COMPARISON CLASSES AND GENERIC POINTERS

Objects as functions
Object as generic pointers (iterators)

Problem Solving with Computers-II

C++

```
#include <iostream>
using namespace std;

int main(){
    cout<<"Hola Facebook\n";
    return 0;
}
```

From last class….

80, 10, 2

```
int main(){
      int arr[]={10, 2, 80};
      priority_queue<int> pq;
      for(int i=0; i < 3; i++)
            pq.push(arr[i]);

      while(!pq.empty()){
            cout<<pq.top()<<endl;
            pq.pop();
      }
      return 0;
}
```

heap

What is the output of this code?

A. 10 2 80
B. 2 10 80
C. 80 10 2
D. 80 2 10
E. None of the above

PQ

PQ has to compare keys → if key(a) < key(b), the a has less priority than b

Comparison class

- A class used to perform comparisons.
- Implements a function operator that compares two keys

```
class cmp{
        bool operator()(int& a, int& b) const {
                return a > b;
        }
};

//Use cmp to compare any two keys
cmp foo;
cout<<foo(x, y);
```

x [10] y [20]

function operator

bool operator==(cmp & other);

Recall the definition of overloaded operators like ==

foo is an object that can be used like a function (see below) also called a functor!

Assume x,y are integers foo(x,y) calls the function operator of cmp.

Configure PQ with a comparison class

```
class cmp{
        bool operator()(int& a, int& b) const {
                return a > b;
        }
};
int main(){
        int arr[]={10, 2, 80};
        priority_queue<int, vector<int>, cmp> pq;
        for(int i=0; i < 3; i++)
                pq.push(arr[i]);

        while(!pq.empty()){
                cout<<pq.top()<<endl;
                pq.pop();
        }
        return 0;
}
```

What is the output of this code?

A. 10 2 80
B. 2 10 80
C. 80 10 2
D. 80 2 10
E. None of the above

type of key

underlying representation of the heap

comparison class

can use greater<int> instead

STL has a compare class called greater<int> that has the same implementation as cmp

PQ uses cmp in the following way

```
cmp foo;
if foo(a,b) {
  // a has lower priority than b
} else {
  // b has lower or same priority as a
}
```

W

std::priority_queue template arguments

The template for priority_queue takes 3 arguments:

```
template <
        class T,
        class Container= vector<T>,
        class Compare = less <T>
        > class priority_queue;
```

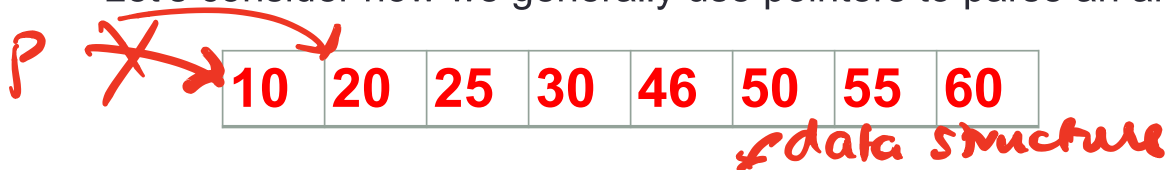
- The first is the type of the elements contained in the queue.
- If it is the only template argument used, the remaining 2 get their default values:
  - a **vector<T>** is used as the internal store for the queue,
  - **less is a comparison** class that provides priority comparisons

CHANGING GEARS: C++STL

- The C++ Standard Template Library is a very handy set of three built-in components:
  - Containers: Data structures stack, set, list, array
  - Iterators: Standard way to search containers
  - Algorithms: These are what we ultimately use to solve problems

C++ Iterators

- Iterators are generalized pointers.
- Let's consider how we generally use pointers to parse an array




```
void printElements(int arr[], int size) {
    int* p= arr;
    for(int i=0; i<size; i++) {
            std::cout << *p << std::endl;
            ++p;
    }
}
```

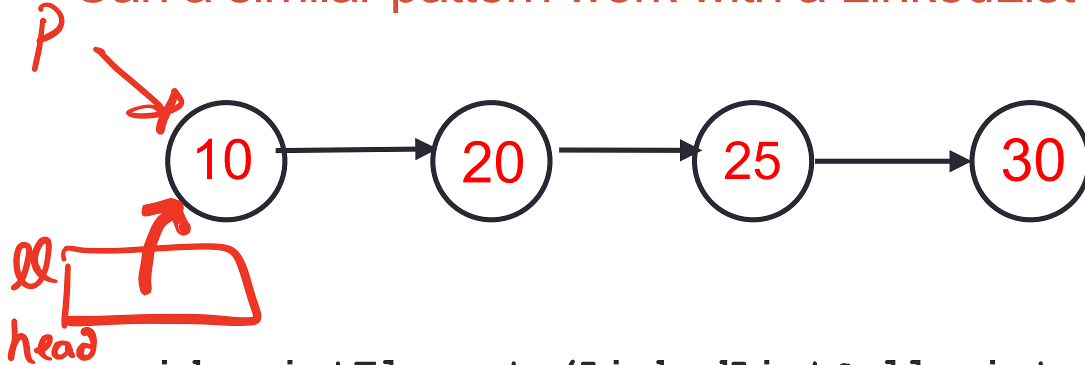
pointer to the first key

get the key

point to the next element.

- We would like our print "algorithm" to also work with other data structures
- E,g Linked list or BST

Can a similar pattern work with a LinkedList? Why or Why not?



No access restrictions

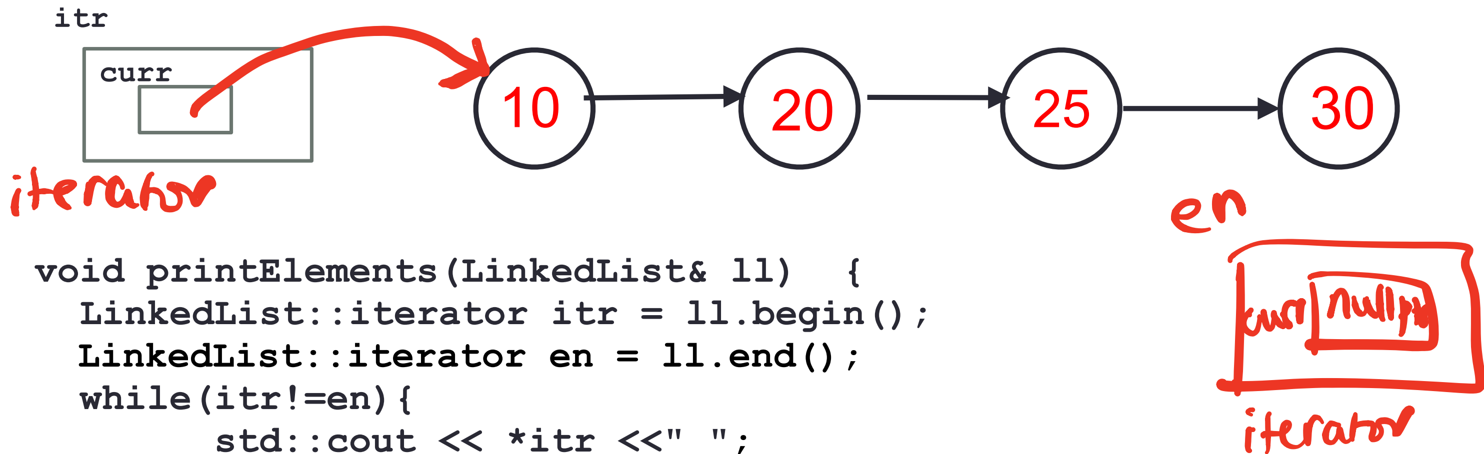
```
void printElements(LinkedList& ll, int size) {
    Node *p = ll.head;                    //How should we define p?
        for(int i=0; i<size; i++) {
            std::cout << *p << std::endl;
            ++p;
        }
}
```

This code doesn't quite work.

The * & ++ operators cannot be overloaded for primitive types

C++ Iterators

- To solve this problem the LinkedList class has to supply to the client (printElements) with a generic pointer (an iterator object) which can be used by the client to access data in the container sequentially, without exposing the underlying details of the class




```
void printElements(LinkedList& ll)  {
   LinkedList::iterator itr = ll.begin();
   LinkedList::iterator en = ll.end();
   while(itr!=en){
          std::cout << *itr <<" ";
          ++itr;
   }
   cout<<endl;
}
```

iterator is a class that contains a pointer
In this case a pointer to a node in the Linked List

C++ Iterators: Initializing the iterator

type → class

function of linkedlist

```
void printElements(LinkedList& ll) {
  LinkedList::iterator itr = ll.begin();
  LinkedList::iterator en = ll.end();
  while(itr!=en){
        std::cout << *itr <<" ";
        ++itr;
  }
  cout<<endl;
}
```

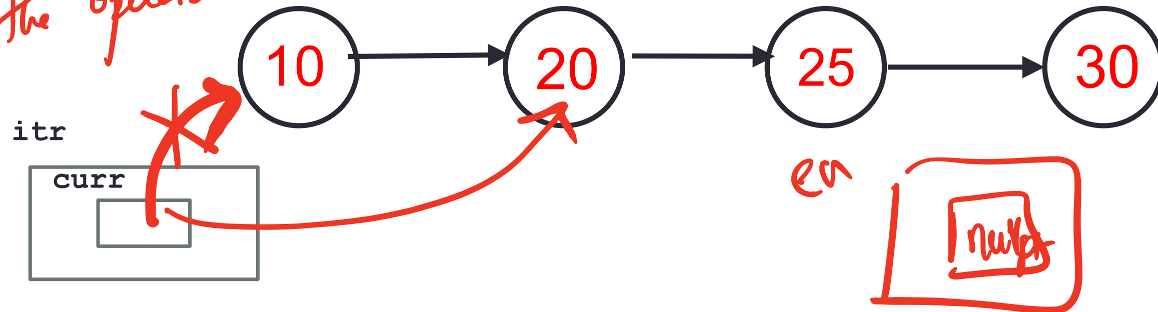
What is the return value of **begin()** ?

A. The address of the first node in the linked list container class

B. An iterator type object that contains the address of the first node

C. None of the above

Need to overload the operators !=, *, ++ for iterator type



C++ Iterators: Overloading operators

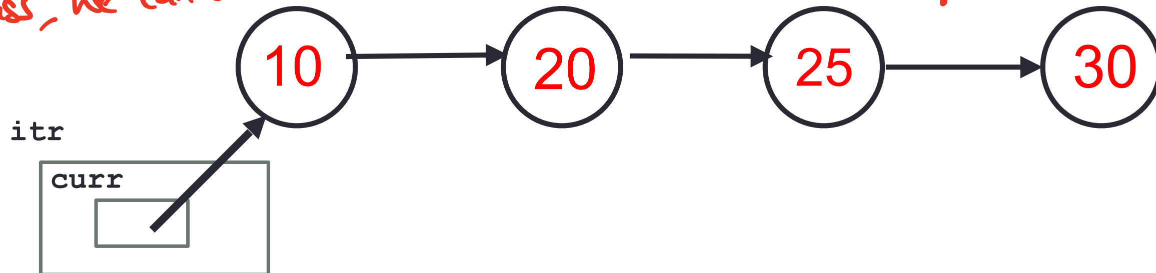
```
void printElements(LinkedList& ll)   {
   LinkedList::iterator itr = ll.begin();
   LinkedList::iterator en = ll.end();
   while(itr!=en){
         std::cout << *itr <<" ";
         ++itr;
   }
   cout<<endl;
}
```

List the operators that must be overloaded for iterator objects?

A. *
B. ++
C. !=
D. All of the above
E. None of the above

(*, ++, !=)

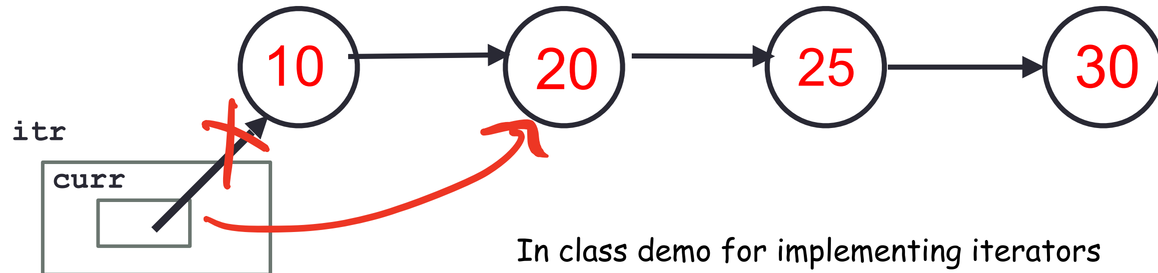
Since iterator is a class, we can overload all these operators on iterator type objects



C++ Iterators

```
void printElements(LinkedList& ll)  {
  LinkedList::iterator itr = ll.begin();
  LinkedList::iterator en = ll.end();
  while(itr!=en){
        std::cout << *itr <<" ";
        ++itr;
  }
  cout<<endl;
}
```

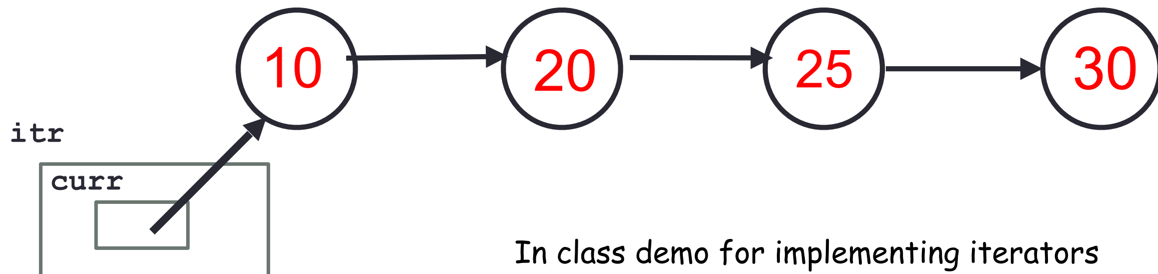
How should the diagram change as a result of the statement ++itr; ?



In class demo for implementing iterators

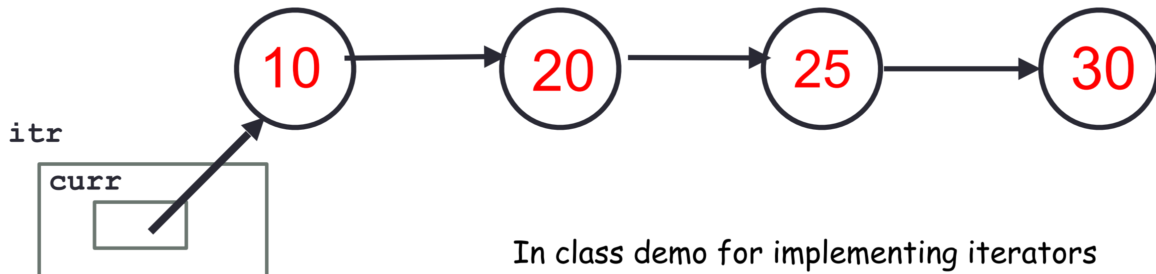
C++ shorthand: auto

```
void printElements(LinkedList& ll)  {
  auto itr = ll.begin();
  auto en = ll.end();
  while(itr!=en){
        std::cout << *itr <<" ";
        ++itr;
  }
  cout<<endl;
}
```



Finally: unveiling the range based for-loop

```
void printElements(LinkedList& ll)  {
  for(auto item:ll){
        std::cout << item <<" ";
  }
  cout<<endl;
}
```



Practice functors and PQs:

```
int main(){
    int arr[]={10, 2, 80};
    priority_queue<int*> pq;
    for(int i=0; i < 3; i++)
        pq.push(arr+i);

    while(!pq.empty()){
        cout<<*pq.top()<<endl;
       pq.pop();
    }
    return 0;
}
```

What is the output of this code?

A. 10 2 80
B. 2 10 80
C. 80 10 2
D. 80 2 10
E. None of the above

Memory locations are stored in the heap & organized as a max-heap

Sort array elements using a pq storing pointers

```
int main(){
      int arr[]={10, 2, 80};
      priority_queue<int*> pq;
      for(int i=0; i < 3; i++)
            pq.push(arr+i);

      while(!pq.empty()){
            cout<<*pq.top()<<endl;
           pq.pop();
      }
      return 0;
}
```

How can we change the way pq prioritizes pointers?

Write a comparison class to print the integers in the array in sorted order

```
int main(){
      int arr[]={10, 2, 80};
      priority_queue<int*, vector<int*>, cmpPtr> pq;
      for(int i=0; i < 3; i++)
            pq.push(arr+i);

      while(!pq.empty()){
            cout<<*pq.top()<<endl;
          pq.pop();
      }
      return 0;
}
```