COMP322: Assignment 1 Background Information- Winter 2013

Due at 11:30pm EST, 5 Feb 2013

1 Introduction

This file is designed to accompany assignment 1 as it includes important information on the assignment. We will be talking about this in class but you should read it on your own first.

2 Background Information

2.1 Storing a sparse polynomial

The most natural way to store a polynomial in C is to create an array where each index of the array represents a coefficient. For example, we might specify that the ith index of the array corresponds with the coefficient of the term of degree i

For example:

```
int main() {
int numbers[4];
numbers[0]= 1;
numbers[1] = 2;
numbers[2] = 3;
numbers[3] = 5;
}
```

could store the polynomial

$$1 + 2x + 3x^2 + 5x^3$$

While this is a very natural way to store a polynomial, there are some applications for which this is not the most efficient means of storing a polynomial. For example, to store the polynomial $x^{100} + 1$ one would need to create an array of size 101, a waste of space. It's a waste of space since most coefficients are 0.

This problem is compounded when we deal with polynomials in multiple variables where you need to create a multidimensional array. In this case, suppose you wanted to store polynomials in 4 variables, each of which could have degrees up to 1000, then the total size of your array would be 1000^4 most values of which are 0. This is a huge waste of memory, and in fact if you try to create an array this large in C++, you may very well get an error (try this)

```
int largeMatrix[1000][1000][1000][1000]; //most likely causes compiler error
```

If a polynomial is very *sparse*, a smarter way to represent it is to store a list of individual *elements*. Each element will have 2 things in it, a degree and a coefficient. If a specific degree appears in the list, then there is a term with that coefficient and degree. If a specific degree does NOT appear in the list, then we assume the coefficient of that degree is 0.

For example, the above polynomial $3x^{9000} + 2x^4 + x + 3$ would be stored as follows:

$degree$	$coefficient$
9000	3
4	2
1	1
0	3

(To store a polynomial in multiple variables, one only needs to store multiple degrees for each term–for example, a degree for x and a degree for y.) Unfortunately, this sort of table is difficult to implement in practice because it's size is constantly changing. This brings us to the next idea:

2.2 Linked List

A *linked list* is a data structure used in computer programming that can change sizes quickly. (You can read more about it at `http://en.wikipedia.org/wiki/Linked_list`)

The idea is to define a type which contains a *link* to another element of the same type. Then you can keep track of the entire list by maintaining a pointer (in Java terms a reference) to the first element in the list. You can then access the 2nd element in the list by following the *link* from the first element to it's next element of the same type. If the value is NULL, then this means we have reached the end of the list. Each element in the list is often referred to as a *node*

To do this in C++(or C), you should define a new type by defining a `struct` which consists of the data you need to store as well as a pointer to the next element in the list. For example, to create a linkedlist of ints, you would define a new type:

```
struct IntNode {
       int value; // stores the actual value
       IntNode* next; //stores a pointer to the next element in the list
};
```

The equivalent definition in Java would be:

```
public class IntNode {
     public int value; //elements of a struct are public by default
     public IntNode next; //note that in Java an IntNode would be a reference by default
}
```

A couple key differences between the C style version and the Java style version:

- By default, everything in a struct in C is public. In Java by default it's package protected (closer to private)
- In Java, there is no way to create an IntNode that is NOT a reference type. In C, you can store an IntNode directly if you wanted, thus the * is necessary to denote that you want to create a pointer.

- In C++NULL is equivalent to null in Java
- You need a ; after the final closing brace in C++

In C++, C, and Java, the dot operator means the same thing. If you have an expression that evaluates to an element of a certain struct or class, you can access it's public properties via a dot operator. For example:

```
IntNode x;
x.value = 3;
x.next = NULL;
```

It is important to note though that in C or C++, by default the type is NOT a reference or pointer. In fact, with every type of data in C or C++, you can specify whether you want to store it directly or using a pointer.

If we want to *traverse* a linked list (that is, if we want to visit every value), we'll use a loop and keep following the *next* pointer. We'll then check to make sure that current node is not null. For example:

```
//assume that root is a pointer to the head or first element of a linked list
IntNode* current = root;
while (current != NULL) {
     //do something with current. for example, print it's value:
     cout << current->value << " "; //see later for explanation on ->
     current = current->next; //update pointer
}
```

2.3 Pointers in C and C++

We'll talk about this in class, but it is useful to outline a few tips on using pointers in C.

In general, in C or C++, any type can be accessed via a variable storing it's value or via a pointer storing the address of the value. Storing something in C or C++as a pointer is equivalent to storing a reference in Java.

To create a pointer variable in C, you add a * to the type. For example, to create a pointer to an int, you would write:

```
int* x;
```

(One technical thing: to create 2 pointers at once you actually have to write `int *p, *q;` That is, a star for each variable)

To make this useful, you need to specify which address you want the int pointer to point to. There are two ways to do this:

1. Use the & operator to get the address of an existing int variable.

```
int* p;
int x = 3;
p = &x; //p refers to the address of x
```

2. Use the *new* operator to create space in the computers memory free store to store an integer.

   ```
   int* p = new int;
   ```

   Of course in this case we have no idea what value is stored at the location pointed to by p. Since we used the new operator, it will be necessary to, before our program exits, use the `delete` operator as well, to assure there is no memory leak.

Any time you want to access the data pointed to by a pointer, you can use the * operator. This is known as *dereferencing a pointer*.

```
int x = 3;
int* p = &x; //p points to the variable x now
(*p) = 1; //change the value pointed to by p to be 1
cout << x ; // prints 1 now
```

When you have a pointer to a struct, you can still use the * operator to deference it.

```
IntNode foo;
IntNode* fooPointer = &foo;
(*fooPointer).value = 3;
```

In the above `fooPointer` is an expression of type `IntNode*`, this means it can be dereferenced (since it is a pointer). `*fooPointer` thus has the type `IntNode`. Since `*fooPointer` is an `IntNode` that means it has a property `value` and we can set it equal to 3. Note that the parenthesis ARE required as if you omit them and write simply `*fooPointer.value`, it will be interpreted as `*(fooPointer.value)`

Because this is used very frequently, a little of what's known as *syntactic sugar* (or perhaps more accurately in this case syntactic iodine) was introduced. If you have a pointer `p` to a struct or class in C++, you can write:

```
p->value = 3;
```

Where the `->` means the same thing as the properly parenthesized * and .

Since there is no automatic garbage collection in C++, you need to `delete` the data pointed at by a pointer when you are through with it. If the pointer is called `p` you can do this by writing `delete p;` You have to be careful though, because if `p` is NULL or if `p` has already been deleted you'll have problems. For example:

```
int *p = new int;
int *q = p; //creates variable q and makes q and p aliases
delete p;
delete q; //EEK! q has already been deleted. Program crashes!
```

In theory, every `new` should be paired with a `delete`.