Data Structures

Today's Agenda

- **Data Abstraction**
  - Given what we talked about last time, we need to step through an example of an abstract data type using
    - classes,
    - constructors,
    - member functions,
    - data hiding,
    - distinguishing the difference between what the client can do and what the ADT can do

List Abstraction

- **Let's begin by building a "list" abstraction**
- **Remember, the data structure used should be able to "plug and play" (i.e., be replaced without affecting the client program**
- **Operations will incloude to:**
  - insert, remove, retrieve, and display
  - create, destroy

List Abstraction

- **Once the operations are understood, we can begin to examine the data**
- **Let's build a list to be used for a student roster**
  - so, the data will include the student name, current grade % in the class, and psu id#
  - we can use a struct or a class to represent the underlying data
  - we will examine both of these approaches today

List Abstraction

- The list of students can be implemented using a variety of data structures
- Our choices are:
  - array (statically allocated),
  - array (dynamically allocated),
  - linear linked list
  - circular linked list
  - doubly linked list

List Abstraction

```
struct data {
        char * name;
        char * psu_id;
        float grade;
};
```

- Options for placement:
  - before the class in the header file
  - in the implementation file
  - nested within the class

Placed in the .h file

```
struct data {
        char * name;
        char * psu_id;
        float grade;
};
class list {
  public:
   ...
```

Placed in the .cpp file

```
//.h file
struct data;
class list {
  public:
   ...
//.cpp file
struct data {
          char * name;
          char * psu_id;
          float grade;
};
```

Nested...

```
class list {
  public:
  ...
  private:
     struct data {
          char * name;
          char * psu_id;
          float grade;
     };
};
```

Structs vs. Classes

```
class data {
        char * name;
        char * psu_id;
        float grade;
        friend class list;
};
```

- If a class had been used, the members would have been private by default requiring the list class to be declared as a friend...

Structs vs. Classes

- My personal preference is to use structures for the underlying data
  - this has the benefit of grouping various data types together
  - the data is accessible directly by the class that uses the underlying data
  - and there is little overhead to access the data members (of the name, psu id#, and grade)

Structs vs. Classes

- But, doesn't this violate data hiding? no!
  - think about what code can access this data? You would have to have the 'array' of students, or the 'head' pointer to the first node containing a student to actually access the data
  - certainly, the client can "see" the structure, but they can't access the data if properly hidden within the class!

Structs vs. Classes

- But, couldn't I use a class instead and specify the members as public to avoid using friends? yes…but…
  - then what is the difference between a class and a struct?
  - personally, I consider that structs should be used when you simply want to group different data items and that classes be used when you want to create new data types, new abstractions, or are doing OOP

Defining the List Class

```
class list {
  public:
     list();
     ~list();
     int insert(const data &);
     int retrieve(char *, data &);
     int display();
     int remove (char *);
```

List class...continued...

```
  private:
     data s_array[SIZE];
     int number_of_students;
};
```

- This provides for a statically allocated array of students
- Notice, none of the member functions uses an index or passes the array
- Why do we need the 2nd data member?

Or...List class...continued...

```
  private:
     data * d_array;
     int number_of_students;
     int size_of_array;
};
```

- This provides for a dynamically allocated array of students
- Notice, none of the member functions changes in their client interface
- Add: list (int size);

Or...List class...continued...

- By replacing the constructor with:

  list (int size=SIZE); //in prototype only

- We can have only one constructor act as either the default constructor (with the same size of an array as the previous example), or with a client-specified size
- The constructor would need to:

  d_array = new data [size];

  size_of_array = size;

Or...List class...continued...

```
  private:
     node * head;
};
```

- This provides for a linked list (linear, circular, or doubly linked)
- Notice, none of the member functions changes in their client interface
- That's right. Insert and retrieve have the same arguments as before!!!!

Defining the node structure

- But, what does the node look like?

```
struct node {    //linear/circular
  data student;
  node * next;
};
```

- Add, node * previous to the structure for a doubly linked list.
- Where do we place this?
- Should the “student” member be a pointer to a data instead?

Constructors

- The purpose of the constructor is to initialize the data members
- So, for the dynamic array it simply allocated the array's memory and initialized the size. What else?
- Yes, it also needs to set the number of items currently stored in the list to zero.
- Why doesn't it need to initialize each element of the array?

Constructors

- For the linked list implementation, the constructor would simply set the head pointer to null
- In some situations, you will also want a second list data member, called tail, to keep track of the end of the list
- This is important when you want to frequently add to the end of the list
- Why doesn't a tail pointer help when removing the last item?

Destructors

- The purpose of a destructor is to deallocate all dynamic memory and close down any resources being used
- For the statically allocated array, the destructor had no purpose
- For the dynamically allocated array, it would be: delete [] d_array;
- Question: Do we need to set the other data members to zero? And, d_array to zero?

Destructors

- For a linked list, the purpose of the destructor is to deallocate all memory
- This requires traversing through the linked data structure, deallocating all items
- Will this do it?

```
delete head;
```

- Of course not. This will deallocate one node unless the “node” has a destructor of its own…

Destructors

- OK, so what is wrong with this:

```
while (head) {
        delete head;
        head = head->next;
}
delete head;
```

- Two things. We are accessing memory within the loop that has already been deallocated
- Second, we have an extra “delete”

Destructors

- OK, is this correct?

```
while (head) {
    node * temp = head;
    delete head;
    head = temp;    }
```

- Close…but why reallocate the memory for temp each time through the loop
- Think about the inefficiency of this!!!
- Therefore, remove the “underlined” portion

Other Member Functions

- For insert, we would need to define what it is that insert actually does
- Does it insert at the beginning, at the end, in sorted order?
- The approach should be well documented in the header file for any client program using the software
- Remember, insert should not prompt or read from the user but rather get its data through arguments from the client

Other Member Functions

- Let's examine some various prototype statements for insert and discuss the pros/cons of each:

```
int insert(const data &);
bool insert(const data &);
void insert(const data &);
void insert(data &, bool);
int insert(const node &);
```

Other Member Functions

- What about retrieve?
- Should it display the matching item or "return" it to the client program?
- Which prototype is best?

```
int retrieve(char *, data &);
data retrieve(char *, bool);
data & retrieve (char *, bool);
```

Efficiency...

- Now that we have seen a partial example of a list abstraction, we should examine the efficiency of using an array versus a linked list for the data structure
- An array allows for direct access, which is only useful if you know the position of where the data is located
- If the array is sorted, then a binary search can be used to get to the correct position

Efficiency...

- If a linked list is used, we must traverse to the correct spot to insert into sorted order
- If the array is not sorted, and we are inserting at the beginning or the end, an array is extremely efficient...direct access for both arrays and linked lists why?
- Would you need a tail pointer now????

Efficiency...

- But, what about retrieval?
- An array, if sorted, will allow us to use the binary search
- A linked list - regardless of if it is sorted or not - will require traversal starting at the head
- Think of the overhead if you have a list of 10,000 items? How many comparisons would we have in the worst case?

Efficiency...

- With an array that is not sorted, the same is true. An array provides no additional advantage in this case!
- Ok, so what about memory efficiency?
- An array's size must be determined prior to actually needing to use the memory
  - A statically allocated array must be decided upon at compile time
  - A dynamically allocated array's size can be determined later, at run time

Next Time...

- Now that we have seen a simple list example, and started to examine how we can use different data structures to solve the same problem
- We will move on to examine linked list, circular linked list, linked list of linked lists, doubly linked lists, etc. beginning next time!
- Our next ADT will be an “ordered list” adt!

Data Structures