Object-Oriented Design and Programming

C++ Container Classes

Outline

Introduction

- Container classes are an important category of ADTs
  - They are used to maintain collections of elements like stacks, queues, linked lists, tables, trees, etc.
- Container classes form the basis for various C++ class libraries
  - Note, making class libraries is a popular way to learn C++...
- C++ container classes can be implemented using several methods:

  (0) *Ad hoc,* rewrite from scratch each time
  (1) Preprocessor Macros
  (2) A **genclass** Facility (*e.g.*, GNU libg**++**)
  (3) Parameterized Types
  (4) **void** Pointer Method
- Note, methods 1–3 apply to *homogeneous* collections; method 4 allows *heterogeneous* collections

Container Class Objectives

- *Application Independence*
  - Transparently reuse container class code for various applications
- *Ease of Modification*
  - Relatively easy to extend classes to fit smoothly into a new application
- *Ease of Manipulation*
  - Implementation must hide representation details, *e.g.*, iterators

Container Class Objectives (cont'd)

- *Type Safety*
  - Insure that the collections remain type safe
    * This is easy for parameterized types, harder for **void** pointers...
- *Run-Time Efficiency and Space Utilization*
  - Different schemes have different tradeoffs
    * *e.g.*, extra indirection vs flexibility

Object-Oriented Class Library Architecture

- Two general approaches are *tree* vs *forest* (differ in their use of inheritance):

  Tree: create a single rooted tree of classes derived from a common base class, *e.g.*, *object*

  – *e.g.*, standard Smalltalk libraries or NIHCL

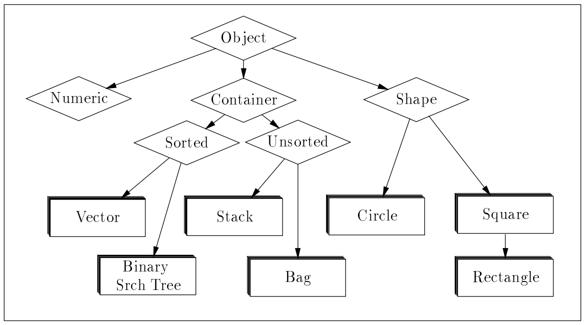
  Forest: a collection of generally independent classes available for individual selection and use

  – *e.g.*, GNU libg++ library, Borland C++ class library, Booch components, Rogue Wave, USL Standard components

- Tradeoffs:

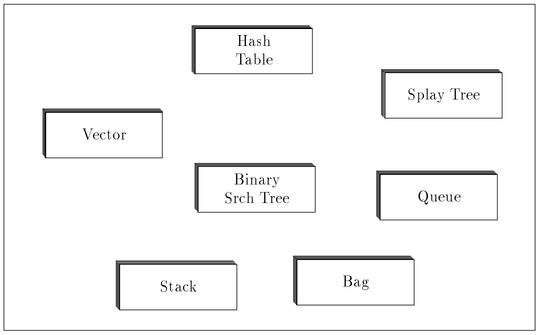
  1. *Uniformity* (Tree) vs *flexibility* (Forest)
  2. *Sharing* (Tree) vs *efficiency* (Forest)
     – Forest classes do not inherit unnecessary functions

Object-Oriented Class Library Architecture (cont'd)



- Tree-based class library

Object-Oriented Class Library Architecture (cont'd)



- Forest-based class library

Parameterized Types

- Parameterized list class

```
template <class T>
class List {
public:
      List (void): head_ (0) {}
      void prepend (T &item) {
            Node<T> *temp =
                  new Node<T> (item, this->head_);
            this->head_ = temp;
      }
      /* ...*/
private:
      template <class T>
      class Node {
      private:
            T value_;
            Node<T> *next_;
      public:
            Node (T &v, Node<T> *n)
                  : value_ (v), next_ (n) {}
      };
      Node<T> *head_;
};

int main (void) {
      List<int> list;
      list.prepend (20);
      // ...
}
```

Parameterized Types (cont'd)

- Parameterized Vector class

```
template <class T = int, int SIZE = 100>
class Vector {
public:
        Vector (void): size_ (SIZE) {}
        T &operator[] (size_t i) {
                return this->buf_[i];
        }
private:
        T buf_[SIZE];
        size_t size_;
};
int main (void) {
        Vector<double> d; // 100 doubles
        Vector<int, 1000> d; // 1000 ints
        d[10] = 3.1416;
}
```


Preprocessor Macros

- Stack example (using GNU g++)

```
#ifndef _stack_h
#define _stack_h

#define name2(a,b) gEnErIc2(a,b)
#define gEnErIc2(a,b) a ## b
#define Stack(TYPE) name2(TYPE,Stack)

#define StackDeclare(TYPE) \
class Stack(TYPE) { \
public: \
        Stack(TYPE) (size_t size): size_ (size) { \
                this->bottom_ = new TYPE[size]; \
                this->top_ = this->bottom_ + size; \
        } \
        TYPE pop (void) { \
                return *this->top_++; \
        } \
        void push (TYPE item) { \
                *--this->top_ = item; \
        } \
        bool is_empty (void) { \
                return this->top_ == this->bottom_ \
                        + this->size_; \
        } \
        bool is_full (void) { \
                return this->bottom_ == this->top_; \
        } \
        ~Stack(TYPE) (void) { delete this->bottom_; } \
private: \
        TYPE *bottom_; \
        TYPE *top_; \
        size_t size_; \
}
#endif
```


Preprocessor Macros (cont'd)

- Stack driver

```
#include <stream.h>
#include "stack.h"
StackDeclare (char);
typedef Stack(char) CHARSTACK;
int main (void) {
        const int STACK_SIZE = 100;
        CHARSTACK s (STACK_SIZE);
        char c;
        cout << "please enter your name..: ";

        while (!s.is_full () && cin.get (c) && c != '\n')
                s.push (c);

        cout << "Your name backwards is..: ";
        while (!s.is_empty ())
                cout << s.pop ();
        cout << "\n";
}
```

- Main problems:

(1) Ugly ;-)
(2) Code bloat
(3) Not integrated with compiler

genclass

- Technique used by GNU libg++
  - Uses **sed** to perform text substitution

```
sed -e "s/<T>/$T1/g" -e "s/<T&>/$T1$T1ACC/g"
```

- Single Linked List class

```
class <T>SLList {
public:
        <T>SLList (void);
        <T>SLList (<T>SLList &a);
        ~<T>SLList (void);
        <T>SLList &operator = (<T>SLList &a);
        int empty (void);
        int length (void);
        void clear (void);
        Pix prepend (<T&> item);
        Pix append (<T&> item);
        /* ...*/
protected:
        <T>SLListNode* last_;
};
```

void Pointer Method

- General approach:
  - **void** * pointers are the actual container elements
  - Subclasses are constructed by coercing **void** * elements into pointers to elements of interest
- *Advantages*:
  1. Code sharing, less code redundancy
  2. Builds on existing C++ features (*e.g.*, inheritance)
- *Disadvantages*:
  1. Somewhat awkward to design correctly
  2. Inefficient in terms of time and space (requires dynamic allocation)
  3. Reclamation of released container storage is difficult (need some form of garbage collection)

void Pointer Example

- One example application is a generic ADT *List* container class. It contains four basic operations:
  1. *Insertion*
     - add item to either front or back
  2. *Membership*
     - determine if an item is in the list
  3. *Removal*
     - remove an item from the list
  4. *Iteration*
     - allow examination of each item in the list (without revealing implementation details)
- The generic list stores pointers to elements, along with pointers to links
  - This allows it to hold arbitrary objects (but watch out for type-safety!!)

void Pointer Example (cont'd)

- Generic_List.h

```
#ifndef Generic_List
#define Generic_List
class List {
public:
    List (void);
    ~List (void);
    void remove_current (void);
    // Used as iterators...
    void reset (void);
    void next (void);
protected:
    class Node {
    friend List;
    public:
        Node (void *, Node *n = 0);
        ~Node (void);
        void add_to_front (void *);
        void add_to_end (void *);
        Node *remove (void *);
    private:
        void *element_; // Pointer to actual data
        Node *next_;
    };
```


- Generic_List.h (cont'd)

```
protected:
    // used by subclasses for implementation
    void add_to_end (void *);
    void add_to_front (void *);
    Node *current_value (void);
    void *current (void);
    bool includes (void *);
    void *remove (void *);

    // important to make match virtual!
    virtual bool match (void *, void *);
private:
    Node *head_;
    Node *iter_; // used to iterate over lists
};
```

- Generic_List.h (cont'd)

```
// Iterator functions
inline List::Node *List::current_value (void) {
	return this->iter_;
}

inline void List::reset (void) {
	this->iter_ = this->head_;
}

inline void *List::current (void) {
	if (this->iter_)
		return this->iter_->element_;
	else
		return 0;
}

inline void List::next (void) {
	this->iter_ = this->iter->next_;
}
```


- Generic_List.C

```
// Node methods
inline List::Node::Node (void *v, List::Node *n)
	: element_ (v), next_ (n) {}

inline List::Node::~Node (void) {
	if (this->next_) // recursively delete the list!
		delete this->next_;
}

inline void List::Node::add_to_front (void *v) {
	this->next_ = new List::Node (v, this->next_);
}
void List::Node::add_to_end (void *v) {
	if (this->next_) // recursive!
		this->next_->add_to_end (v);
	else
		this->next_ = new List::Node (v);
}
List::Node *List::Node::remove (void * v) {
	if (this == v)
		return this->next_;
	else if (this->next_) // recursive
		this->next_ = this->next_->remove (v);
	return this;
}
```


- Generic_List.C

```
// List methods
void List::add_to_front (void *v) {
	this->head_ = new List::Node (v, this->head_);
}
void List::add_to_end (void *v) {
	if (this->head_) // recursive!
		this->head_->add_to_end (v);
	else
		this->head_ = new List::Node (v);
}
bool List::includes (void *v) {
	// Iterate through list
	for (this->reset (); this->current (); this->next ())
		// virtual method dispatch!
		if (this->match (this->current (), v))
			return true;
	return false;
}
bool List::match (void *x, void *y) {
	return x == y;
}
```


- Generic_List.C (cont'd)

```
void List::remove_current (void) {
	if (this->head_ == this->iter_)
		this->head_ = this->iter_->next_;
	else
		this->head_ = this->head_->remove (this->iter_);
	this->iter_->next_ = 0;
	delete this->iter_; // Deallocate memory
	this->iter_ = 0;
}

void *List::remove (void *v) {
	for (this->reset (); this->current (); this->next ())
		if (this->match (this->current (), v)) {
			void *fv = this->current ();
			this->remove_current();
			return fv;
		}
	return 0;
}

inline List::List (void): head_ (0), iter_ (0) {}

List::~List (void) {
	if (this->head_) delete this->head_; // recursive!
}
```

void Pointer Example (cont'd)

- Card.h

```
#include "Generic_List.h"
class Card {
      friend class Card_List;
public:
      enum Suit {
            SPADE = 1, HEART = 2, CLUB = 3, DIAMOND = 4
      };
      enum Color { BLACK = 0, RED = 1 };
      Card (int r, int s);
      int rank (void);
      Suit suit (void);
      Color color (void);
      bool operator == (Card &y);
      void print (ostream &);
private:
      int rank_;
      Suit suit_;
};
```


- Card.h

```
inline int Card::rank (int) { return this->rank_; }
inline Card::Suit Card::suit (void) { return this->suit_; }

inline bool Card::operator == (Card &y) {
      return this->rank () == y.rank ()
            && this->suit () == y.suit();
}

inline void Card::print (ostream &str) {
      str << "suit " << this->suit ()
            << "rank " << this->rank () << endl;
}

inline Card::Card (int r, Card::Suit s)
            : rank_ (r), suit_ (s) {}

inline Card::Color Card::color (void) {
      return Card::Color (int (this->suit ()) % 2);
}
```


- Card_List.h

```
#include "Card.h"

class Card_List : public List {
public:
      void add (Card *a_card) {
            List::add_to_end (a_card);
      }

      Card *current (void) {
            return (Card *) List::current ();
      }

      int includes (Card *a_card) {
            return List::includes (a_card);
      }

      void remove (Card *a_card) {
            List::remove (a_card);
      }

      void print (ostream &);

protected:
      // Actual match function used by List!
      virtual bool match (void *, void *);
};
```


- Card_List.C

```
// Virtual method
bool Card_List::match (void *x, void *y) {
      Card &xr = *(Card *) x;
      Card &yr = *(Card *) y;
      // Calls Card::operator ==
      return xp == yp;
}

void Card_List::print (ostream &str) {
      for (this->reset (); this->current (); this->next ())
            this->current ()->print (str);
}
```

- main.C

```
#include "Card.h"
int main (void) {
      Card_List cl;

      Card *a = new Card (Card::HEART, 2);
      Card *b = new Card (Card::DIAMOND, 4);
      Card *c = new Card (Card::CLUB, 3);

      cl.add (a); cl.add (b); cl.add (c); cl.add (b);

      cl.print (cout);

      if (cl.includes (new Card (Card::DIAMOND, 4)))
            cout << "list includes 4 of diamonds\n";
      else
            cout << "something's wrong!\n";

      cl.remove (new Card (Card::CLUB, 3));
      cl.print (cout);
      return 0;
}
```

- Main problem:
  - Must dynamically allocate objects to store into generic list!
    - Handling memory deallocation is difficult without garbage collection or other tricks...