Lecture 6

CSE 110

8 July 1992

1 I Made a Mistake

When I said that the declaration `int foo;` initialized `foo` to be 0, I lied. The compiler is a allowed to initialize `foo` to anything it wants, including 0 or a random garbage value. To initialize `foo` to 0, write `int foo = 0;`.

2 Loops

Frequently we want a program to keep doing the same thing over and over again until something happens. That's called *looping*.

2.1 `while`

For example, we might play a game and then ask the user if he or she wants to play again. We'd want to do that over and over as long as the user kept saying 'yes', until finally the user said 'no'.

We use a `while` statement for that sort of control flow. The form of a `while` statement is:

`while (`*condition*`)` *statement*

To execute this kind of statement, the computer first evaluates the condition, which is just an expression. If the condition is true, it executes the statement. If the condition is false, it skips the whole thing and goes on to the next statement.

So far this is just like `if`. The difference is this: When the computer is done executing the statement in the body of an `if`, it goes on to the next statement. When the computer is done executing the statement in the body of a `while`, it goes back and repeats the whole statement. The computer will test the condition and execute the statement over and over, until finally when the condition is false, it gets to go on to the next statement.

Here's code to print out the numbers between 1 and 50, inclusive:

```
int x = 1;

while (x <= 50)
  printf("%d ", x++);
```

Control is stuck in the `while` loop until the value of `x` exceeds 50. Each time through the `while` loop, we call `printf()` to print the value of `x`, and we increment `x`. When we've incremented `x` so many times that its value exceeds 50, the test in the `while` statement fails and we drop down to the next statement.

Here's another example. This time the body of the `while` statement is a compound statement:

```
int n=0;

while ( n<1 || n>10 ) {
  printf("Please enter an integer between 1 and 10.\n");
  scanf("%d", &n);
}
```

We'll keep prompting the user and reading an integer until we get one between 1 and 10. Note that `n` is initialized to 0; if it were initialized to 7 instead for some reason, we'd never prompt at all, because the condition would be false the first time we got to the `while` statement.

3 More about `scanf()`

`scanf()` is a function, and so it has a return value. It returns a value to its caller to say whether or not it succeeded in doing what the caller asked. It returns an <`int`>.

In the quadratic equation program we did

```
/* get inputs a, b, and c from user. */
printf("Please input a, b, and c, one per line.\n");
scanf("%lf", &a);
scanf("%lf", &b);
scanf("%lf", &c);
```

but there's a somewhat simpler way:

```
/* get inputs a, b, and c from user. */
printf("Please input a, b, and c, separated by white space.\n");
scanf("%lf %lf %lf", &a, &b, &c);
```

`scanf()`'s first argument, the *format string*, can contain more than one conversion specifier (The `%lf`s are *conversion specifiers* because they specify how to convert the input to values), and it can contain white space. The format string is like a pattern that tells `scanf()` what things to look for in the input, in what order, and what to do with them if it finds them.

A `%lf` tells `scanf()` to start looking for input in the form of a decimal numeral, and that there will be an extra argument of type <pointer to `double`> which will say where to store the value that the numeral represents. White space in the format string tells `scanf()` to expect some white space characters in the input and to read them and throw them away.

The format string we gave to `scanf()` above tells it to read a decimal numeral into the <`double`> variable pointed to by `&a`, then read and discard some white space characters in the input, then to read another decimal numeral into the space pointed to by `&b`, discard more white characters, and finally read a last decimal numeral into the space pointed to by `&c`. When `scanf()` reaches the end of the format string, it stops reading the input, and any characters still unread stay there until the next time a function asks for input.

Now, we already know what happens if the `scanf()` call above gets the input `12 39.07 -.00003`, but what if you gave it `12 foo -.00003` instead? The `12` gets read and put into the space pointed to by `&a`, and then the blank character is all right because it's white space, but as soon as `scanf` sees the character `f`, it knows something's wrong—it's expecting something it can interpret as a <`double`>, and no <`double`> starts with `f`. What does it do?

It leaves the `f` and all the following characters unread, does not store anything into the spaces pointed to by `&b` or `&c`, and returns immediately. It returns the value 1, because it was only able to read, interpret, and store one of the conversions. If we had given `scanf()` the correct input with three numerals in it, it would have been able to read, interpret and store all three of the conver-

sions, and so it would have returned 3. `scanf()` always returns the number of conversions it was able to read, interpret, and store.

3.1 A Heinous Error

Beginners often make this mistake:

```
int num = 0;

while ( num < 1 || n > 50 ) {
  printf("Enter an integer between 1 and 50.\n");
  scanf("%d", &num);
}
```

What's wrong here is that if the user enters a bogus input, such as `foo`, instead of a decimal numeral, the `scanf()` will fail to convert the input, will return 0, will leave the value of `num` unchanged, and *will leave the bogus input unread*. Then the next time through the loop, the bogus input will still be there, and it must be read before anything else is,[1] so `scanf()` will fail again in the same way. This code loops forever if the user enters a bogus input, printing the prompt over and over again.

Because of situations like this, `scanf()` is rarely used to read input from users—it is much better for reading input from other computers, which are more predictable. But we can use it, if we are careful.

4 Flushing Input

To recover from bogus[2] input to `scanf()` above, we first need to detect the bogusness of the input, and then we need to discard the bogus input.

We already know how to do the former: We check the return code from `scanf()` and see if it's what we expected. This section is about how to do the latter.

[1] It would be a terrible disaster for the program to receive the input characters in any order other than the order the user typed them in. If you want your program to look at the characters in the input in some other order, you must read them into memory, and then you can look at the memory any way you want.

[2] I am not being cute here. 'Bogus' is an English word meaning 'counterfeit'.

4.1 getchar()

`getchar()` accepts no arguments. It reads one character from the input and returns that character as its return value. The character we read with`getchar()` is no longer available to be read again; next time we call `getchar()` or `scanf()` or any other input function, reading will commence with the character after the one we just read.

If there is no more input, or if `getchar()` can't read a character for some reason, it returns `EOF`. `EOF` stands for "End Of File".

Whatever `EOF` is `#defined` as, it can't have type <`char`>. Why not? Because if it did, then that <`char`> value might actually appear in the input, and then you wouldn't be able to distinguish between when `getchar()` returned `EOF` because it had happened to read that character in the input and when `getchar()` returned `EOF` because there was an error.

The return type of `getchar()` therefore can't be <`char`>, because `getchar()` needs to be able to return `EOF`, which is not a <`char`>. Accordingly, `getchar()`'s return type is <`int`>, and the characters it returns are converted to <`int`>s when they're returned.

You have to `#include <stdio.h>` to use `getchar()`, because you need the definition of `EOF`, and also because on most systems `getchar()` is actually a macro, and the definition of that macro is in `<stdio.h>`.

4.2 Character Constants

If you write the sequence `'f'`, the compiler generates a value of type <`int`> which represents the value of the character `f`. If `c` is an <`int`> variable in which a character is stored, you can see whether the character stored in `c` is the letter `f` by doing

```
if (c == 'f') { ... }
```

`'f'` is called a *character constant*. Some characters, such as newlines, are hard to type or would confuse the compiler if you actually put them into the source code. The compiler lets you write `'\n'` to represent the newline character; similarly, `'\t'` is a TAB character, `'\\'` is a backslash (\) character, and `' '` is a blank character. Then

```
if (c == '\n') { ... }
```

checks to see if the character stored in `c` is the newline character, and executes the statements in the body of the `if` if it is.

4.3 A Program to Count Words

This program counts the number of characters, words, and lines in its input.

```
#include <stdio.h>

int main(void)
{
  int c = 0, w = 0, l = 0;       /* Count of characters, words, lines */
  int ch;                        /* Current character */

  while ((ch = getchar()) != EOF) {
    c++;
    if ( ch == ' ' || ch == '\t' || ch == '\n') w++;
    if ( ch == '\n' ) l++;
  }

  printf("Characters: %d.  Words: %d.  Lines: %d.\n", c, w, l);
  return 0;
}
```

There aren't any notes for this program because there's nothing new here.

4.4 Flushing the Input Line

Now we know enough to recover from the heinous error of section 3.1. If `scanf()` doesn't return what we expect, we'll use `getchar()` to discard all the input up to the end of the line, to give `scanf()` a fresh start next time we call it. Here's the code:

```
int num = 0;

while ( num < 1 || num > 50 ) {
  printf("Enter an integer between 1 and 50.\n");
```

```
  if (scanf("%d", &num) < 1) {
    /* Gobble up rest of input line */
    while (getchar() != '\n')
      /* do nothing */ ;
  }
}
```

There's only one new feature here: a `;` by itself is a *null statement*: it behaves syntactically like a statement, but it doesn't actually do anything. The gobbling up of characters happens in `getchar()`, which is in the condition part of the `while` loop, so the `while` doesn't have to do anything in its body. But the syntax rules say that `while` must be followed by a statement. So we follow it with a null statement, which doesn't do anything.