

### Chapter 3 Array Based Implementation

**Chapter 1 Data Abstraction**

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

**Chapter 2 Recursion**

**Important note:**When you are writing application code to use a class template, you must specify the actual data type that you want to use in the class template. For example:

    Node<double> \*myPtr;

When you are writing code inside a template where ItemType is the name that represents the type of data that is being processed, you need:

    Node<ItemType> \*myPtr;

In question 1 below, you are asked to write application code.

**1. Using the Node class template from Chapter 4, this code will create a linked list with 3 nodes:**

**// Create an empty list with pointers to first and last node**  
**Node <int> \*first = nullptr;    // create head pointer for list**  
**Node <int> \*last = nullptr;    // pointer to last node in list**  
  
**Node<int> \*temp;            // temporary pointer**  
  
**first = new Node<int>(2);            // add first node to list**  
**last = first;**  
  
**temp = new Node<int>(4);    // create a second node**  
**last->setNext(temp);        // and add it to end of list**  
**last = last->getNext();**  
  
**temp = new Node<int>(6);    // create a third node**  
**last->setNext(temp);        // and add it to end of list**  
**last = last->getNext();**

**Write code to remove the second node from the list. Be sure to free the memory occupied by the freed node, and that the linked list still contains the remaining nodes.**

Code to remove the second node:

    temp = first->getNext();            //make temp point to node to be removed      
    first->setNext(temp->getNext());    // make first node point to third node  
    delete temp;                        // free removed node

Code to display the items in the linked list (putting them in a function is left as an exercise):

    temp = first;  
    while (temp != nullptr)  
    {  
        cout << temp->getItem() << ' ';  
        temp = temp->getNext();  
    }

Chap. 2

* Recursion is an alternative to iteration
* Box Trace: each box conducts a recursive call
* Binary search: array must be sorted, also find what side the value is on (Right or Left side)

Chap. 5

* Recursion is inefficient bc of constant function call
* Recursion is efficient bc it’s easy to code (clarify complex solution)
* Clear, efficient iterative solution is always better than recursion
* Backtracking: Backing up when you encounter a barrier, get a new sequence of steps, combine recursion and backtracking to solve problems, always go back to origin

Chap. 4

* Linked list are linked together by pointers
* They have regular pointers and null pointers
* headPtr = new Node<std::string>();
* headPtr = nullPtr;
* isempty always bool
* getCurrentSize returns int
* Shallow Copy: 2 copies
* Deep Copy: 1 copy
* Array = easy to use, but has fixed size, can waste memory, good for small project, access array items directly
* Linked based can change size, don’t have fixed time, requires more memory, traverse the linked chain

Chap. 3

* ADT is a collection of data
* Class = public and privates
* Specification = header file
* 1st step for implementation = choose data structure
* Interface = add, contains, remove
* Core methods = Constructors, add(setters, mutators), display(getters, assessors), remove
* Driver client tests core methods

Chap. 1

* Object oriented design = Describe solution to problem
* Polymorphism = Virtual function that checks type at run time
* Cohesion = high -> well defined easy to use program(documentation)
* Coupling = low ->Measure of dependence from another module
* Abstraction = bare bone implementation functions, does not dictate how to implament(used without knowing how to implament)
* Minimal interface for class = public files only
* Complete interface = all including public and private

**1. What is infix notation?**

Infix notation is the notation commonly used in mathematics and many programming languages where the operator is placed between its operands. For example, to indicate the sum of operands a and b, we write:a + b

**2. What is prefix notation?**

In prefix notation the operator is placed before its operands. For example, to indicate the sum of operands a and b, we write: + a  b

**3. What is postfix notation?**

In postfix notation the operator is placed after its operands. For example, to indicate the sum of operands a and b, we write:  a  b +

**4. Which of these 3 notations is the most complex?**

Prefix and postfix expressions can be written without using parentheses, operator precedence rules or associativity rules. To completely specify the order of evaluation for operators, parentheses, operator precedence rules and/or associativity rules are often needed. This makes the processing of infix expressions more complex.

**Chapter 5**

**Chapter 4 Linked-Based Implementations**

**int list[SIZE];  
      
    for (int i=0; i<SIZE; i++)  
        list[i] = 2 \* i;  
      
    for (int i=0; i<SIZE; i++)  
        cout << list[i] << endl;**

To use a resizable array, the code is:

    const int SIZE = 5;  
    int \*list = new int[SIZE];     // allocate array  
      
    for (int i=0; i<SIZE; i++)  
        list[i] = 2 \* i;  
      
    for (int i=0; i<SIZE; i++)  
        cout << list[i] << endl;  
      
    delete [] list;                         // de-allocate array

1. When we pass an array to a function in C++, we normally use 2 parameters. That are those two parameters?

We pass the array itself and the number of elements in the array. The parameter used to pass the array actually contains the memory address (location of) the first element of the array. For example, given the array:

    int list[5] = {1, 2, 3, 4, 5};

We might have a function called print that we call like this:

    print(list, 5);

The function might look like this:

    void print( int array[], int size)  
    {  
        for (int i=0; i<size; i++)  
            cout << array[i] << ' ';  
        cout << endl;  
    }

2. When an array is not full of data (i.e., some array elements contain useful data and others do not), what is a common technique for keeping track of which parts of the array contain useful data and which parts are "empty"?

We put the useful data at the beginning of the array (at the smaller indexes) and leave the unused/empty elements at the end of the array. We use a variable, say called size, to hold the number of elements that we are currently using. Array elements with indexes 0 through size - 1 contain useful data. Elements at indexes size through the end of the array are unused.

1. Given the function

    int func(int num)  
    {  
        if (num == 2)  
            return 2;  
        else  
            return num \* func(num - 1);  
    }

What is output when this function is called using:

    cout << func(5) << endl;

The output is "120". The recursive calculation is 2 \* 3 \* 4 \* 5 = 120.

There is a problem with this recursive function. Identify the problem.

If the function is called with an argument that is smaller than 2, the base case will never be reached. The recursion will go on "forever". Actually, the program will abort when it runs out of memory to store the activation records.

2. Write a recursive function that has two parameters - an array of strings and the number of strings in the array. The function should print the elements of the array (onr per line) in reverse order.

    void printBackwards( string array[], int length )  
    {  
        if (length == 0)  
            return;  
        else  
        {  
            cout << array[length-1] << endl;  
            printBackwards(array, length - 1);  
        }  
    }  
Sample call:  
  
    string array[] = {"zero", "one", "two", "three", "four"};  
    printBackwards(array, 5);

1. What is an ADT? An ADT is is a logical description of a group of data values and the operations that are allowed on them. This specification should not indicate how the data will be stored or how the operations will be implemented.

2. According to the textbook, what is the final step in designing an ADT (assuming that you plan to implement the ADT in C++)? Write an abstract base class for the ADT that documents the ADT specifications.

3. Should a client programmer be able to write an application based on the information contained in an ADT interface file? Yes. According to the textbook, the interface file should contain comments that completely specify the methods of an ADT. For each method, the interface file should contain:

* the purpose of the method
* a description of all parameters
* a description of the return value
* any preconditions and postconditions

4.What do we call the place where code modules meet and interact? an interface

5. What is a module? A module is an independent piece of code that provides specific and tightly coupled functionality. Examples in C++ are functions, classes and methods.

6. What is a client programmer? A client programmer is the programmer who uses a class (or class template) to create an application program.

7. What is an object? An object is a variable whose data type is defined by a class. That is, if we declare variables of a class type, we call them objects. Objects are also called instances of a class.

**C++ Interlude 1 C++ Classes**

**1. What is a class template?**

A class template allows the user to separate the functionality of a class from the type of data that it operates on. A class template is a pattern or blueprint that describes how to build a family of classes that all behave basically the same. The client programmer can specify the data type they want to use, and the compiler automatically generates the appropriate class.

2. Write a class definition (not a class template) for a class called *Rectangle*. All data members should be private and all methods should be public. Only include prototypes for the methods - the implementations will be written in the next question. The class should include:

* two data members called *length* and *width* (double values)
* a default (no argument) constructor that sets the length and width to zero
* a parameterized constructor that has parameters that allow the client to specify the initial values for the rectangle length and width
* an accessor (get) method for each data member that returns its value (2 methods)
* a mutator (set) method for each data member that allow the client to change that value stored in the data member
* a method to return the area of the rectangle (length times width)

Important: methods that are not intended to modify the data members of an objects must be constant

   class Rectangle

   {  
 private:  
 double length;  
 double width;  
 public:  
 Rectangle();  
 Rectangle(double initLength, double initWidth);  
 double getLength() const;  
 double getWidth() const;  
 void setLength(double newLength);  
 void setWidth(double newWidth);  
 double getArea() const;  
 };

3. Write implementations for al methods in question 2.

Rectangle::Rectangle()

{

length = 0.0;

width = 0.0;

}

Rectangle::Rectangle(double initLength, double initWidth)

{

length = initLength;

width = initWidth;

}

double Rectangle::getLength() const

{

return length;

}

double Rectangle::getWidth() const

{

return width;

}

void Rectangle::setLength(double newLength)

{

length = newLength;

}

void Rectangle::setWidth(double newWidth)

{

width = newWidth;

}

double Rectangle::getArea() const

{

return length \* width;

}

**1. A disadvantage of using a fixed array is that you can run out of room for elements. If you use a dynamically allocated array (the textbook calls them resizable arrays), you have the option of re-allocating the array to a larger size.**

**To show that you know how to use a resizable array, convert the following code to use a resizable array. Include code to free the memory used by the array at the end of your code.**

-When each module performs one well defined task. We say it that it is: cohesive

-A(n) \_\_\_ allows two modules to comm. with each other: interface

-What is wrong with Fibonacci sequence recursively?: same values over and over