




---

---

---

---

---

---

---

---

- ...
- Singly Linked List
- **Each node linked to next**
    - Pointer in node points to next node
    - Last node pointer is null
    - Separate pointer identifies first node
  - **Simpler than doubly linked list**
    - As implemented in STL list
    - See text, pages 725-748
- .....

---

---

---

---

---

---

---

---

- ...
- List Terminology
- **Node - list element**
    - Contains data, link
  - **Head - pointer to first node**
    - Or null if no list elements
  - **Data - actual element info**
  - **Link - pointer to next node**
    - Or null if last element
- .....

---

---

---

---

---

---

---

---

⋮

**Example Program (1)**

```

ListInt list1;
list1.push_back(10);
list1.push_back(20);
list1.push_back(30);

ListIntIter iter = list1.begin();
iter++;
iter = list1.insert(iter,55);

cout << "Inserted data = "
      << (*iter) << endl;

```

.....

⋮

**Example Program (2)**

```

cout << "List size = " << list1.size()
      << endl;
cout << "List data = [ ";

for (iter = list1.begin();
     iter != list1.end();
     iter++)
{
  cout << (*iter) << " ";
}

cout << "]" << endl;

```

.....

⋮

**Simple List Class**

```

class ListInt
{ public:
  ListInt(); ...
  unsigned int size() const;
  ListIntIter begin();
  ListIntIter end();
  bool empty() const;
  void push_back(int val);
  ListIntIter insert(ListIntIter it,
                    int val=0);

  void clear();
private:
  NodeInt* head;
};

```

.....

---

---

---

---

---

---

---

---



---

---

---

---

---

---

---

---



---

---

---

---

---

---

---

---

⋮

**Simple Node Class**

```

class NodeInt
{
public:
  NodeInt(int dta = 0);
  ~NodeInt();
  NodeInt* GetNext() const;
  void SetNext(NodeInt* nxt);
  int& GetDataRef();
private:
  int data;
  NodeInt* next;
};

```

.....

---

---

---

---

---

---

---

---

⋮

**List & Node Constructors**

```

ListInt::ListInt()
: head (0)
{
  // Init empty list
}

NodeInt::NodeInt(int dta)
: data(dta), next(0)
{
  // Init node
}

```

.....

---

---

---

---

---

---

---

---

⋮

**Test for Empty List**

```

bool ListInt::empty() const
{
  return (head == 0);
}

```

.....

---

---

---

---

---

---

---

---

⋮

**Adding a List Element**

```

void ListInt::push_back(int val)
{
  NodeInt* pnew = new NodeInt(val);
  if (head == 0)
  {
    head = pnew;
  }
  else
  { // Add element at back of
    // list. (How?)
  }
}

```

.....

---

---

---

---

---

---

---

---

⋮

**Deleting All List Elements**

```

void ListInt::clear()
{
  while (head != 0)
  {
    NodeInt* p2 = head->GetNext();
    delete head;
    head = p2;
  }
}

```

```

Delete elements in list destructor
ListInt::~ListInt ()
{
  clear();
}

```

.....

---

---

---

---

---

---

---

---

⋮

**List Iterators**

- **So far, cannot traverse list**
  - Move from element to element
- **Must add iterator**
  - As in STL list class
  - Separate object
  - Represents position in a list

.....

---

---

---

---

---

---

---

---

.....

**Iterator Definition**

```

class ListIntIter
{
public:
    ListIntIter(NodeInt* np = 0);
    ...
    int& operator*();
    bool operator!=(const ListIntIter& r);
    ListIntIter& operator++();
    ListIntIter operator++(int);
private:
    [NodeInt* node_ptr;] ← Pointer to
};                               node in a list

```

.....

---

---

---

---

---

---

---

---

.....

**Constructor & Access Operator**

```

ListIntIter::ListIntIter (NodeInt* np)
: node_ptr (np)
{ }

```

```

int& ListIntIter::operator*()
{
    assert (node_ptr != 0);
    [return node_ptr->GetDataRef();]
}

```

Access data in node

.....

---

---

---

---

---

---

---

---

.....

**Iterator Increment (++iter)**

```

ListIntIter& ListIntIter::operator++()
{
    if (node_ptr != 0)
    {
        node_ptr = node_ptr->GetNext();
    }
    return (*this);
}

```

Unary operator -- no argument

.....

---

---

---

---

---

---

---

---

```

...
...
Iterator Increment (iter++)
  Dummy int argument specifies postfix

ListIntIter ListIntIter::operator++(int)
{
  ListIntIter ret (*this);
  operator++();
  return ret;
}

```

Annotations:  
 - Arrow from "Dummy int argument specifies postfix" points to `(int)`.  
 - Arrow from "Retain copy of prior value and return" points to `ListIntIter ret (*this);`.  
 - Arrow from "Call prefix ++ operator" points to `operator++();`.

---

---

---

---

---

---

---

---

```

...
...
List Iterator Functions

ListIntIter ListInt::begin()
{
  return ListIntIter(head);
}

ListIntIter ListInt::end()
{
  return ListIntIter(0);
}

```

---

---

---

---

---

---

---

---

- ```

...
...
Class Exercise

```
- **Complete list implementation**
    - push\_back
    - insert
    - etc.
  - **Convert to doubly linked list?**
    - See textbook example
    - Bidirectional iterators?
    - Simplify insert?




---

---

---

---

---

---

---

---

```
// tstlisti.cpp - Test program for integer list class.
// Mark Sebern
// Version 1.00
// 26 April 1999

#include <iostream>
using namespace std;

#include "listint.h"

void main()
{
    // Create a list and append some elements.
    ListInt list1;

    list1.push_back(10);
    list1.push_back(20);
    list1.push_back(30);

    // Move an iterator into the list and insert an element.
    ListIntIter iter = list1.begin();
    iter++;

    iter = list1.insert(iter,55);

    // Verify the inserted element.
    if (iter != list1.end())
    {
        cout << "Inserted data = " << (*iter) << endl;
    }
    else
    {
        cout << "Insert failed?" << endl;
    }

    // Write out the list data.
    cout << "List size = " << list1.size() << endl;
    cout << "List data = [ ";

    for (iter = list1.begin(); iter != list1.end(); iter++)
    {
        cout << (*iter) << " ";
    }
    cout << "]" << endl;
}
```

```
// listint.h - Definition of integer list class.
// Mark Sebern
// Version 1.00
// 26 April 1999

#ifndef LISTINT_H
#define LISTINT_H

// List node class.
class NodeInt
{
public:
    // Constructor.
    // Arguments:
    //   dta - data value for this node.
    NodeInt(int dta = 0);

    // Destructor.
    ~NodeInt();

    // Get "next" pointer from node.
    // Returns: pointer to "next" node.
    NodeInt* GetNext() const;

    // Set "next" node pointer.
    // Arguments:
    //   nxt - pointer to next node in list.
    void SetNext(NodeInt* nxt);

    // Get reference to data in node.
    // Returns: non-const reference to node data.
    int& GetDataRef();

private:
    // Node data.
    int data;

    // Pointer to following node.
    NodeInt* next;
};

// Forward reference to iterator class. This reference is needed
// because ListInt definition refers to the iterator class.
class ListIntIter;

// List class -- singly linked list.
class ListInt
{
public:
    // Constructor.
    ListInt();

    // Copy constructor.
    ListInt(const ListInt& lis);

    // Destructor.
    ~ListInt();

    // Get number of elements in list.
    unsigned int size() const;

    // Get iterator to beginning of list.
    ListIntIter begin();

    // Get iterator just past end of list.
    ListIntIter end();

    // Test if list is empty.
    // Returns: true if list is empty, false otherwise.
    bool empty() const;

    // Append an element to the list.
    // Arguments:
    //   val - value to append to list.
    void push_back(int val);
};
```

```
// Insert an element in the list.
// Arguments:
//   it - iterator specifying position of element to
//       insert before.
//   val - value of new element.
// Returns: iterator specifying position of new element.
ListIntIter insert(ListIntIter it, int val=0);

// Delete all elements in list.
void clear();

private:
// Pointer to first node in list; null if no elements.
NodeInt* head;
};

// Non-const list iterator class.
class ListIntIter
{
public:
// Constructor.
// Arguments:
//   np - pointer to node iterator is to reference; null if none.
ListIntIter(NodeInt* np = 0);

// Copy constructor.
ListIntIter(const ListIntIter& it);

// Destructor.
~ListIntIter();

// Assignment operator.
ListIntIter& operator=(const ListIntIter& right);

// Get node of element referenced by iterator.
// Returns: pointer to node of referenced element.
NodeInt* GetNodePtr() const;

// Dereference iterator, returning reference to element data.
int& operator*();

// "Not equal" comparison for iterators.
// Returns: true if iterators do not reference same element,
//         false if they do.
bool operator!=(const ListIntIter& right);

// Pre-increment operator.
// Positions iterator to next element if not "end()";
// if iterator value is "end()" the value is not changed.
// Returns: updated iterator value.
ListIntIter& operator++();

// Post-increment operator.
// Positions iterator to next element if not "end()";
// if iterator value is "end()" the value is not changed.
// Returns: original (before increment) iterator value.
ListIntIter operator++(int);

private:
// Pointer to node of element referenced by iterator; null
// if iterator value is "end()".
NodeInt* node_ptr;
};

#endif // LISTINT_H
```

```
// listint.cpp - Implementation of integer list class.
// Mark Sebern
// Version 1.00
// 26 April 1999

// Incomplete! (to be finished as class exercise)

#include <cassert>
#include "listint.h"

NodeInt::NodeInt(int dta)
: data(dta), next(0)
{
    // No further initialization.
}

NodeInt::~NodeInt()
{
    // ??
}

NodeInt* NodeInt::GetNext() const
{
    return next;
}

void NodeInt::SetNext(NodeInt* nxt)
{
    next = nxt;
}

int& NodeInt::GetDataRef()
{
    return data;
}

ListInt::ListInt()
: head(0)
{
    // No further initialization.
}

ListInt::ListInt(const ListInt& lis)
: head(0)
{
    // ??
}

ListInt::~ListInt()
{
    clear();
}

unsigned int ListInt::size() const
{
    return 999; // ??
}

ListIntIter ListInt::begin()
{
    return ListIntIter(head);
}

ListIntIter ListInt::end()
{
    return ListIntIter(0);
}

bool ListInt::empty() const
{
    return (head == 0);
}

void ListInt::push_back(int val)
{

```

```
    NodeInt* pnew = new NodeInt(val);
    if (head == 0)
    {
        head = pnew;
    }
    else
    {
        // ??
    }
}

ListIntIter ListInt::insert(ListIntIter it, int val)
{
    // ??
    return it; // ??
}

void ListInt::clear()
{
    while (head != 0)
    {
        NodeInt* p2 = head->GetNext();
        delete head;
        head = p2;
    }
}

ListIntIter::ListIntIter(NodeInt* np)
: node_ptr (np)
{
    // No further initialization.
}

ListIntIter::ListIntIter(const ListIntIter& it)
{
    // ??
}

ListIntIter::~ListIntIter()
{
    // ??
}

ListIntIter& ListIntIter::operator=(const ListIntIter& right)
{
    // ??
    return *this;
}

NodeInt* ListIntIter::GetNodePtr() const
{
    return node_ptr;
}

int& ListIntIter::operator*()
{
    assert (node_ptr != 0);
    return node_ptr->GetDataRef();
}

bool ListIntIter::operator!=(const ListIntIter& right)
{
    return false; // ??
}

ListIntIter& ListIntIter::operator++()
{
    if (node_ptr != 0)
    {
        node_ptr = node_ptr->GetNext();
    }
    return (*this);
}

ListIntIter ListIntIter::operator++(int)
```

```
{  
    ListIntIter ret (*this);  
    operator++();  
    return ret;  
}
```