Object-Oriented Design and Programming

C++ Container Classes

Outline

Introduction
Container Class Objectives
Class Library Architecture
Parameterized Types
Preprocessor Macros
genclass
void Pointer Method
void Pointer Example

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

Type Safety
- Ensure 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)

- Forest-based class library

Parameterized Types

- Parameterized list class

```cpp
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

```cpp
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;
};
```

```cpp
template <class TYPE>
class StackDeclarer {
    Stack(TYPE) (size_t size) {
        this->bottom_ = new TYPE[size];
        this->top_ = this->bottom_ + size;
    }
}
```

```cpp
template <class TYPE>
class Stack {
public:
    Stack(TYPE) (size_t size) {
        this->bottom_ = this->top_ + size;
        this->top_ = this->bottom_ + size;
    }
    void push (TYPE item) {
        this->top_ = item;
        this->bottom_ == item;
    }
    bool is_empty (void) {
        return this->top_ == this->bottom_;
    }
    bool is_full (void) {
        return this->bottom_ == this->top_;
    }
};
```

```cpp
template <class TYPE>
class SLList {
private:
    TYPE *bottom_; 
    TYPE *top_; 
    size_t size_
};
```

```cpp
#ifndef STACK_H
#define STACK_H

define name2(a,b) gEnEric2(a,b)
define gEnEric2(a,b) a ## b
define Stack(TYPE) name2(2,Stack)

```Preprocessor Macros

- Stack example (using GNU g++)

```cpp
#include <stream.h>
#include "stack.h"
StackDeclarer (char);
typedef Stack(char) CHARSTACK;
```

```cpp
int main (void) {
    const int STACK_SIZE = 10;
    CHARSTACK s (STACK_SIZE);
    char c;
    cout << "please enter your name.:");

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

    cout << "Your name backwards is.:");
    while (s.is_empty ()
        cout << s.pop (c);
        cout << 
    }

- Main problems:

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

Preprocessor Macros (cont'd)

- Stack driver

```cpp
#include <include_main
#include <include_stream.h

```

```cpp
genclass
```

- Technique used by GNU libg++

```cpp
- Uses sed to perform text substitution

```cpp
```

```cpp
- Single Linked List class

```cpp
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 prepent (<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
  
- 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

```c
#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.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::add_to_front (void *v)
        { if (this->next_) // recursive!
            this->next_ = new List::Node (v, this->next_);
        else
            this->head_ = new List::Node (v);
    }
    
    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;}
    
    void List::remove (void *v)
        { if (this->head_ == this->iter_.next_)
            this->head_ = this->iter_.next_.next_;
        else
            this->head_ = this->head_->remove (this->iter_.next_);
        this->iter_.next_ = 0;
        delete this->iter_; // Deallocate memory
        this->iter_ = 0;
    }
    
    void List::remove_current (void)
        { if (this->head_ == this->iter_.next_)
            this->head_ = this->iter_.next_.next_;
        else
            this->head_ = this->head_->remove (this->iter_.next_);
        this->iter_.next_ = 0;
        delete this->iter_;}
    
    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

```cpp
#include "GenericList.h"
class Card {
    friend class CardList;
public:
    enum Suit {
        SPADE = 1, HEART = 2, CLUB = 3, DIAMOND = 4
    }
    enum Color {
        BLACK = 0, RED = 1
    }
    int rank, int suit;
    Color color;
    bool operator == (Card &y) {
        return this->rank() == y.rank ()
            && this->suit() == y.suit();
    }
    inline Card::Color Card::color (void) {
        return Card::Color (int (this->suit()) % 2);
    }
private:
    int rank, suit;
};
```

- CardList.h

```cpp
#include "Card.h"
class CardList : public List {
public:
    void add (Card *a_card) {
        List::add_to_end (a_card);
    }
    Card *current (void) {
        return 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:
    virtual bool match (void *, void *);
};
```

- CardList.C

```cpp
bool CardList::match (void *x, void *y) {
    Card &xr = *(Card *) x;
    Card &yr = *(Card *) y;
    // Calls Card::operator ==
    return xp == yp;
}
void CardList::print (ostream &str) {
    for (this->reset (); this->current (); this->next ())
        this->current ()->print (str);
}
```
main.C

```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. .