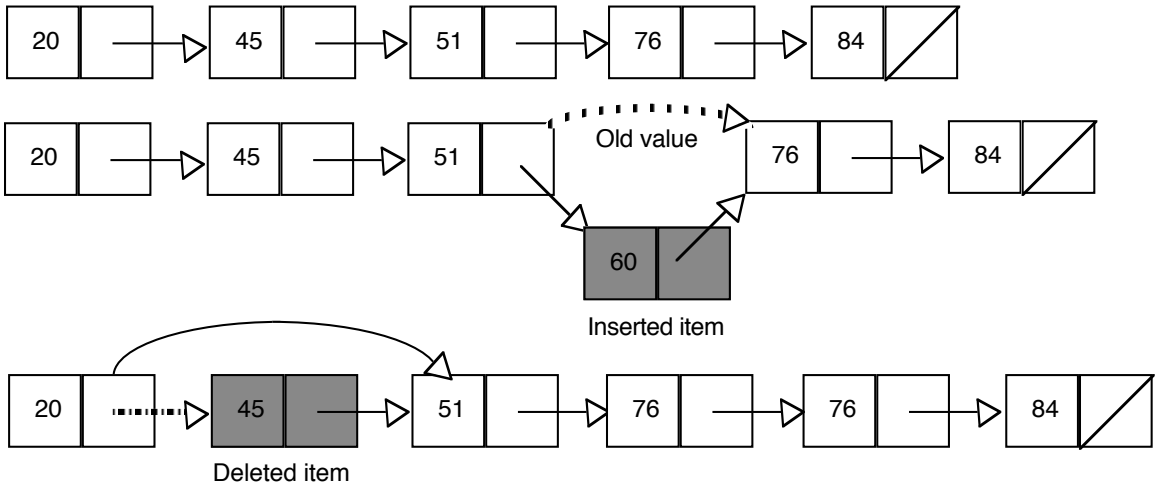
Chapter 4-1, Lecture notes

Preliminaries

1. The ADT list defined in chapter 3 has
   1.1 Operations of insert, delete, and retrieve
   1.2 Arrays for ADT not always best since they are of fixed size and lists are not.
   1.3 When you insert an item, you typically must move items over to insert the new item and when you delete an item, you must move items to close up the gap in the list.
      1.3.1 Have too many items and the array is full, you are stuck
      1.3.2 Can delete and reallocate an array but
         1.3.2.1 Not commonly done
         1.3.2.2 Still need to copy the data from the old array to the new array (waste space and time)
      1.3.3 Another method: linked lists, via pointers, that can grow and shrink dynamically
      1.3.4 Figure 4-1 on page 172 shows how a linked list would work in inserting and deleting an item:



2 With the linked list, have data then a pointer that literally points to NEXT item.
   2.1 If know where the item is then can know where the next item or successor is.
   2.2 With insert:
      2.2.1 Allocate a new item
      2.2.2 Set the value into the item,
      2.2.3 Link the new item's link to the successor and
      2.2.4 THEN link the current item to the new item
      2.2.5 Note: order is important since can, literally, lose an item if not careful.
   2.3 With the delete:
      2.3.1 Take the item before the one going to delete
      2.3.2 Move its pointer to the deleted item's successor

2.3.3 THEN delete the correct item

2.4 Each item is "linked" to the successor: the linked list.

2.5 It can grow and shrink as needed.

3 Let's review pointers.

3.1 If declare an ordinary variable x as an integer, the C++ compiler allocates a memory cell that can hold an integer.

3.2 Like so:

```
int x;
x = 5;
```

3.3 If wanted to display the variable, all that's needed is:

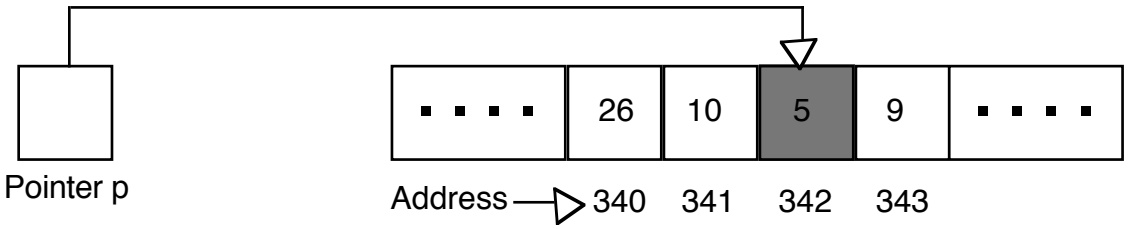
```
cout << "The value of x is " << x << endl;
```

3.4 A "pointer variable" or just "pointer" contains the location or "address" of a memory cell

3.4.1 A pointer is a bit tricky concept

3.4.2 The pointer p's value would be "where" the value is and not the "what".

3.4.3 Figure 4-2 on page 173 shows this:



3.5 Now two big questions:

- How to get a pointer variable p to point to a memory cell
- How to use p to get to the content of the memory cell to which p points?

3.5.1 You declare the pointer variable p as:

```
int *p;
```

3.5.2 That would declare an integer pointer p that would point onto to integers.

3.5.3 Be careful defining more that one variable at a time.

```
int *p, q;
```

3.5.4 Would have p as a pointer but q would be a regular integer variable.

3.5.5 It is like the following:

```
int *p;
```

```
int q;
```

3.5.6 To declare two integer pointers, you can do:

```
int *p;
int *q;
```

or:

```
int *p, *q;
```

3.5.7 And doing:

```
int x;
```

3.5.8 Would declare a regular integer variable.

3.6 The pointers and integer variable would be allocated before the program executes: “static allocation”.

3.6.1 Initially, the contents of p, q, and x are undetermined.

3.6.2 But can place the address of x into p and therefore point to x by using the C++ address-of operator & as follows:

```
p = &x;
```

3.6.3 However, the following is NOT allowed:

```
p = x;  // THIS STATEMENT IS ILLEGAL
```

3.6.4 That is illegal because of a type class: p is a pointer and x is an integer.

3.6.5 The notation of “*p” represents the memory cell to which p points.

3.6.6 So can store a value into the memory cell that p points to by doing:

```
p = &x;
*p = 6;
```

3.6.7 Now if do:

```
cout << *p << “ “ << x << endl;
```

3.6.8 It would print out:

```
6 6
```

3.6.9 Since *p points to the memory cell of x.

3.7 Can use the “new” command to allocate memory to the pointer.

3.7.1 The memory comes from a part of memory allocated to program by the operating system.
3.7.2 That part of memory is typically referred to as the "heap".
3.7.3 If tried to allocate memory and could not, the new command would throw a "std::bad_alloc" error
3.7.3.1 Have to allocate in a "try" block and have a corresponding "catch".
3.7.3.2 We'll assume we have enough memory for now and skip the try and catch.
3.7.4 So, if did the following:

```
p = new int;
*p = 7;
cout << *p << " " << x << endl;
```

3.7.4.1 That would print out:

```
7 6
```

3.7.4.2 The point p pointed to a different memory cell than x.
3.7.4.3 Now if had pointer q, could have p and q point to the same memory location like so:

```
q = p;
```

3.7.4.4 If no longer needed the memory pointed to by q, could release it by doing:

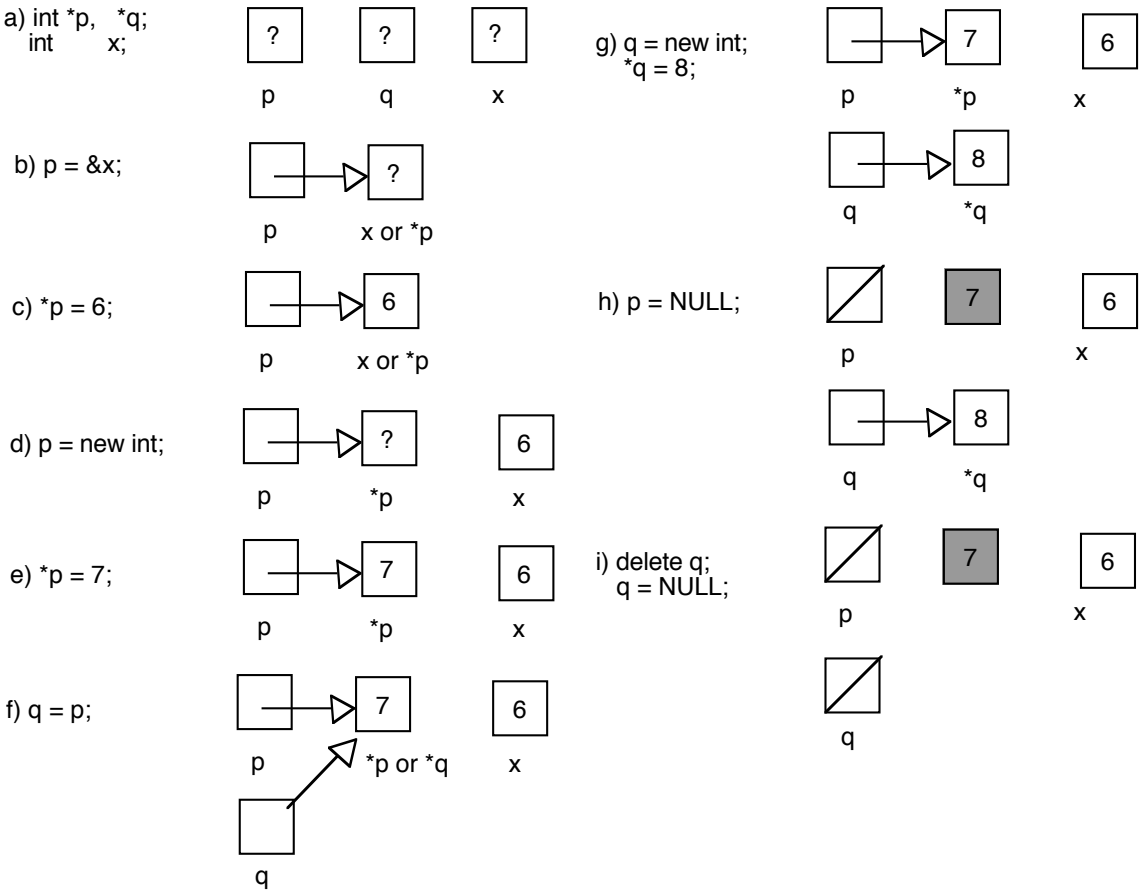
```
delete q;
```

3.7.4.5 If don't release it and assign another memory location to q, would lose the memory and cause a **memory leak**.
3.7.4.6 Note: the address in q is unchanged when the memory is released.
3.7.4.7 Need to set the pointer either to another memory location or to NULL – and that would be for all pointers that pointed to the deleted memory location. Like so:

```
p = new int;
q = p;
delete p;
p = NULL;
```

3.7.4.8 The pointer q still points to the deleted memory location and, if used, can cause difficulties
3.7.4.9 .Figure 4-3 on page 177 shows the memory leak with part h where p is set to NULL before deleting the memory.

3.7.4.10 Part i shows the correct way of deleting memory then setting q to NULL.



3.8 Figures 4-4 and 4-5 on pages 178 and 179 show more difficulties with pointers. An ADT that uses pointers is called “pointer-based”.

4 Arrays are by their nature in C++ pointers.

4.1 Can actually dynamically allocate an array.

4.1.1 The static way of allocating is:

```
const int MAX_SIZE = 50;
double anArray[MAX_SIZE];
```

4.1.2 The compiler allocates a specify number of memory cells for the array.

4.1.3 Typically, the size of memory cells differs according to the type.

4.1.4 Typical sizes are: double – 8 bytes, int – 4 bytes, bool – 1 byte, and char – 1 byte.

4.1.5 For the above array, the number of bytes allocated for anArray would be 50 * 8 or 400 bytes.

4.1.6 Most of the time you don’t have to be overly concerned with the size of the array unless it is VERY large.

4.2 Now, can treat anArray as a pointer and to the following:

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

4.2.1 That would dynamically allocate the array using a variable instead of a constant.
4.2.2 Can access the array as usual: anArray[0], anArray[1], and so on.
4.2.3 Though can use pointer offset notation to access the array also (and any array, dynamically allocated or not). Like so:

*anArray is the same as anArray[0]
*(anArray+1) is the same as anArray[1]

4.2.4 The "+1" would not just add a 1 to the address in the pointer of anArray. It actually does "+(1*8)" with 8 being the size of one element in the array.
4.2.5 Therefore if anArray started at location 1000, then *(anArray+1) would point to 1008.
4.2.6 Most of the time, we merrily ignore what is happening in the background and just use the indexing.

4.3 One thing we can do is to increase the size of the array by allocating a new block, copying the old values, and releasing the old array. Like so:

```
double *oldArray = an Array;                // copy pointer to array (don't lose old array)
anArray = new double[2*arraySize];          // double array size
for (int index = 0; index < arraySize; ++index)
        anArray[index] = oldArray[index];   // copy old array
delete [] oldArray;                         // deallocate old array
```

4.4 Most of the time we'll use statically allocated arrays but there will be times to use dynamically allocated arrays.

5 The pointer-based linked lists are a powerful tool (can simulate linked lists using arrays but we'll go with the pointers for now).
5.1 In figure 4-1, we showed linked lists.
5.2 Each component of the linked list is called a "node" – consists of both a data item (or items) and a "pointer" to the next item.
5.3 The way linked lists work is not the easiest concept to grasp. Let's define a structure called Node like below:

```
struct Node
{       int item;
        Node *next;
}; // end Node
```

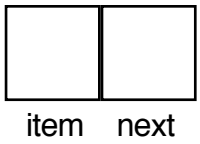
5.4 Now the statement:

Node *p;

5.4.1 That makes pointer p be able to point to a Node. Therefore we can do:

p = new Node;

5.4.2 Would get a node like:



5.4.3 If wanted to access the item, use the pointer notation like so:

p->item

5.4.4 Can access the item part of Node.

5.4.5 Could use a clumsier notation of *(p).item but we'll use the "->" instead.

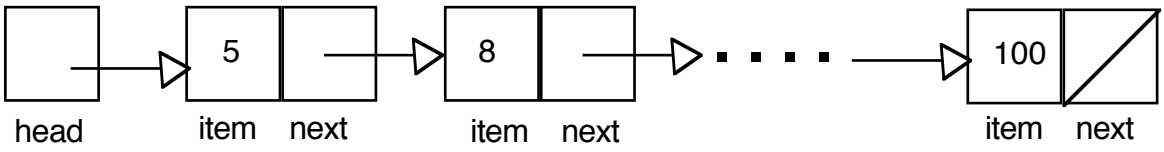
5.5 For a linked list to work, need a "head pointer" to point to the first of the linked lists.

5.5.1 Then the first node points to the second node, the second node points to the third node, and so on.

5.5.2 Declare the head pointer, or head, as:

Node * head;

5.5.3 Figure 4-7 on page 183 shows how this would work:



5.5.4 The last node has a NULL in its next to indicate the end of the linked list.

6 By the way, don't fall into the common mistake of assuming must use the "new" command before assigning a pointer a value – you don't.

6.1 In fact, if do the following:

```
head = new item;  // incorrect use of new
head = NULL;
```

6.2 Will LOSE a chunk of memory.