**CS-255 Exam 2 Study Guide**

Dynamic Memory Allocation

* This includes creating memory manually using pointers
* This includes pointers:
* **SEE ALL POINTER EXAMPLES**
* Variable whose content is a memory address
* We have used this with reference parameters and whenever we used arrays for parameters
* Use \* to declare a pointer:
  + Datatype \*identifier;
* Ex:

Int \*ptr; //ptr is a pointer to an int

Double \*d; //d is a pointer to a double

Double \*e,f; //e is a pointer an f is a double

* Use & sign to return the address of an operand
* Ex:

Int data;

Int \*ptr;

Data = 47;

Ptr = &data;

* Use the \* as a dereferencing operator
* Refers to the object to which the operand points

Cout << data << endl;

Cout << pointer << endl;

Cout << \*ptr << endl; //example of dereferencing

\*ptr = 86; //the 47 will change to 86

Cout << data << endl;

Operator Overloading

* We can overload specific operators so that the operators can work effectively in a specific application.
* You cannot create new operators
* In order to overload an operator, you must write functions (in the header and the body) to define what operation the overload operator indicates should be performed.
* **See Assignment 6 Project**
* Advantages:
  + Enhances the flexibility of the class
  + It is more preferred to use operators instead of functions
* Friend Function:
  + Nonmember function of the class but has access to all of the members (public or non-public)
  + In order to make a function a friend of the class, the reserve word “friend” appears only in the function prototype in the class definition.
  + EX:

**Class classIllusFriend**

**{**

**Friend void two(/\*parameters\*/);**

**};**

* Two is a nonmember function of classIllusFriend
* Any object of classIllusFriend can access its private members within the definition of the function “two”
* Ex: Use in front of >> and << overloading

Recursion

* Used what a problem can be broken down into successive problems that are identical to the overall problem
* See Lecture 2-22 CodeBlocks
* Two cases
  + Base Case: The size of the list is 1 the only element in list is the largest element
    - 0! = 1
  + General Case: the size of the list is greater than 1 to find the largest element in the list
    - n! = n x (n-1) if n>0
* General Formulas
* Recursion Tree
  + N->n/2->n/4 and so on

Complexity

* This regards understanding which algorithm is faster
* Big O Notation: Describes the worst-case scenario
  + You must look at the time it takes for an algorithm when the problem size gets larger (N)
  + Examples:
    - Search: Un ordered array with a key and we are looking f+or it within the array
    - Use linear search
    - There are N items in our array
    - Unordered: O(N)
    - Ordered: Use Binary Search O(LogN)
    - Insert (Allow duplicates)
    - Unordered: Add to the end of the array O(1)
    - Ordered: Search first O(LogN)
    - Add O(N) Because you might have to shift the whole list so O(LogN) +O(N)
    - Insert (No duplicates)
    - Unordered: O(N)+O(1) = O(N)
    - Ordered: O(LogN) + O(N) + O(N)
    - Delete
    - Unordered: Remove item, take bottom and override deleted value and subtract 1 from array size
    - O(N) + O(1) = O(N)
    - Ordered: O(LogN) + O(N) = O(N)
    - See Chapter 16

Linked Lists

* Dynamic structure that consists of a collection of nodes
* These nodes consist of 2 things:
  + Data
  + A pointer to another node on the list
* When initializing a linked list, a pointer is created called the “Head” of the list
* The null pointer indicates the end of a linked list
* Common operations of a linked list:
  + addHead

//Function

void addHead(ClassPtr &head, int theData)

{

Head = new Class(theData, head);

}

//Needed constructor

Class::Class(int theData, ClassPtr theLink)

{

data = theData;

link = theLink;

}

* addTail

addTail(ClassPtr &head, int theData)

{

if(head == nullptr)

{

head = new Class(theData, head);

}

Else

{

ClassPtr temp = nullptr, next = nullptr;

Temp = head;

Next = head->getLink();

While(next != nullptr)

{

Temp = next;

Next = next -> getLink();

}

Insert(temp, theData);

}

}

//Needed insert function

Void insert(ClassPtr afterMe, int theData)

{

ClassPtr temp = nullptr;

Temp = new Class(theData, afterMe ->getLink());

afterMe -> setLink(temp);

}

* displayList

void displayList(ClassPtr head)

{  
 ClassPtr temp = head;

Cout << “Head: “;

While(temp != nullptr)

{

Cout << “ -> “ << temp -> getData();

Temp = temp -> getLink();

}

Cout << endl;

* search

search(ClassPtr head, int key)

{

ClassPtr temp = head;

While(temp != nullptr)

{

If(temp -> getData() == key)

{

Cout << “Found Item!” << endl;

}

Temp = temp -> getLink();

}

}

deleteNode(ClassPtr &head, int target)

{

If(head == nullptr)  
 {

Cout << “Cannot delete from an empty list.” << endl;

}

Else

{

ClassPtr pred = head, temp = head-> getLink();

If(pred-> getData() == target)

{

Cout < “Deleted item successfully.” << endl;

Head = temp;

Delete pred;

}

Else

{

While((temp -> getData() != target) && (temp -> getLink() != nullptr)

{

Pred = temp;

Temp = temp -> getLink();

}

If(temp -> getData() == target)

{

Pred ->setLink(temp ->getLink());

Delete temp;

Cout << “deleted item successfully.”;

}

Else

{

Cout << “Item was not found in the list.” << endl;

}

}

}

}

Binary Files

* Difference between binary files and text files
  + Text/ASCII file- plain characters – 1 byte per character
  + Binary File- encoded in a specific format
    - Ex: Word, Excel mp3, jpg, png, etc.
  + Advantages of using binary over text in C++ programming:
    - Less amount of code to read/write with binary files
    - Faster to execute when reading/writing with binary files