Indentation, `for` Loops and Strings

This lecture covers several useful details of the C language.
After this lecture you should be able to indent your code according to K&R conventions and use the `++` *operator and* `for` *loops in your programs.*

Indentation

Since the C compiler considers any sequence of white space characters to be equivalent there are many ways to format a program. However, some formatting conventions have been established to help convey the logical structure of the program. The most important of these conventions is the use of indentation.

The general rule is that statements embedded within another statement (i.e. inside pairs of braces) are indented several (3 to 8) spaces more than the indentation of the immediately surrounding statement. The placement and indentation of braces also helps convey the meaning of the code. Using a consistent indentation convention will greatly help you and others understand the structure of the program.

There are several popular indentation styles. Any one of these would normally be acceptable if used consistently. However, many students have tended not to use any style at all and this has caused much confusion.

From now on all programs submitted will be required to use the "K&R" indentation style. This style was first used by the designers of the C language (Kernighan and Ritchie). This is also the indentation style used in the examples and solutions given in this course.

Marks will be deducted for any C program written for a lab, assignment or exam that does not follow the K&R indentation conventions.

Increment Operator

The unary operator `++` is used in C to increment a variable by one. If the operator is placed before variable (e.g. `++i`) the variable is incremented before its value is used in the expression. If the operator is used after the operator (e.g. `i++`) the *initial* value of the variable is used in the expression. This operator has a lower precedence than any other unary operator but a higher precedence than non-unary operators.

Exercise: If `n` has the value 5, what is the value of the expression `++n - 1`? What is the value of the expression `2 * n++`?

The `--` operator is used in the same way as `++` but it decrements the variable by 1 instead of incrementing it.

The `for` Loop

Iterative loops are used so often that most computer languages have special language constructs to implement them. In C this is done with a statement called a `for` loop.

Every iterative loop must have three parts: initialization of the loop control variable (done once before the start of the loop), testing of the loop variable before each execution of the statements in the loop, and the increment (or other change) of the loop variable at the end of the loop. The `for` loop allows us to define a loop more compactly by specify each of these actions in one statement. Here's an example of a `for` loop:

```
for ( i=0 ; i<10 ; i++ ) {
   sum = sum + x[i] ;
}
```

The `for` loop is similar to a `while` loop except that it contains three expressions in parentheses after the `for` keyword. The first expression is evaluated immediately before the start of the loop. The second expression is evaluated before each iteration of the loop and terminates the loop if it evaluates to zero. The final expression is evaluated at the end of each iteration of the loop. The above code is equivalent to the following:

```
i=0 ;
```

```
while ( i<10 ) {
   sum = sum + x[i] ;
   i=i+1 ;
}
```

Exercise: Use a `for` loop to print all the powers of 2 with values between 1 and 256.

Nested Loops

It's often necessary to use one loop inside another. These are called *nested* loops. A different loop variable needs to be used to control each loop. For example, the following code would print all the possible combinations of two dice:

```
for ( i=1 ; i<=6 ; i++ ) {
   for ( j=1 ; j<=6 ; j++ ) {
      printf ( "%d and %d\n", i, j ) ;
   }
 }
```

Flag Variables

Sometimes it's not practical to put the code that controls whether a loop should continue inside the control expression in a `while` loop. In this case we can use an auxiliary "flag" to control the loop instead. The value of this variable is initialized outside the loop and changed inside the loop when it's necessary to terminate the loop. For example:

```
   int c, done ;
...
   done = 0 ;
   while ( ! done ) {
      c = getche() ;
      if ( c == 'x' ) {
         done = 1 ;
      }
   }
```

Exercise: What does the above code do?

Strings

Strings (sequences of characters) in C are declared and stored as character arrays. By convention each string in C is terminated with a null character (the character with value zero).

The C language supports this convention by creating properly-terminated arrays when a sequence of characters are surrounded by double quotes. Note that there are no operators that operate on strings so these string constants can only be used in variable initializations and as function arguments.

There are built-in functions that can manipulate null-terminated strings. For example, the `strlen()` function returns the length of a string, and the `strcmp()` function compares two strings. To use these functions you must include the `string.h` include file in your program (using `#include <string.h>`). For example:

```
#include <string.h>
...
char name[5] = "Jane" ;
...
n = strlen(name) ;
```

Exercise: Why is the string `name` declared as having 5 elements if it is being initialized with a 4-character value?

The functions `fgets()` and `fputs()` can be used to read and write a string from a file or the keyboard/display.