LECTURE 13: FLOW OF CONTROL (K&R 3)

VOCABULARY: STATEMENTS / BLOCKS

Recall that an expression is a combination of values/constants/variables/operators/functions that evaluates to another value.

An expression becomes a statement when it is followed by a semicolon.

E.g.,
```
x = 0;
```

Curly braces are used to group declarations and statements into a compound statement, or block.

Note that the closing brace is not followed by a semicolon.

Syntactically, the grouped statements are equivalent to a single statement.

E.g.,
```
{
  x = 0;
  y = 1;
}
```

IF STATEMENTS

Things to note:

1. The if condition is just testing a numeric value.
   We can use a shortcut in this test:
   `if(expression)` same as `if(expression != 0)`
   `if(!expression)` same as `if(expression == 0)`
2. An else always goes with the closest previous if that does not have an else. If you don't want this, you need to use braces to force the desired association.

ELSE-IF

Things to note:

1. Can have as many as you want.
2. They are evaluated in order, and if the condition evaluates to true for one, its statement is executed and we don't look at the rest.
3. An else at the end is equivalent to "none of the above."

SWITCH STATEMENTS

Familiar to those with Java experience.

Another way to do multi-way decisions.

```
switch (expression) {
  case const-expr1: statements
  case const-expr2: statements
  default: statements
}
```

This will test whether expression matches each of the constant expressions and execute the corresponding statements if so.

The constant expressions must be integer-valued.

Execution will fall through a switch unless you add break after statements.

counting digits/white space/other (K&R, page 59)

```
main() {
  int c, i, nwhite, nother, ndigit[10];
  nwhite = nother = 0;

  for (i = 0; i < 10; i++)
    ndigit[i] = 0;

  while ((c = getchar()) != EOF) {
    switch (c) {
      case '0': case '1': case '2': case '3': case '4':
      case '5': case '6': case '7': case '8': case '9':
        ndigit[c-'0']++;
        break;
      case ' ': case '\n': case '\t':
        nwhite++;
        break;
      default:
        nother++;
        break;
    }
  }
  printf("digits =");
    for (i = 0; i < 10; i++)
  printf(" %d", ndigit[i]);
  printf(", white space = %d, other = %d\n", nwhite, nother);
  return 0;
}
```

Two approaches:

if-else	test if i == 1 if that fails, test if i == 2 if that fails, test if i == 3 ... when we test i == 27, we have done 26 prior tests.
switch	usually compiled into assembly as a jump table.
	an array of "go to" instructions subscripted by the value of i.
	if i = 27, we look up the go to at address 27 in the table
	this way we only execute that one go to.

Falling through can be useful, but you should be careful with it as it may create unintended behavior if the program is modified later.

Good practice is to use break statements.

LOOPS

These are equivalent:

```
for (expr1; expr2; expr3)
  statement;
```

```
expr1;
while (expr2) {
  statement;
  expr3;
}
```

Note that any part of a for loop can be left out:
`for(init; loop-test; increment)`

left out	effect
init or increment	nothing is evaluated (the program must initialize and increment by other means)
loop-test	assumes permanently true condition and loops forever
	(must use other means to exit the loop)

comma operator

Most often use is in for loop statements.

Pairs of expressions separated by , are evaluated left-to-right.

Value of comma expression is value of right-most comma-separated expression.

Example use in a for statement:
`for(i = 0, j = strlen(s) - 1; i < j; i++, j--)`

DO WHILE

```
do {
  statement(s);
} while(expression);
```

Guaranteed to execute the statement(s) at least once, regardless of whether expression is true or false.

Used infrequently.

BREAK / CONTINUE

break	allows departure from a loop
	can be used in for, while, and do loops (similar to its use in switch)
	allows you to exit the current loop (one level only; important to remember when using break in nested loops)
continue	skips to the next iteration of the loop
	used to selectively execute statements in a loop iteration

GOTO / LABELS

Per K&R:
With a few exceptions [...] code that relies on goto statements is generally harder to understand and to maintain than code without gotos. [...] goto statements should be used rarely, if at all.

goto	redirects flow of control to a label
	can be used to break two or more levels
label	has the same form as a variable name
	is followed by a comma
	can be attached to any statement in the same function as the goto.

BINARY SEARCH

```
/* find x in v[0] <= v[1] <= . .  <= v[n-1]
   returns subscript of x if found, -1 if not */
int binsearch ( int x, int v[ ], int n) {
  int low, high, mid;
  low = 0;
  high = n - 1;
  while ( low <= high) {
    mid = (low + high)/2;
    if (x < v[mid])
      high = mid - 1;
    else if (x > v[mid])
      low = mid + 1;
    else    /* found match */
      return mid;
  }
  return -1;  /* no match */
}
```

LECTURE 13: FUNCTIONS (K&R 3)

FUNCTIONS/ PROGRAM STRUCTURE

C makes functions efficient and easy to use.
We want to use many small functions, NOT a few big functions.
Functions may reside in separate source files.
An Introduction to MAKEFILE is coming. (A process that K&R does not go into)

FUNCTION PROTOTYPES

Review:	

Comprised of:	Return type, function name, and (). E.g.,
	int fname();
	float fname();

	With argument list: types and, optionally, variable names. E.g.,
	int fname(int*, int, float);
	int fname(int array[], int i, float j);

Special case for null argument lists (a function that takes zero arguments):
Code prototype for this kind of function as: int fname(void);
Including void keeps compiler parameter checking enabled.
WRONG: int fname(); [...] x = fname(i, j, k); /* compiler will not catch this */

FUNCTION DECLARATIONS

Same as function prototype, except:	If the function takes arguments, variable names must be included in the argument list.
	Followed by { function statements;} not ;