Annotated C Programs: Arithmetic Expressions

Marathon distance conversion: Version 1

1. Consider the following problem statement: Write a program to convert the distance of a marathon race consisting of 26 miles and 385 yards to kilometers using conversion factors of 1.609 kilometers for 1 mile and 1760 yards for 1 mile.
2. The algorithm for converting the marathon distance in miles and yards to kilometers looks like this:

   Algorithm Marathon-Distance-Kms
   Input: Marathon distance of 26 miles and 385 yards
   Output: Marathon distance in $kilometers$

   1. $[1760 \text{ yards are } 1 \text{ mile}]$
   2. $[385 \text{ yards are } 385/1760 \text{ mile}]$
   3. $[\text{Marathon distance is: } 26 + 385/1760 \text{ miles}]$
   4. $[1 \text{ mile is } 1.609 \text{ km}]$
   5. $kilometers = 26 + 385/1760 \times 1.609$

3. The following code in source file `marathon.c` represents the *initial attempt* at a program for the algorithm:

```
1  #include <stdio.h> // for printf prototype
2  
3  /*
4  A marathon race is 26 miles, 385 yards
5  1760 yards make a mile and thus 385 yards is (385/1760) miles
6  A mile is 1.609 kilometers
7  Therefore, a marathon race is 26 miles + 385/1760 miles *1.609 kilometers
8  */
9  
10 int main(void) {
11   double kms;
12 
13   kms = 26 + 385/1760 * 1.609;
14   printf("A marathon race is %f kilometers\n", kms);
15 
16   return 0;
17 }
18 
```

   Annotations for `marathon.c` are provided. Since inclusion of header files using `include` directive and C comments have been previously discussed, the annotations begin with line 11.

4. Line 11 contains the *declaration*, and more specifically, the *definition* of variable `kms`.
   - `double` is a C keyword that represents $64-$bit double-precision floating-point values.

- A *variable* is an identifier associated with a location in memory. The interpretation of the memory associated with a variable by a compiler and ultimately by the machine depends on the variable's *type*. Therefore, for a compiler to correctly interpret a variable, the programmer must make known to the compiler the association between the variable's identifier and the variable's type.
- A *declaration* specifies the interpretation given to an identifier; it doesn't necessarily reserve storage associated with the identifier. A declaration that reserves storage is called a *definition*. A C object such as a function or variable can have many declarations but only one definition. Line 11 is a definition statement that says to the compiler to reserve storage of a $8-$byte chunk of memory for values of type `double` and associate the identifier `kms` with this memory
- Recall that a data type is a set of values and a set of operations on these values. Thus, line 11 is additionally telling the compiler to flag an error if the programmer uses this variable of type `double` to perform actions that are not compatible with its declared type.
- The values of variables that were not given initial values during their definition are *unspecified*. Sometimes these values are called *garbage values* because they're values left behind by the previous occupant either from the same program or from a previous program. Thus, the value of variable `kms` after execution of line 11 is unspecified or garbage.

5. Line 13 looking like this: `kms = 26 + 385/1760 * 1.609;` is an *expression statement* with arithmetic and assignment *expressions* containing arithmetic *operators* and *assignment operator*.

    - The general form of an expression statement is `expr;` where `expr` refers to C syntax that expresses computations.
    - An *expression* is composed of one or more *operands* and zero or more *operators*. *Operator*s represent actions while *operands* represent the objects on which actions are applied.
    - The purpose of an expression is to have the machine evaluate it during the program's execution. Therefore, for each expression in the source code, the compiler will generate machine code that will *evaluate* the expression. Evaluating an expression implies application of operators on operands yielding a *result* which will have a *value* and a *type*.
    - Any evaluation resulting in a non-zero value is considered *true* while an evaluation resulting in a `0` value is considered *false*.
    - The expression on line 13 consists of four operators: `=`, `+`, `/`, `*` and five operands: `kms`, `26`, `385`, `1760`, and `1.609`, and several intermediate operands that are temporarily created in the process of evaluating the larger expression. How will the compiler combine these operators and operands to come up with the appropriate value that must be stored in variable `kms`?
    - Let's begin to unravel the expression on line 13 by understanding the operands in the expression: `kms`, `26`, `385`, `1760`, and `1.609`.
        - In C, operands such as `26` are called *literal constants*: *literal* because we can speak of that operand only in terms of its value and *constan*t because its value cannot be changed in the program. A literal constant is non-addressable; although its value is stored somewhere in memory, programmers have no means of accessing that storage. In addition to its value, every literal constant also has a *type*. For example, literal constants `26`, `385`, and `1760` are of type `int` while literal constant `1.609` is of type `double`.

        - The other type of operand is `kms` which is a variable of type `double`. Variables have been previously discussed.
        - The difference between a constant and a variable is that values of constants cannot be changed by the program while variables (or more specifically their memory locations) can be updated with new values using expressions.
    - Let's now look at arithmetic operators and understand their behavior.
        - Just as in math, symbols `+`, `-`, `/`, and `*` represent binary *addition*, *subtraction*, *division*, and *multiplication operators*, respectively.
        - They are *binary* operators because they require two operands.
        - These operators behave exactly as their arithmetic counterparts in math. The result of evaluating expression `2+12` is value `14` of type `int` since operands `2` and `12` are both of type `int`. The result of evaluating expression `2.0*12.2` is value `24.4` of type `double` since operands `2.0` and `12.2` are both of type `double`.
        - What happens if `+` and `*` operators are used in *mixed expressions*, that is, expressions involving operands with differing types? For example, expression `2+12.4` is mixed because left operand `2` has type `int` while right operand `12.4` has type `double`. The first thing to note is that arithmetic and logic units in CPUs cannot evaluate mixed expressions - they can only perform arithmetic computations when both operands have the same type. That is, ALUs can only add two values of type `int` or multiply two values of type `double` and so on. Rather than flagging mixed expressions as errors, the designers of C decided to have compilers implicitly add code that would silently convert an operand's type to match the other operand's type. This decision leads to the question of choosing which operand should be converted to match the other operand's type? For example, in mixed expression `2+12.4` is operand `2` of type `int` converted to `2.0` of type `double` or is operand `12.4` of type `double` converted to `12` of type `int`? C designers opted for *lossless implicit conversions* compared to *lossy implicit conversions*. The idea of lossless implicit conversion is to *promote* the operand of lower type to higher type so that there is no loss of information and ensuring the expression is evaluated with operands of the same type. For example, if the expression involves an `int` operand and a `double` operand, the `int` operand will be promoted to a `double` before the expression is evaluated, and the result of the evaluation will be a value of type `double`. Thus, mixed expressions `2+12.4` and `12.4+2` will both evaluate to value `14.4` of type `double` by promoting `int` value `2` to higher type `double` with value `2.0`.
        - Division operator `/` behaves differently based on the types of its operands. *Integer division* is performed when both operands are integral values with the quotient being the result of the evaluation while the remainder is discarded. Therefore, expression `3/5` will evaluate to value `0` of type `int`, while expression `16/3` will evaluate to value `5` of type `int`.
        - When either or both operands to `/` operator are floating-point types, *floating-point division* is performed so that the fractional part of the result is retained. If only one operand is of floating-point type, the other is promoted to the same floating-point type. This means mixed expressions `7/5.0` and `7.0/5` and `7.0/5.0` will evaluate to value `1.4` of type `double`.
        - There's more to learn about arithmetic operators. If you want expression `7/5` to be evaluated as in math (to obtain a result of `1.4`), then expression `7/5` must be augmented with the *cast operator*, as in `(double)7/5`. Here, expression `(double)7` *casts* (that is, converts) `int` value `7` to type `double`. Since the

numerator in the expression is cast to type `double`, the compiler will *implicitly promote* the denominator of type `int` to type `double` and evaluate expression `(double)7/5` as division of two `double` operands: `7.0/5.0`. More generally, the cast operator has the form `(type)`.

- There's one final and important detail with arithmetic operators that relates to *precedence* and *associativity*. If an expression consists of zero or one operator, the compiler will *unambiguously* evaluate the expression. For example, expression `4*2` is evaluated unambiguously to a value `8` having type `int`.
- If an expression contains more than one operator and has no parentheses, the possibility exists that the expression can be evaluated *ambiguously*. For example, expression `2+3*4` can be evaluated as either `(2+3)*4` or as `2+(3*4)`. The first evaluation leads to result `20` while the second evaluation leads to result `14`. To ensure unambiguous evaluation of expressions containing more than one operator, C introduces the notions of *operator precedence level* and a *rule of associativity*.
- Where parentheses don't explicitly indicate the grouping of operands with operators, operands are grouped with the operator having *higher* precedence. Binary operators `+` and `-` have the same precedence level while binary operators `*` and `/` also have the same precedence level. However, binary operators `+` and `-` have lower precedence level than binary operators `*` and `/`. Hence, expression `2+3*4` is evaluated as `2+(3*4)` and not as `(2+3)*4` since binary `*` operator has higher precedence than binary `+` operator.
- A *tie-breaking rule* is required if two operators in an expression have the same precedence. In this case, operands are grouped with the left or right operator first according to whether the operators are *left-associative* or *right-associative*. Operators sharing the same precedence level will always have the same associativity. Consider expression `4*5/6` which could be evaluated as `(4*5)/6` (with value `3`) or `4*(5/6)` (with value `0`). Since operators `*` and `/` have the same precedence level, the tie-breaking left-associative rule will come into play (since both `*` and `/` are left-associative operators). This means that in expression `4*5/6`, the left-most operator `*` will have operands `4` and `5` grouped with it, like this `(4*5)/6` (evaluating to value `3`). Thus, expression `4*5/6` will result in value `3` of type `int`.
- Now, for the final part where the complex expression `2+3-4*5/6` must be evaluated. Using precedence levels and associativity of the four operators, the expression will be evaluated as `(2+3) - ((4*5)/6)` to value `2` of type `int`.
- Remember that the compiler will rely on precedence and associativity of operators only if expressions don't explicitly group operands with operators using parentheses. Programmers can break the built-in precedence and associativity rules by using parentheses to group operands with operators. For instance if you had wanted the earlier expression `2+3-4*5/6` to be evaluated differently, then you'd tell the compiler so using explicit parentheses, as in: `(2 + 3 - 4 * 5)/6` which evaluates to value `-2` of type `int`.
- [This]() table lists the the precedence and associativity of C operators.

- The final operator to be studied is operator `=` which is called *assignment operator*. The assignment operator is used to assign a value to a variable. The general form of assignment operator is

$$variable = expression$$

where $expression$ can be a constant, another variable, or the result of an expression. Consider the following code fragment that defines and assigns values to variables `weight` and `sum`:

```c
double weight;
int sum;
/* other code here */
// replace previous value of variable weight with constant 54.72
weight = 54.72;
// replace previous value of variable sum with constant 30
sum = 30;
```

It is important that you don't confuse the assignment operator `=` with algebra symbol $=$ which implies equality. Consider the following assignment expression

```c
sum = sum + 10
```

In algebra, this expression is invalid, because a value cannot be equal to itself plus 10. However, this assignment expression should not be read as an equality; instead it should be read as "`sum` is assigned the value of `sum` plus `10`." With this interpretation, the expression indicates that the value stored in variable `sum` is incremented by `10`. Thus, if the value of `sum` is `20` before this expression is executed, then the value of `sum` will be `30` after the expression is executed.

Multiple assignments are also allowed in C. Suppose variables `x`, `y`, and `z` are declared as type `int`:

```c
x = y = z = 25;
```

The assignment operator is right-associative, therefore the compiler will evaluate it as

```c
(x = (y = (z = 25));
```

The result of the evaluation of expression `(z = 25)` is value `25` of type `int` assigned to `z`. The result of the evaluation of expression `(y = (z = 25))` is value `25` of type `int` assigned to `y` which is then further assigned to `x`.

- Now, you're in a position to evaluate the expression on line 13: `kms = 26 + 385/1760 * 1.609`. Based on previous discussions about mixed expressions, precedence and associativity of operators, the compiler will evaluate this expression as `(kms = (26 + ((385/1760) * 1.609)))`. What is the value resulting from the evaluation of this expression?
  - Begin by looking at subexpression `((385/1760) * 1.609)`. The left operand of `*` operator is `385/1760` which evaluates to value `0` of type `int` while the right operand is `1.609` of type `double`. Since this is a mixed expression, `int` value `0` is promoted to value `0.0` of type `double` resulting in subexpression `(0.0 * 1.609)` which evaluates to value `0.0` of type `double`. Therefore, the subexpression `((385/1760) * 1.609)` evaluates to value `0.0` of type `double`.
  - The subexpression `(26 + ((385/1760) * 1.609))` is a mixed expression with left operand `26` of type `int` and right operand `0.0` of type `double`. After promotion of `26` to value `26.0` with type `double`, the entire expression evaluates to value

`26.0` with type `double`. Therefore, the subexpression `(26 + ((385/1760) * 1.609))` evaluates to value `26.0` of type `double`.
        - Expression `(kms = (26 + ((385/1760) * 1.609)))` will now look like this: `(kms = 26.0)`. Thus, value `0.0` of type `double` will be assigned to variable `kms` of type `double`. The entire expression will evaluate to value `26.0` of type `double`.
    - Line 14 contains a call to the standard library function `printf` with two arguments. The first argument `"A marathon race is %f kilometers\n"` is a character string called *format string* that describes how the second argument `kms` is to be displayed. Characters inside the format string that are not preceded by `%` sign are written literally to standard output. The first argument contains a `%` sign followed by character `f` thereby requiring the second argument `kms` to be displayed using floating-point numbers. The `%` sign and character `f` are referred to as *conversion specifiers*.
    - Compile and link the source file to create an executable file:

```
$ gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror marathon.c -o marathon.out
```

    - Run the executable and compare the result with hand calculations:

```
$ ./marathon.out
A marathon race is 26.000000 kilometers

```

Marathon distance conversion: Version 2

1. The result from running the executable is wrong because expression `kms = 26 + 385/1760 * 1.609` was incorrectly authored. What is required is to convert 385 yards to an appropriate fraction of a mile; add this fraction to 26 to determine miles in a marathon; and multiply the miles with conversion factor 1.609 to convert miles to kilometers.
2. Since operands `385` and `1760` are of type `int`, expression `385/1760` will evaluate to value `0` with type `int`. To obtain the fraction of a mile for 385 yards, either of the two operands must be cast to `double`, as in `(double)385/1760`.
3. Recall that programmers can avoid precedence and associativity rules by grouping operands with operators using parentheses. The expression to add fraction of mile to 26 miles is: `26 + (double)385/1760`. In this mixed expression, `26` is promoted to value `26.0` with type `double`.
4. The result of the previous expression must be multiplied by conversion factor 1.609. The following expression does that: `(26 + (double)385/1760)*1.609`.
5. Finally, the resulting value must be assigned to variable `kms`, as in: `kms = (26 + (double)385/1760)*1.609`.
6. Rather than explicitly using the cast operator, the expression can be simplified by using constants of type `double`, as in: `kms = (26.0 + 385.0/1760.0)*1.609`.
7. The complete program will look like this in source file `marathon-v2.c`:

```c
#include <stdio.h> // for printf prototype

// convert marathon race in miles and yards to kilometers
int main(void) {
  double kms;

  kms = (26.0 + 385.0/1760.0) * 1.609;
  printf("A marathon race is %f kilometers\n", kms);

  return 0;
}

```

8. Compiling and link the source file `marathon-v2.c`:

```
$ gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror marathon-v2.c -o marathon-v2.out
```

9. Running the executable `marathon-v2.out`:

```
$ ./marathon-v2.out
A marathon race is 42.185969 kilometers

```

Marathon distance conversion: Version 3 using `define` preprocessor directive

The previous conversion program uses *hard-coded* constants `26.0`, `385.0`, `1760.0`, and `1.609` to convert marathon distance in miles and yards to kilometers. These values appear as *magic* numbers for other programmers since there is no contextual information associated with them. Hard-coded constants are more difficult to change, especially if they appear many times in a program. For example, if the number of yards in a marathon is determined to be `386` and not `385`, then programmers will have to search for every occurrence of the value `385` with `386`. This is an error-prone process because the value `385` might be used in the program independently for other reasons. Hard-coding values might also show variations in different occurrences. For example, the conversion factor from miles to kilometers could occasionally be written `1.6093` or `1.60934` by accident.

You've seen the use of `include` preprocessing directives for inclusion of header files. The preprocessor provides additional text editing facilities. A *symbolic constant* can be defined with `define` preprocessor directive that assigns an identifier to the constant. The `define` directive can appear anywhere in a C program; the compiler will replace each occurrence of the directive identifier with the constant value in all statements that follow the directive. The idea of symbolic constants through the use of `define` preprocessor directive to create names for constants is illustrated in lines 3 through 6 of `marathon-define.c`:

```c
1 #include <stdio.h> // for printf prototype
2
3 #define YARDS_TO_MILES (1.0/1760.0)
4 #define MILES_TO_KMS   (1.609)
5 #define MILES_IN_MAR   (26.0)
6 #define YARDS_IN_MAR   (385.0)
```

```c

// convert marathon race in miles and yards to kilometers
int main(void) {
  double kms;

  kms = (MILES_IN_MAR + YARDS_IN_MAR*YARDS_TO_MILES) * MILES_TO_KMS;
  printf("A marathon race is %f kilometers\n", kms);

  return 0;
}

```

Compile and link source file `marathon-define.c`:

```
$ gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror
marathon-define.c -o marathon-define.out
```

Run the executable like this:

```
$ ./marathon-define.out
A marathon race is 42.185969 kilometers

```

Constants that arise in science and engineering such as π or g (acceleration due to gravity) are good candidates for symbolic constants. For example, consider the following preprocessor directive

```c
#define PI 3.141593
```

where the directive identifier is `PI` and the symbolic constant is `3.141593`. Expressions that need to use the value of π would then use the identifier `PI` instead of `3.141593`:

```c
area = PI * radius * radius;
```

Wherever directive identifier `PI` appears in the source file, the preprocessor substitutes the identifier with the corresponding constant.

Using symbolic constants have several advantages [from Section 14.3 of text]:

- They make programs easier to read.
- They make programs easier to modify.
- They help avoid inconsistencies and typographical errors.

Marathon distance conversion: Version 4

To maintain simplicity and readability in longer and more complex problem solutions, programs should be developed to use a `main` function and additional functions, instead of using one large `main` function. Breaking a problem solution into a set of functions has many advantages. By separating a solution into a group of functions, each function is easier to understand, and adheres to the basic guidelines of structured programming. Because a function has a specific purpose - think of function `printf` - it can be written and tested separately from the rest of the problem solution. Since an individual function is smaller than the compete solution, testing a function is easier compared to testing the entire problem solution. Also, once a function has been carefully tested, it can be used in new problem solutions without being retested. This reusability is a very important issue in the development of large software systems, because it can save both development time and costs. In fact, this is one of the concepts behind the standard C library.

The advantages of encapsulating algorithms in functions motivate a function `miles_yards_to_kms` that will take distance specified by two parameters `miles` and `yards` - both of type `double` - and return the distance in kilometers. The function prototype is first declared in a file `conversion.h`:

```
/*!
@author pghali
@brief  Computes the distance in kilometers given the distance
        in miles and yards.

This function takes as input a distance measured in miles and
yards and returns that distance in kilometers.

@param m - double-precision floating-point value specifying miles.
@param y - double-precision floating-point value specifying yards.
@return  - a double-precision floating-point value measuring the
input distance in kilometers.
*//*______________________________________________________________*/
double miles_yards_to_kms(double m, double y);

```

Notice that clients can read the function header in `conversion.h` to obtain the information necessary to use the function correctly. The definition of the function in source file `conversion.c` looks like this:

```
#include "conversion.h"

#define YARDS_TO_MILES (1.0/1760.0)
#define MILES_TO_KMS   (1.609)

double miles_yards_to_kms(double miles, double yards) {
  return (miles + yards*YARDS_TO_MILES)*MILES_TO_KMS;
}

```

Source file `conversion.c` is compiled (only) in the usual manner:

```
$ gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror -c conversion.c -o conversion.o
```

It is straightforward to write a `main` function in source file `test-conv.c` to test function `miles_yards_to_kms`:

```c
#include <stdio.h>
#include "conversion.h"

int main(void) {
  double miles = 26.0, yards = 385.0;
  printf("Marathon with %lf miles and %lf yards is %lf kilometers\n",
          miles, yards, miles_yards_to_kms(miles, yards));
  return 0;
}

```

Source file `test-conv.c` is compiled (only) in the usual manner:

```
$ gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror -c test-conv.c -o test-conv.o
```

The two object files are linked (along with C standard library function `printf`):

```
$ gcc test-conv.o conversion.o -o test-conv.out
```

To run executable program `test-conv.out`, type the executable's pathname like this:

```
$ ./test-conv.out
Marathon with 26.000000 miles and 385.000000 yards is 42.185969 kilometers

```

Things to review

1. What is an *identifier*? What is the legal way to write an identifier?
2. What is a *data type* in the context of a programming language?
3. What is a *variable*?
4. What is a *literal value*? How does it differ from a variable?
5. What is an *operator*? An *operand*? An *expression*?
6. What does it mean to *evaluate* an expression? What is the result of an expression's evaluation?
7. What is a *statement*?
8. What is a *binary operator*? List the binary operators in this tutorial.
9. What is the *cast* `(type)` operator?
10. What is the *precedence* level of an operator? What is the rule of *associativity* of operators?
11. What is the result of (that is, the value and the value's type obtained by) the evaluation of expression `2+3*4` ?
12. What is the result of the evaluation of expression `2+3/4`?
13. What is the result of the evaluation of expression `2*3/4`?