Chapter #: Topic

Tuesday, February 23, 2021 2:38 PM

#.1 Header Here

1. Text here

Chapter 1: Data Abstraction

Wednesday, January 27, 2021 9:01 AM

Object-oriented analysis (OOA) is the process of understanding what the problem is and what the requirements of a solution are

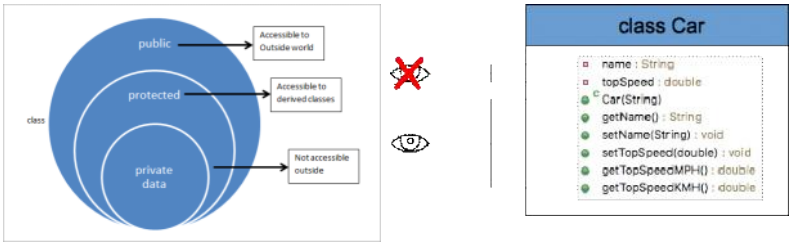
In OOA, you express the problem and the requirements of a solution in terms of relevant objects - describe these objects and their interactions among one another.

During Object-Oriented Design (OOD), you describe a solution to the problem, fulfilling the requirements you discovered during analysis.

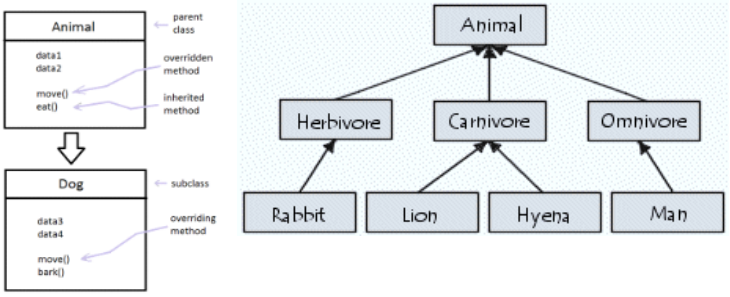
- Algorithms: step-by-step recipes for performing a task within a finite period of time
- Attribute: characteristics of objects of a single type
- Behaviors: the object's operations
- Data members: the individual data items specified in a class

Three Principles of OOP

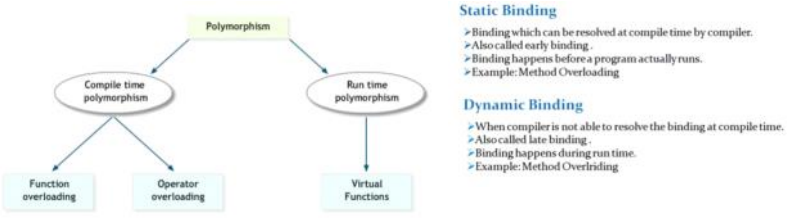
- Encapsulation: a technique that hides inner details



- Inheritance: allows you to reuse classes for you defined earlier for a related purpose by extending that implementation or making slight modifications; may make it impossible for the compiler to determine which operation you require in a particular situation - but polymorphism fixes that

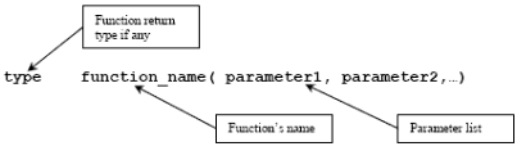


- Polymorphism: enables determination of which operation is required in a particular situation; decides the operation to be executed depending on the object that performs the operation; allows the compiler to simply note that the meaning of an operation is unknown until execution time

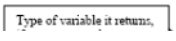


Specifications of a Module

- Operation contract: documents how a method can be used and what limitations it has; helps programmers understand the responsibilities the module will have to other modules in the function
- Unusual conditions: how the module should react to invalid circumstances; in C++, the desired reaction is to throw an exception.
- Abstraction: separates the purpose of a module from its implementation; specifies each model clearly BEFORE you implement it in a programming language
- Information hiding: tells you not only to hide such details within the module, but also ensures that no other modules can tamper with the information
- Minimal and complete interfaces: called the signature; a function's prototype
  - Complete interface: one that will allow the programmer to accomplish any reasonable task given the responsibilities of the given class
  - Minimal interface: one that contains a method if and only if the method is essential to the class responsibilities



General form of function header



- Complete interface: one that will allow the programmer to accomplish any reasonable task given the responsibilities of the given class
- Minimal interface: one that contains a method if and only if the method is essential to the class responsibilities

Abstract data type (ADT):

a collection of data and a set of operations on the data. Ultimately, someone will implement the ADT by using a data structure, which is a construct that you can define within a programming language to store a collection of data. For example, you can store the data in a C++ array of strings or an array of objects.

Additional information:
- TutorialsPoint: Abstract Data Type in Data Structures
- GeeksForGeeks: Abstract Data Types

COHESION

- Each module should perform one well-defined task; that is, it should be highly-cohesive.
- Benefits of a highly-cohesive module:
  1. Well-named, self-documenting, easy-to-understand code
  2. Easy to reuse in other software projects
  3. Much easier to maintain
  4. Is more robust - less likely to be affected by change

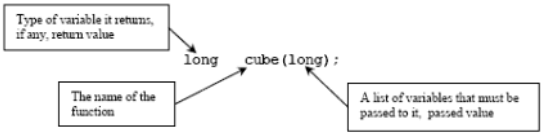
COUPLING

- Coupling is a measure of the dependence among modules. This dependence could involve sharing data structures or calling each other's methods.
- Ideally, modules in a design should be independent of each other.
- Benefits of loosely coupled modules
  1. Tends to create a system that is more adaptable to change
  2. Creates a system that is easier to understand
  3. Increases the reusability of the module
  4. Has increased cohesion
- Tightly coupled modules is where modules have a high degree of dependence on each other

ADTs can implement other ADTs.

When defining methods in an ADT, you should not print anything or create any output - that's the client's business regarding what they want to output. Only return Boolean values to indicate whether a method succeeded or not, and gather all ADT bag components into a singular vector if the client wants to see all of them - but DO NOT output the values.

Create an abstract base class as an interface template for the ADT. An abstract base class has at least one method that is virtual and has no implementation - also, cannot be instantiated. The subclass must implement all methods that were specified but not defined in the abstract base class.




```
// function prototype code snippet example
// this example cannot be compiled or run
double  squared (double);      // function prototype
void  print_report (int);      // function prototype

double  squared(double number)   // function header
{                                  // opening bracket
     return (number * number);     // function body
}                                   // closing bracket

void   print_report(int report_number)
{
     if(report_number  == 1)
          printf("Printing Report 1");
     else
          printf("Not printing Report 1");
}
```

https://www.tenouk.com/Module4.html

http://www-h.eng.cam.ac.uk/help/languages/C++/c++_tutorial/functions.html

SUMMARY

1. Object-oriented analysis (OOA) is used during the initial stage in problem solving to understand what the problem is and what the requirements of a solution are.
2. During object-oriented design (OOD), you describe a solution to a problem whose requirements are discovered during OOA. You express the solution in terms of software objects.
3. Encapsulation is a principle of object-oriented programming whereby you hide the inner details of functions and objects. Functions encapsulate behavior, and objects—that is, instances of a class—encapsulate data as well as behavior.
4. Inheritance is another concept related to object-oriented programming, allowing you to reuse already defined classes by extending their definitions or making slight modifications.
5. Polymorphism is the third principle of object-oriented programming, whereby objects determine appropriate operations at execution time.
6. Each module should be highly cohesive; that is, it should perform one well-defined task.
7. Coupling is a measure of the dependence among modules. Module should be loosely coupled. A function or method should be as independent as possible and perform one well-defined task.
8. UML is a modeling language used to express object-oriented designs. It provides a notation to specify the data and operations and uses diagrams to show relationships among classes.
9. An operation contract documents how a module can be used and what limitations it has.
10. A function or method should always include an initial comment that states its purpose, its precondition—that is, the conditions that must exist at the beginning of a module—and its postcondition—the conditions at the end of a module's execution.
11. For problems that primarily involve data management, encapsulate data with operations on that data by designing classes. Practice abstraction—that is, focus on what a module does instead of how it does it.
12. Data abstraction is a technique for controlling the interaction between a program and its data structures. It builds walls around a program's data structures, just as other aspects of modularity build walls around a program's algorithms. Such walls make programs easier to design, implement, read, and modify.
13. The specification of a set of data-management operations, together with the data values on which they operate, defines an abstract data type (ADT).
14. Only after you have fully defined an ADT should you think about how to implement it. The proper choice of a data structure to implement an ADT depends both on the details of the ADT operations and on the context in which you will use the operations.

C++ Interlude 1: C++ Classes

Friday, January 29, 2021 6:06 PM

Class templates: allows us to specify the data type of the items contained in a data structure in a very generic way

Header/specification files: provide a mechanism to partially separate the design of a class from the implementation in the source , or implementation (.cpp), file .

Can use a typedef statement to rename a datatype to make it more readable to the programmer. For example:

We can define `ItemType` as the type of item stored in the box by using a `typedef` statement. For example, to have the box hold a `double`, we can write

```
typedef double ItemType;
```

More information on typedef statements: https://www.lix.polytechnique.fr/~liberti/public/computing/prog/c/C/SYNTAX/typedef.html

C1.1 A Problem to Solve

1. **PRIVATE DATA FIELDS**

- Every data field in every class should be declared as *private*
- Why? Because clients and derived classes should not have direct access to the data fields of a class - instead, they use set and get methods.

2. **CONSTRUCTORS AND DESTRUCTORS**

- Constructor: allocates memory for new instances of a class and can initialize the object's data to specified values.
- Destructor: destroys an instance of a class when the object's lifetime ends
- A typical class has many constructors and only one destructor
- Default constructor: has no parameters, just initializes data fields
- Parameterized constructor: initializes data fields to values chosen by client

Note: *If you do not define any constructors for a class, the compiler creates a default constructor—one without parameters. Once you define a constructor, the compiler does not create any of its own. Therefore, if you define a parameterized constructor but not a default constructor, your class will not have a default constructor.*

3. **METHODS**

- An accessor method in a class accesses the value of a data field.
- A mutator method in a class changes the value of a data field.
- Passing parameters by constant reference. The method *setItem* and the parameterized constructor both have a parameter *theItem* that is passed by constant reference:

```
void setItem(const ItemType& theItem);

PlainBox(const ItemType& theItem);
```

- Passing a parameter by constant reference provides benefits to both the client and programmer.
- To keep the efficiency of passing by reference and still protect the data of our

client, we use the keyword *const* before the declaration of that parameter. Then, the method treats the parameter as a constant that cannot be modified. Our method can still access and use an object passed by constant reference, but the compiler flags any modifications to the object as errors.

Programming Tip: Methods declared `const` can access and use data fields, so labeling all accessor methods with the `const` declaration is appropriate and a good idea.

Note: Guidelines for safe and secure programming

- Declare all data fields in the private section of the class declaration.
- Declare as **`const`** any method that does not change the object (accessor methods).
- Declare any parameter passed by reference as **`const`**, unless you are certain it must be modified by the method, in which case the method should be either protected or private.

1. **PREVENTING COMPILER ERRORS**

- You can declare C++ variables only once within each programming block - a second declaration (redefinition) of a variable results in a compiler error.
- We need a way to prevent the compiler from reading the class definition a second time. So, we use the preprocessor directives *#ifndef*, *#define*, and *#endif*.

LISTING C1-1 The header file for the class `PlainBox`

```
/** @file PlainBox.h */

#ifndef _PLAIN_BOX
#define _PLAIN_BOX

// Set the type of data stored in the box
typedef double ItemType;
```

  - *#ifndef* means that "If *_PLAIN_BOX* is not defined then…" If the compiler had not defined the name *_PLAIN_BOX*, it would process the code that follows until it reached the *#endif* directive at the end of the file.
  - *#define* defines the name *_PLAIN_BOX*. If another file includes the class definition of PlainBox, the name *_PLAIN_BOX* would have already been defined, and the *#ifndef* directive will cause the preprocessor to skip any of the code that follows - ensuring that the class is not defined more than once.

C1.2 Implementing a Solution

- To tell the compiler that the methods defined in the header file are a part of the implementation file, you must precede the contsructor and method names with a namespace indicator *PlainBox :: methodName()*.

```
ItemType PlainBox::getItem() const
{
   return item;
}  // end getItem
```

- A namespace is a syntax structure, such as a class, that allows you to group together declarations of data and methods under a common name, such as *PlainBox*.
- Therefore, a namespace indicator is a prefix that indicates to the compiler that the method is a part of the *PlainBox* namespace.

C1.3 Templates

- If we want to store different object types, then instead of changing the *typedef* of

each class and then renaming the constructors and namespace indicators to reflect the new class names, we can make a template header file.

LISTING C1-3 Template header file for the PlainBox class

```
/** @file PlainBox.h */

#ifndef _PLAIN_BOX
#define _PLAIN_BOX

template<class ItemType>; // Indicates this is a template definition

// Declaration for the class PlainBox
class PlainBox
{
private:
   // Data field
   ItemType item;
```

- Templates enable the programmer to separate the functionality of an implementation from the type of data used in the class.
- After making the header file a template, in the single implementation class, you must write the same template statement prior to each method's definition as such:

LISTING C1-4 Implementation file for the PlainBox template class

```
/** @file PlainBox.cpp */

template<class ItemType>;
PlainBox<ItemType>::PlainBox()
{
}  // end default constructor

template<class ItemType>;
PlainBox<ItemType>::PlainBox(const ItemType& theItem)
{
   item = theItem;
}  // end constructor
```

- To instantiate an instance of *PlainBox*, you write the data type of the item to be placed in a box surrounded by angle brackets:

```
PlainBox<double> numberBox;    // A box to hold a double
PlainBox<string> nameBox;      // A box to hold a string object
PlainBox<MagicWand> wandBox;   // A box to hold a MagicWand object
```

Note: *By using templates, you can define a class that involves data of any type, even data types that are created after you designed and implemented your class.*

C1.4 Inheritance

- *Most of this you already know…*
- Inheritance does not imply access
- A method in a derived class overrides, or redefines, a method in the base class if the two methods have the same name and parameter declarations (signatures)

```
template<class ItemType>
void MagicBox<ItemType>::setItem(const ItemType& theItem)
{
   if (!firstItemStored)
   {
      PlainBox<ItemType>::setItem(theItem);
      firstItemStored = true; // Box now has magic
   }  // end if
} // end setItem
```

- Because *MagicBox* has two *setItem* methods (its own and another one from inherited base-class *PlainBox*).

- We would precede a method name with the base-class name-space indicator to tell the compiler to use the base-class version of the method.

C1.5 Virtual Methods and Abstract Classes

- Using the keyword *virtual* in front of the header of the method tells the C++ compiler that the code this method executes is determined at runtime, NOT when the program is compiled.
- We do this because we want the version of the method implementation executed to depend on the derived object class calling it - not just the base-class version.

If we declare PlainBox's method setItem to be virtual in the PlainBox header file by writing

```
virtual ItemType setItem() const;
```

- Declaring our ADT methods as virtual allows an application using our class to take advantage of polymorphism when the ADT's methods are invoked.
- For these kinds of ADTs, we do NOT want to provide an implementation of the ADT methods - instead, we want to force the classes derived from the ADT to provide the implementations.
- To avoid the compiler error due to not implementing the method, we write our methods as pure virtual methods, which is a method that has no implementation.
- An abstract class is one that has at least one pure virtual method.

```
virtual void setItem(const ItemType& theItem) = 0;
```

- To indicate that our class is derived from an interface, use this header format:

```
class PlainBox : public BoxInterface<ItemType>
```

Chapter 2: Recursion

Friday, January 29, 2021 6:02 PM

Recursion is an extremely powerful problem solving technique. It is an alternative to iteration, which involves loops.

Complex problems can have simple recursive solutions.

For example, suppose that you could solve problem P1 if you had the solution to problem P2, which is a smaller instance of P1. Suppose further that you could solve problem P2 if you had the solution to problem P3, which is a smaller instance of P2. If you knew the solution to P3 because it was small enough to be trivial, you would be able to solve P2. You could then use the solution to P2 to solve the original problem P1.

A base case is a special case whose solution you know. Recursion uses a *divide-and-conquer strategy*, where you continue to divide the problem into smaller and smaller parts until you reach the base case. All recursive algorithms MUST have a base case - otherwise it will generate an infinite sequence of calls.

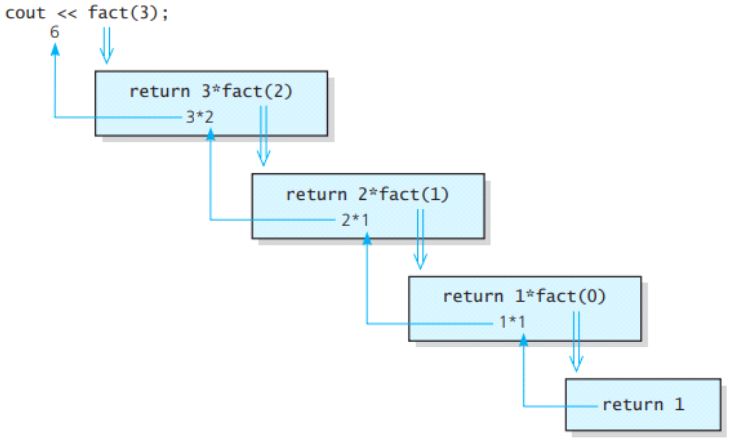
1. A recursive function calls itself
2. Each recursive call solves an identical but smaller, problem
3. A test for the vas case enables the recursive calls to stop
4. Eventually, one of the smaller problems must be the base case

Recursive Valued Function vs. Recursive Void Function

Recursive Valued Function:

Finding the factorial of *n* is an example of a problem that can be solved using a recursive valued function.

```
/** Computes the factorial of the nonnegative integer n.
 @pre  n must be greater than or equal to 0.
 @post  None.
 @return  The factorial of n; n is unchanged. */
int fact(int n)
{
   if (n == 0)
      return 1;
   else // n > 0, so n-1 >= 0. Thus, fact(n-1) returns (n-1)!
      return n * fact(n - 1); // n * (n-1)! is n!
}  //  end fact
```



We can use a box trace to help us understand recursion and to debug recursive functions. The box

trace illustrates how compilers frequently implement recursion. Each box roughly corresponds to an activation record , which a compiler typically uses in its implementation of a function call.

Recursive Void Function:

Recursive functions need not return a value; they can be void functions.

For example, we can write a string backwards using a recursive void functions.

Base case: string length of 1

Goal: write a string of length *n* backwards

Notice that the recursive calls to the function use successively shorter versions of the string s, ensuring that the base case will be reached

Because the function does nothing when it reaches the base case, it does not deal with the base case explicitly - the base case is implicit.

There are two different ways to approach this problem - compare the code and output streams of the two different methods:

writeBackward():

```
writeBackward(s: string)

   cout << "Enter writeBackward with string: " << s << endl;
   if (the string is empty)
      Do nothing—this is the base case
   else
   {
      cout << "About to write last character of string: "
           << s << endl;
      Write the last character of s
      writeBackward(s minus its last character) // Point A
   }
   cout << "Leave writeBackward with string: " << s << endl;
```

Output stream:

```
Enter writeBackward with string: cat
About to write last character of string: cat
t
Enter writeBackward with string: ca
About to write last character of string: ca
a
Enter writeBackward with string: c
About to write last character of string: c
c
Enter writeBackward with string:
Leave writeBackward with string:
Leave writeBackward with string: c
Leave writeBackward with string: ca
Leave writeBackward with string: cat
```

writeBackward2():

```
writeBackward2(s: string)
   cout << "Enter writeBackward2 with string: "
        << s << endl;
   if (the string is empty)
      Do nothing—this is the base case
   else
   {
     writeBackward2(s minus its first character) // Point A
     cout << "About to write first character of string: "
          << s << endl;

     Write the first character of s
   }
   cout << "Leave writeBackward2 with string: " << s << endl;
```

Output stream:

```
Enter writeBackward2 with string: cat
Enter writeBackward2 with string: at
Enter writeBackward2 with string: t
Enter writeBackward2 with string:
Leave writeBackward2 with string:
About to write first character of string: t
t
Leave writeBackward2 with string: t
About to write first character of string: at
a
Leave writeBackward2 with string: at
About to write first character of string: cat
c
Leave writeBackward2 with string: cat
```

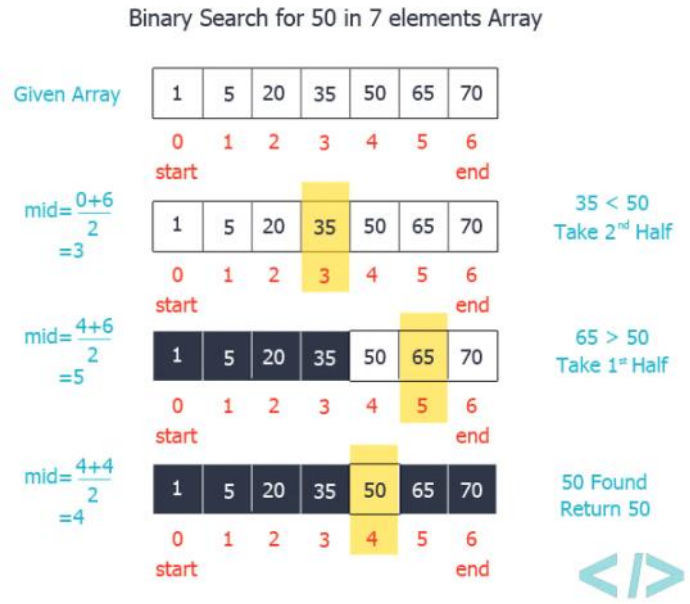
Recursion With Arrays

- **REVERSING AN ARRAY**
  - Goal: How will we pass anArray minus its last character to writeArrayBackward()?
  - Process:
    - Pass the number of characters left in the array
    - At each recursive call, we would decrease this number by one

```
/** Writes the characters in an array backward.
  @pre  The array anArray contains size characters, where size >= 0.
  @post  None.
  @param anArray  The array to write backward.
  @param first  The index of the first character in the array.
  @param last  The index of the last character in the array. */
void writeArrayBackward(const char anArray[], int first, int last)
{
   if (first <= last)
   {
      // Write the last character
      cout << anArray[last];

      // Write the rest of the array backward
      writeArrayBackward(anArray, first, last - 1);
   }  // end if

   // first > last is the base case - do nothing
}  // end writeArrayBackward
```

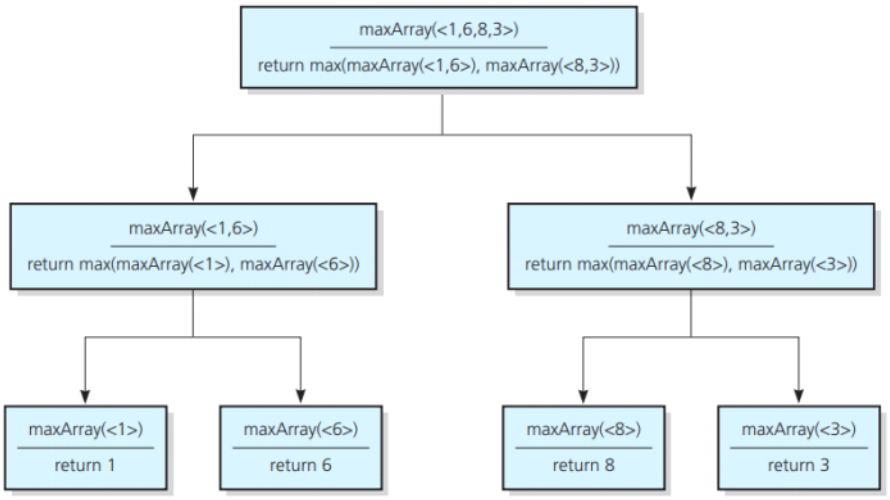
- **THE BINARY SEARCH**
  - A binary search conquers one of its subproblems at each step. It is a recursive algorithm to search for a value within in array.

```
/** Searches the array anArray[first] through anArray[last]
    for a given value by using a binary search.
 @pre  0 <= first, last <= SIZE - 1, where SIZE is the
       maximum size of the array, and anArray[first] <=
       anArray[first + 1] <= ... <= anArray[last].
 @post  anArray is unchanged and either anArray[index] contains
        the given value or index == -1.
 @param anArray  The array to search.
 @param first  The low index to start searching from.
 @param last  The high index to stop searching at.
 @param target  The search key.
 @return  Either index, such that anArray[index] == target, or -1.
*/
int binarySearch(const int anArray[], int first, int last, int target)
{
   int index;
   if (first > last)
      index = -1;   // target not in original array
   else
   {
      // If target is in anArray,
      // anArray[first] <= target <= anArray[last]
      int mid = first + (last - first) / 2;
      if (target == anArray[mid])
         index = mid; // target found at anArray[mid]
      else if (target < anArray[mid])
         // Point X
         index = binarySearch(anArray, first, mid - 1, target);
      else
         // Point Y
         index = binarySearch(anArray, mid + 1, last, target);
   }  // end if

   return index;
}  // end binarySearch
```



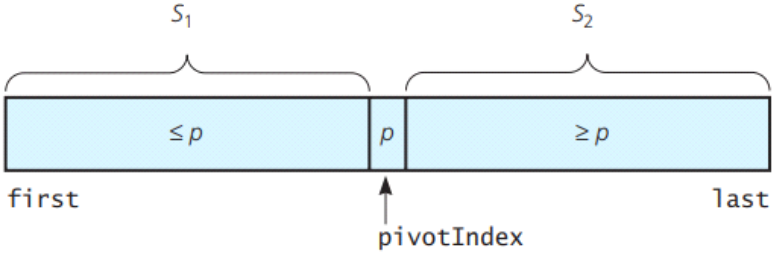
- **FINDING THE LARGEST VALUE IN AN ARRAY**
  - This strategy fits the divide-and-conquer model similar to the previous binary search algorithm used.
  - However, although the binary search algorithm conquers only one of its subproblems at each step, maxArray conquers *both*. This is called multipath recursion. It must then reconcile (find the maximum of) the two solutions.

```
if (anArray has only one entry)
   maxArray(anArray) is the entry in anArray
else if (anArray has more than one entry)
   maxArray(anArray) is the maximum of
      maxArray(left half of anArray) and maxArray(right half of anArray)
```



How to write a recursive solution:

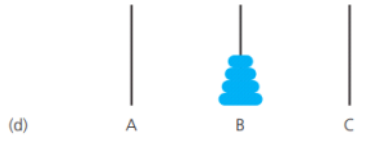
1. Selecting a pivot value in the array
2. Cleverly arranging or partitioning the values in the array about this pivot value
3. Recursively applying the strategy to one of the partitions



Organizing Data

- **THE TOWERS OF HANOI**
  - Conditions:
    - There are *n* disks and three poles (A: source, B: destination, C: spare)
    - Disks can be placed only on top of disks larger than themselves
    - Goal: Efficiently transfer all disks from pole A to pole B
  - This problem can be solved by essentially by recursively solving multiple smaller Towers problems

  -



```
void solveTowers(int count, char source, char destination, char spare)
{
   if (count == 1)
   {
      cout << "Move top disk from pole " << source
           << " to pole " << destination << endl;
   }
   else
   {
      solveTowers(count - 1, source, spare, destination);  // X
      solveTowers(1, source, destination, spare);          // Y
      solveTowers(count - 1, spare, destination, source);  // Z
   }  // end if
}  // end solveTowers
```

```
Move top disk from pole A to pole B
Move top disk from pole A to pole C
Move top disk from pole B to pole C
Move top disk from pole A to pole B
Move top disk from pole C to pole A
Move top disk from pole C to pole B
Move top disk from pole A to pole B
```

Recursion and Efficiency

- Recursion is a powerful problem-solving technique that often produces very clean solutions to even the most complex problems.
- Two factors contribute to the inefficiency of some recursive solutions:
  - The overhead associated with function calls
  - The inherent inefficiency of some recursive algorithms
- *Recursion is only valuable when a problem has no simple iterative solution*

SUMMARY

1. Recursion is a technique that solves a problem by solving a smaller problem of the same type.
2. When constructing a recursive solution, keep the following four questions in mind:
   a. How can you define the problem in terms of a smaller problem of the same type?
   b. How does each recursive call diminish the size of the problem?
   c. What instance of the problem can serve as the base case?
   d. As the problem size diminishes, will you reach this base case?
3. When constructing a recursive solution, you should assume that a recursive call's result is correct if its precondition has been met.
4. You can use the box trace to trace the actions of a recursive function. These boxes resemble activation records, which many compilers use to implement recursion. Although the box trace is useful, it cannot replace an intuitive understanding of recursion.
5. Recursion allows you to solve problems—such as the Towers of Hanoi—whose iterative solutions are difficult to conceptualize. Even the most complex problems often have straightforward recursive solutions. Such solutions can be easier to understand, describe, and implement than iterative solutions.
6. Some recursive solutions are much less efficient than a corresponding iterative solution due to their inherently inefficient algorithms and the overhead of function calls. In such cases, the iterative solution can be preferable. You can use the recursive solution, however, to derive the iterative solution.
7. If you can easily, clearly, and efficiently solve a problem by using iteration, you should do so.

Chapter 3: Array-Based Implementations

Saturday, February 6, 2021 6:49 PM

Implementing an ADT as a C++ class provides a way for you to enforce the wall of an ADT, thereby preventing access of the data structure in any way other than by using the ADT's operations. A client then cannot damage the ADT's data.

3.1. Approach

1. **CORE METHODS**
   a. Defining a class that implements an ADT --> Identify a group of core methods to both implement and test before continuing with the rest of the class definition. A group of core methods is called a core group
   b. Ex: In the ADT bag, the method toVector() allows us to see the container's data, so it is a core method.
2. **USING FIXED-SIZE ARRAYS**
   a. When implementing an ADT that represents a data collection, you need to store the data items and track their number
   b. Array-based implementation - you store the items in an array
      i. Objects in the bag can begin at index 0 of the array
   c. We need to establish assertions - certain truths about our planned implementation so that the action of each method is not detrimental to other methods
      i. Ex: method toVector() must know where add has placed the entries

3.2 An Array-Based Implementation of The ADT Bag

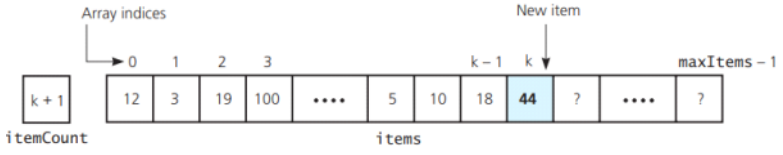
1. **THE HEADER FILE**
   a. DEFAULT_CAPACITY assigns the default capacity of the bag
   b. Add another constructor that enables the client to set the capacity of the bag during program execution (maxItems)

```
class ArrayBag : public BagInterface<ItemType>
{
private:
   static const int DEFAULT_CAPACITY = 6;// Small size to test for a full bag
   ItemType items[DEFAULT_CAPACITY];     // Array of bag items
   int itemCount;                        // Current count of bag items
   int maxItems;                         // Max capacity of the bag

   // Returns either the index of the element in the array items that
   // contains the given target or -1, if the array does not contain
   // the target.
   int getIndexOf(const ItemType& target) const;

public:
   ArrayBag();
   int getCurrentSize() const;
   bool isEmpty() const;
   bool add(const ItemType& newEntry);
   bool remove(const ItemType& anEntry);
   void clear();
   bool contains(const ItemType& anEntry) const;
   int getFrequencyOf(const ItemType& anEntry) const;
   vector<ItemType> toVector() const;
}; // end ArrayBag
```

1. **DEFINING THE CORE METHODS**
   a. The core methods include
      i. The default constructor (use initializer notation)

```
ArrayBag<ItemType>::ArrayBag(): itemCount(0), maxItems(DEFAULT_CAPACITY)
{
} // end default constructor
```

      i. The add method



      i. The toVector method
      ii. The getCurrentSize method
      iii. The isEmpty method

2. **TESTING THE CORE METHODS**
   a. An incomplete definition of a method is called a stub, and it only needs to pass the syntax checker using dummy return values.

More info on the const keyword at the end of a function declaration https://stackoverflow.com/questions/751681/meaning-of-const-last-in-a-function-declaration-of-a-class

When you add the const keyword to a method the this pointer will essentially become a pointer to const object, and you cannot therefore change any member data.

The const keyword is part of the functions signature which means that you can implement two similar methods, one which is called when the object is const, and one that isn't.

The const member functions are the functions which are declared as constant in the program. The object called by these functions cannot be modified. It is recommended to use const keyword so that accidental changes to object are avoided.

A const member function can be called by any type of object. Non-const functions can be called by non-const objects only.

b. Test program output:

Output

```
Testing the Array-Based Bag:
The initial bag is empty.
isEmpty: returns 1; should be 1 (true)
The bag contains 0 items:


Add 6 items to the bag:
The bag contains 6 items:
one two three four five one

isEmpty: returns 0; should be 0 (false)
getCurrentSize: returns 6; should be 6
Try to add another entry: add("extra") returns 0
All done!
```

3. **IMPLEMENTING MORE METHODS**
   a. The getFrequencyOf method
   b. The contains method

You should form a test program incrementally, so you test all the methods you have defined so far

4. **METHODS THAT REMOVE ENTRIES**
   a. The remove method
      i. Simply remove the first instance of anEntry that we encounter while searching for it - return either true or false to indicate whether the removal was successful
   b. The clear method
      i. Makes a bag appear empty simply by setting itemCount to zero (all other methods depend on itemCount, so setting it to zero would virtually erase all the elements)
      ii. For example, the toVector method depends on itemCount (to avoid confusion, keep in mind that this class is NOT an array, it is only structured to RESEMBLE an array):

```
vector<ItemType> ArrayBag<ItemType>:: toVector() const
{
   vector<ItemType> bagContents;
   for (int i = 0; i < itemCount; i++)
      bagContents.push_back(items[i]);
   return bagContents;
}  // end toVector
```

5. **TESTING**
   a. Just add more cout << statements to show the functionality of the new methods

3.3 Using Recursion In The Implementation

1. **THE METHOD getIndexOf**
   a. We can use recursion to search the rest of the array beginning with items[searchIndex+1]
   b. Base cases:
      i. If items[searchIndex] is the entry we seek, we are done.
      ii. If searchIndex equals itemCount , we are done, because the entry is not in the array

```
int ArrayBag<ItemType>::getIndexOf(const ItemType& target, int searchIndex) cons
{
   int result = -1;
   if (searchIndex < itemCount)
   {
      if (items[searchIndex] == target)
      {
        result = searchIndex;
      }
      else
      {
        result = getIndexOf(target, searchIndex + 1);
      } // end if
   } // end if

   return result;
}  // end getIndexOf
```

2. **THE METHOD getFrequencyOf**
   a. Recursive steps
      i. If items[searchIndex] is the entry we seek, the frequency of occurrence of this entry is one more than its frequency of occurrence in the rest of the array.
      ii. If items[searchIndex] is not the entry we seek, the frequency of occurrence of the entry is the same as its frequency of occurrence in the rest of the array.

SUMMARY

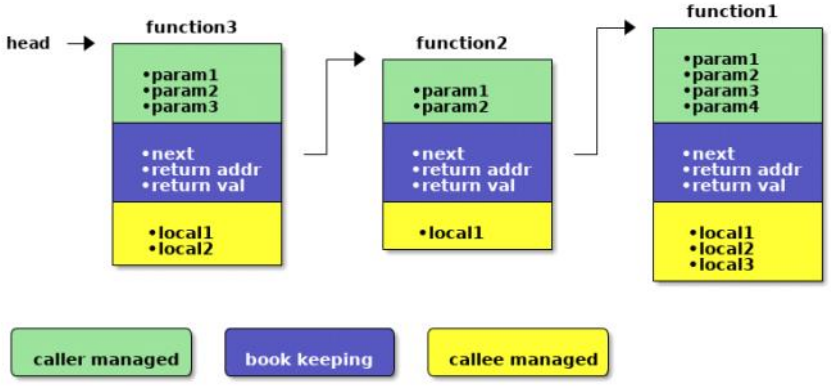
1. By using a class to implement an ADT, you encapsulate the ADT's data and operations. In this way, you can hide implementation details from the program that uses the ADT. In particular, by making the class's data members private, you can change the class's implementation without affecting the client.
2. Given an interface that specifies an ADT in an implementation-independent way, derive a class from the interface and declare the class within a header file. Choose a data structure to contain the ADT's data. Then implement the class's methods within an implementation file.
3. You should make a class's data members private so that you can control how a client can access or change the data.
4. An array-based implementation of an ADT stores the ADT's data in an array.
5. Generally, you should not define an entire class and then attempt to test it. Instead, you should identify a group of core methods to both implement and test before continuing with the rest of the class definition.
6. Stubs are incomplete definitions of a class's methods. By using stubs for some methods, you can begin testing before the class is completely defined.
7. A client must use the operations of an ADT to manipulate the ADT's data.

C++ Interlude 2: Pointers, Polymorphism, & Memory Allocation

Thursday, February 18, 2021 7:24 PM

2.1 Memory Allocation for Variables and Early Binding of Methods

- When you declare an ordinary variable with a particular data type, the C++ compiler allocates a memory cell that can hold that data type.
- Ex: `int x = 5;`
- A function's locally declared variables are placed in activation records that are store in an area of the application's memory called the run-time stack. Each time a function is called, an activation record is automatically created on the run-time stack.



The Run-time stack

During execution of a program, the run-time stack stores essential information for each of the currently active function calls.

Whenever a function is invoked during execution of a program, storage for:
- the **return value** (*if any*)
- the function **parameter(s)** (*if any*)
- the **return address**, and
- the function's **local variable(s)** (*if any*)

is allocated on the run-time stack.

- When the function ends, the activation record is destroyed and the memory is freed up, meaning the function's local variables are no longer accessible.
- Two different ways to invoke default constructors:

```
PlainBox<string> myPlainBox;
MagicBox<string> myMagicBox = MagicBox<string>();
```

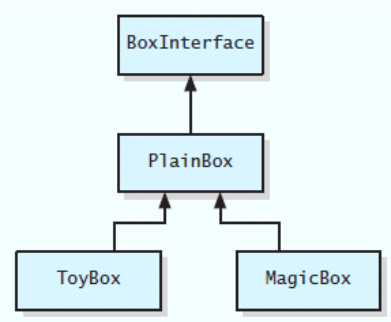
^ when these objects are instantiated, their data fields are placed on the runtime stack.
- If you invoke an inherited method on two different objects, the compiler knows which version of the method to call for each object. This is called early binding and is made during compilation and cannot be altered during execution.

```
myPlainBox.setItem("Fun Item");
myMagicBox.setItem("Secret Item");
```

- Two situations in which automatic memory management and early binding are insufficient:
  - You want to take advantage of polymorphism.
  - You must access an object outside of the function or method that creates it.

2.2 A Problem to Solve

Write a function that takes two arguments: an object of any of the three types of boxes and an item of type string. The function should place the item in the box by invoking the box's setItem method.

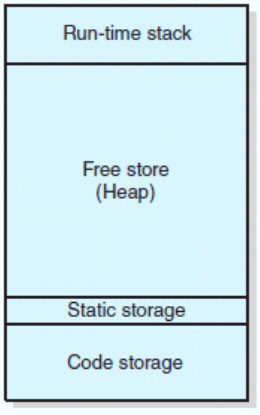



```
void placeInBox(PlainBox<string>& theBox, string theItem)
{
   theBox.setItem(theItem);
}  // end placeInBox
```

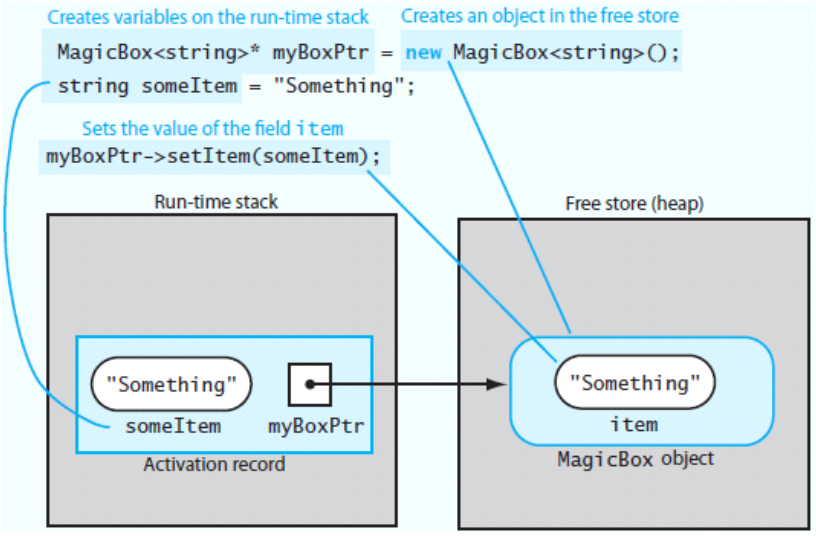
- The parameter *theBox* can accept an argument of any *PlainBox* or *PlainBox*-derived object.
- The statement *theBox.setItem(theItem);* invokes the *PlainBox* version of the *setItem* method instead of the *MagicBox* version. This is because the compiler determined the version of the method to invoke from the type of the parameter *theBox* instead of from the type of the actual *argument*.
- We need a way to communicate to the compiler that the code to execute should not be determined until the program is running. This is called late binding, and to do so, we need pointer variables and virtual methods.

2.3 Pointers and the Program's Free Store

- To take advantage of late binding, we do NOT want our objects to be in an activation record on the runtime stack.
- In addition to runtime stack, the operating system sets aside memory for the code (called code storage or text storage) and for global and static variables (called static storage). Your program is also given extra memory called the free store or application heap.



- Allocate memory for a variable on the free store using the operator *new*. The memory address returned by the *new* operator after allocating memory for the variable is placed in a pointer.



- Unlike the runtime stack, variable placed in free store persist in memory even when the function or method that created them ends. Therefore, they must be manually deallocated when no longer needed. Otherwise, a memory leak may occur, which is memory that has been allocated for use but is no longer needed and cannot be accessed or deallocated.
- Indicate a pointer type variable by writing an asterisk after the data type when making the declaration.

```
MagicBox<string>* myBoxPtr = new MagicBox<string>();
```

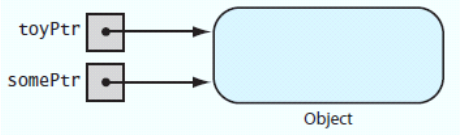
- To call a method of an object that is in the free store, use the notation ->.

```
string someItem = "Something Free";
myBoxPtr->setItem(someItem);
```

- Pointer variables can only point to objects of the same type.

```
ToyBox<string>* toyPtr = new ToyBox <string>();   // OK
ToyBox<string>* boxPtr = new MagicBox<string>();  // Error!
ToyBox<string>* somePtr = new ToyBox <double>();  // Error!
```

- You can also have multiple pointer variables that point to the same object using the following syntax: *somePtr = toyPtr*.



DEALLOCATING MEMORY

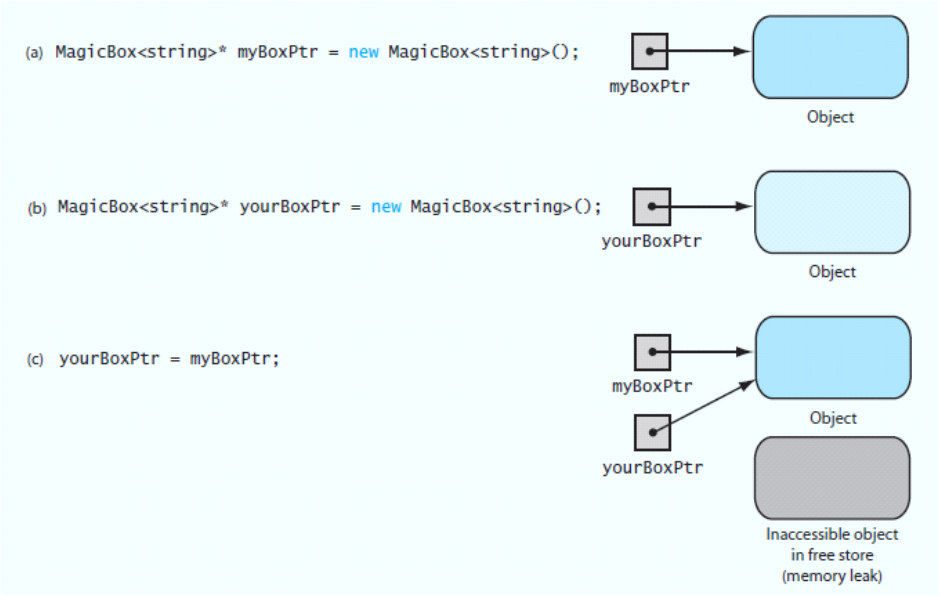
- Use the following syntax to properly deallocate a pointer variable:

```
delete somePtr;
somePtr = nullptr;
```

- If we did not set *somePtr* to *nullptr* (to indicate it no longer references or points to an object) then it would be considered a dangling pointer - meaning that it still contains the address of an object that was deallocated.

AVOIDING MEMORY LEAKS

- Example of how poorly written code can cause a memory leak:



- Create an object in the free store and return a pointer to it. The caller must delete the object when it is no longer needed. -> This does not prevent a memory leak, but it warns against it.

```
ToyBox<double>* pluggedLeakyFunction(const double& someItem)
{
   ToyBox<double>* someBoxPtr = new ToyBox<double>();
   someBoxPtr->setItem(someItem);
   return someBoxPtr;
} // end pluggedLeakyFunction
```

- Example of how you'd call the function above:

```
double boxValue = 4.321;
ToyBox<double>* toyPtr = pluggedLeakyFunction(boxValue);
```

- To prevent a memory leak, do NOT use a function to return a pointer to a newly created object.

AVOID DANGLING POINTERS

- Set pointer variables to nullptr either initially or when you no longer need them. If a class has a pointer variable as a data field, the constructor should always initialize that data fi eld, to point either to an object or to nullptr .
- Test whether a pointer variable contains nullptr before using it to call a method.
- Try to reduce the use of aliases in your program. As you will see, that is not always possible or desirable in certain situations.
- Do not delete an object in the free store until you are certain that no other alias needs to use it.
- Set all aliases that reference a deleted object to nullptr when the object is deleted.

2.4 Virtual Methods and Polymorphism

- To allow the compiler to perform the late binding necessary for polymorphism, you must declare the methods in the base class as virtual.
- Virtual methods are methods that use the keyword *virtual* to indicate that they can be overridden in a derived class.

```
/** @file PlainBox.h */
#ifndef _PLAIN_BOX
#define _PLAIN_BOX

template<class ItemType>; // Indicates this is a template

// Declaration for the class PlainBox
class PlainBox
{
private:
   // Data field
   ItemType item;

public:
   // Default constructor
   PlainBox();

   // Parameterized constructor
   PlainBox(const ItemType& theItem);

   // Mutator method that can change the value of the data field
   virtual void setItem(const ItemType& theItem);

   // Accessor method to get the value of the data field
   virtual ItemType getItem() const;
}; // end PlainBox

#include "PlainBox.cpp" // Include the implementation file
#endif
```

- You MUST implement a class's virtual methods (pure virtual methods are exempt)
- A derived class does not need to override an existing implementation of an inherited virtual method
- If you do not want a derived class to override a particular method, the method should NOT be virtual.
- Destructors SHOULD be virtual to ensure children of the object can deallocate themselves correctly.
- A virtual method's return type CANNOT be overridden.

2.5 Dynamic Allocation of Arrays

- An ordinary C++ array is statically located
- Use the *new* operator to allocate an array dynamically:

```
int arraySize = 50;
double* anArray = new double[arraySize];
```

- When you allocate an array dynamically, you need to return its memory cells to the system when you no longer need them:

```
delete [ ] anArray;
```

- The advantage of allocating an array dynamically is that it is resizable. For example, if you want to double the size of the array to hold more items than originally intended, you can allocate a new and larger array, copy the old array into the new array, and finally deallocate the old array:

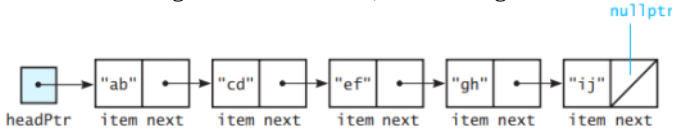
```
double* oldArray = anArray;                    // Copy pointer to array
anArray = new double[2 * arraySize];           // Double array size

for (int index = 0; index < arraySize; index++)   // Copy old array
   anArray[index] = oldArray[index];

delete [ ] oldArray;                           // Deallocate old array
```

Chapter 4: Link-Based Implementations

Saturday, February 6, 2021 7:45 PM

4.1 Preliminaries

1. **THE CLASS NODE**
   a. Nodes: can be linked together; each one contains both a data item and a "pointer" to the next item.
   b. The head pointer points to the first node; because nothing so far points to the first node, and if we cannot get to the first node, we cannot get to the second node, and so on.



   c. Header file for template class Node:

```
/** @file Node.h */

#ifndef _NODE
#define _NODE

template<class ItemType>
class Node
{
private:
   ItemType        item; // A data item
   Node<ItemType>* next; // Pointer to next node
public:
   Node();
   Node(const ItemType& anItem);
   Node(const ItemType& anItem, Node<ItemType>* nextNodePtr);
   void setItem(const ItemType& anItem);
   void setNext(Node<ItemType>* nextNodePtr);
   ItemType getItem() const ;
   Node<ItemType>* getNext() const ;
}; // end Node
#include "Node.cpp"
#endif
```

? More info on template<class ItemType>: https://www.cplusplus.com/doc/oldtutorial/templates/

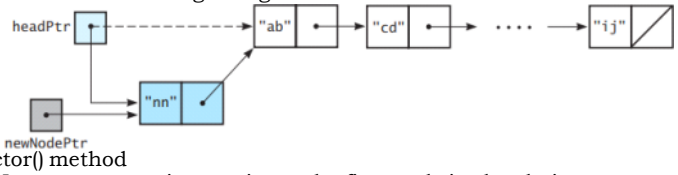
? More info on virtual vs pure virtual functions: https://stackoverflow.com/questions/2652198/difference-between-a-virtual-function-and-a-pure-virtual-function

4.2 A Link-Based Implementation of the ADT Bag

1. **THE HEADER FILE**
   a. Class LinkedBag
   b. Each element will be stored as a Node<ItemType>
   c. Must make destructors virtual

```
private:
   Node<ItemType>* headPtr; // Pointer to first node
   int itemCount;           // Current count of bag items

   // Returns either a pointer to the node containing a given entry
   // or the null pointer if the entry is not in the bag.
   Node<ItemType>* getPointerTo(const ItemType& target) const;

public:
   LinkedBag();
   LinkedBag(const LinkedBag<ItemType>& aBag); // Copy constructor
   virtual ~LinkedBag();                       // Destructor should be virtual
   int getCurrentSize() const;
   bool isEmpty() const;
   bool add(const ItemType& newEntry);
   bool remove(const ItemType& anEntry);
   void clear();
   bool contains(const ItemType& anEntry) const;
   int getFrequencyOf(const ItemType& anEntry) const;
   vector<ItemType> toVector() const;
}; // end LinkedBag
```

1. **DEFINING THE CORE METHODS**
   a. Constructor
      i. Initializes head pointer and the current number of items in the bag
   b. add() method
      i. Make headptr point to the new node, and the new node must point to the node that had been at the beginning of the chain



   c. toVector() method

i. Let a current pointer point to the first node in the chain
ii. While the current pointer is not the null pointer, assign the data portion of the current node to the next element in a vector and then set the current pointer to the next pointer of the current node

2. IMPLEMENTING MORE METHODS

a. getFrequencyOf() method
b. contains() method
c. remove() method
d. clear() method: cannot simply set *itemCount* to zero because the nodes in the chain were allocated dynamically - we must go through and deallocate them.

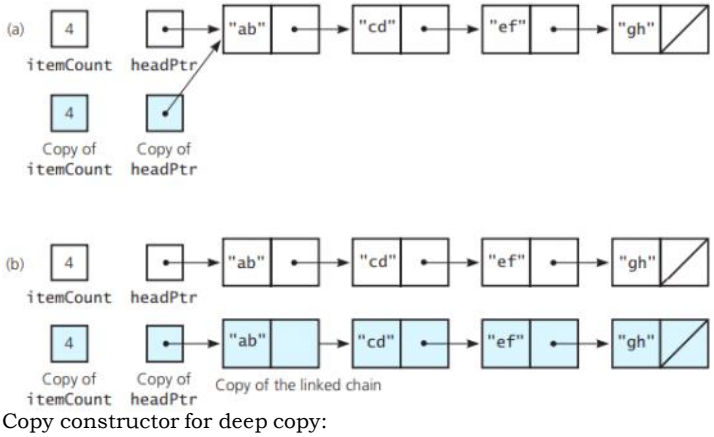
```
while (headPtr != nullptr)
{
   Node<ItemType>* nodeToDeletePtr = headPtr;
   headPtr = headPtr->getNext();

   // Return node to the system
   nodeToDeletePtr->setNext(nullptr);
   delete nodeToDeletePtr;
}  // end while
// headPtr is nullptr

nodeToDeletePtr = nullptr;
itemCount = 0;
```

e. Destructor method (invoked implicitly at the end of the block in which the object was created)

```
template<class ItemType>
LinkedBag<ItemType>::~LinkedBag()
{
   clear();
}  // end destructor
```

A destructor's name is a tilde (~) followed by the class name. A destructor cannot have arguments, has no return type—not even `void`—and cannot use `return` to return a value.

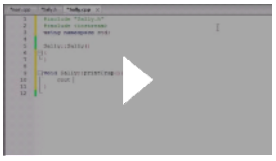
f. Copy constructor
   i. (invoked implicitly when you either pass an object to a function by value, return an object from a valued function, or define and initialize an object) --> LinkedBag bag2(bag1);
   ii. When copying an object involves only copying the values of its data members, the copy is called a shallow copy
   iii. If a shallow copy is sufficient, you can *omit the copy constructor*, and the compiler generates a copy constructor that performs a shallow copy
   iv. If you need to create a deep copy of the linked chain, you must write your own copy constructor
   v. Linked chain shallow copy (a) vs. linked chain deep copy (b):



   vi. Copy constructor for deep copy:

More info on -> operator: https://www.tutorialspoint.com/What-is-arrow-operator-in-Cplusplus

Buckys C++ Programming Tutorials - 42 - Arrow Member Selection Operator



*a →b is essentially a shorthand notation for (*a).b, ie, if a is a pointer to an object, then a→b is accessing the property b of the object that points to*

More info on .pushback: https://www.cplusplus.com/reference/vector/vector/push_back/

Adds a new element at the end of the vector, after its current last element.

```
template<class ItemType>
LinkedBag<ItemType>::LinkedBag(const LinkedBag<ItemType>& aBag)
{
   itemCount = aBag->itemCount;
   Node<ItemType>* origChainPtr = aBag->headPtr

   if (origChainPtr == nullptr)
      headPtr = nullptr; // Original bag is empty; so is copy
   else
   {
      // Copy first node
      headPtr = new Node<ItemType>();
      headPtr->setItem(origChainPtr ->getItem());

      // Copy remaining nodes
      Node<ItemType>* newChainPtr = headPtr;      // Last-node pointer
      while (origPtr != nullptr)
      {
         origChainPtr = origChainPtr ->getNext(); // Advance pointer

         // Get next item from original chain
         ItemType nextItem = origChainPtr->getItem();

         // Create a new node containing the next item
         Node<ItemType>* newNodePtr = new Node<ItemType>(nextItem);


          // Link new node to end of new chain
          newChainPtr->setNext(newNodePtr);

          // Advance pointer to new last node
          newChainPtr = newChainPtr->getNext();
       }  // end while

       newChainPtr->setNext(nullptr); // Flag end of new chain
    }  // end if
}  // end copy constructor
```

Errata sheet corrections:

- In the 4th line, replace -> with a period:

```
itemCount = aBag.itemCount;
```

- In the 5th line, replace -> with a period, and add a semicolon at the end:

```
Node<ItemType>* origChainPtr = aBag.headPtr;
```

- In the `while` statement, replace `origPtr` with `origChainPtr`:

```
while (origChainPtr != nullptr)
```

- The first statement in the body of the `while` loop,

```
origChainPtr = origChainPtr->getNext(); // Advance pointer
```

is misplaced. It should appear just before the `while` statement and again on page 148 as the last statement in the loop's body.

4.3 Using Recursion In Link-based Implementations

1. **RECURSIVE DEFINITIONS OF METHODS IN LinkedBag**
   a. Method: toVector()
      i. Goal: create a vector with the data items of all nodes in the LinkedChain
      ii. Process:
         1) After creating a vector, toVector() fills it with the data in the chain of linked nodes whose head pointer is headPtr by calling fillVector()
         2) Define recursive method fillVector():
            a) Base case: curPtr != nullptr
            b) First add the data curPtr->getItem() to the vector and recursively fill the vector with the chain that begins at curPtr->getNext()

```
template<class ItemType>
vector<ItemType> LinkedBag<ItemType>::toVector() const
{
   vector<ItemType> bagContents;
   fillVector(bagContents, headPtr);
   return bagContents;
}  // end toVector
```

         3) fillVector():

```
if (curPtr != nullptr)
{
   bagContents.push_back(curPtr->getItem());
   fillVector(bagContents, curPtr->getNext());
}  // end if
```

   b. Private method: getPointerTo()
      i. Goal: locates a given entry within the linked chain.
      ii. Process:
         1) Revise the declaration of getPointerTo in the header file for the class LinkedBag

to include a second parameter curPtr
2) Base cases:
   a) When chain is empty, causing method to return nullPtr
   b) When we locate the desired entry at curPtr->getItem()

Errata sheet corrections:

In the definition of the method `getPointerTo` at the bottom of the page, `LinkedBag<ItemType>::` is missing before the method's name:

```
Node<ItemType>* LinkedBag<ItemType>::getPointerTo(const ItemType& target,
                                                  Node<ItemType>* curPtr) const
```

4.4 Testing Multiple ADT Implementations

Just…test it like the array-based implementation one…and add some fancy polymorphism stuff, idk just check the textbook *dies*

Sample Output 2

```
Enter 'A' to test the array-based implementation
or 'L' to test the link-based implementation: L
Testing the Link-Based Bag:
The initial bag is empty.
isEmpty: returns 1; should be 1 (true)
Add 6 items to the bag:
The bag contains 6 items:
one five four three two one

isEmpty: returns 0; should be 0 (false)
getCurrentSize returns : 6; should be 6
Try to add another entry: add("extra") returns 1
All done!
```

4.5 Comparing Array-Based and Linked-Based Implementations

Weigh advantages and disadvantages before deciding on the type of implementation.

ARRAYS

- Arrays are easy to use, but they have a fixed size
- To decide, ask the question of whether the fixed-size restriction of an array-based implementation presents a problem in the context of a particular application:
  - One factor: for a given application, can you predict in advance the maximum number of items in the ADT at any one time? --> if you can't, it's possible that the program will fail because the ADT in the context of a particular application requires more storage than the array can provide
  - Another factor: would you waste storage by declaring an array to be large enough to accommodate this maximum number of items?
  - In both cases, the array-based implementation is not desirable
- Increasing the size of a dynamically allocated array can waste storage and time
- An array-based implementation is a good choice for a small bag

LINKED CHAIN

- A link-based implementation can solve any difficulties related to the fixed size of the array-based implementation
- Because you will be allocating storage dynamically, you won't waste storage.
- The item after an array item is implied in a chain of linked nodes, an item points explicitly to the next item
- An array-based implementation require less memory than a link-based implementation because it does not have to store explicit information about where to find the next data item
- You can access array items directly with equal access time
- You must traverse a linked chain to access its ith node - time to access depends on i

SUMMARY

1. You can link objects—called nodes—to one another to form a chain of linked data. Each node contains a data item and a pointer to the next node in the chain. An external pointer variable—called the head pointer—points to the first node. The last node in the chain has `nullptr` in its pointer portion, so it points to no other node.

2. You use the `new` operator to dynamically allocate a new node, whereas you use the `delete` operator to deallocate a node.

3. Inserting a new node at the beginning of a linked chain or deleting the first node of a linked chain are easier to perform than insertions and deletions anywhere else in the chain. The insertion requires a change to two pointers: the pointer within the new node and the head pointer. The deletion requires a change to the head pointer and an application of the `delete` operator to the removed node.

4. Unlike an array, which enables you direct access to any of its elements, a linked chain requires a traversal to access a particular node. Therefore, the access time for an array is constant, whereas the access time for a linked chain depends on the location of the node within the chain.

5. When traversing a linked chain by using the pointer variable `curPtr`, you must be careful not to reference `curPtr` after it has "passed" the last node in the chain, because it will have the value `nullptr` at that point. For example, the loop

   ```
   while (value > curPtr->getItem())
      curPtr = curPtr->getNext();
   ```

   is incorrect if `value` is greater than all the data values in the linked chain, because `curPtr` becomes `nullptr`. Instead you should write

   ```
   while ((curPtr != nullptr) && (value > curPtr->getItem()))
      curPtr = curPtr->getNext();
   ```

   Because C++ uses short-circuit evaluation (see Appendix A) of logical expressions, if `curPtr` becomes `nullptr`, the expression `curPtr->getItem()` will not be evaluated.

6. A class that allocates memory dynamically needs an explicit copy constructor that copies an instance of the class. The copy constructor is invoked implicitly when you pass an object to a function by value, return an object from a valued function, or define and initialize an object. If you do not define a copy constructor, the compiler will generate one for you. A compiler-generated copy constructor is sufficient only for classes that use statically allocated memory.

7. A class that allocates memory dynamically needs an explicit destructor. The destructor should use `delete` to deallocate the memory associated with the object. If you do not define a destructor, the compiler will generate one for you. A compiler-generated destructor is sufficient only for classes that use statically allocated memory.

8. Although you can use the `new` operator to allocate memory dynamically for either an array or a linked chain, you can increase the size of a linked chain one node at a time more efficiently than you can increase the size of an array. When you increase the size of a dynamically allocated array, you must copy the original array entries into the new array and then deallocate the original array.

Chapter 5: Recursion as a Problem-Solving Technique

Saturday, February 13, 2021 10:50 AM

(5.1-5.3.1 ONLY)

- Formal grammar: enable you to define, for example, syntactically correct algebraic expressions, which we explore in some detail.
- Backtracking: a problem-solving technique that involves guesses at a solution.
- In this chapter, you will learn how to us mathematical induction to study properties of algorithms.

5.1 Defining Languages

- Language: nothing more than a set of strings of symbols from a finite alphabet
- Whereas all programs are strings, not all strings are programs
- A C++ compiler is a program that sees whether a given string is a member of the language *C++Programs*
- Grammar: states the rules of a language
  - The grammars in this chapter are recursive in nature
- Recognition algorithm: a straightforward recursive algorithm, based on the grammar, that determines whether a given string is in the language

1. THE BASICS OF GRAMMARS

A grammar uses several special symbols:

- $x \mid y$ means x or y.
- $x\,y$ (and sometimes $x \cdot y$) means x followed by y.
- $<word>$ means any instance of *word*, where *word* is a symbol that must be defined elsewhere in the grammar.

A grammar for the language

$$C++Identifiers = \{\text{string } s : s \text{ is a legal C++ identifier}\}$$

a. The grammar for the language *C++Identifiers:*

$<identifier> = <letter> \mid <identifier><letter> \mid <identifier><digit>$

$<letter> = a \mid b \mid \ldots \mid z \mid A \mid B \mid \ldots \mid Z \mid \_$

$<digit> = 0 \mid 1 \mid \ldots \mid 9$

b. The above definition reads as follows: *An identifier is a letter, or an identifier followed by a letter, or an identifier followed by a digit.*

The **identifier appears in its own definition**: this grammar is recursive, as are many grammars.

Pseudocode for C++ identifiers recognition algorithm:

```
// Returns true if s is a legal C++ identifier;
// otherwise returns false.
isId(s: string): boolean

   if (s is of length 1)                    // Base case
      if (s is a letter)
         return true
      else
         return false
   else if (the last character of s is a letter or a digit)
      return isId(s minus its last character)  // Point X
   else
      return false
```

2. TWO SIMPLE LANGUAGES

a. Palindrome: a string that reads the same from left to right as it does from right to left

b. Define the language of palindromes as:
Palindromes = {string *s*: *s* reads the same left to right as right to left}

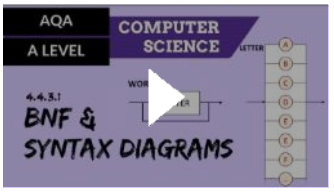
Process:

1. Need to devise a rule that allows you to determine whether a given string *s* is a palindrome - you should state this rule in terms of determining whether a smaller string is a palindrome:
   i. The first and last characters of *s* are the same
   ii. *s* minus its first and last characters is a palindrome
   iii. Two base cases: empty string or string of length 1 are palindromes
2. Use these rules to write the following grammar for the language of palindromes:

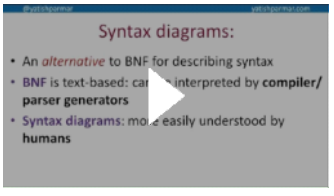
The C++ identifier is a name used to identify a variable, function, class, module, or any other user-defined item.

https://www.tutorialspoint.com/Identifiers-in-Cplusplus#:~:text=The%20C%2B%2B%20identifier%20is%20a,digits%20(0%20to%209).

Syntax diagrams explained: AQA A'Level BNF and syntax diagrams



Syntax diagrams - A Level Computer Science



<pal> = empty string | <ch> |a <pal> a|b <pal> b|. . .| Z <pal> Z
<ch> = a | b |. . .| z | A | B |. . .| Z

3. Based on this grammar, write the pseudocode for a recognition algorithm for palindromes:

```
// Returns true if the string s of letters is a palindrome; otherwise returns false.
isPalindrome(s: string): boolean

   if (s is the empty string or s is of length 1)
      return true
   else if (s's first and last characters are the same letter)
      return isPalindrome(s minus its first and last characters)
   else
      return false
```

5.2 ALGEBRAIC EXPRESSIONS

- Three different languages for algebraic expressions; the expressions in these languages are easy to recognize and evaluate but are generally inconvenient to use.
- The stricter the definition for an algebraic expression, the easier it is to recognize a syntactically legal expression. However, overly-strict definitions can be inconvenient for programmers.

1. KINDS OF ALGEBRAIC EXPRESSIONS

a. Infix expressions: the algebraic expressions you learned about in school. "Infix" indicates that every binary operator appears *between* its operands
b. Ex:
   i. *(a + b) * c*
c. *NOTE:* common practice is to associate from left to right

2. PREFIX EXPRESSIONS

a. Prefix expressions: the operator precedes its operands
b. Ex:
   i. *+ a * b c ---equivalent to the infix expression---> a + (b * c)*
c. Grammar that defines the language of all prefix expressions is

<prefix> = <identifier> | <operator> <prefix> <prefix>
<operator> = + |−|*|/
<identifier> = a | b |. . .| z

d. Algorithm must check that;
   i. The first character of the string is an operator
   ii. The remainder of the string consists of two consecutive prefix expressions
e. Pseudocode for a recognition algorithm for prefix expressions:
   i. (check textbook for more details on the *endPre()* algorithm - I can't do this anymore)

```
// Sees whether an expression is a prefix expression.
// Precondition: strExp contains a string with no blank characters.
// Postcondition: Returns true if the expression is in prefix form; otherwise returns false.

isPrefix(strExp: string): boolean

   lastChar = endPre(strExp, 0)
   return (lastChar >= 0) and (lastChar == strExp.length() - 1)
```

3. POSTFIX EXPRESSIONS

a. Postfix expressions: the operator follows its operands
b. Ex:
   i. *a b c * + ---equivalent to the infix expression---> a + (b * c)*
c. Grammar that defines the language of all prefix expressions is:

<postfix> = <identifier>|<postfix><postfix><operator>
<operator> = + |−|*|/
<identifier> = a | b |. . .| z

a. Pseudocode for an algorithm that converts a prefix expression to postfix form

```
// Converts a prefix expression to postfix form.
// Precondition: The string pre is a valid prefix expression with no blanks.
// Postcondition: Returns the equivalent postfix expression.
convert(preExp: string): string

    preLength = the length of preExp
    ch = first character in preExp
    postExp = an empty string

    if (ch is a lowercase letter)
        // Base case—single identifier
        postExp = postExp • ch              // Append to end of postExp
    else // ch is an operator
    {
        // pre has the form <operator> <prefix1> <prefix2>
        endFirst = endPre(preExp, 1)     // Find the end of prefix1

        // Recursively convert prefix1 into postfix form
        postExp = postExp • convert(preExp[1..endFirst])

        // Recursively convert prefix2 into postfix form
        postExp = postExp • convert(preExp[endFirst + 1..preLength - 1))

        postExp = postExp • ch              // Append the operator to the end of postExp
    }
    return post
```

1. **FULLY PARENTHESIZED EXPRESSIONS**
    a. Grammar for the language of all fully parenthesized infix expressions:

$$\langle infix \rangle = \langle identifier \rangle \mid (\langle infix \rangle \langle operator \rangle \langle infix \rangle)$$
$$\langle operator \rangle = + \mid - \mid * \mid /$$
$$\langle identifier \rangle = a \mid b \mid \ldots \mid z$$

    a. Although the grammar is simple, the language is rather inconvenient for programmers.

5.3 Backtracking

Backtracking is a strategy for guessing at a solution and backing up when an impasse is reached

1. **SEARCHING FOR AN AIRLINE ROUTE**
    a. Goal: must find a path from some point of origin to some destination point
    b. Problem details:

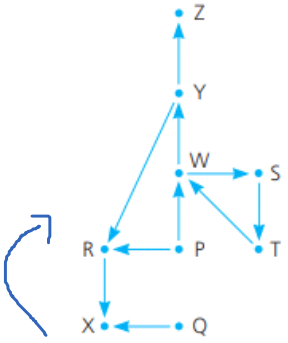
The High Planes Airline Company (HPAir) wants a program to process customer requests to fly from some origin city to some destination city. So that we can focus on recursion, we will simplify the problem: For each customer request, just indicate whether a sequence of HPAir flights from the origin city to the destination city exists.

Imagine three input text files that specify all of the flight information for the airline as follows:

- The names of cities that HPAir serves
- Pairs of city names, each pair representing the origin and destination of one of HPAir's flights
- Pairs of city names, each pair representing a request to fly from some origin to some destination

The program should then produce output such as

```
Request is to fly from Providence to San Francisco.
HPAir flies from Providence to San Francisco.

Request is to fly from Philadelphia to Albuquerque.
Sorry. HPAir does not fly from Philadelphia to Albuquerque.

Request is to fly from Salt Lake City to Paris.
Sorry. HPAir does not serve Paris.
```



    a. called a directed graph. Each arrow is called a directed path.
    b. Must develop an algorithm that searches the flight map for a directed path from the origin city to the destination city. Such a path might involve either a single flight or a sequence of flights. This solution is called an exhaustive search,

meaning that starting from the origin city, the solution will try every possible sequence of flights until either it find a sequence that gets to the destination city or it determines that no such sequence exists.

c. A recursive strategy:

```
To fly from the origin to the destination:
    Select a city C adjacent to the origin
    Fly from the origin to city C
    if (C is the destination city)
        Terminate— the destination is reached
    else
        Fly from city C to the destination
```

a. Drawbacks of this recursive strategy:
   i. You reach a city C from which were are no departing flights
   ii. You go around in circles and the algorithm might not terminate

b. We can use backtracking to recover from a wrong city choice:

"After discovering such a mistake, the algorithm can retrace its steps, or backtrack , to the city C' that was visited just before city C was visited. Once back at city C', the algorithm can select a flight to some city other than C. Notice that it is possible that there are no other flights out of city C'. If this were the case, it would mean that it was a mistake to visit city C', and thus, you would want to backtrack again—this time to the city that was visited just before city C'."

c. Refined version of the recursive search algorithm:

```
// Discovers whether a sequence of flights from originCity to destinationCity exists.
searchR(originCity: City, destinationCity: City): boolean

    Mark originCity as visited
    if (originCity is destinationCity)
        Terminate—the destination is reached
    else
        for (each unvisited city C adjacent to originCity)
            searchR(C, destinationCity)
```

d. ADT implementation of a flight map:

```
// Reads flight information into the flight map.
+readFlightMap(cityFileName: string, flightFileName: string): void

// Displays flight information.
+displayFlightMap(): void

// Displays the names of all cities that HPAir serves.
+displayAllCities(): void

// Displays all cities that are adjacent to a given city.
+displayAdjacentCities(aCity: City): void

// Marks a city as visited.
+markVisited(aCity: City): void

// Clears marks on all cities.
+unvisitAll(): void

// Sees whether a city was visited.
+isVisited(aCity: City): boolean

// Inserts a city adjacent to another city in a flight map.
+insertAdjacent(aCity: City, adjCity: City): void

// Returns the next unvisited city, if any, that is adjacent to a given city.
// Returns a sentinel value if no unvisited adjacent city was found.
+getNextCity(fromCity: City): City

// Tests whether a sequence of flights exists between two cities.
+isPath(originCity: City, destinationCity: City): boolean
```

SUMMARY

1. A grammar is a device for defining a language, which is a set of strings of symbols. By using a grammar to define a language, you often can construct a recognition algorithm that is directly based on the grammar. Grammars are frequently recursive, thus allowing you to describe vast languages concisely.
2. To illustrate the use of grammars, we defined several different languages of algebraic expressions. These languages have their relative advantages and disadvantages. Prefix and postfix expressions, though difficult for people to use, have simple grammars and eliminate ambiguity. On the other hand, infix expressions are easier for people to use but require parentheses, precedence rules, and rules of association to eliminate ambiguity. Therefore, the grammar for infix expressions is more involved.
3. Backtracking is a solution strategy that involves both recursion and a sequence of guesses that ultimately lead to a solution. If a particular guess leads to an impasse, you retrace your steps in reverse order, replace that guess, and try to complete the solution again.
4. A close relationship between mathematical induction and recursion exists. You can use induction to prove properties about a recursive algorithm. For example, you can prove that a recursive algorithm is correct, and you can derive the amount of work it requires.

Exp

Important Tags

Saturday, March 6, 2021 7:10 PM

- #*define* defines the name *_PLAIN_BOX*. If another file includes the class definition of PlainBox, the name *_PLAIN_BOX* would have already been defined, and the #*ifndef* directive will cause the preprocessor to skip any of the code that follows - ensuring that the class is not defined more than once.
- An abstract class is one that has at least one pure virtual method.
- An array-based implementation require less memory than a link-based implementation because it does not have to store explicit information about where to find the next data item
- clear() method: cannot simply set *itemCount* to zero because the nodes in the chain were allocated dynamically - we must go through and deallocate them.
- *Implementing an ADT as a C++ class provides a way for you to enforce the wall of an ADT, thereby preventing access of the data structure in any way other than by using the ADT's operations. A client then cannot damage the ADT's data.*
- Inheritance does not imply access
- Notice that the recursive calls to the function use successively shorter versions of the string s, ensuring that the base case will be reached
- Passing parameters by constant reference. The method *setItem* and the parameterized constructor both have a parameter *theItem* that is passed by constant reference:
- *Recursion is only valuable when a problem has no simple iterative solution*
- The **identifier appears in its own definition**: this grammar is recursive, as are many grammars.
- The stricter the definition for an algebraic expression, the easier it is to recognize a syntactically legal expression. However, overly-strict definitions can be inconvenient for programmers.
- To prevent a memory leak, do NOT use a function to return a pointer to a newly created object.
- Unlike the runtime stack, variable placed in free store persist in memory even when the function or method that created them ends. Therefore, they must be manually deallocated when no longer needed. Otherwise, a memory leak may occur, which is memory that has been allocated for use but is no longer needed and cannot be accessed or deallocated.
- Use the following syntax to properly deallocate a pointer variable:
- We would precede a method name with the base-class name-space indicator to tell the compiler to use the base-class version of the method.
- When copying an object involves only copying the values of its data members, the copy is called a shallow copy
- *You should form a test program incrementally, so you test all the methods you have defined so far*