Chapter 2: Types, Operators, and Expressions

- Variables and constants are the basic data objects manipulated in a program.
- Declarations list the variables to be used, state what type they are, and possibly initialize values.

2.1 Variable Names.

- There are restrictions on the names of variables and symbolic constants.
    - Names are made up of letters and digits.
        - The first character must be a letter.
        - The underscore counts as a letter.
            - It's useful for improving readability.
            - Library routines often begin function names with an underscore.
    - C uses all lowercase letters for variable names and upper case for symbolic constants.

2.2 Data Types and Sizes.

- There are only a few basic data types in C:
    - Char: Single byte capable of holding one character in local character set.
    - int: an integer, typically reflecting the natural size of ints on local machines.
    - float: single-precision floating point.
    - double: double-precision floating point.

- There are a number of qualifiers that can be applied to the basic types.
    - short
    - long

2.3 Constants

- An integer constant like 1234 is an int.
- A long constant will have an L appended to the end.
    - An integer too big for int will be taken as long.
- Unsigned constants will have U or UL appended to the end.
- The value of an integer can be specified in octal, hex, or decimal.
- A *character constant* is a character written within single quotes 'x'.
    - The integer value is the numeric value in the machine char set.
- A *constant expression* is an expression that involves only constants.
- A *string constant* is a sequence of zero or more chars in double quotes.
    - "This is a string literal or string constant"
    - The quotes delimit the string constant and are not a part of it.
    - Technically a string constant is an array of characters.
        - There is a null character appended to the end '\0'
- There is a difference between a char constant and a string constant.
- *enumeration constants* An enumeration is a list of constant integer values.
    - Example: `enum boolean {NO, YES}`
    - The first name in an enum value is 0, the next 1, and so on.
    - Enumerations provide a way of associating constant values with names.

2.4 Declarations

- All variables must be declared before use.
    - certain declarations can be made implicitly by context.

- A declaration specifies a type, and contains a list of one or more variables of that type.
- The qualifier *const* can be applied to the declaration of any variable to specify that its value will not be changed.
    - *const* can be used with array arguments to indicate a function does not change the array.

2.5 Arithmetic Operators.

- The binary arithmetic operators are +, -, *, /, and modulus %.
    - % can't be applied to float or double precision variables.
- Precedence:
    - *, /, % have the same precedence.
    - +, - have lower precedence than the above.
    - Precedence moves left to right.

2.6 Rational and Logical Operators

- Relational operators are: >, >=, <, <=
    - They all have the same precedence.
- The equality operators are: ==, !=
    - These have lower precedence than the relational operators.
- Relational operators have lower precedence than the arithmetic operators.

2.7 Type Conversions

- When an operator has operands of different types they are converted to a common type according to rules:
    - The only automatic conversions convert a "narrower" operand to a "wider" operand without losing information.

        - Converting an integer into a floating point is an example.
    - Converting to a shorter type losing precision is not illegal.
- Converting characters to integers:
    - The language does not specify whether variables of type char are signed or unsigned.
        - Can a char conversion to int ever produce a negative number?
        - The answer varies from machine to machine.
        - On some machines a char with the left most bit 1 will be converted to a negative integer.
        - For portability specify signed or unsigned if non-character data is to be stored in char variables.
- If a binary operator has two inputs of different types the standard is that the "lower" type is promoted to the "higher" type before the operation is executed.
- The following are informal rules if there are no unsigned operands;
    - If either operand is long double, convert the other to long double.
    - If either operand is double convert the other to double.
    - If either operand is float convert the other to float.
    - convert char and short to int.
    - if either operand is long convert the other to long.
- Conversions take place across assignments.
    - The value of the right side is converted to the type of the left.
- Long integers are converted to shorter ones or to char by dropping the high-order bits.
    - This happens regardless of the involvement of a sign extension.
- Forced type conversions can be done by casting `(type name) expression`
    - The expression is converted to the type name.
    - When casting the value of the variable is given as the proper type, the variable is not changed.

2.8 Increment and Decrement Operators.

- C provides two operators for incrementing and decrementing variables.
  - ++ adds 1 to its operand.
  - -- subtracts 1 from its operand.
- Both may be used as prefix or postfix operators.
  - prefix increments before the value is used.
  - postfix increments after the value is used.
  - So the context is different on how the value is being used.
  - Example:
    - Suppose that n is 7.
      - `x = n++;` sets x to 7. assigns then increments.
      - `x = ++n;` sets x to 8 because the value is incremented before the operation of assignment.
        - In both cases n is incremented to 8 but the assignment to x is different.
  - Only variables can use increment and decrement, using with expressions is illegal.
  - When no value is wanted and only the incrementing is important prefix and postfix don't matter.

2.9 Bitwise Operators.

- C provides six operators for bit manipulation.
  - may only be applied to integral operands.
  - &: bitwise AND
  - |: bitwise inclusive OR
  - ^: bitwise exclusive OR
  - <<: left shift
  - : right shift

    - ~: one's complement
- Bitwise AND operator:
    - Used to mask off some set of bits.
    - Example: `n = n & 0177;` sets all but the low-order 7 bits of n to 0.
- Bitwise OR operator:
    - Used to turn bits on.
    - Example: `x = x | SET_ON;` sets the bits in x to 1 that are 1 in SET_ON.
- Bitwise exclusive OR operator:
    - Sets the bits where the operands differ to 1 and 0 where they are the same.
    - Example: `011010 ^ 111001 = 100011`.
- The << and >> operators perform left and right shifts of the left operand by the number of bit positions in the right operand.
    - The right operand must be positive.
    - Since we are operating in binary we can consider this as multiplying or dividing by the powers of 2.
    - Example: `8 = 1000` and `8 >> 2 = 1000 >> 2 = 10 = 2` which is equivalent to division by 4 or 2^2.
- The unary operator ~ flips the bits of the operand.

2.10 Assignment Operators and Expressions

- Expressions like `i = i + 2` can be written in the form `i+=2`
    - The += is called an assignment operator.
- The equivalence is `expr op = expr2` is equivalent to `expr = expr op expr2`
    - Example:
        - `x *= y + 1` is equivalent to `x = x * (y + 1)`
        - So the second expression was the entire right side after =.

2.11 Conditional Expressions

- A *conditional expression* written with the ternary operator "?:" provides an alternative way to write conditional statements.
    - Usage `expr ? expr2 : expr3` is equivalent to:
        - if expr is true then expr2 else expr3.
        - Only one of expr2 and expr3 is evaluated.