text{end}() \)?

- Let \( p \) be an iterator of \( t \). If \( p = t.\text{first}() \), then what is \( ++p \)?
  And then \( ++p \) again?
template<class T>
class ListNode {public: T _data; ListNode* _next; 
};

class CHT_Iterator {
private:
ListNode<T> * _ptr;
ListNode<T> ** _pos; // current position in the table
ListNode<T> ** _end; // end position in the table

public:
CHT_Iterator () { _ptr = NULL; _pos = _end = NULL; }
CHT_Iterator (ListNode<T>* ptr, ListNode<T>** pos,
ListNode<T>** End)
{ _ptr = ptr; _pos = pos; _end = End; }
~CHT_Iterator () {} 

CHT_Iterator& operator++ (); // prefix, ++p
CHT_Iterator operator++ (int); // postfix, p++

bool operator== (CHT_Iterator q)
{ return _ptr == q._ptr && _pos == q._pos && _end == q._end; }
bool operator!= (CHT_Iterator q)
{ return _ptr != q._ptr || _pos != q._pos || _end != q._end; }

CHT_Iterator operator+ (int i)
{
    CHT_iterator result(_ptr,_idx);
    for (; i>0; i--) result++; 
    return result; 
} 

};
CHT_Iterator& CHT_Iterator::operator++()
{
    _ptr = _ptr->next;
    if (_ptr!=NULL) return *this;

    _pos++;
    while (*_pos==NULL && _pos<_end)
        _pos++;
    _ptr = *_pos;
    return *this
}

template <class T>
T& CHT_Iterator::operator*()
{
    return _ptr->data;
}

Assume that a chained hash table contains a "ListNode* _table[CAPACITY]," which holds the "head" pointers to respective linked lists.

CHT_Iterator ChainedHashTable::first ()
{
    ListNode** first_pos = _table;
    ListNode** end_pos = _table+CAPACITY;

    while (*first_pos==NULL && first_pos<end_pos)
        first_pos++;

    return CHT_Iterator(first_pos, *first_pos, end_pos);
};

CHT_Iterator ChainedHashTable::end () {  
    ListNode** end_pos = _table+CAPACITY;
    return (end_pos, NULL, end_pos);
}
A Challenging Exercise

Consider binary tree iterators.

- What kinds of information are needed in a tree iterator?
- How to implement ++?

Summary

- An iterator of a data structure contains information necessary to walk thru the data structure.
- To determine the information that goes into an iterator, try to see what it takes to go from one data entry to the next.
- We re-define the pre/post increment operators (++) and the dereference operator (*) to create an “illusion” that iterators behave like regular pointers.
- We provide the first() and end() methods in the data structure class to locate its first data entry and define the “NULL.”
- Depending on your applications, you may also want to provide the comparison operators (== and !=) and the addition operator (for example, p+3 means moving forward 3 steps).