ICOM 4015
Advanced Programming

Lecture 13
Data Abstraction I
Reading: Chapter 6

Prof. Bienvenido Vélez

Data Abstraction Lecture Series

- Lecture 1
  - introduction to classes
- Lecture 2
  - encapsulation
- Lecture 3
  - class specialization (inheritance)
- Lecture 4
  - subtype polymorphism
- Lecture 5
  - parametric polymorphism

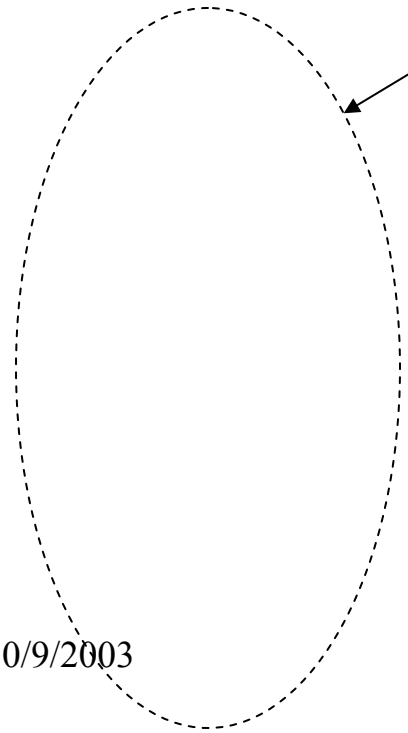
Data Abstraction I
Outline

- Why classes?
- Class declarations
- Method definitions
- Constructors/Destructors

The List ADT

```
// lists.h
// Global declarations for linked lists module

// List data structures

typedef int DatumType;

struct Node {
  DatumType datum;
  Node* next;
  Node* prev;
};

struct List {
  Node* first;
  Node* last;
  Node* cursor;
  bool  atEnd;
  bool  atStart;
  long length;
};

// Operations on linked lists

// List initialization and destruction
void  listInit            (List& l);
void  listDestroy         (List& l);

// List modification
List& listAppend         (List& l, DatumType d);
List& listPrepend        (List& l, DatumType d);
List& listInsert         (List& l, DatumType d);
List& listDelete         (List& l);

// List interation (bidirectional)
DatumType listFirst  (List& l);
DatumType listLast   (List& l);
DatumType listNext   (List& l);
DatumType listPrev   (List& l);
DatumType listCurrent(List l);
bool      listBOL    (List l); // True if at the beginning of l
bool      listEOL    (List l); // True if at the end of l
```

Need to use complex naming schemes to avoid conflicts



Using the List ADT
Without breaking abstraction barrier

```
#include <iostream>

#include "lists.h"

int main() {
  // Define the list
  List l;
  // Initialize the list
  listInit(l);
  // Fill the list
  for(int i=0; i<10; i++) {
    listAppend(l,i);
  }
  // Print the list
  cout << "List contents:" << endl;
  for(DatumType d=listFirst(l); !listEOL(l); d=listNext(l)) {
    cout << "Next datum: " << d << endl;
  }
  // Recicle memory occupied by list
  listDestroy(l);
}
```

No explicit dependencies on pointer implementation of the ADT

Using the List ADT

Breaking the Abstraction Barrier (unintentionally?)

```
#include <iostream>

#include "lists.h"

int main() {
  // Define the list
  List l;
  // Initialize the list
  listInit(l);
  // Fill the list
  for(int i=0; i<10; i++) {
    listAppend(l,i);
  }
  // Print the list
  cout << "List contents:" << endl;
  for(DatumType d=listFirst(l); !listEOL(l); d=listNext(l)) {
    cout << "Next datum: " << d << endl;
  }
  cout << "Breaking the abstration barrier" << endl;
  cout << "List contents:" << endl;
  for(Node* n=l.first; n!=(Node *) NULL; n=n->next) {
    cout << "Next datum: " << n->datum << endl;
  }
  // Recicle memory occupied by list
  listDestroy(l);
}
```

Using the List ADT Output

```
[bvelez@amadeus] ~/icom4015/lec18 >>main
List contents:
Next datum: 0
Next datum: 1
Next datum: 2
Next datum: 3
Next datum: 4
Next datum: 5
Next datum: 6
Next datum: 7
Next datum: 8
Next datum: 9
Breaking the abstration barrier
List contents:
Next datum: 0
Next datum: 1
Next datum: 2
Next datum: 3
Next datum: 4
Next datum: 5
Next datum: 6
Next datum: 7
Next datum: 8
Next datum: 9
[bvelez@amadeus] ~/icom4015/lec18 >>
```

The List Class

```
// listsClass.h
// Global declarations for List ADT


typedef int DatumType;

class Node;

class List {
  Node* first;
  Node* last;
  Node* cursor;
  bool  atEnd;
  bool  atStart;
  long length;

  // Operations on linked lists

 public:
  // List initialization and destruction
  List              ();
  ~List             ();

  // List modification
  List& Append      (DatumType d);
  List& Prepend     (DatumType d);
  List& Insert      (DatumType d);
  List& Delete      ();

  // List interation (bidirectional)
  DatumType First   ();
  DatumType Last    ();
  DatumType Next    ();
  DatumType Prev    ();
  DatumType Current ();
  bool      BOL     (); // True if at the beginning of l
  bool      EOL     (); // True if at the end of l

  // List printing
  void      Dump    ();
};
```

Using the List Class

```
#include <iostream>

#include "listsClass.h"

int main() {
  // Define the list
  List l;
  // Initialize the list. Done transparently by constructor

  // Fill the list
  for(int i=0; i<10; i++) {
    l.Append(i);
  }
  // Print the list
  cout << "List contents:" << endl;
  for(DatumType d=l.First(); !l.EOL(); d=l.Next()) {
    cout << "Next datum: " << d << endl;
  }
  // Destroy list. Done transparently by destructor
}
```

Again, no explicit dependencies on pointer implementation of the ADT

Cannot break abstraction barrier!

```
#include <iostream>

#include "listsClass.h"

int main() {
  // Define the list
  List l;
  // Initialize the list. Done by constructor

  // Fill the list
  for(int i=0; i<10; i++) {
    l.Append(i);
  }
  // Print the list
  cout << "List contents:" << endl;
  for(DatumType d=l.First(); !l.EOL(); d=l.Next()) {
    cout << "Next datum: " << d << endl;
  }
  cout << "Breaking the abstration barrier" << endl;
  cout << "List contents:" << endl;
  for(Node* n=l.first; n!=(Node *) NULL; n=n->next) {
    cout << "Next datum: " << n->datum << endl;
  }
  // Recicle memory occupied by list. Done by destructor

}
```

```
g++ listsClass.cc mainClass.cc -o mainClass
mainClass.cc: In function `int main()':
mainClass.cc:21: member `first' is a private member of class `List'
mainClass.cc:21: invalid use of undefined type `class Node'
mainClass.cc:22: invalid use of undefined type `class Node'

Compilation exited abnormally with code 1 at Wed Mar 29 08:29:55
```

List Class Definition I

```
// lists.cc
// Implementes singly linked lists ADT

// includes ommitted

Node* NullNode = (Node *)NULL;

class Node {
public:
  DatumType datum;
  Node* next;
  Node* prev;
};

// Operations on linked lists

// List initialization
List::List()
{
  List& l = *this;
  l.first = NullNode;
  l.last = NullNode;
  l.cursor = NullNode;
  l.length = 0;
  l.atEnd = true;
}

List::~List()
{
  List& l = *this;
  Node* n=l.first;
  while (n!=NullNode) {
    Node* temp = n;
    n=n->next;
    delete temp;
  }
}
```

List Class Definition II

```
…
…

// List modification
List& List::Append(DatumType d)
{
  List& l = *this;
  Node* temp = new Node;
  temp->next = NullNode;
  temp->prev = NullNode;
  temp->datum = d;
  if (l.first == NullNode) {
    l.first = temp;
    l.last = temp;
  }
  else {
    Node* prev = l.last;
    prev->next = temp;
    temp->prev = prev;
    l.last = temp;
  }
  l.length++;
  return l;
}

…
…
```

Pros of Classes

- No need to explicitly construct/destroy objects
- Better control over namespace
- Better control over abstraction barriers
- Others to come …