Chapter 23

Index Loops

This section is about using loops to look though every letter of a string. For example, finding the first z, or counting how many z's there are. But it's really about using loops and indexes to look through any kind of list. Searching strings is just an easy way to practice.

Once we know how to look through letters in a string, we can use the same tricks to search a list of Customers or enemy monsters.

23.1 Indexes

If you remember, way back I wrote a string is made from individual characters. If you have `string w="cowbell";` it's really a list of 7 characters. `w = "cow"+"bell";` is also a list of 7 characters. It doesn't remember those two parts. Even `""+"cowb"+""+"ell"` works out to the exact same list of 7 characters.

The characters in a string are numbered, starting from 0. Here's the traditional picture of some strings with the numbering underneath:

```
string w="cat";

//  c a t
//  0 1 2

string poem="fish - swimX2.";

//  f i s h   -   s w i m X 2 .
//                      1 1 1 1
//  0 1 2 3 4 5 6 7 8 9 0 1 2 3
```

`"cat"` is obvious – it has three letters, numbered 0, 1, 2. I mixed things up in `poem`, but it's just 14 characters, numbered from 0 to 13. Spaces, dashes, and

numbers all count as 1 character each.

To look up one character, use the number, with square brackets around it:

```
string w = "cat";

print( w[0] ); // c
print( w[1] ); // a
print( w[2] ); // t

string q="cowbell";

print( q[1] ); // o
print( q[2] ); // w
```

The official name for the number in square brackets is an **index**.

Most computer numbering starts at 0. We usually call that a **zero-based index**. Here's a fun list of stuff that happens automatically because we start from zero:

- If a string has 10 letters, they're numbered from 0 to 9.
- The first letter is always `w[0]`.
- If you want the 6th letter, use have to subtract one and write `w[5]`.
- The last letter is always *one less* than the length.

After reading that, it seems like it would have been easier to start from 1, but you get used to 0, and it does work out.

Obviously, indexes are always `int`s. You have to pick out one character (what would `w[1.5f]` even be?)

Using an index really gives us a character. We can assign `w[0]` to a character, or compare it to one. If you remember, the name is `char` and they use single-quotes:

```
string w = "Leopard";
char first = w[0]; // character 'L'

if(w[1] == 'x') {} // compare to character

if(w[2]=="x") {} // ERROR -- can't compare char and string
```

Indexes *always* start at 0. There's no way to tell a string to start numbering itself at 1. Here's an example where I try to trick the computer (but it doesn't work):

```
string a="cat";
print( a[0] ); // c
a = "were"+a; // renumbers, so 'w' is at 0
print( a[0] ); // w
print( a[4] ); // c  cat pushed down to 4,5,6
print( a[5] ); // a
print( a[6] ); // t
```

Out-of-range errors

Using indexing can give us an exciting new error. `w="cat"` has indexes 0, 1, 2. What should happen when someone uses `w[3]` or `w[50]`? It should be some sort of error. But `w[3]` isn't wrong. It's the correct way to ask for the 4th letter, and `w` could have had four letters. Maybe it always will.

So, we can't give you a red error ahead of time. We can only give you the error when we come to the line. The program crashes and gives a red **run-time** error:

```
string w="goat";
print( w[3] ); // t
w="rat";
print( w[3] ); // ERROR
```

The error message is *IndexOutOfRangeException: Array index is out of range.* That's not too bad, since now we know an index is the number in the `[]`'s. I guess *Array* is the technical term for how a string stores its letters.

Using a negative index gives the same error. You'd think the computer would know ahead of time that -1 is never legal. But it's simpler to have off-the-edge in either direction be the same error.

23.1.1 Length of a string

This is another of those things that has a real rule to it, but the explanation uses stuff that's too much trouble to learn now, so I'm just going to say it's magic.

You can find the integer length of a string `w` by using `w.Length`. Examples:

```
string animal="bear";
print ( animal.Length ); // 4
w="camel";
print( w + " has " + w.Length + " letters"); // camel has 5 letters
w=""; print( w.Length ); // 0
```

23.2 Variables as indexes

The most clever and useful trick with indexes is that you can use variables and formulas. We already knew you could do that with other numbers, but the way

it works with indexes is just so extra clever.

These two examples use an all-number equation as the index. They aren't good for anything, except as examples:

```
string w="abcdefg";
// a b c d e f g
// 0 1 2 3 4 5 6
print( w[3+2] ); //  same as w[5], f
print( w[8-2*3] ); //  same as w[2], c
```

These next three are more realistic, using an actual variable, `num`, for the index. Even cooler, since `num` is a variable, we can change it in-between lookups:

```
int num=0;
print( w[num] ); // a
num++;
print( w[num] ); // b

print( w[num+1] ); // c
```

This is really no different than the rule "any place that takes an int, can use anything that works out to be an int."

The most common trick using a variable index is in a loop. Suppose `w` has length ten. It's numbered 0 to 9. If we make a loop counting 0 to 9, we can use the loop variable to look at each position:

```
string w="abcdefghij";
for( int i=0; i<10; i++ ) {
  print( w[i] );
}
// Output (on different lines):
// a, b, c, d ...
```

In our minds, this is a 0 to 9 loop, hitting each index, and also hitting us each letter with `w[i]`. It's a standard "look at each item" loop.

There's one thing left to make it perfect – we should look up the current length of `w` and use that to end the loop:

```
string w="abcdefghij";
for( int i=0; i<w.Length; i++ ) {
  print( w[i] );
}
```

It's easy to be off by 1, so let's double-check with an example: suppose `w` is `"goat"`. That means `w.Length` is 4, and `w` is numbered 0 to 3. The loop runs

while `i<4`, so it counts 0,1,2,3. So it exactly hits every index it should.

That math works for any length string. A length 20 string has indexes 0-19, and a standard "run 20 times" loop sets `i` from 0-19. Using `i<w.Length` stops 1 before the size, which is the last one.

It's good to check a very short example too: suppose `w` is just `"a"`. Length is 1, so the loop runs on 0 and stops when it gets to 1. Which is perfect.

23.3 String Loop examples

Using that "look at every letter" loop, we can fill in the body to do a few things. I'm going to write these as functions because I think they look nicer.

We can count how many copies of a letter are in a string. I think it makes sense here to have the letter be a `char` input:

```
int timesInString( string w, char countMe ) {
  int count=0;
  for(int i=0; i<w.Length; i++) { // <- standard every letter loop
    if( w[i]==countMe )
      count++;
  }
  return count;
}

// samples:
print( timesInString("cattle talk", 't') ); // 3
print( timesInString("banana", 'x') ); // 0
```

The loop driver is showing me every letter, in `w[i]`. Inside the loop, `if(w[i]==countMe)` is really asking "is the current letter the one we want?"

We can use the loop to make a new string from the old one, one letter at a time. This example adds a slash after each letter:

```
string addSlashes( string w ) {
  string ans="";
  for(int i=0; i<w.Length; i++) {
    ans = ans + w[i] + '/';
  }
  return ans;
}

// sample:
print( addSlashes("abcd") ); // a/b/c/d/
```

The same idea can reverse the string. We just add each letter to the *front* of our answer. I'm adding a testing line, so we can watch it work:

```
string reverse( string w ) {
  string ans="";
  for(int i=0; i<w.Length; i++) {
    ans = w[i] + ans;
    print( ">" + ans );
  }
  return ans;
}

// sample:
print( reverse("frog") ); // gorf
// >f      <- the testing lines
// >rf
// >orf
// >gorf
```

That seems pretty impressive for such a short loop.

A note: this returns a reversed copy. We've seen this sort of function before. If we want to change a string to be backwards we'd using it like `ani=reverse(ani);`.

Removing a letter can be done by making a 1-letter-at-a-time copy, but using an `if` to skip the letter we don't want. We don't really skip it, we just don't copy it over:

```
string remove( string w, char removeMe ) {
  string ans="";
  for(int i=0; i<w.Length; i++) {
    if( w[i] != removeMe ) ans += w[i];
  }
  return ans;
}
```

`remove("elephant ear", 'e' );` would give us `"lphant ar"`. It also works if the letter isn't there – it copies everything with no changes. `remove("abc", 'X')` gives you back `"abc"`.

Removing spaces is fun (spaces are perfectly good characters.) `remove("in a tent", ' ')` gives us `"inatent"`.

If we want to remove two letters, we can use it twice:

```
// remove < and >:
w="<goat> <cow> <pig>";
w=remove(w,'<'); // goat> cow> pig>
w=remove(w,'>'); // goat cow pig
```

Very similar to `remove` is doing a replace. We still check for that letter, but instead of skipping it, we add the replacement letter. Since the loop body uses `w[i]` twice, I'm copying `w[i]` into a temp character variable:

```
string replace( string w, char oldChar, char newChar ) {
  string ans="";
  for(int i=0; i<w.Length; i++) {
    char ch=w[i];
    if( ch != oldChar ) ans += ch;
    else ans += newChar
  }
  return ans;
}

// sample:
print( replace("elephant ear", 'e', 'X' ); // XlXphant Xar

print( replace("cat,dog,cow", ',', ' ' ); // cat dog cow
```

23.3.1 Index inputs/outputs

Once we know about indexes, it seems natural to use them as inputs and outputs. For example, removing the 3rd letter from a string, or finding the position of the first `'a'`.

We really are going to use zero-based indexes for everything, which takes a while to get used to. If we call a function to get the first `'m'` in `"camel"`, it will tell us 2, because that's the index. Likewise, if we wanted to remove the 3rd letter, we'll have to use 2.

This function removes a letter from whatever index you say. It does the usual trick building a string from letters, but skips the one you didn't want:

```
string removePos( string w, int removeIndex ) {
  string ans="";
  for(int i=0; i<w.Length; i++) {
    if( i != removeIndex ) ans+=w[i];
  }
  return ans;
}

print( removePos("werecat", 2) ); // weecat
print( removePos("abc", 0) ); // bc
```

The `if` is the interesting part. Before, we compared letters, using `w[i]`. Now we're comparing index numbers, so just using `i`.

Some fun uses: `w=removePos(w,0);` gets rid of the first letter of `w`. And `w=removePos(w, w.Length-1);` gets rid of the last. `w.Length-1` is the mini-formula for the last letter.

Here's a version which removes a range. The first number is the start index, the second is how many to remove. I'm doing it that way, since everyone else does.

The math for the ending index is fencepost stuff. `removeRange(w, 3, 4)` says to remove 4 things, starting at index 3. That works out to 3, 4, 5 and 6. The formula works out to `start+howMany-1`. That's the last index we skip. The function:

```
string removeRange( string w, int startIndex, int howMany ) {
  string ans="";
  int endIndex=startIndex+howMany-1; // "-1" to fix fencepost problem
  for(int i=0; i<w.Length; i++) {
    if( i < startIndex || i > endIndex ) ans+=w[i];
  }
  return ans;
}
```

It's the same as removing one thing, except the `if` checks for a range. Notice how it's a rare "not-in-range" test

Some examples:

```
removePos("abcdefghijk", 6, 4); // abcdefk ("ghij" is gone)
removePos("werecat", 2, 3); // weat ("rec" is gone)
removePos("abc", 0, 1); // bc
removePos("abc, -10, 999); // "" (but not an error)
```

For fun, here's another way to write `removeRange`, using two loops. Loop one adds everything before the range. Loop two adds everything after:

```
string removeRange( string w, int startIndex, int howMany ) {
  string ans="";
  int endIndex=startIndex+howMany-1;
  for(int i=0; i<startIndex; i++) ans+=w[i];
  for(int i=endIndex+1; i<w.Length; i++) ans+=w[i];
  return ans;
}
```

I like this because you have think about not being off by 1. The first loop uses `<startIndex`, since we don't want that one. The second loop begins at `endIndex+1`, since we don't want the letter at `endIndex` either.

We can also write a function for the exact opposite thing – getting only letters in the range. That's officially called a **substring**. This will loop only over the indexes we want:

```
string substring( string w, int startIndex, int howMany ) {
  int endIndex=startIndex+howMany-1;
```

```
  string ans="";
  for(int i=startIndex; i<=endIndex; i++) ans+=w[i];
  return ans;
}

// sample:
print( substring( "werecat bowling", 4, 8) ); // cat bowl
```

The loop from `startIndex` to `endIndex` is obviously right, but only because we've been using that math a lot.

This one can crash on bad indexes. If the start is negative, it crashes right away. If `howMany` is too big it goes off the end and crashes.

A different type of function searches for a letter and tells us the position. For example, find the index of the first "e". If the word has 10 letters, it returns 0-9 for where it was, or -1 for not found (handling not found makes these trickier.)

The function. It cheats using the return-from-middle trick when it finds one:

```
int indexOfFirst( string w, char findMe ) {
  for(int i=0; i<w.Length; i++) {
    if(w[i]==findMe) return i;
  }
  return -1;
}

// samples:
print( indexOfFirst( "old deer", 'd') ); // 2
print( indexOfFirst( "old deer", 'f') ); // -1
```

It looks funny seeing `return -1;` at the end. But, it's like any other return-from-middle – the answer is -1 only if we didn't find our letter and quit early.

Here's the same thing, but not using return-from-middle. The condition says to stop when we find it or get to the end:

```
int indexOfFirst( string w, char findMe ) {
  int ans=-1; // means we haven't found it
  for(int i=0; i<w.Length && ans==-1; i++) {
    if(w[i]==findMe) ans=i;
  }
  return ans;
}
```

`ans=-1;` is the fall-through trick. If we find the letter, we set `ans` and quit the loop. If we never find it, `ans` will still be -1.

This is more awkward and feels a bit hacky, which is the point – return-from-middle is a nice trick. But a double check in the loop, for not off the end

and for something else, is common.

We can improve find-the-first by adding the ability to say where to start looking – find the first `'e'` past index 4. All we do is start at whatever index they tell us:

```
int indexOfFirst(string w, char findMe, int startPos ) {
  for(int i=startPos; i<w.Length; i++) // the only change
    if(w[i]==findMe) return i;
  return -1;
}
```

A fun use of this is finding the second time something is in a string. Find the first one starting from 0, then look again from just past there:

```
int e2=-1; // position of second e
int e1 = indexOfFirst(w, 'e', 0); // find 1st one
if(e1>=0) // quit if there weren't any e's
  e2 = indexOfFirst(w, 'e', e1+1);
print("second e at index " + e2);
```

To find the *last* `'e'` in a string we can use a backwards string loop. This is our first one, but it's not as exciting as we'd hope:

```
string indexOfLast( string w, char findMe ) {
  for(int i=w.Length-1; i>=0; i--) { // standard backwards string loop
    if(w[i]==findMe) return i;
  }
  return -1;
}
```

It's nice to double-check these. `w.Length-1` is the last one. And it runs `i>=0`, which means it hits 0. So it hits exactly last to first.

23.4 Odd string loops

I'd like to check if one string starts with another, for example if `"theater"` starts with `"thi"` (almost, but not quite.) We can do that using a loop which walks through both strings at the same time.

```
t h e a t e r
t h i
0 1 2
```

The loop will run 0, 1, 2, comparing the matching letters. To make it simpler, I'll say the strings have to be in order: longer one first, then the shorter one to check:

```
bool startsWith( string w, string front ) {
  for(int i=0; i<front.Length; i++) { // go to end of shorter word
    if( w[i] != front[i] ) // <- use i to get letters from both words!
      return false;
  }
  return true;
}
```

Using the same index in two side-by-side lists is a common trick.

Going to the end of `front` is being lazy. We'd crash checking past the end of `w` if `front` was longer. When we use this trick for real we compare lengths and compute how far:

```
// go through w1 and w2 together, as much as possible:
int len;
if(w1.Length<w2.Length) len=w1.Length;
else len=w2.Length;
for(int i=0; i<len; i++) {
```

We can make a loop to check whether a string has the same two letters in a row: for examples `"ballet"` yes and `"elephant"` no. We'll compare each letter to the one after it, using `w[i+1]`. This is our first real used of index math.

We have to stop at the second-to-last position, since we can't compare the last letter to the one after it:

```
bool hasARepeat(string w) {
  for(int i=0; i<w.Length-1; i++) { // only to 2nd-to-last letter
    if( w[i] == w[i+1] ) // compare letter at i with letter at i+1
      return true;
  }
  return false; // we didn't find any == pairs
}
```

We can do this another way. Instead of comparing to the letter after `i`, we can compare to the letter before it, We shift the loop by +1: start on the second letter and go all the way to the end:

```
// version where we check the letter before i
bool hasARepeat(string w) {
  for(int i=1; i<w.Length; i++) {
    if( w[i-1] == w[i] ) return true;
  }
  return false;
}
```

A funny thing is that both versions compare the same sequence: 0 and 1, 1 and 2 .... But they "feel" different, and easier-to-read code is important (I

prefer the before version. Skipping 0 is easier, and an obvious way to warn about loop hijinks.)

This next one is very different. A palindrome is the same forward and backward. The fun ones are sentences, like "Madam, I'm Adam" (if you remove spaces and punctuation and ignore upper/lowercase.) But it's really anything where the back half is the front half reversed: `abcdcba`. Notice there's only one `d`. When the length is odd the middle letter can be anything.

My plan is to make two markers, at the first and last letter. A loop will compare them, then move both towards the middle. Quit when they cross. I've never written a loop like this, but the plan sounds good:

```
bool isPalindrome(string w) {
  // start 2 index variables, at first and last letter:
  int i1=0, i2=w.Length-1;
  while(i1<i2) { // keep going until they cross in middle
    if(w[i1] != w[i2]) return false;
    // move both 1 step towards the middle:
    i1++; i2--;
  }
  return true;
}
```

This is the first loop we've ever written where 2 things in the test are changing at once. Doing that was the plan, but a quick test is good: `i1` goes up, `i2` goes down, so clearly `i1<i2` will eventually be false. This won't run forever.

Adding a test print is a nice way to check (especially when we get wrong answers.) Let's try printing the letters, and the next indexes:

```
 ...
  while(i1<i2) {
    print("compare "+w[i1]+" and "+w[i2]);
    if(w[i1] != w[i2]) return false;
    i1++; i2--;
    print("move to "+i1+" and "+i2);
  }
  ...
```

Running lets us watch it. If we didn't get the idea, it might help. It we had bugs, we might see wrong numbers:

```
isPalindrome("abccba");
compare a and a
move to 1 and 4
compare b and b
move to 2 and 3
compare c and c
move to 3 and 2  <-- the loop quits here
```

If we hate the double-moving-vars plan, we can rewrite it as a regular loop. We'll go from 0 to the middle, using a formula to find the opposite position:

```
bool isPalindrome(string w) {
  int mid=w.Length/2;
  for(int i=0; i<mid; i++) { // loop goes only through 1st half
    int i2=(w.Length-1)-i;
    if( w[i] != w[i2] ) return false;
  }
  return true;
}
```

It's comparing the exact same spots as version 1. I even re-used `i2`.

Formulas like `(w.Length-1)-i` are easy to get wrong, especially off-by-one. I tried to use an extra set of parens to show it's the last spot, `w.Length-1`, moving down with `i`. I usually write out a numbered word to double-check.

Here's one last loop with even trickier index math. It checks whether a string *ends* with a certain word. Here's a picture of how we line up `"edwinton"` and `"ton"` to check for a match:

```
e d w i n t o n
0 1 2 3 4 5 6 7
          t o n
          0 1 2
```

We obviously want a simple loop going through `"ton"`, comparing matching letters. The problem is the indexes in the words aren't the same – they're off by 5. They're *all* off by 5, which means we can compute one number to fix it.

It takes a lot of thinking, but the off-by is the difference in their lengths (8 for "edwinton" minus 3 for "ton" means we skip the 5 letters in front.) Otherwise it's like the `startsWith` loop:

```
bool endsWith(string w, string end) {
  // compute so end[0] lines up with w[shift]:
  int shift=w.Length - end.Length;

  for(int i=0; i<end.Length; i++) {
   if(end[i] != w[i+shift] ) return false;
  }
  return true;
}
```

This is new. Checking for doubles had index math (`w[i]!=w[i+1]`) and we compared side-by-side in startsWith. Now we're using both tricks together.

The math seems ugly, so as usual we can watch with debugging lines:

```
bool endsWith(string w, string end) {
  print("running with ("+w+") and ["+end+"]");

  int shift=w.Length - end.Length;
  print("shift="+shift);

  for(int i=0;i<end.Length;i++) {
   int i2=i+shift;
   print("compare "+i+"/"+i2); // print the numbers
   print("  "+end[i]+"/"+w[i2]); // and the letters
   if(end[i] != w[i2] ) return false;
  }
  return true;
}
```

A sample test:

```
endsWith("cowbell", "bezl");
running with (cowbell) and [bezl]
shift=3
compare 0/3
 b/b
compare 1/4
 e/e
compare 2/5
 z/l // <- returns false here
```