Introduction to Object Oriented Programming 2E Timothy A. Budd

Chapter 15

Case Study: Container Classes

© Timothy A. Budd 1996 (Modifications © H. C. Cunningham 1997)

Strong Typing versus Containers

Strong typing useful because it helps to discover programming errors

But strong typing makes development of reusable collection (or container) classes difficult

- To be reusable, container classes must be independent of any application
- Values should not lose their types when placed in container
- Programmer creating reusable container classes will not know type objects held

© Timothy A. Budd 1996 (Modifications © H. C. Cunningham 1997)

Conventional Solution

Conventional containers closely tie the container and the element type

```
type
   Link = Record
      value : integer;
      nextELement : ^ Link;
   end;
```

© Timothy A. Budd 1996 (Modifications © H. C. Cunningham 1997)

Containers in Dynamic Languages

- Containers in dynamic languages not difficult – values "know" their own types
- Containers hold values of type Object
- Containers in strongly typed languages are much more difficult

© Timothy A. Budd 1996 (Modifications © H. C. Cunningham 1997)

Three Solutions

- Containers maintain elements derived from a common parent class
- Containers maintain pointers, which can be cast to appropriate type
- Templates

© Timothy A. Budd 1996 (Modifications © H. C. Cunningham 1997)

Common Base Link Type

- Programmer creating linked list abstraction creates generic "link" class, independent of any application
- Programmer using list abstraction then subclasses this generic class for values specific to problem

© Timothy A. Budd 1996 (Modifications © H. C. Cunningham 1997)

The Generic List Class

```
class List {
public:
      // constructor
   List() : firstElement(0) { }

      // operations
   void Add(Link * n)
      { n→next = firstElement; firstElement = n; }

   int isEmpty()
      { return firstElement == 0; }

   Link * First();
      { return firstElement; }

   void RemoveFirst();
      { Link * p = firstElement;
        firstElement = firstElement→ next;
        delete p; }

   int  Includes(Link * n)
      { for (Link * p = firstElement; p; p = p→next)
          if (*p == *n) return 1;
        return 0; }
```

© Timothy A. Budd 1996 (Modifications © H. C. Cunningham 1997)

```
private:
        // data field
    Link * firstElement;
};

class Link {
public:
        // constructors
    Link () : next(0) { }
    Link (Link * n) : next(n) { }

        // operations
    virtual int operator == (Link *) = 0;

        // data field
    Link * next;
};
```

© Timothy A. Budd 1996 (Modifications © H. C. Cunningham 1997)

A Link for Double Precision Numbers

```
class  DoubleLink : public Link {
public:
    // data area
    double theValue;

    // constructor
    DoubleLink (double v) : theValue(v) {}

    // operations
    int operator == (Link * d)
        { return thisValue == ((DoubleLink *) d)→theValue; }
};
```

© Timothy A. Budd 1996 (Modifications © H. C. Cunningham 1997)

Using the List Class

Using the list class involves explicitly creating links.

```
List aList;

aList.Add(new DoubleLink(2.71828));
aList.Add(new DoubleLink(1.61803));
aList.Add(new DoubleLink(3.14159));

DoubleLink * pi;

pi = (DoubleLink *) aList.First();
aList.RemoveFirst();
```

© Timothy A. Budd 1996 (Modifications © H. C. Cunningham 1997)

Hiding the Links

Can avoid explicit manipulation of links by subclassing the List class as well.

```
class DoubleList : public List {
public:
    void Add(double v)
        { List::add(new DoubleLink(v)); }

    double First()
        { return firstElement→value; }

    int Includes(double v)
        { DoubleLink v1(v);
          return List::includes(v1); }
};
```

Containers that Hold Only Pointers

- In many applications containers used only to hold pointers, not values themselves.
- Often because only pointers or references are polymorphic variables in C++
- Create generic link class that maintains a void pointer – no need to subclass Link

© Timothy A. Budd 1996 (Modifications © H. C. Cunningham 1997)

List and Link

```
class List {
public:
    ... // same as before

    void Add(void * v)
        { firstElement = new Link(v, firstElement); }

    void * First()
        { return firstElement→value; }

    ... // same as before
};

class Link {
        // constructors
    Link (void * v) : next(0), value(v) {}
    Link (void * v, Link * n) : next(n), value(v) {}

        // data areas
    Link * next;
    void * value;
};
```

Example: list of Windows

```
class WindowList : public List {
public:
    void Add(Window * w)
        { List::add(w); }

    Window * First()
        { return (Window *) List::first(); }
};
```

© Timothy A. Budd 1996 (Modifications © H. C. Cunningham 1997)

Best Alternative: Templates

- Third alternative is the best
- Template allows class description to be parameterized
- Parameter values are *types* – used as type names in class description

© Timothy A. Budd 1996 (Modifications © H. C. Cunningham 1997)

A Template List Class

```
template <class T>
class List {
      // constructors
   List();

      // operations
   void Add(T v)
      { firstElement = new Link<T>(v, firstElement); }

   int isEmpty()
      { return firstElement == 0; }

   T First()
      { return firstElement→value; }

   void RemoveFirst()
      { Link<T> * p = firstElement;
        firstElement = p→next;
```

© Timothy A. Budd 1996 (Modifications © H. C. Cunningham 1997)

```
        delete p; }

    int Includes(T v)
        { for (Link<T> * p = firstElement; p; p = p->next)
            if (v == p->value)
                return 1;
          return 0; }
};

template <class T>
class Link {
    // constructor
    Link (T v, Link * n) : value(v), next(n) { }

    // data fields
    Link * next;
    T value;
};
```

Using Template Classes

```
List<double> aList;

aList.Add(2.71828);
aList.Add(1.61803);
aList.Add(3.14159);

double pi;

pi = aList.First();
aList.removeFirst();
```

© Timothy A. Budd 1996 (Modifications © H. C. Cunningham 1997)

Advantages of Templates (Generics)

- Do not require any new subclasses to be created for use
- General facility, now part of C++ standard definition

Generics are available in other OOP languages such as Eiffel and Ada 95, but not in Java

© Timothy A. Budd 1996 (Modifications © H. C. Cunningham 1997)

Iteration and Iterators

Want mechanism for iteration over containers of values without exposing implementation

Can be performed using an *iterator*

In C++ can be made to look like a pointer

```
list<int>::iterator start = aList.begin();
list<int>::iterator end = aList.end();

for ( ; start != end; ++start)
    cout << (*start) << endl;
```

Iterator Class

```
template <class T> class listIterator {
    ListIterator (Link<T> & sl) : currentLink(sl) { }

    void operator ++ () { currentLink = currentLink→nextLink; }

    T operator * () { return currentLink→value; }

    bool operator == (ListIterator<T> & right)
    {   return currentLink == right.currentLink; }

private:
    Link<T> * currentLink;
};
```

© Timothy A. Budd 1996 (Modifications © H. C. Cunningham 1997)

Containers in Java

- Common parent class for all objects, class Object
- All object variables hold references (pointers)
- Object wrapper classes for primitive types
- Values "know" their type, type-safe casting provided
- No template (generic) facility (possibly in future)
- API package java.util provides several container classes (e.g., Stack, Vector, Hashtable)
- Java 1.1 has reflection API, Java Beans component architecture

© Timothy A. Budd 1996 (Modifications © H. C. Cunningham 1997)