Lab # 1

Due midnight on Sept. 13, 2013

Collaboration is encouraged. You may discuss the problems with other students, but you must **write up your own solutions**, *including all your C programs, by yourself. If you submit identical or nearly identical solutions to someone else, this will be considered a violation of the code on academic honesty.*

Submit your assignment by email to your TA as a zipped collection of files called `123456789.zip`, where `123456789` is your student ID. This zipped collection must include:

- A file called `goodbye-cruel-world.c`
- A file called `miles-feet-inches.c`
- A file called `factorial.c`
- A file called `perm.c`

1. **Your first C program**.

   As tradition would have it, everyone's very first program is one that prints the greeting "hello, world."

   ```
   # include <stdio.h>
   int main()
   {
      printf("hello, world\n");
   }
   ```

   Please see the reference document on the class webpage on how to compile and run this program.

   A C program, whatever its size, consists of *functions* and *variables*. A function contains *statements* that specify the computing operations to be done, and the variables store the values used during the computation. Normally, you are liberty to give functions whatever names you like, but "`main`" is special – your program begins executing at the beginning of main. This means that every program must have a `main` somewhere.

   The first line of the program

```
#include <stdio.h>
```

tells the compiler to include information about the standard input/output library. This includes the `printf` statement that is use to print out your polite greeting to the world.

Problem: Write a program that prints out "goodbye cruel world". Call this program `goodbye-cruel-world.c`.

2. **A C program that does conversion.**.

The next program uses the formula $°C = (5/9)(°F - 32)$ to print the following table of Fahrenheit temperatures and their centigrade equivalents:

```
#include <stdio.h>
/*
 * print Fahrenheit-Celsius table for fahr = 0, 20, ..., 300;
 */
main()
{
      float fahr, celsius;
      int lower, upper, step;
      lower = 0;     /* lower limit of temperatuire scale */
      upper = 300;   /* upper limit */
      step = 20;     /* step size */
      fahr = lower;
      while (fahr <= upper) {
            celsius = (5.0 / 9.0) * (fahr - 32.0);
            printf("%3.0f %6.1f\n", fahr, celsius);
            fahr = fahr + step;
      }
}
```

This program introduces several new ideas, including comments, declarations, variables, arithmetic expressions, loops, and formatted output.

Everything that appears between the symbols `/*` and `*/` consists of comments. These are ignored by the compiler, but they are a crucial part of programming. Comments are how people (including the author when looking back at his or her own code later) make sense of what's written. We'll emphasize proper commenting throughout the course.

All variables must be *declared* before they are used, usually at the beginning of the function before any executable statements. A declaration announces the properties of variables; it consists of a name and a list of variables, such as

```
float fahr, celsius;
int lower, upper, step;
```

The type `int` means that the variables listed are integers; by contrast with `float`, which means floating point, i.e., numbers that may have a fractional part. C provides several other data types besides `int` and `float`, including:

- `char` character - a single byte
- `short` short integer
- `long` long integer
- `double` double-precision floating point

Computation in the temperature conversion program begins with the assignment statements

```
lower = 0;
upper = 300;
step = 20;
```

which set the variables to their initial values. Individual statements are terminated by semicolons. Each line of the table is computed the same way, so we use a loop that repeats once per output line; this is the purpose of the while loop

```
while (fahr <= upper) {
...
}
```

The while loop operates as follows: The condition in parentheses is tested. If it is true (`fahr` is less than or equal to `upper`), the body of the loop (the three statements enclosed in braces) is executed. Then the condition is re-tested, and if true, the body is executed again. When the test becomes false (`fahr` exceeds `upper`) the loop ends, and execution continues at the statement that follows the loop. There are no further statements in this program, so it terminates.

Most of the work gets done in the body of the loop. The celsius temperature is computed and assigned to the variable celsius by the statement

```
celsius = (5.0 / 9.0) * (fahr - 32.0);
```

The `printf` conversion specification `%3.0f` says that a floating-point number (here `fahr`) is to be printed at least three characters wide, with no decimal point and no fraction digits. `%6.1f` describes another number (`celsius`) that is to be printed at least six characters wide, with 1 digit after the decimal point.

There are plenty of different ways to write a program for a particular task. Let's try a variation on the temperature converter.

```
# include <stdio.h>

/* print Fahrenheit-Celsius table */
main()
{
      int fahr;
      for (fahr = 0; fahr <= 300; fahr = fahr + 20) {
            printf("%d %f\n", fahr, (5.0 / 9.0) * (fahr - 32));
      }
}
```

This produces the same answers, but it certainly looks different. One major change is the elimination of most of the variables; only `fahr` remains, and we have made it an `int`. The lower and upper limits and the step size appear only as constants in the for statement, itself a new construction, and the expression that computes the celsius temperature now appears as the third argument of `printf` instead of a separate assignment statement. This last change is an instance of a general rule – in any context where it is permissible to use the value of some type, you can use a more complicated expression of that type. Since the third argument of `printf` must be a floating-point value to match the `%6.1f`, any floating-point expression can occur here.

The `for` statement is a loop, a generalization of the while. If you compare it to the earlier `while`, its operation should be clear. Within the parentheses, there are three parts, separated by semicolons. The first part, the initialization

```
fahr = 0
```

is done once, before the loop proper is entered. The second part is the test or condition that controls the loop:

```
fahr <= 300
```

This condition is evaluated; if it is true, the body of the loop (here a single `printf`) is executed. Then the increment step

```
fahr = fahr + 20
```

is executed, and the condition re-evaluated. The loop terminates if the condition has become false. As with the `while`, the body of the loop can be a single statement or a group of statements enclosed in braces. The initialization, condition and increment can be any expressions.

The choice between `while` and `for` is arbitrary, based on which seems clearer. The `for` is usually appropriate for loops in which the initialization and increment are single statements and logically related, since it is more compact than while and it keeps the loop control statements together in one place.

Problem: Write a program that prints out lengths in miles, feet and inches from 1 to 100 miles. (You crazy Americans with your Imperial system of measures. Don't even get me started on ounces per pound and fluid ounces per gallon...) Call this program `miles-feet-inches.c`. Here's the first line that it should print out:

```
1 5280 63360
```

3. **A C program that does some math.**

   A function provides a convenient way to encapsulate some computation, which can then be used without worrying about its implementation. With properly designed functions, it is possible to ignore how a job is done; knowing what is done is sufficient. C makes the use of functions easy, convenient and efficient; you will often see a short function defined and called only once, just because it clarifies some piece of code.

   Earlier we used the function `printf` that has been provided for us; now it's time to write a few of our own. Since C has no exponentiation operator, let us illustrate the mechanics of function definition by writing a function `power(m,n)` to raise an integer `m` to a positive integer power `n`. That is, the value of `power(2,5)` is 32. This function is not a practical exponentiation routine, since it handles only positive powers of small integers, but it's good enough for illustration. Here is the function `power` and a `main` program to exercise it, so you can see the whole structure at once.

```c
#include <stdio.h>

int power(int m, int n);

/* test power function */
main()
{
      int  i;
      for (i = 0; i < 10; ++i)
            printf("%d %d %d\n", i, power(2, i), power(-3, i));
      return 0;
}

/* power: raise base to n-th power; n >= 0 */
int
power(int base, int n)
{
      int i, p;
      p = 1;
      for (i = 1; i <= n; ++i)
            p = p * base;
```

```
    return p;
}
```

A function definition has this form:

```
return-type function-name(parameter declarations)
{
   declarations
   statements
}
```

Function definitions can appear in any order, and in one source file or several, although no function can be split between files. If the source program appears in several files, you may have to say more to compile and load it than if it all appears in one, but that is an operating system matter, not a language attribute. For the moment, we will assume that both functions are in the same file, so whatever you have learned about running C programs will still work. The function `power` is called twice by `main`, in the line

```
printf("%d %d %d\n", i, power(2,i), power(-3,i));
```

Each call passes two arguments to `power`, which each time returns an integer to be formatted and printed. In an expression, `power(2,i)` is an integer just as 2 and `i` are. (Not all functions produce an integer value.)

The first line of `power` itself,

```
int power(int base, int n)
```

declares the parameter types and names, and the type of the result that the function returns. The names used by `power` for its parameters are local to `power`, and are not visible to any other function: other routines can use the same names without conflict. This is also true of the variables `i` and `p`: the `i` in power is unrelated to the `i` in main.

We will generally use *parameter* for a variable named in the parenthesized list in a function. The value that power computes is returned to `main` by the `return:` statement. Any expression may follow return.

The declaration

```
int power(int base, int n);
```

just before `main` says that `power` is a function that expects two int arguments and returns an int. This declaration, which is called a *function prototype*, has to agree with the definition and uses of power.

Problem: Write a program that prints out `n!` (`n` factorial) for values of `n` from 1 to 10. Call this program `factorial.c`. Here are the first few lines that it should print out:

```
1
2
6
24
```

4. **A C program that does some more math.**

   Modularity is one of the most important concepts in programming. We can define functions and then call them repeatedly, when needed. In Problem 3, you wrote a function that computes `n!`:

```
int factorial(int n)
```

   Here's a program that computes the number of **combinations**: the number of ways one can choose `k` objects from among a given set of `n` objects. It uses the function that you wrote for Problem 3 (which, obviously, we're omitting here).

```
#include <stdio.h>

int factorial(int n);

/* combinations */
main()
{
      int  i, j;
      for (i = 1; i <= 10; ++i) {
         for (j = 1; j <= i; ++j) {
           printf("%d %d %d\n", i, j,
                  factorial(i)/(factorial(j) * factorial(i - j)));
         }
      }
      return 0;
}

/* factorial function */
int factorial(int n)
{
   /* include it here */
}
```

   Notice here that we have two `for` loops, with the inner one *nested* inside the outer one. The upper bound on the inner loop depends on the index variable from the outer loop. Here are the first few lines that this program prints out (`n` in the first column, `k` in the second column, the answer in the third column):

```
1  1  1
2  1  2
2  2  1
3  1  3
3  2  3
3  3  1
4  1  4
4  2  6
4  3  4
4  4  1
5  1  5
5  2  10
5  3  10
5  4  5
5  5  1
```

Problem: Write a program that prints out the number of **permutations**: the number of ways `r` objects can be chosen from a total of `n` distinct objects and arranged in different ways. Call this program `perm.c`. Here are the the first few lines that it should print out (`n` in the first column, `k` in the second column, the answer in the third column):

```
1  1  1
2  1  2
2  2  2
3  1  3
3  2  6
3  3  6
4  1  4
4  2  12
4  3  24
4  4  24
5  1  5
5  2  20
5  3  60
5  4  120
5  5  120
```