NAME:**____________________________________**

CS211
Lab02
FA2022
streller/dutta

PROBLEM STATEMENT:

C SIMPLE I/O AND EXPRESSIONS

CODE :

background:

When dealing with hardware (nics) and OSs', the C language is the most used. Thus, we must gain some experience with that language. While we will not explore the whole of C, we do need to gain some familiarity with the language.

The C language itself does not provide any input or output statements. You can write your own I/O functions or use the 'standard library". This library is implemented by functions that are provided with each C compiler implementation.

The standard input and output files for a program consist of three files: stdin, stdout, and stderr. The input to a C program is assumed to be a stream of data from stdin, the standard input device. Stdout, the standard output device is a stream of data output from a C program. The default device for stdin is the terminal keyboard, while stdout is defaulted to the terminal screen.

The standard error file is stderr. This file is always mapped to the terminal screen. C programmers use this file to report errors in a program. You must however, write messages specifically to this file.

The standard C functions, scanf and printf, perform formatted I/O. Scanf is used for input. It translates character representations of numbers to an internal binary format. Printf is used for output. It translates binary representations of numbers to an external character format.

printf and scanf use a format string. They also use arguments, for printf what is to be printed, for scanf where the data is to be saved. The size of each location is determined by the format string conversion specification used.

The conversion specifications used by scanf for input are almost exactly the same as those used in the printf function. There is one minor exception. The printf function uses the conversion characters, %f, for a double precision (double) or for a floating point (float) value. printf does not need to distinguish between the two because both type arguments look the same when printf sees them. However, scanf must know that it is reading a double precision value so that it stores the correct size data. Therefore, the conversion specification for scanf must be %lf (long float) for double precision variables.

Data Type	Conversion Specification
int	%d
long	%ld
char	%c
float	%f
double	%lf for scanf, %f for printf

Now take a look at a program that calculates the distance traveled using input entered through scanf.

```
//*************************************************************
//
// File Name: lab2a.c
//
// A Program to calculate the distance traveled given the
// speed (mph) and time (hr)
//
// Programmer: “Your Name Goes Here”
//
// Date Written: today’s date
//
// Date last revised: “place the date of last revision here”
//
//*************************************************************


#include <stdio.h>

int main()
 {
   int time,
       mph;

   // INPUT SECTION
   printf(”Enter your speed in miles per hour: “);
   scanf(“%d”,&mph);

   printf(”Enter the number of hours: “);
   scanf(“%d”,&time);

  // CALCULATION AND OUTPUT SECTION
   printf(”Miles traveled = %d \n”, mph * time);

  return 0;
 }
```

It is important to notice that scanf’s arguments are preceded by the address-of operator, &. The purpose of scanf is to store input values into variables, it must be passed the locations of the variables.

To fully understand why the address-of operator is required, you need to be aware that there are two ways of passing arguments to a C function: pass by value and pass by reference. When functions are called in C, the variables are passed to the called function by value (pass by value). The function only knows the value of the variable and not the location where the variable is stored. This is fine if you do not want to change the value of the variable, but what happens if you do, like for scanf? By passing the address of the variable (using the & operator) to scanf, you can change the value of the variable. This is known as passing the variable by reference. Scanf does the conversion of the input to its binary representation and then stores the value at the address of the appropriate variable.

In the program, if by accident you leave off the & before the variables you will have a problem. scanf will be passed the value of mph, not its location. scanf will then attempt to store the input, using the value of mph as a memory address. Consequently, an area of memory, possibly containing the operating system or the executable code of your program, will be destroyed. Using scanf to input from the terminal increases the program’s flexibility.

Now for this lab:

Create a directory called cs211_lab2 in your home directory to put the program files created for this lab. Change to the directory cs211_lab2. Obtain the source file cs211_lab2a.c for the above program; rename it to lab2a.c. If you have theses files on flash drive, see cs211 Notes section on how to mount and copy the files to your home (cs211_lab2a.c is included with this lab)

compile and link it using the following command in the shell:

```
gcc –o lab2a lab2a.c
```

run the program by entering the following in the shell

```
./lab2a
```

run the program lab2a and test the following values.
mph = 40, time = 5; miles traveled = ______
mph = 35, time = 4; miles traveled = ______
mph = 65, time = 6; miles traveled = ______

note:
the above commands
gcc calls the C compiler/linker
-o is a switch (optional) which specifies the name of the executable output file (here named lab2a)
the last argument is always the name of the source code file (lab2a.c)
To run the executable, one just enters ./executable file name (that's dot forward slash executable file and assuming one is living in the executable's directory).

Now for the program lab2b.c (below), take a close look at the way this program gets its input. Before each scanf function call, there is a function call to printf. This call outputs a prompting message on the terminal so that you know the program is waiting for input. You also know by these messages, the appropriate data to input. By using scanf for input, this program will not have to be recompiled every time new data is tried.

A conversion specification, %6.2f, is used in this example. This prints floating point numbers in a field of at least 6 digits in width with 2 digits to the right of the decimal point. If the argument has fewer characters than the field width then the number is padded on the left with blanks.

```
//**************************************************************
//
// File Name: cs211_Lab2b.c
//
// A Program to calculate the gross profit, net profit
//                          and sales commission on an item
//
// Programmer: “Your Name Goes Here”
//
// Date Written: today’s date
//
// Date last revised: “place the date of last revision here”
//
//**************************************************************

#include <stdio.h>

int main()
{
  long total, stock, sales;
  double price, commission, cost, income, profit;
  char type;
```

```
 /* input values to be used */
 printf("\n\n Please enter the following items ");
 printf( "to calculate \n gross profit,\n net profit,");
 printf("\n and sales commission on a item. \n\n\n");
 printf(" Enter the type of item (1 char.): ");
 scanf( "%c",&type);
 printf("\n Enter the number of items sold: ");
 scanf("%ld", &sales);   /* items sold */

 printf("\n Enter the number of items on hand");
 printf(" at the beginning of the month: ");
 scanf("%ld", &stock);   /* items on hand */

 printf("\n Enter the current selling price: ");
 scanf("%lf", &price);  /* selling price */

 printf("\n Enter the cost of the item: ");
 scanf("%lf", &cost);    /* cost of item */

 /* calculate values */

 total  =  stock -  sales;
 income =  price * sales;
 profit =  (price -  cost) *  sales;
 commission =  profit *  0.03;

 /* print answers */
 printf("\n\n\n\n End of month total inventory =  %ld of type %c\n", total, type);
 printf("\n On a gross income of  ");
 printf("%6.2f profit = %6.2f\n",income, profit);
 printf("\n Commissions at a rate of 0.03 = %6.2f\n", commission);
 printf("\n\n");

 return 0;
}
```

This file is also included herein. Now compile and link it using the following command:

> ____(fill in the blank)

run the program created and test the following values

> type of item = ‘m’;
> \# items sold = 1000;
> \# items at the beginning of the month = 2500
> selling price = 1.73;
> cost of item = 0.35;
> end of month inventory = ____;
> gross income = ____;
> profit = ____;
> commissions = ____

Another feature of the C language that supports easy changing of program data is the symbolic constant. Symbolic constants (usually referred to as *macros*), are defined using a compiler control named #define. A symbolic constant is simply a name used to reference a string of characters. Every place the name occurs, the string of characters is substituted.

The #define is followed by at least one blank, the name of the constant, then at least one blank, and the string of characters. Upper case names are usually chosen for symbolic constants as a way to alert you that they are not the same as variables. Also notice that the definition of a symbolic constant does not need a terminating semicolon. Let's take a look at a few definitions.

```
#define PI 3 .14159
#define MAX pp
#define RSQUARED radius * radius
#define FORMULA PI * RSQUARED
```

You can use a symbolic constant anytime after its definition. When the constant's name is encountered, the compiler substitutes the associated string of characters. This is sometimes referred to as macro expansion. Here are some examples using the above definitions. The expanded code shows what the compiler actually processes.

source code	expanded code
dim = radius * PI;	dim = radius * 3.14159;
cc = 5 * aa + MAX,.	cc = 5 * aa + pp;
printf("MAX = ",MAX);	printf("MAX = ",pp);
FORMULA;	3.14159 * radius * radius;

Did you notice that the characters MAX inside the string constant did not get expanded as a symbolic constant? Also take a look at the nested symbolic constant definitions of FORMULA and RSQUARED. First, the compiler substitutes PI * RSQUARED for FORMULA. It then examines
the text again and determines that PI is a symbolic constant also. It then expands the code to 3.14159 * RSQUARED. When the second constant is encountered the code is expanded to 3.14159 * radius * radius;.

A symbolic constant (or macro) may also be defined with one or more string arguments. Arguments provide the means of specifying the contents of the string each time the macro is used. An argument is a name that appears twice in the macro definition. First, it appears inside parentheses immediately following the macro name. Second, it appears somewhere inside the string of characters that define the macro. The argument is simply a name that can assume any string value. The string value is passed to the macro each time the macro is used. Therefore a single macro with an argument can expand into many different strings of characters. The result of the expansion depends on the value of the argument. The following example illustrates the use of macros.

```
//*************************************************************
//
// File Name: cs211_Lab2c.c
//
// A Program to calculate the cost, tax and total cost of
//          purchasing several items with a 6.125% sales tax
//
// Programmer: "Your Name Goes Here"
//
// Date Written: today's date
//
// Date last revised: "place the date of last revision here"
//
//*************************************************************


#include <stdio.h>

#define TAXRATE 0.06125
#define READ_LONG(longvar) scanf( "%ld", &longvar)
```

```
        #define READ_DOUBLE(doubleVar) scanf("%lf", &doubleVar)

        int main()
         {

                long number;
                double price, cost, tax;

                printf("\nEnter number of units: ");
                READ_LONG(number);

                printf("Enter price per unit: ");
                READ_DOUBLE(price);

                printf("\nCost = %8.2f\n", cost = price * number);
                printf("Tax = %8.2f\n", tax = cost * TAXRATE);
                printf("Total = %8.2f\n", cost + tax);
                printf("\n");

                return 0;
         }
```

The name TAXRATE is used in the program to define a floating point constant value. It is good practice to use symbolic constants rather than actual values. The actual value of a symbolic constant is defined in only one place even though it might be used in many places. If the value needs to be changed at a later time, you will only have to modify one line in the program.

Notice that the READ LONG and READ_FLOAT macros each have one argument. The compiler expands the statement READ LONG(number) into scanf("%ld", &number). The string number is substituted for the argument longvar in the READ_LONG macro definition. The compiler expands the macro READ__FLOAT(price) into scanf("%f", &price). The string price is substituted for the argument floatvar in the READ__FLOAT macro definition. Macros with arguments are useful for defining new ways of expressing C statements.

Also notice that the variables cost and tax are both assigned values inside the calls to the printf function. In C, the assignment statement may be used anywhere that an expression is legal.

Obtain cs211_Lab2c.c) compile and link it:

____(fill in the blank with the required command)

run the program created and test the following values (save a copy of your source)

3 of units = 67; price = 13.31; cost = ____; tax = ____; total = ____

You have been introduced to a few language features that can be used as tools to build useful C functions. Before you can effectively utilize these and the C operators you know about however, you must understand more about C expressions. For instance, what happens when you multiply an integer number times a floating point number? Or, in what order are the operators applied in this expression, 3 + 3 * 5 (i.e. is the answer 18 or 30?)

Now let's look at why these are problems. When several operators are in one expression, some of the operators are acted on before others. In the expression 3+3*5 there are two operators, * and +. The arithmetic operators * (multiplication) and / (division) are higher in precedence than + (addition) or - (subtraction). This means that multiplication and division are performed before addition and subtraction. Therefore, the expression would be evaluated as three times five (fifteen), plus three, resulting in a value of eighteen. The rule determining the order of evaluation is the operator with the highest precedence is evaluated first.

If you have the expression 3+2-4+6, all of these operations are at the same precedence level. In this case, another rule is applied to the order of evaluation. This is known as associativity (grouping). All the arithmetic operators are binary (involving two operands) and group left to right. Therefore, 3+2-4+6 is evaluated as three plus two which is five, subtract four which yields one, plus six yields seven. Let's look at another example of associativity, a=b=c=2. How is this expression grouped? The assignment operator groups right to left so it is evaluated as: a=(b=(c=2)), the variable c gets the value 2, then b gets the value of c (2), then a gets the value of b (2).

When expressions are evaluated, the precedence rule is applied first. Then the grouping rule is applied if there are operators of an equal precedence level in the expression. If you want to force a calculation to occur in a manner that does not fit the precedence or associativity rules, you can use parentheses. They are used in the same manner that parentheses are used in algebra: to force evaluation of the expression inside the innermost pair of parentheses, followed by the next innermost pair of parentheses, etc. Try to work these examples of C expressions and see if you understand both the above rules:

C expression	Equivalent to	Answer
a=b=c=2*3-6*2	a=(b=(c=(2*3)-(6*2)))	a=-6,b=-6,c=-6
24%5+7*4/7	(24%5)+((7*4)/7)	8
(3+2)*4%(6+3)	((3+2)*4)%(6+3)	2

The C language has quite a large number of operators. It also has 15 different levels of precedence.

You now can figure out what happens with operator precedence, but what happens when you execute the following that has a mixed type expression?

Example (cs211_Lab2d.c):

```
#include <stdio.h>

int main()
{

      int a;
      a =  3.4 * 5;
      printf("a = %d \n",a);

 return 0;
}
```

Is the value of the variable a, seventeen or fifteen? In other words, was the multiplication performed as integer or floating point? To learn what happens, let's look at conversion of operands in C expressions.

When two operands in a binary operation are of different types, an automatic type conversion will occur. To understand the rules of this conversion process, you must understand that C types have a specific ordering. The lowest type is char and the highest type is double. The following lists lowest type to highest type in a left to right order:

char < int < unsigned < long < float < double

The types of the two operands are compared and the lower type operand is converted to the higher type. The type of the result is the same as the higher type operand. In particular, the following conversion rules are applied in the order listed:

I. Any operand of char is converted to int.
2. Any operand of float is converted to double.
3. If either operand is double, then convert the other to double.
4. If either operand is long, then convert the other to long.
5. If either operand is unsigned, then convert the other to unsigned.

Obtain cs211_Lab2d.c and determine the value of a. a = __________
(save a copy of your source)

Another kind of type conversion takes place when the assignment operator is used. No matter what type expression appears on the right hand side of the assignment, the value is converted to the type of what appears on the left hand side. Conversion of float to int truncates the fractional part of the number and long to int drops the excess high order bits. Integer to character drops excess high order bits also. Conversions such as these can lead to unexpected results unless you are aware that they are occurring. The following example shows some of the conversions that can take place.

Example (you save as cs211_Lab2e.c)

```
        #include <stdio.h>

        int main()
          {
                char c1, c2, c3;
                short i1, i2, i3;
                double f1, f2, f3;
                int d1;

                /* char converts to int */

                c1 = ‘c’;
                i1          = c1 - ‘a’ + ‘A’;
                c3 = i1;   /* truncate to character */
                printf(”c1 = %c, i1 = %d, c3 = %c\n”, c1,i1,c3);
                i1          =  321;
                c2 = i1;   /* convert short to char */
                c3 = i1 + 1;            /* truncates value */
                 printf(”i1 = %d, c2 = %c, c3 = %c\n”, i1 ,c2,c3);

                /* automatic conversion from int to double */

                f1 = 200;               /* converted to float */
                f2 = 350 * f1;  /* 350 converted to double */

                /* 7 converted to double; result truncated */
                i3 = 3.4 * 7;

                /* 350 converted to double;  result truncated */
                 i1         = f3 = f1 / 350;
                printf(”f1 = %f, f2 = %f,\n”,f1,f2);
                printf(”i3 = %d, f3 = %f, i1 = %d\n”, i3,f3,i1);

                /* values produced in the following
                assume a 16 bit short */
                d1 = 69631;             /* In hex 10FFF */
                i2 =d1;    /* truncates to 1000 */
                printf(”i2 = %d\n”, i2);

        return 0;
        }
```

Determine the output of the program:

c1 = _________, i1 = _________, c3 = __________
i1 = _________, c2 = _________, c3 = __________
f1 = ___________, f2 = ______________
i3 = ___________, f3 = _____________, i1 = ____________
i2 = _________________

Take a look at this output carefully. In the statement,

```
        i1          = c1 - 'a' + 'A';
```

the two character constants, ‘a’ and ‘A’ are converted to short before the calculation is performed. You might notice this expression actually is converting the lowercase letter ‘c’ to uppercase ‘C’.

The next part of the program,

```
         i1 = 321;
        c2 = i1;   /* convert short to char */
        c3 = i1 + 1;          /* truncates value */
```

shows a short that is truncated to a character. Exactly which letters are printed as a result is not really important. This example shows that such conversion is possible. Usually, this occurs as a programming accident.

The next part of the example,

```
        f1 = 200;                       /*200      converted to double*/
        f2 = 350 * f1;                  / *350     converted to double*/
        i3 = 3.4 * 7;                   /*7 converted to double, result truncated*/
        i1 = f3 = f1 / 350;             /*350      converted to double, result truncated*/
```

shows some common floating point conversions. Notice that the expression 3.4 * 7 is calculated in double floating point arithmetic and then truncated when the result is assigned to the short i3. Also note in the next expression, the constant 350 is converted to floating point. The variable f3 is assigned the floating point value(0.571429), but when the value is stored in i1, the floating point number is truncated to integer   (0).

The conversion of int to short is shown in the following code:

```
        d1 = 69631;          /* In hex 10FFF */
        i2 = d1;   /* truncates to 1000 */
```

The exact representation in bits of this example is machine dependent. If a short is 16 bits and an integer is 32 bits in length, then the variable d1 which has the value, 69631, expressed in bits is, 00000000000000010000111111111111. When the value of d1 is converted to a short, the high order bits are lost. The value of i2 becomes, 0001000000000000(due to round even truncation), or 4096. Notice that this truncation of bits can drastically change the value of a variable!

This example just illustrates some of the rules we've discussed earlier. What is important to learn from it is that C may do some conversions automatically during calculations, sometimes making the results not what you intended.

The assignment and arithmetic operators that you've seen previously have some shorthand versions. These are known as assignment operators and they combine one arithmetic operator and the assignment operator, such as +=. For example, x += 3 is the same as x = x + (3). There are other operators that can be used in this manner, but the following are the ones you've seen at this time:

Expression	Equivalent
x += 4;	x = x + (4);
y -= 1;	y = y - (1);
z *= -4;	z = z * (-4);
w /= 23;	w = w / (23);
x %= 2;	x = x % (2);

The increment and decrement operators also require that their operand be a variable and not a constant. The increment operator is used to add one to the operand, whereas the decrement operator subtracts one. Both are unary operators and therefore require only one operand. For example, a++, is equivalent to a = a+1 and a-- is equivalent to a = a-1.

In the statement, bb = aa++;, the value of aa is incremented by one after assigning the current value of aa to the variable bb. In this case ++ is a postfix operator, it increments the value of the variable aa after assigning the current value of the variable aa to the variable bb.

The increment and decrement operators can also be written prior to the operand. This is known as a prefix operator, the value of aa in this statement, bb = --aa;, is decremented before the assignment to the variable bb. The precedence of the increment and decrement operators are higher than anything you've seen before except the parentheses. For example:

The statement:	The statement:
b = c + a++;	b = c + --a;
is equivalent to:	is equivalent to:
b = c + a;	a = a – 1;
a = a+1;	b = c + a;

This next example will show some assignment operators as well as increment and decrement operators.

Example (cs211_Lab2f.c)

```
You place comment header here.
#include <stdio.h>

int main()
{
    int xx,ii, yy, jj;

    jj = ii = 0;

    xx      = jj++ + ++jj;
   printf("xx = %d, jj = %d, ii = %d\n",xx,jj,ii);

   yy = jj *= xx++ + ++ii + 3;
   printf("yy = %d, jj = %d, xx = %d, ii = %d\n",yy,jj,xx,ii);

   ii = ++ii;
   printf("ii = %d\n",ii);

   printf("1st --yy = %d, 2nd --yy = %d", --yy, --yy);

return 0;
}
```

Determine the output from the program: (save a copy of your source as lab2f.c)

```
xx= ______, jj = ______, ii = _______
yy = ______, jj = ______, xx = ______, ii = _______
ii = ________
1st --yy = ______, 2nd --yy = _________
```

Take time to analyze the output from this example. Every expression shown has a side effect. That is, the increment/decrement operators change the value of a variable during evaluation of the expression. But when is the value changed? Let's examine the statement:

```
xx = jj++ + ++jj;
```

The increment operators have the highest precedence and will therefore be evaluated first. The left operand of the addition (+) is first evaluated. The result is the current value of jj, which is zero. But then jj is incremented to the value one. The right operand of + is then evaluated. The variable jj, currently equal to one, is incremented to two. This is the result of the right operand. The result of the left operand (0) is then added to the result of the right operand (2). The value two is then assigned to xx. After this statement has finished, jj has the value two and xx has the value two. Also note the following expression:

```
yy = jj  *= xx++ + ++ii + 3;
```

has five plus signs in a row. You must separate the symbols by blanks to make clear that you desire a post-incremented xx added to a preincremented ii. The compiler will try to take the longest string of characters that will form an operator. So, if you wrote +++++, the compiler would see ++, then another ++, followed by a + which is not the desired set of operations.

Turn in the filled in lab (***STAPLED***, of course) along with your source files copied to a directory called Lab2 in your home directory.

Deliverables:

A printout of ANSSHEET, with all questions answers. Also you must have a lab2 directory in your home folder for grading with all the files

cs211_Lab2a.c cs211_Lab2c.c cs211_Lab2e.c
cs211_Lab2b.c cs211_Lab2d.c cs211_Lab2f.c

DUE DATE : 11:00AM 27 September 2022 in class.