Lesson #3

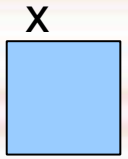
Variables, Operators, and Expressions

Variables

- We already know the three main types of variables in C: int, char, and double. There is also the float type which is similar to double with only single precision. Here we will use double exclusively for floating-point values.
- int is more precise and faster than double.
- A variable, like a memory cell, can only contain **one** value at a time.
- Putting a value in a variable that contains another value destroys the previous value.

Variable Declarations

- To reserve space in memory for variables, a declaration statement must be written. A simple declaration consists of the type and the identifier (for example: int x;).
- When you declare int x; in a C program, the operating system reserves 32 bits at a certain location in the computer's memory to store the variable named x.
- The same process applies to *char* and *double* variable declarations except that the operating system allocates 8 and 64 bits, respectively.



Variables and Memory

- Each variable used in C is stored at a specific address in the computer's memory. We do not really care what that address is but it is important for the operating system. The **&** operator provides us with the actual address of the variable in memory.

- For example, by declaring **int x;** I create a variable named **x**. This variable is stored at the address **&x** in the computer's memory.

Pointer Variables

- We know that the address of **x** can be represented by **&x**. It is possible to put such an address into a variable (known as a *pointer variable* or simply *pointer*).
- A pointer is in fact a variable that contains the address of another variable.
- A pointer variable can be int*, char*, or double*, meaning respectively pointer-to-int, pointer-to-char, and pointer-to-double. The type of the pointer must match the type of the variable it points to.

Pointer Variables

Let's have

```
int x = 10;
int* ptr;

ptr = &x;
```

- In the third instruction, we place the address of x into the pointer variable **ptr**. It is said that ptr *points to* x.
- The variable **ptr**, however, must be of a special type ready to hold addresses, specifically addresses of integers (since x is int). So to declare ptr we use the **int*** type, not int. **int*** means *pointer to integer*.

Pointer Variables

- Note that **int* ptr;** can also be written **int * ptr** or **int *ptr;**.

- The * operator reveals the value of the variable pointed by the pointer variable. Note that the * operator can only be applied to a pointer variable. *ptr will *follow the arrow* to the variable x and reveal its value. So, *ptr is in reality x.
- *ptr means: Go to ptr, follow the arrow, get the value.
- printf ("%d", *ptr); /* will display 10. */

Inaccuracies

- Putting certain values in a variable can lead to inaccuracies.
- **Cancellation error:** happens when the magnitude of the operands are too different.
  - Ex: 10000.0+0.0000015 would give 10000.0 (This is just an example. In reality the magnitudes must be much more different).
- **Arithmetic underflow:** happens when a number too small appears as 0.
  - Ex: 0.0000001*0.0000001 would give 0.0 (again just an example).

Inaccuracies

- **Arithmetic overflow:** happens when the result is too large to be represented. The result is unpredictable. It is quite easy to get an arithmetic overflow using integers.
  - Ex: 2000000000+2000000000 (int)

Arithmetic Operators

- Addition (+): 3+4 or 55.1+43.58
- Subtraction (-): 50-20 or 45.3-0.78
- Multiplication (*): 5*10 or 0.6*3.4
- Division (/): 50.0/2.0 or 45/2
- Remainder (%): Also called modulus

Ex: 30%7 is 2, 45%3 is 0, 23%77 is 23.

Important: % works only with integers!

Integer Expressions

- Expressions containing only integers are called integer expressions. The result of an integer expression is always an integer. This is particularly important for the division operator.
- For example, 5/2 is an integer division and will give 2, not 2.5.
- **There is never a rounding up of values. 99/100 will give 0 not 1.**
- Now that we know about integer division, we find that a%b is the same as *a - ((a / b) * b)*.

Double Expressions

- Expressions containing only doubles are called double expressions. The result of an double expression is always a double.
- For example 5.0/2.0 is a double division and will give 2.5.
- 99.0/100.0 will give 0.99.

Mixed Expressions

- Expressions containing doubles and integers are called mixed expressions. The result of a mixed expression is always a **double**.
- For example 5/2.0 or 5.0/2 is a mixed division and will give 2.5.
- 35*2.0 will give 70.0.

Explicit Conversion (Casting)

- The casting (*type*) operator is used to do explicit conversions when necessary. Let's suppose I want to calculate the average of three integer numbers.

```
int a = 4, b = 3, c = 7, sum = 0; /*note the initialization*/
double average; /* need double for average */
sum = a + b + c;
average = sum / 3; /* 4.0 - that is not the correct average! */
```

- The solution is to convert either the sum or 3 into a double to have a mixed expression.
  - average = (double) sum / 3;
  - or
  - average = sum / 3.0;

Multiple Operator Expressions

- What if an expression contains multiple operators?
- What would be the answer to 3.0 + 4.0 / 2.0? 3.5 or 5.0?
- There must be rules to evaluate expressions; otherwise the result is unpredictable.
- How do you evaluate an expression like (a + b) / c + a / c – a + b / c * b ?

Evaluating Expressions

- **Rule #1:** Parentheses rule: All parentheses must be evaluated first from the inside out.
- **Rule #2:** Operator precedence rule:
  - 2.1 Evaluate unary operators first.
  - 2.2 Evaluate *, /, and **%** next.
  - 2.3 Evaluate **+** and – next.
- **Rule #3:** Associativity rule: All binary operators must be evaluated left to right, unary operators right to left.

Unary Operators

- Binary operators are the operators with two operands.
  - Ex: a+b, b-c, b*a, a%b, b/c
- Unary operators are the operators with only one operand.
  - +: the unary plus does **nothing** (+2 is 2).
  - –: the unary minus reverses the sign (–(–2)) is 2, –a reverses the sign of the value of a).

Unary Operators and Memory

- It is very important to note that the unary minus (–) operator does not affect the value of the variable. Only an assignment operator (or a scanf/fscanf) can change the value.

for example:

```
x = -3;
printf ("%d", -x);  /* will display 3 but x is still -3! */
x = -x;  /* now x is 3! */
```

Expression Building

- Let's have an expression to compute the speed of an object.
- Speed is position2 minus position1 divided by time2 minus time1.
- s = (p2 – p1) / (t2 – t1);
- Parentheses can always be used to enhance expression clarity even if they are not necessary.

Expression Evaluation

Let's evaluate the following expression:

z – (a + b / 2) + w * –y

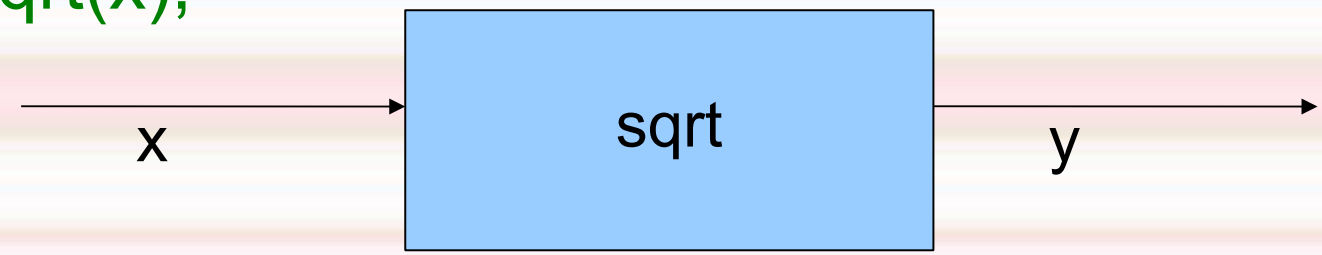
1. The parenthesis is evaluated first:
   Do b/2 first then add a to the result.
2. The unary operator is evaluated next:
   –y is evaluated.
3. Next, –y is multiplied by w.
4. Next, (a+b/2) is subtracted from z.
5. Finally, add the result of step #4 to the result of step #3.

Additional Operators

- Some operations cannot be performed with predefined operators. In that case we need special functions.
- A function is a program unit that carries out an operation.
- A function is a “black box” where only what goes in and comes out is known, not its inside mechanisms.

Square Root

- Square roots in C are computed with a special function taken from a special library: the math library.
- To use that library, we need to include the proper header file: #include <math.h>
- The square root function is called sqrt and is used by calling it this way: **sqrt (x)** where x is the number we wish to know the square root of. We can put that answer in another variable y=sqrt(x);



Math Functions

- Math functions can be integrated in other C statements and expressions. All math functions use doubles.
  - z = a + sqrt (b-c);
  - printf ("The square root of %lf is %lf", x, sqrt(x));

Other Math Functions

- y=floor (x): the largest whole number <= x. If x is 3.7, y will be 3.0. If x is -14.2, y will be -15.0.
- y=ceil (x): the smallest number >= x. If x is 3.7, y will be 4.0. If x is -14.2, y will be -14.0.
- y=log(x): finds the natural log of x (ln).
- y=log10(x): finds the decimal log of x (log).
- y=fabs(x): finds the absolute value of x.

Other Math Functions

- sin(x), cos(x), and tan(x) and are trigonometric functions giving the sine, cosine, and tangent of an angle expressed in radians (not degrees).
- radians = degrees * PI / 180
- y = exp (x): gives **e** to the power of x.
- z = pow (x,y): gives x to the power of y.
- atan(x): calculates the arc tangent of a real number giving an angle expressed in radians.

Other Functions

- Other functions can be found in the standard library (also need to #include <stdlib.h>).
- b=abs(a): gives the absolute value of an integer.
- n = rand(): will give a random integer number between 0 and RAND_MAX, a predefined constant macro. To find the value of RAND_MAX on your computer just try this:

```
printf ("%d", RAND_MAX);
```

Shortcut Operators

- In some books, you will see some operations in a short form when a variable value is changed by an operation on itself.
- x=x*5; may be shortened to x*=5;
- a=a/2; may be shortened to a/=2;
- i=i+1; may be shortened to i+=1;
- Since adding and subtracting 1 is very common, there is a shorter version still.
- i=i+1; may be shortened to ++i;
- i=i-1; may be shortened to - -i;

Increment and Decrement

- ++i is called an increment; --i a decrement.
- i++ and i-- can also be used.
- There is no difference between the prefix ++i and postfix i++ forms as far as the value of i is concerned.
- If an assignment is used, there is a difference. In b=++i; i is incremented and the answer is then placed into b. In b=i++, the value of i is placed in b and then i is incremented.
- Note that it is not recommended to use increment and assignment in the same statement.

End of Lesson