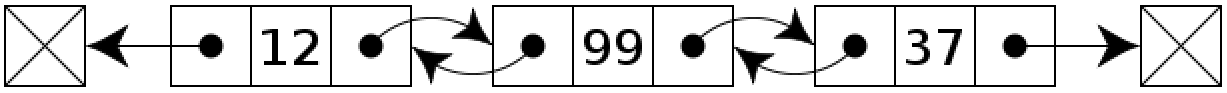
Project #2 – dlist

In this project you are asked to implement a doubly-linked list container (class template `dlist`). A doubly-linked list is a data structure that contains a set of sequentially linked records called nodes. Each node contains data, a link (pointer) to the previous node, and a link to the next node. The node at the beginning of the list is called the head. The node at the end of the list is called the tail. The head's previous link points to null. The tail's next link points to null.

Here is a diagram showing a doubly-linked list containing the values 12, 99, and 37:



For more information on doubly-linked lists how they are implemented internally see *http://en.wikipedia.org/wiki/Doubly_linked_list.*

The following sample demonstrates some of the capabilities of this container:

```
int data[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }

// Iterator range construction
dlist<int> d(data, data + 10);

// Support for begin/end iterators
copy(d.begin(), d.end(), ostream_iterator<int>(cout));
```

You are provided with the following starting point for your `dlist` and `dlist::iterator`:

```
namespace Project2
{
    template <typename T>
    class dlist
    {
    public:
        // Types
       class iterator;
       typedef size_t size_type;
       typedef T value_type;
       typedef const T const_reference;

       // Default constructor
       dlist();

       // Copy constructor
       dlist(const dlist &);

       // Iterator range constructor
       template <typename InputIterator> dlist(InputIterator first,
                                               InputIterator last);
```

```
   // Destructor
   ~dlist();

   // Copy assginment operator
   dlist &operator=(const dlist &);

   // empty() & size()
   bool empty() const;
   size_type size() const;

   // front() & back()
   T &front();
   const T &front() const;
   T &back();
   const T &back() const;

   // Modifiers
   void push_front(const T &);
   void pop_front();
   void push_back(const T &);
   void pop_back();
   iterator insert(iterator, const T &);
   iterator erase(iterator);

   // Comparision
   bool operator==(const dlist &) const;
   bool operator!=(const dlist &) const;
   bool operator<(const dlist &) const;
   bool operator<=(const dlist &) const;
   bool operator>(const dlist &) const;
   bool operator>=(const dlist &) const;

   // Iterators
   iterator begin();
   const iterator begin() const;
   iterator end();
   const iterator end() const;

private:
   // You decide what goes here
};

template <typename T>
class dlist<T>::iterator : public std::iterator<bidirectional_iterator_tag, T>
{
    friend class dlist<T>;

public:
    typedef const T const_reference;

    iterator();
    explicit iterator(typename dlist<T>::node *);

    bool operator==(const iterator &) const;
    bool operator!=(const iterator &) const;

    T &operator*();
    const T &operator*() const;
```

```
            T *operator->();
            const T *operator->() const;

            iterator &operator++();
            const iterator operator++(int);
            iterator &operator--();
            const iterator operator--(int);

        private:
            // You decide what goes here
        };
    }
```

To complete this project you will need to implement dlist and dlist::iterator so that the 50 provided unit tests all pass. In addition, your implementation must be exception safe and exception neutral.

Grading (90 total points available)

1. **(50 points)** One point for each passing unit tests (there are 50 unit tests provided).
2. **(10 points)** dlist is strongly exception safe.
3. **(10 points)** dlist is exception neutral.
4. **(10 points)** dlist has no memory leaks.
5. **(10 points)** dlist is implemented using only ANSI-compliant C++ features, the code is clean (e.g. no duplicate code), the code uses best practices (e.g. operator+ implemented in terms of operator+=).

Turning in the assignment

- Place your dlist.h file in a zip file. Submit this zip file.