CS113: Lecture 3

Topics:

- Variables
- Data types
- Operators
- Order of Evaluation
Variables, Data types

● Names of variables:
  – Composed of letters, digits, and the underscore ("_") character. (NO spaces; use underscore instead.)
  – First character must be a letter.
  – Only the first 31 characters matter.
  Similar rules for naming functions, etc.

● Data types.
  – C’s basic types and typical sizes:
    * `char` - a single byte, capable of holding one character (8/16 bits)
    * `int` - an integer (16/32 bits)
    * `float` - single-precision floating point (32 bits)
    * `double` - double-precision floating point (64 bits)
  – Size is compiler- and machine-dependent.
  – Qualifiers (e.g. `unsigned`, `long`) can be applied.
  – There are rules for conversion (e.g. adding an `int` to a `double` and then assigning the outcome to a `float`). See K&R page 42.
Variable declarations

- Variables must generally be declared before use.

```c
int lower;
int upper;
int step;
char c;
char d;
```

- Variables with the same type can be grouped together:

```c
int lower, upper, step;
char c, d;
```

- Variables can also be initialized in the declaration.

```c
int lower = 0, upper = 8, step = 1;
char c = 'f', d = 'z';
```

- What happens if a variable is not initialized and then used?

```c
void main()
{
    int a;
    printf( "The value of a is: %d\n", a );
}
```
Examples of Constants

- Integer constant: 1234
- long int constant: 12345789L
- Integers can be specified in octal (leading zero) or hexadecimal (leading 0x or 0X): 037, 0x1f. Representation of integers in different bases gives rise to OCT 31 = DEC 25 joke.
- Floating-point constant: 123.4
Character constants

• Escape codes corresponding to characters
  – For use in single-quotes; or in double-quotes, for instance in passing a string to `printf`
  – Examples: `\n` (newline), `\` (backslash), `\"` (double quote)

• Variables of type `char` can be thought of as either a character of an integer.

  ```c
  printf( "%c", 'a' ); /* a is printed */
  printf( "%d", 'a' ); /* 97 is printed */
  printf( "%c", 97 ); /* a is printed */
  ```

• Lower-case letters, upper-case letters, digits “consecutive”

  ```c
  'a' == 97, 'b' == 98, ..., 'z' == 122
  'A' == 65, 'B' == 66, ..., 'Z' == 90
  '0' == 48, '1' == 49, ..., '9' == 57
  ```

• Some more examples of the integer values corresponding to character constants:

  ```c
  '&' == 38, '*' == 42, '\n' == 10, '\\' == 92, ...
char Example

void main()
{
  char i;
  printf( "Here’s the alphabet, in lower-case:\n" );
  for( i = 97; i <= 122; i++ )
  {
    printf( "%c", i );
  }
  printf( \nHere’s the alphabet, in upper-case:\n" );
  for( i = 65; i <= 90; i++ )
  {
    printf( "%c", i );
  }
}

void main()
{
  char i;
  printf( "Here’s the alphabet, in lower-case:\n" );
  for( i = ’a’; i <= ’z’; i++ )
  {
    printf( "%c", i );
  }
  printf( \nHere’s the alphabet, in upper-case:\n" );
  for( i = ’A’; i <= ’Z’; i++ )
  {
    printf( "%c", i );
  }
}
Operators

- Recall the relational operators (>, >=, <, <=), equality operators (==, !=), and the logical operators (!, &&, ||).

- C has a number of arithmetic operators.
  - Assignment operator: =
  - Binary arithmetic operators: +, -, *, /, 
    * Can be applied to int, float, or double, except for % which can only be applied to ints.
    * % is the "modulus" or "mod" operator: a % b is equal to the remainder when a is divided by b. We won’t worry about what happens on non-positive values (implementation dependent). Example: 8 % 3 == 2.
  - Unary arithmetic operator: -. Example:
    \[ x = -y; \]
  - Shortcut operators: +=, -=, *=, /=
    \[ x += 2; \quad \text{/* equivalent to } x = x + 2; \quad */ \]
    \[ x *= 2; \quad \text{/* equivalent to } x = x * 2; \quad */ \]
  - Increment/decrement operators: ++, --
    \[ x++; \quad \text{/* equivalent to } x += 1; \text{ or } x = x + 1; \quad */ \]
    \[ x--; \quad \text{/* equivalent to } x -= 1; \text{ or } x = x - 1; \quad */ \]
++ and --: two tricky guys

- Two ways to use: before a variable, and after.

```c
/* First example */
int a = 10;
printf( "%d", a++ );

/* Second example */
int a = 10;
printf( "%d", ++a );
```

- I prefer not to mix ++ or -- into more complicated expressions.

- Note that the expression that ++ or -- is applied to must be an lvalue, e.g. a variable.

```
(x + 2)++;       /* no good! */
```

- True for assignment statements as well.

```
x + 2 = 8;       /* no good! */
```

- The result of applying ++ or -- to an lvalue is NOT a lvalue.

```
(x++)++;         /* no good! */
```

- Another complication (occurs in general when expressions have side effects):

```c
int a = 10;
printf( "%d %d", a++, a++ );
```
Order of Evaluation

How are expressions with many operators evaluated?

Two considerations:

- Precedence
  - How is $1 + 2 * 3$ evaluated? Is it $(1 + 2) * 3$, or $1 + (2 * 3)$?
  - It's the latter: the * operator has higher precedence than the + operator.
  - Parentheses must be used if we want the addition to be performed first.

- Associativity
  - What about expressions containing operators at the same precedence level? E.g., $(12 / 6 * 2)$ or $(5 - 3 - 1)$?
  - These parse as $((12 / 6) * 2)$ and $((5 - 3) - 1)$: they are left associative. (Most operators are left associative.)

See table on p. 53 of K&R.