COMP225

Annabelle McIver (Lectures Weeks 1--6).

Consultation: Mondays 1--3pm. Other times by appointment.

Get a Review Quiz from the COMP225 web pages.

Quick review of pointers (Carrano, chapter 4)

int *p;   -- statically allocates a pointer variable p, with undefined value.   (Is it NULL?)

p = new int; -- dynamically allocates a new memory cell, which p “points” to. (What happens if the cell cannot be allocated?)

*p -- the contents at address p.

delete p; -- deallocates memory. (Does p still “exist” after a delete?)

What's wrong with this program?

```
struct Node
{   int item;
    Node *link;
};

void print ( Node* list) {
 // POST: prints the contents of a linked list.
 while ( list != NULL ) {
   cout << list ;
   list = list -> Node;
 }}
```

This is an address, not the contents.

This is rubbish.

What's wrong with this program?

```
struct Node
{   int item;
    Node *link;
};

void print ( Node* list) {
 // POST: prints the contents of a linked list.
 while ( list != NULL ) {
   cout << list-> item ;
   list = list -> link;
 }}
```

We need to preserve the head of the list.

```
struct Node
{   int item;
    Node *link;
};

void print ( Node* &list) {
 // POST: prints the contents of a linked list.
Node* temp= list;
while ( temp != NULL ) {
   cout << temp-> item ;
   temp = temp -> link;
 }}
```

What's wrong with this program?

```
struct Node
{   int item;
    Node *link;
};

void delete ( Node* &list) {
 // POST: deletes the head of a list.
  list = NULL;
 }
```

This loses the entire list without deallocating memory.

The list variable must point to the next item.

```
struct Node
{   int item;
    Node *link;
};

void delete ( Node* &list) {
 // POST: deletes the head of a list.
     Node* temp;
 if (list != NULL) {
     temp= list; list= list-> link;
      delete temp;
 } }
```

Quick review of dynamic data structures

- stacks (Carrano, chapter 6)
- queues (Carrano, chapter 7)

```
class Stack
 {  public:
      Stack(); // constructor

      bool isEmpty();
         // Returns true iff stack is empty.

      void push (ItemType newItem);
        // Pushes newItem onto the stack.

      void pop (ItemType & t);
        //Puts top of stack into t and removes it.
};
```

Example: Correcting input.

When inputting text , it's common to make mistakes. Those mistakes may be corrected using the backspace.

Eg. Hekko ⌫ ⌫ ⌫ llo Worp ⌫ ld

comes out as "Hello World"

Actual Input	Corrected Input
Hekko	Hekko
⌫	Hekk
⌫	Hek
⌫	He
l	Hel
l	Hell
o	Hello

Example: Correcting input.

The change in the Corrected Input with respect to the Actual Input suggests that we can use a stack to store the Corrected Input.

What if cInput is empty?

```
Stack cInput;
char ch, c;
cin.get(ch);
while (ch != 'EOL')
{ if (ch == '⌫') cInput.pop(c);
  else cinput.push(ch);
  cin.get(ch);
}
```

Example: Correcting input.

The change in the Corrected Input with respect to the Actual Input suggests that we can use a stack to store the Corrected Input.

```
Stack cInput;
char ch, c;
cin.get(ch);
while (ch != ‘EOL’)
{ if (ch == ‘⌫’) {if (!isEmpty.cInput()) cInput.pop(c);}
  else cinput.push(ch);
  cin.get(ch);
}
```

Example: Printing a stack in reverse order (bottom to top).

We need a temporary stack and two loops.

```
void printStack (Stack s) {
       ch x;
       Stack temp;
       while (!s.isEmpty() {
         s.pop(x);
         temp.push(x);
        }
        while (!temp.isEmpty()) {
           temp.pop(x);
           cout << x;
           s.push(x);
       }
 }
```

```
class Queue
 {  public:
      Queue(); // constructor

      bool isEmpty();
         // Returns true iff queue is empty.

      void push (ItemType newItem);
        // Pushes newItem onto the back of the queue.

      void pop (ItemType & t);
        //Puts the front item into t and removes it.
};
```

Queues and Stacks may be implemented by arrays or linked lists.

```
class Stack {

 public:
   .....
 private:
  Node* top; // Top of stack.
};
```

```
class Queue {

 public:
   .....
 private:
  Node* front; // Front of queue.
  Node* back; // Back of queue.
};
```

Stacks and queues will be significant for processing tree-like data structures.

Recursion review

Recursion is a technique that solves a problem by solving a smaller problem of the same type, and using the solutions of the smaller problems to construct a solution to the whole problem.

Examples:

- Searching a sorted array may be done by splitting the array in two and searching only one of the two halves (binary search);

- Printing the items of a stack may be done from bottom to top by removing the top item, printing the remaining smaller stack (from top to bottom), and then printing the top item.

Recursion review

The technique of recursion only works provided the following issues are borne in mind.

- The recursion must stop at some point i.e. there must be a base case (sometimes more than one!)

- The smaller problems must actually approach the base case;

- The solutions to the smaller problems must really contribute to the construction of a solution to the original problem.

Other issues to bear in mind:

• If many values are recomputed many times, then recursion can be very inefficient;

• If the specification of a recursive function requires it to return a value (of some type), then it *must* return a value. If your implementation does not, then there is something wrong with it.

• There is a space overhead involved in the use of recursion, as with each recursive call, the values of the local variables together with the details of the calling function. Effectively a stack data structure is used to store this information.

Example:

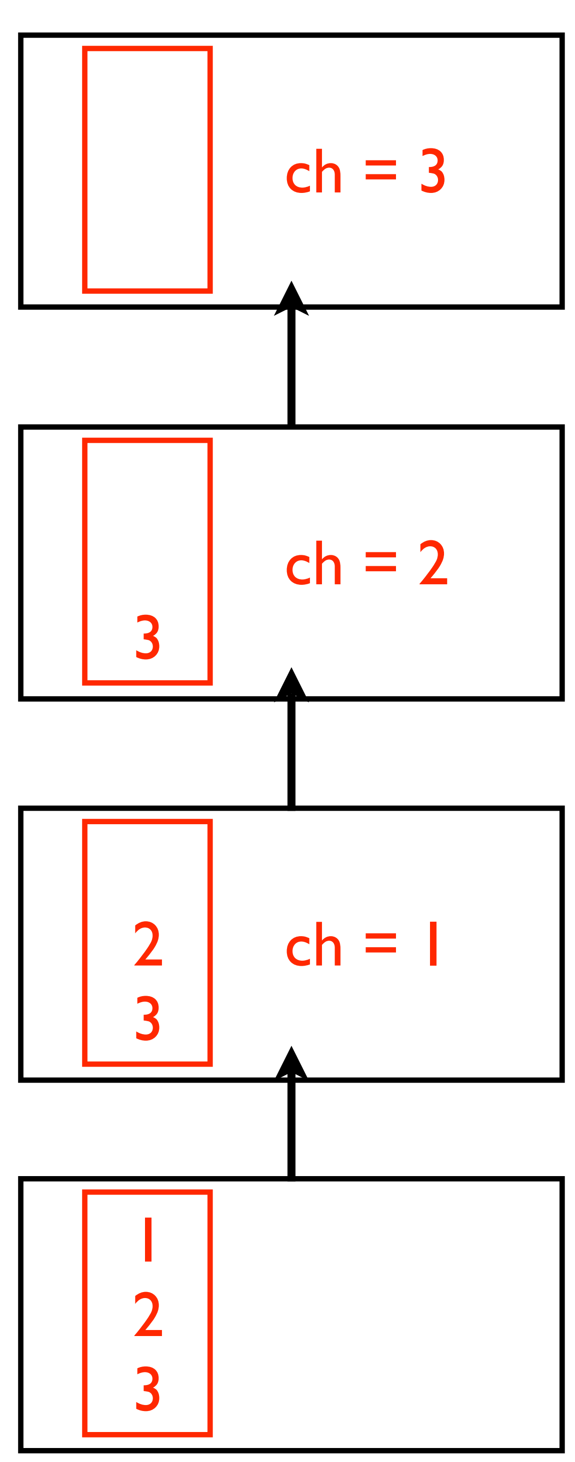
Printing a stack in reverse order, without using a temporary stack.

```
void reverseStack (Stack s)
 {
   if (!s.isEmpty()) {
        char ch;
        s.pop(ch);
        reverseStack(s);
        cout << ch;
        s.push(ch);
  }
 }
```

Base case: do nothing

Subproblem is smaller, and contributes to the whole solution only because we output the previously-saved ch

```
void reverseStack (Stack s)
 { //Print stack from bottom to top
   if (!s.isEmpty()) {
        char ch;
        s.pop(ch);
        reverseStack(s);
        cout << ch;
        s.push(ch);
    }
}
```



Beware the space overhead!

We don’t need to use a temporary stack (as we do in an iterative implementation) since the recursion effectively builds a stack “for free”.

Example:

Printing a linked list.

```
void printList (Node *head)
 {
   if (head != NULL) {
        cout << head -> item;
        printList(head-> link);
  }
 }
```

Base case: do nothing

Subproblem is smaller,
and contributes to the whole
solution only
because we output the item at the
current top first.

Note: when control returns to the calling function, has the original list changed? What does "head" point to?

Example: Binary search

When designing a recursive solution, it's important to be aware of what variables need to be passed to the recursive calls so that the calls can be made to subproblems.

```
int binarySearch (char A[], int first, int last, char key)
 {// PRE:  A[first] <= key and A is sorted.
  // POST: Returns the greatest index i, st A[i] <= key

    if (first >= last) return first;
    else {
          int mid = (first+last)/2;
          if (A[mid] <= key) return binarySearch(A, mid, last, key);
          else return binarySearch(A, first, mid-1, key);
    }
 }
```

Error! Only a smaller instance if first + 1 < last.

```
int binarySearch (char A[], int first, int last, char key)
 {// PRE:  A[first] <= key and A is sorted.
  // POST: Returns the greatest index i, st A[i] <= key

    if (first + 1 >= last) {
        if (A[last] <= key) return last;
        else return first;
     }
    else {
            int mid = (first+last)/2;
            if (A[mid] <= key) return binarySearch(A, mid, last, key);
            else return binarySearch(A, first, mid-1, key);
    }
}
```

Two base cases.

first and last required to specify subproblems

Using program invariants helps to ensure that the implementation is correct (more on this later). In this case the implementation always ensures that A[first] <= key, and it moves first and last as far as they can be moved given the available information.

```
int binarySearch (int A[], int first, int last, char key)
 { // Returns the greatest index i, st A[i] <= key, if A is sorted,
   // and A[0] <= key
    if (first + 1 >= last) { if (A[last] <= key) return last;
                               else return first;
                             }
    else {
            int mid = (first+last)/2;
            if (A[mid] <= key) return binarySearch(A, mid, last, key);
            else return binarySearch(A, first, mid-1, key);
   }
```

Testing and debugging.

Testing and debugging.

Specification -- this says what the program is supposed to do, as well as setting out initial conditions.

```
void mySort(int A[], int n);
 // PRE:  A is an array with n items;
 // POST: Prints contents of A sorted  in increasing order.
```

Try to get used to using a more precise mathematical-style for specifications.

```
void mySort(int A[], int n);
// PRE: The length of A is n.
// POST: for all i st. 0 <= i < n-1,  A[i] <= A[i+1]
```

You should get used to deciding what tests to do =before= you write any code. That way you'll better understand the problem.

Precise specifications can tell you what to expect from your program before you test it.

Consider the following searching problem:
Let A be a sorted array containing n items. You are asked to write a program such that given a target K, it returns the index of the item with value K.

On Monday, you are told to assume that K is contained in A, and that all the items are distinct.

On Tuesday, you are told the customer made a mistake and that the items might not be distinct.

On Wednesday, you find out that sometimes K is not in A, and the index returned should be the value of the index if K were inserted.

On Thursday, it turns out that the index might not be a current index of A.

On Friday, you discover that sometimes A is empty.

Let’s look at a shorter way to say all of those things, in a way that will avoid inconsistencies.

Return the greatest index i such that:

- i <= n, and
- for all i <= j < n,   K <= A[j], and
- for all 0 <= j < i,   K > A[j]

Monday: The spec says return i, st A[i] == K

Wednesday: The spec says return the largest i, where A[i] >= K, but all values to the left are less than K.

Thursday: If K > all values, return n, else if K < all values, return 0.

On Friday: Return 0.

Testing and debugging.

For arrays, lists, stacks, queues etc. use boundary testing. A common place where implementations are buggy are

empty lists, lists with one or two items, at the end of the list etc. with, or without repeated items.

Testing and debugging.

"Out of bounds"

Copying data into array items which are not there. (This is actually how a lot of so-called malicious code is malicious.)

```
int array[5];
int n=0;
while (n <= 5) {
 A[i]= i;
}
```

The last assignment writes to array index 5, which does not exist. Not all C++ compilers warn about errors like this --- typically it generates a segmentation fault at runtime.