Lecture 7

CSE 110

9 July 1992

1 More Loops

Looping is so important that there are three ways to do it.

1.1 `do...while`

Consider this example:

```
int n = 0;

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

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. We had to be careful to fix up `n` in advance to make sure that the loop gets executed at least once.

There's another loop construct like `while`, which isn't as often used, but which might have been better for the example above. It looks like this:

`do` *statement* `while (`*condition*`) ;`

The difference is that the condition is tested *after* the statement is executed, instead of before. That means that the statement is always executed at least once. For example:

```
int n = 7;

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

Now it doesn't matter that `n` is initialized to 7, because we prompt and ask for a number from the user at least once anyway, and the 7 is obliterated by the call to `scanf()` before we ever look at the value of `n`.

Use `while` when it might be appropriate for the actions in the body to never happen at all; use `do`...`while` when you always want the actions to be executed at least once.

1.2 `for`

This code exemplifies the notion that every loop has four logical parts:

```
int x = 1;

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

There's an *initialization* to set up the variable or variables that form the main control of the loop; that's the `x = 1`. There's a *control expression* that says how long to run the loop and when to stop; that's `x <= 50` here. There's a *control update*, which updates the values the main control variables; that's `x++` here. And there's a *loop body*, which contains the statements that the control expression actually controls; that's the `printf()`.

There's a loop construct that makes all four parts explicit. It looks like this:

```
for ( initialization ; control ; update )
    loop body
```

This is functionally identical[1] to this code:

[1] Not quite identical, but the only difference is in the behavior of `continue` statements, which we'll see later.

initialization
`while (` *control* `) {`
    loop body
    update
`}`

For example, here's our example that prints the numbers from 1 to 50 again:

```
int x;

for ( x=1; x<=50; x++ )
    printf("%d ", x);
```

When to use `for` and when to use `while` is a matter of style. Typically, we like to use `for` when the initialization, control, and update expressions are all closely related, because it keeps them together, and `while` in other cases, because it doesn't emphasize a connection that isn't there.

Sometimes for some reason we don't want to have an initialization in a `for` statement. In that case we just omit it and leave a blank space between the open parenthesis and the first semicolon after the `for`. Similarly, we can omit the update expression. We can even omit the control condition, in which case the computer will take it as being always true. So as a special case,

```
for ( ; ; ) { ... }
```

performs the statement delimited by the braces over and over, forever.[2] This is functionally identical with

```
while ( 1 ) { ... }
```

2 Example: A Program to Computer Prime Numbers

A *prime number* is a positive integer, n, which is evenly divisible only by n and by 1. For example, 2, 3, 5, 7, 11, and 13 are prime numbers. 4 is not prime,

[2]There are ways to get out of this statement: You could use `return`, for example. Other ways we haven't seen yet include `goto` and `break`.

because it is divisible by 2; 6 and 9 are not prime because they are divisible by 3.

This program reads number inputs from the user and says whether they are prime or not, until the user enters 0, when it exits. It checks a number n to see if it is prime by dividing it by each number j between 2 and $\frac{n}{2}$; if the remainder, computed with the `%` operator, is zero, then n is evenly divisible by j and so is not prime.

```
#include <stdio.h>
#include <math.h>                /* For sqrt() */

int main(void)
{
  int n;

  for (;;) {
    printf("Enter a number, 0 to quit.  ");
    if (scanf("%d\n", &n) < 1)  /* Bogus input */
      while (getchar() != '\n') /* Discard input characters to end of line */
        /* nothing */ ;
    else {
      if (n == 0)
        return 0;
      else if (is_prime(n))
        printf("That number is prime.\n");
      else
        printf("That number is not prime.\n");
    }
  }
}

int is_prime(int n)
{
  int divisor;

  for (divisor=2; divisor < sqrt(n); divisor+=1)
    if (n%divisor == 0)
      return 0;

  return 1;
}
```