Linked Lists, Pointers to Objects, & the Queue Abstract Data Type

Topics Covered

- Reading: 503-560 (Chapter 12, review 1st half)
- Classes and implicit member function review
- Using pointers to objects
- Using Linked Lists
- Code reuse [using opaque types]
- The Queue ADT
- Homework 9, 10

Implicit Member Functions

- C++ "writes" five member functions for you
  - Often, the built-in functions don't "do the right thing"
  - When that's true, you must write your own
- The five "built-in" member functions are
  - The default [no argument] constructor
  - The destructor
  - The copy constructor
  - The assignment operator
  - The address operator
The Default Constructor I

- **When is the default constructor called?**
  - 1) When you create an object variable with no args
    - `Frog f1; // NOT Frog f1();`
  - 2) When you create an array of objects
    - `Frog herd[10];`

- **What does the built-in constructor do?**
  - Nothing; the variable's fields are undefined

The Default Constructor II

- **When should you write a default constructor?**
  - 1) If users will create object arrays or no-arg objects
  - 2) You have written other constructors

- **When should you avoid writing a default ctor?**
  - You want to force users to use a specific constructor

- **How should you write the default constructor?**
  - `Frog::Frog() { legs = 4; }
  - `Frog::Frog(int lg = 4) { legs = lg; }

The Default Destructor

- **When is the destructor invoked?**
  - 1) For automatic objects, when goes out of scope
  - 2) For static objects, when program ends
  - 3) For dynamic objects, when programmer deletes

  ```
  int func() {
      Frog *pFrog = new Frog;
      static Frog staticFrog;
      Frog autoFrog;
      delete pFrog;
  }
  ```
The Default Destructor II

- When should you write a destructor?
  - 1) If the constructor allocates memory with new
  - 2) If the constructor creates an O/S resource
     » "Resource acquisition is initialization"
     » Use destructor and local objects to simplify code
  - 3) If your class can be extended [almost always]
     » We’ll learn about this when we look at inheritance &
       virtual functions

The Copy Constructor I

- When is the copy constructor invoked?
  -Whenever a copy of an object is created from another
     » Explicit or implicit initialization
     » Passing arguments by value
     » Temporary objects created by overloaded operators
     » Returning objects from methods (sometimes)

- What does the default copy constructor do?
  - It makes a member-wise copy of all fields

The Copy Constructor II

- When do you need to write a copy constructor?
  - When you have pointers as data fields that refer to
    allocated memory or resources

- What must your copy constructor do?
  - Allocate space for a copy of the external data
    » Update pointer field with address of new memory
  - Copy data from one external area to another
  - Update all the rest of the data members
The Assignment Operator

- When is the assignment operator invoked?
  - When the value of an existing object is copied to another existing object

- How does the built-in assignment operator work?
  - Performs a member-wise copy of all fields

- When do you need to write your own?
  - If your constructor allocates external memory
  - You must perform a deep copy like copy constructor

The Assignment Operator II

- How do you write the assignment operator?
  - Similar to copy constructor, with a few differences
  - 1) Make sure that you are not self-assigning
     » Test the value of this against address of argument
  - 2) Release existing resource [which will be replaced]
  - 3) Perform your allocation and deep copy
  - 4) Return a reference to caller [usually *this]

Using new with Constructors

- Here are some useful rules from your book:
  - 1) Use new in constructor? Use delete in destructor
  - 2) Use new[ ] in ctor? Use delete [ ] in dtor;
  - 3) All versions of ctor must use new in same way
  - 4) You must write a copy constructor [deep copy]
  - 5) You must write an assignment operator

- What if you don't want to support copy or assign?
  - Add dummy member functions in private section
Creating pointers to objects
- Use the same syntax as pointers to primitives:
  - int i, *pi; // int and pointer to int
  - Bird b, *pb; // Bird and pointer to bird

- Creating a pointer does not call constructor
- You can assign address of object just like primitive
  - pi = &i;
  - pb = &b;

Pointers to Objects II

- Often, pointers to objects use dynamic memory
  - Constructor is called when new operator invoked

- Bird b; // Default constructor
- Bird *pb; // No constructor
- pb = &b; // Points to existing Bird
- delete pb; // ILLEGAL - didn't use new
- pb = new Bird; // Default constructor
- delete pb; // Destructor, created with new

Pointers to Objects III

- Use a pointer to object just like a regular pointer
  - To access a member function, use member selection

- Bird *pb = new Bird;
  - pb->sing(); // OK, member selection
  - pb.sing(); // Not OK

- Use the indirection operator to get object

- Bird *pb = new Bird, cb;
  - (*pb).sing(); // OK
  - cb = *pb; // OK
**Pointers to Objects IV**

- Object pointers can be passed to functions
  - Destructors and constructors are not called
    ```
    void display(Bird b) // Copy constructor
    {
    cout << b << endl;
    } // Destructor
    void display(Bird * pb) // No constructor
    {
    cout << *(pb) << endl;
    } // No destructor
    ```

**Pointers to Objects V**

- You can use pointers to create arrays of objects
  - Always uses the default constructor [no choice]
    ```
    Bird *flock = new Bird[size];
    ```
  - Don't have a default constructor? Use pointer array:
    ```
    Bird **flock = new Bird*[size];
    for (int i = 0; i < size; ++)
    flock[i] = new Bird(…);
    ```
  - Complication? Must delete each item separately.

**Introduction to Linked Lists**

- Basis for most advanced programming
- **Static allocation: Arrays**
  - Know location of next item by deduction
    - size of object + address of array
- **Linked allocation: Linked Lists & Trees**
  - Address of next item must be explicitly described
  - To find a particular item, you follow the trail of addresses until you get the item you want.
The List node

- Linked-list elements are called nodes
- A node is composed of two parts:
  - The data part: holds information
  - The pointer part: holds address of next element
- Often a node is created using struct:

  ```
  struct node {
    DATA_ITEM data;
    node * next;
  };
  ```

A Linked-List Class

- A Linked-List object is separate from the nodes
  - Here's one possible simple implementation
  - Use three node pointers to "anchor" list
    - node *head, *current, *tail;

Building a List I

- A list can store any kind of data
  - We'll create a CStringList class that holds C-style character strings.
- Here are the methods we'll use:
  - 1. Constructor sets all of the fields to 0
  - 2. A destructor to free all the items in the list
  - 3. An add() function to add a new string
  - 4. A first() and next() method to navigate
  - 5. An eol() method to tell when you're at last item
  - 6. A c_str() method to retrieve the current data
Building a List II

- Let's declare our node struct inside the class
  - In private section, it is hidden outside. Here's how a CStringList class would look start: [CStringListII.cpp]

```cpp
class List {
  struct node {
    char data[SIZE];
    node * next;
  };
  node * next, *current, *tail;
 public: // Methods go here
};
```

Building a List III

- To construct an empty list, set all fields to 0

```cpp
CStringList::CStringList()
{
  head = current = tail = 0;
}
```

- To move to start of the list, set current to head

```cpp
void CStringList::first()
{
  current = head;
}
```

Building a List IV

- If current is 0, then you are on the last item

```cpp
bool CStringList::eol() { return current == 0; }
```

- To move to next, set current to current->next
  - Only if current is not 0

```cpp
void CStringList::next()
{
  if (current != 0)
    current = current->next;
}
To return the data part, write an accessor

```c
const char * c_str()
{
    if (current != 0)
        return current->data
    static const char * emptyString = "";
    return emptyString;
}
```

May want to handle with an assertion instead

---

Adding data is the most complex part
- 1) Allocate a new node, using current
- 2) Set current->next to 0 [NULL]
- 3) Copy the data into current->data
- 4) If head is 0 [first time], set head to current
   otherwise, set tail->next to current
- 5) Set tail to current

Member function `add()` shown on following page

---

The `CStringList::add()` member function

```c
void CStringList::add(const char * newStr)
{
    current = new node;
    current->next = 0;
    strncpy(current->data, newStr, SIZE);
    if (head == 0) head = current;
    else tail->next = current;
    tail = current;
}
```
Building a List VIII

- Cleaning up when done in destructor
  - Move through list using head and current

  ```cpp
  CStringList::~CStringList()
  {
      while(head != 0)
      {
          current = head;
          head = head->next;
          delete current;
      }
  }
  ```

Using a List

- Here's an example of using CStringList

  ```cpp
  int main()
  {
      CStringList lst;
      char buf[80];
      while (cin.getline(buf, 80))
      {
          lst.add(buf);
      }
      for (lst.first(); ! lst.eol(); lst.next())
      {
          cout << lst.c_str() << endl;
      }
  }
  ```

Bi-directional Lists

- A doubly-linked list allows forward/backwards
- Changing CStringList1 to List2
  - Step 1: Add a prev pointer to the structure
  - Step 2: Initialize current->prev to tail
- Traversing backwards
  - Step 1: Set current to tail
  - Step 2: Change loop to use current->prev
A List ADT I

- CStringList is really too specific
  - List class should be able to store any kind of data
  - Solution in book uses "copy, paste, & typedef"
    ```
    typedef struct YourClass Item;
    class List {
        struct node {
            Item data;
            node * next;
        };
    };
    ```

A List ADT II

- Problems w/ cut 'n' paste for code reuse
  - Several copies of same code floating around
  - Hard to maintain, bugs don't get fixed in all versions
- C++ has two better solutions to hide contents
  - Opaque types
    - Container holds void pointers, not objects
  - Class templates
    - Compiler writes "cut 'n' paste" code from blueprint

A List ADT III

- A generic list holding void pointers [List02.cpp]
  - Data portion is a void pointer
  - A void pointer can point to any kind of object
  - A void pointer cannot be dereferenced or new'd
    - User must cast to a specific type before dereferencing
- Generic [opaque] list stores pointers, not objects
  - User must create pointers to store in list
  - User must free objects when removed
  - A very flexible, but not a safe or simple design
A List ADT IV

- You'll normally create typesafe wrappers
  - A class that uses a generic class as a data member
    - Example: CStringList03.h, CStringList03.cpp
  - Generic class is private so client can't bypass
  - Wrapper class provides
    - Meaningful constructors and destructors
    - Meaningful way to add and delete objects
    - Delegates majority of work to generic class

The Queue ADT

- A queue is a linear data structure
  - Similar to a stack, except FIFO, not LIFO
  - Used when input and output are not synchronized
  - Operations on a queue
    - enqueue: add an item to the end of the queue
    - dequeue: remove the item in front of the queue
    - isEmpty() or isFull()
- Several different implementations: List, array

Homework

- Due Sunday, March 14, [Grade early Mon. AM]
  - Homework 9: The BigUInt class
    - An "unlimited-size" unsigned integer as ASCII digits
    - Use (unchanged) hmwk09.cpp, BigUInt01.h
    - Add copy constructor, assignment operator, and relational operator >= to BigUInt01.cpp (submit)
  - Homework 10: Infix to Postfix Notation
    - Implement a Stack as a linked list
    - Use hmwk10.cpp, make changes where specified
    - Note the difference in the Stack member functions