Experiment # 1:
Implementation of PRINT statement in C

Some Useful Operations	Keyboard Shortcuts
Copy text	Alt + w
Paste text	Ctrl + y
Cut text	Ctrl + w
Undo	Ctrl + /
Open multiple emacs files	Ctrl + x then Ctrl + f
Close a file in emacs	Ctrl + x then Ctrl + c
Go to the terminal without closing the emacs file	Ctrl + z
From the terminal, go back to the already opened emacs file	Type fg and press enter
Pressed a prefix key (e.g C-x) or invoked a command (e.g. Find File)	Hold Ctrl and then press g repeatedly

1. Eliminate all the syntax errors in the code to get the output as shown below. Note: The name of the code file is e1t1.c

```
#include <studio.h>

int main(void)

{
  print(Welcome to Programming Fundamentals\n)
  print(Nice to meet you.\n)
  return
}
```

Output

Prepared By: Rehan Naeem and Dr. Suleman Sami Qazi

```
root@DESKTOP-9OVP93E: ~
root@DESKTOP-9OVP93E:~# gcc -Wall -Wstrict-prototypes -ansi -pedantic e1t1.c -o e1t1
root@DESKTOP-9OVP93E:~# ./e1t1
Welcome to Programming Fundamentals
Nice to meet you.
```

2. Complete the following code to get the output as shown below. Don't add extra printf statements.

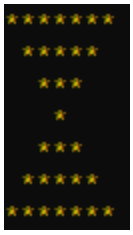
```c
#include <stdio.h>

int main(void)

{
  printf("Part a:");
  printf();
  printf();
  printf();

  printf("Part b:");

  printf("Part c:");
  printf();
  printf();
  printf();

  printf("Part d:");

  return 0;
}
```

Output

```
Part a:
1
2
3
Part b:
1
2
3
Part c:123
Part d:123
```

3. Use multiple printf statements to print the hourglass shown below.

Output



4. Use multiple printf statements to get the output as shown below. Pay special attention to the alignment and position of each text and each hourglass at the output. The integers "1234….0123" will help you to align and position the texts and the hourglasses properly. Also pay attention to the number of rows and the number of asterisks (*) in each row of each hourglass.

Output

Experiment # 2:
Implementation of Variables and Arithmetic Operators in C

Things to be covered in this experiment

- Variables
- scanf() statement
- Arithmetic operations
- math.h to use pow() and trigonometric functions
- comments

Note: Use comments to describe what each program does.

1. Remove all the errors in the program given below to get the output as shown below.

```
void main(int)
{
  cows, legs;
  printf("How many cow legs did you count?\n);
  scanf("%c", legs);
  cows = legs / 4;
  printf("That implies there are %f cows.\n", cows)
 }
```

Output

```
rehan@DESKTOP-44H03JG:~$ c_style_check.txt e2t1.c
rehan@DESKTOP-44H03JG:~$ gcc -Wall -Wstrict-prototypes -ansi -pedantic e2t1.c -o E2t1
rehan@DESKTOP-44H03JG:~$ ./E2t1
How many cow legs did you count?
8
That implies there are 2 cows.
```

2. Write a program that takes as inputs three different integers a, b and c and then displays their sum, the average, and the product.

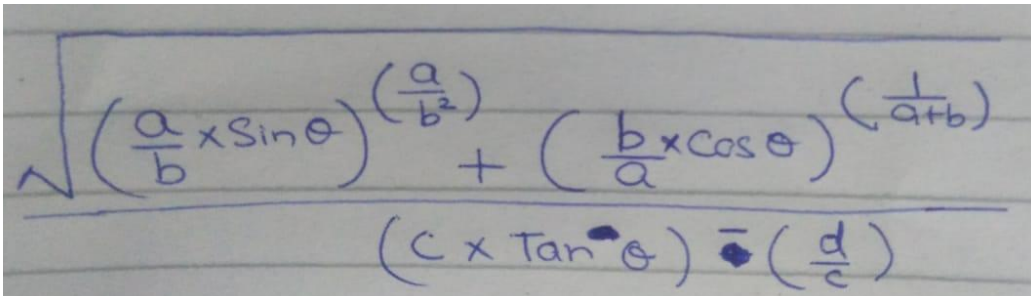
e.g., 13, 27 and 14. Now this input could be hard coded or user-generated via the keyboard (choice is yours). Show the output as below:

Output

```
Enter three integers:13 27 14
The sum is 54
The product is 4914
The average is 18.000000
```

3. Write a program to evaluate the following expression when a=10, b=5, c=15, d=20 and $\boldsymbol{\theta} = 45°$. The program should work correctly for any values of the variables. Don't forget to include the header <math.h> to use trigonometric and pow() functions. Use 22/7 as value of π.The output should be as shown below.

Expression



Output

```
rehan@DESKTOP-44H03JG:~$ gcc -Wall -Wstrict-prototypes -ansi -pedantic e2t3.c -o E2t3 -lm
rehan@DESKTOP-44H03JG:~$ ./E2t3
The answer is 0.105502
```

4. Write a program to calculate and display the value of the following expression.

$$v = A * sin(2\pi ft + \phi)$$

where A = amplitude of the sine wave

f = frequency of the sine wave in Hertz (Cycles/Second)

ϕ = Phase shift of the sine wave

t = time in seconds

Note: C considers the units of the angle of the function sin() to be radians i.e. sin(90) = 0.893 instead of sin(90) = 1. Use 22/7 as value of π.

Output

```
rehan@DESKTOP-44H03JG:~$ gcc -Wall -Wstrict-prototypes -ansi -pedantic e2t4.c -o E2t4 -lm
rehan@DESKTOP-44H03JG:~$ ./E2t4
Enter A:1
Enter f:50
Enter t:0.006
Enter p in degrees:45
v = 0.453032
```

5. Write a program that requests the download speed in megabits per second (Mbs) and the size of a file in megabytes (MB). The program should calculate the download time for the file. Note that in this context one byte is eight bits. Use type float, and use / for division. The program should report all three values (download speed, file size, and download time) as in the following:

Output

```
rehan@DESKTOP-44H03JG:~$ gcc -Wall -Wstrict-prototypes -ansi -pedantic e2t5.c -o E2t5
rehan@DESKTOP-44H03JG:~$ c_style_check.txt e2t4.c
rehan@DESKTOP-44H03JG:~$ ./E2t5
At 18.120001 Mbs, a file of 2.200000 MB downloads in 0.971302 seconds.
```

Experiment # 3:
Implementation of Conditionals in C

1. Part a. Complete the following code and remove all the errors to get the output as shown below. The program displays a message based on the age entered by the user.

```
printf("Enter your age:");
scanf("%f", age);

if (age >= 18):
  {
    printf("You are eligible for the driving license.\n");

else:

    print('You are not eligible for the driving license.\n")
  }
```

Output 1

```
Enter your age:18
You are eligible for the driving license.
```

Output 2

```
Enter your age:5
You are not eligible for the driving license.
```

Part b. Now add a few lines of code to the code you have written which first asks the user whether he wants to proceed or not. If yes, then the program should do what it did in part a. If no, then the program should display a message "End of the program". If neither yes nor no then the program should display "Invalid Input". Write and submit only one code for this question.

Output 1

```
rehan@DESKTOP-44H03JG:~$ gcc -Wall -Wstrict-prototypes -ansi -pedantic e3t1.c -o E3t1
rehan@DESKTOP-44H03JG:~$ ./E3t1
Do you want to proceed? Enter y for yes and n for no:y
Enter your age:19
You are eligible for the driving license.
```

Output 2

```
rehan@DESKTOP-44H03JG:~$ ./E3t1
Do you want to proceed? Enter y for yes and n for no:n
End of the program
```

2. Write a program which displays the message "Let's visit Murree." if the temperature of Lahore is greater than or equal to 40℃ and the temperature of Murree is less than or equal to 20℃. If the temperatures are not within the ranges then the message "Murree tour is not essential." Use only the IF - ELSE IF statements in this task.

Output 1

```
rehan@DESKTOP-44H03JG:~$ gcc -Wall -Wstrict-prototypes -ansi -pedantic e3t2.c -o E3t2
rehan@DESKTOP-44H03JG:~$ c_style_check.txt e3t2.c
rehan@DESKTOP-44H03JG:~$ ./E3t2
Enter temperature of Lahore:50
Enter temperature of Murree:10
Let's visit Murree.
```

Output 2

```
rehan@DESKTOP-44H03JG:~$ ./E3t2
Enter temperature of Lahore:20
Enter temperature of Murree:20
Murree tour is not essential.
```

3. Write a program which takes three integers from the user and displays the maximum integer of the three as shown below.

Output

```
rehan@DESKTOP-44H03JG:~$ c_style_check.txt e3t3.c
rehan@DESKTOP-44H03JG:~$ gcc -Wall -Wstrict-prototypes -ansi -pedantic e3t3.c -o E3t3
rehan@DESKTOP-44H03JG:~$ ./E3t3
Enter value of a:5
Enter value of b:1
Enter value of c:7
c has the maximum value which is 7.
```

4. Write a program which takes 3 integers -- x, bound1, and bound2, where bound1 is not necessarily less than bound2. If x is between the two bounds, just display it unmodified. Otherwise, if x is less than the lower bound, display the lower bound, or if x is greater than the upper bound, display the upper bound.

Output

```
rehan@DESKTOP-44H03JG:~$ gcc -Wall -Wstrict-prototypes -ansi -pedantic e3t4.c -o E3t4
rehan@DESKTOP-44H03JG:~$ c_style_check.txt e3t4.c
rehan@DESKTOP-44H03JG:~$ ./E3t4
Enter x, bound1 and bound2:1 3 5
3
rehan@DESKTOP-44H03JG:~$ ./E3t4
Enter x, bound1 and bound2:4 3 5
4
rehan@DESKTOP-44H03JG:~$ ./E3t4
Enter x, bound1 and bound2:6 3 5
5
rehan@DESKTOP-44H03JG:~$ ./E3t4
Enter x, bound1 and bound2:6 5 3
5
```

5. In the program shown below, what should be written in place of ? so that the truth table for AND operation is fully implemented. *You are only allowed to write some code in place of ? in the code below and nothing else.* The output is as shown below

```c
#include <stdio.h>

int main(void)
{
  ? x, y;

  printf("Enter x:");
  scanf( ? , &x);
  printf("Enter y:");
  scanf( ? , &y);

  if (?)
    {
      printf("%? AND %? is TRUE\n", x, y);
    }
  else if (?)
    {
      printf("%? AND %? is FALSE\n", x, y);
    }
  return 0;
}
```

Output

```
rehan@DESKTOP-44H03JG:~$ c_style_check.txt e3t5.c
rehan@DESKTOP-44H03JG:~$ gcc -Wall -Wstrict-prototypes -ansi -pedantic e3t5.c -o E3t5
rehan@DESKTOP-44H03JG:~$ ./E3t5
Enter x:t
Enter y:f
t AND f is FALSE
```

AND Truth Table

X	Y	X and Y
True	True	True
True	False	False
False	True	False
False	False	False

6. Write a program which takes a floating number from the user. The program should display the message "The value of x is 0.1" if the user enters 0.1 otherwise the program should do nothing.

Output

```
rehan@DESKTOP-44H03JG:~$ gcc -Wall -Wstrict-prototypes -ansi -pedantic e3t6.c -o E3t6
rehan@DESKTOP-44H03JG:~$ c_style_check.txt e3t6.c
rehan@DESKTOP-44H03JG:~$ ./E3t6
Enter a number:0.1
The value of x is 0.1
```

Experiment # 4:
Implementation of Loops in C

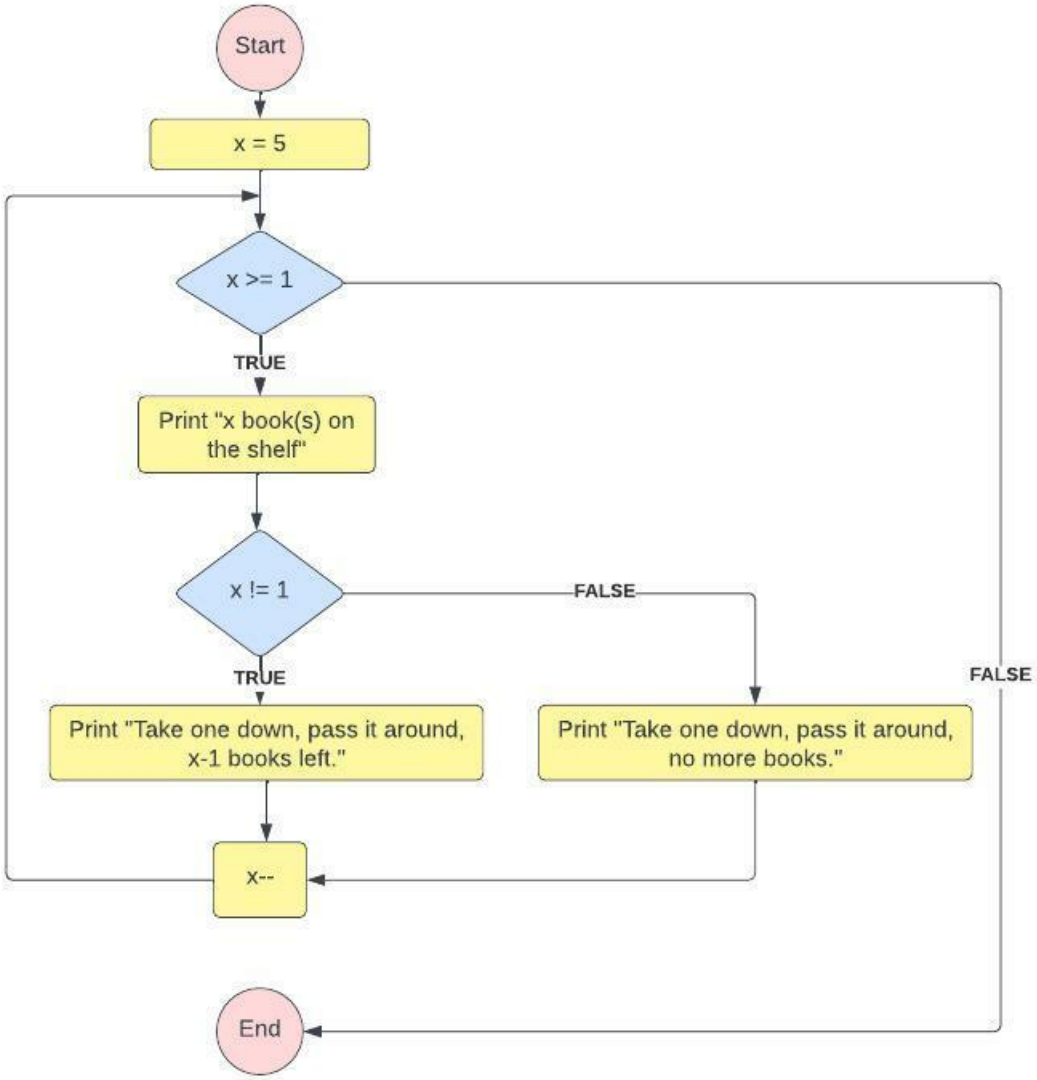
1. Complete the following program to get the output as shown below. Use the concept of field width to introduce spaces in the output.

```c
#include <stdio.h>
int main(void)
{
  int x;
  printf("123456789012345678901 23\n");
  printf("%s%s%s\n", "Integer", "Square", "Cube");

  for(x = 1; x <= 5; x++)
    {
      printf("%d%d%d\n", x, x * x, x * x * x);
    }
  return 0;
}
```

Output

```
rehan@DESKTOP-44H03JG:~$ c_style_check.txt e4t1.c
rehan@DESKTOP-44H03JG:~$ gcc -Wall -Wstrict-prototypes -ansi -pedantic e4t1.c -o E4t1
rehan@DESKTOP-44H03JG:~$ ./E4t1
12345678901234567890123
Integer   Square   Cube
      1        1      1
      2        4      8
      3        9     27
      4       16     64
      5       25    125
```

2. Convert the following flow chart into C code to get the output as shown below.




```
rehan@DESKTOP-44H03JG:~$ c_style_check.txt e4t2.c
rehan@DESKTOP-44H03JG:~$ gcc -Wall -Wstrict-prototypes -ansi -pedantic e4t2.c -o E4t2
rehan@DESKTOP-44H03JG:~$ ./E4t2
5 book(s) on C on the shelf
Take one down, pass it around, 4 books left
4 book(s) on C on the shelf
Take one down, pass it around, 3 books left
3 book(s) on C on the shelf
Take one down, pass it around, 2 books left
2 book(s) on C on the shelf
Take one down, pass it around, 1 books left
1 book(s) on C on the shelf
Take one down, pass it around, no more books.
```

3. Write a program to calculate the GPA of a student in a semester. The formula, grade points table and the output is shown below. Use the concept of precision to skip digits after the decimal point of a floating number.

Formula

$$GPA = \frac{\sum(CH \times Grade\ Points)}{\sum(CH)}$$

Table

Letter Grades & Corresponding Grade Points

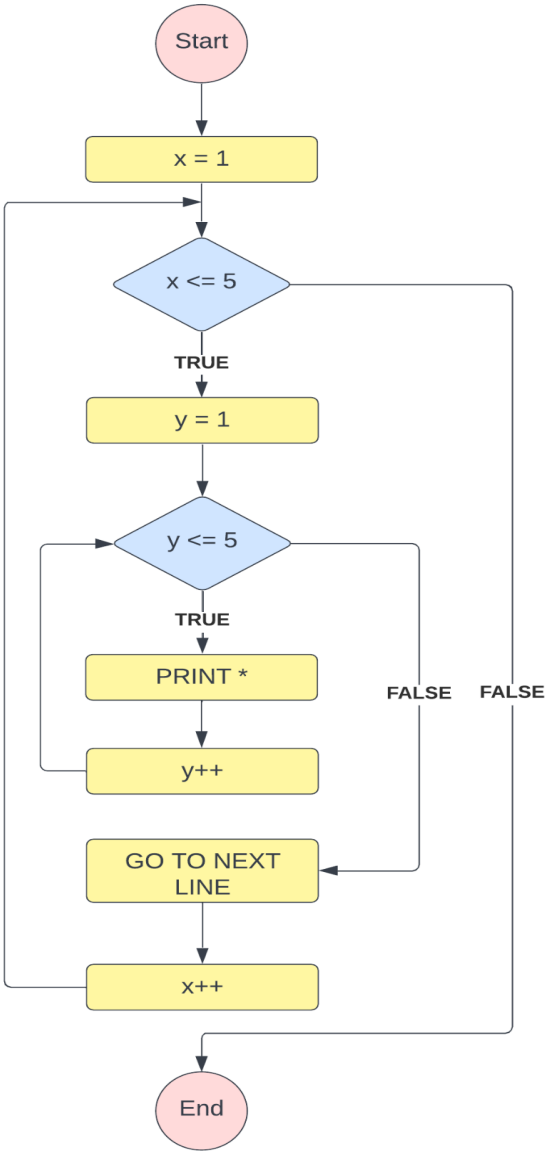
A+	A	A-	B+	B	B-	C+	C	C-	D+	D	F	W	WF	I	IP
4.0	4.0	3.7	3.3	3.0	2.7	2.3	2.0	1.7	1.3	1.0	0	-	-	-	-

Course Code	Course Title	CH	Grade	GPs	Status
Fall 2018					
EE-103	Introduction to Computing	2.0	B-	5.4	Confirmed
EE-103L	Introduction to Computing	1.0	A-	3.7	Confirmed
MA-123	Calculus	3.0	C+	6.9	Confirmed
ME-100L	Workshop Practice	1.0	A	4.0	Confirmed
ME-110	Applied Thermodynamics	3.0	A-	11.1	Confirmed
ME-110L	Applied Thermodynamics	1.0	A	4.0	Confirmed
MGT-103	Sociology for Engineering	2.0	A-	7.4	Confirmed
PHY-111	Applied Physics	2.0	B+	6.6	Confirmed
PHY-111L	Applied Physics	1.0	B+	3.3	Confirmed
Semester CH: 16.0		**GPA: 3.275**	**CGPA: 3.275**	**Status: Promoted**	

Output

```
rehan@DESKTOP-44H03JG:~$ gcc -Wall -Wstrict-prototypes -ansi -pedantic e4t3.c -o E4t3
rehan@DESKTOP-44H03JG:~$ c_style_check.txt e4t3.c
rehan@DESKTOP-44H03JG:~$ ./E4t3
Enter No. of subjects: 9
Enter credit hour and grade of subject 1: 2 B-
Enter credit hour and grade of subject 2: 1 A-
Enter credit hour and grade of subject 3: 3 C+
Enter credit hour and grade of subject 4: 1 A
Enter credit hour and grade of subject 5: 3 A-
Enter credit hour and grade of subject 6: 1 A
Enter credit hour and grade of subject 7: 2 A-
Enter credit hour and grade of subject 8: 2 B+
Enter credit hour and grade of subject 9: 1 B+
GPA = 3.275
```

4. Print the following SQUARE using the flowchart given below .



Output

```
rehan@DESKTOP-44H03JG:~$ c_style_check.txt e4t4.c
rehan@DESKTOP-44H03JG:~$ gcc -Wall -Wstrict-prototypes -ansi -pedantic e4t4.c -o E4t4
rehan@DESKTOP-44H03JG:~$ ./E4t4
*****
*****
*****
*****
*****
```

5. Print the following using nested for-loops.

Output

```
rehan@DESKTOP-44H03JG:~$ ./E4t5
1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
1 x 4 = 4
1 x 5 = 5

2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
```

6. Write a program to print a hourglass of 5 rows as shown below. Divide this problem into two smaller problems i.e. the first small problem prints line 1 to 3 of the hourglass whereas the second small problem prints line 4 to 5. Follow the think-code-test-debug procedure to get this task done. Draw a flowchart before writing the code.

Output

```
rehan@DESKTOP-44H03JG:~$ gcc -Wall -Wstrict-prototypes -ansi -pedantic e4t6.c -o E4t6
rehan@DESKTOP-44H03JG:~$ ./E4t6
*****
 ***
  *
 ***
*****
```

```
suleman@Desktop-corei7-6700K:~/C$ gcc Lab2a.c -o Lab2a
suleman@Desktop-corei7-6700K:~/C$ ./Lab2a
Input a string without spaces
ThisIsExperiment2
Number of vowels in the string: 6
```

Experiment # 4: Loops (continued)

1. Determine the output of the following codes without using a computer.

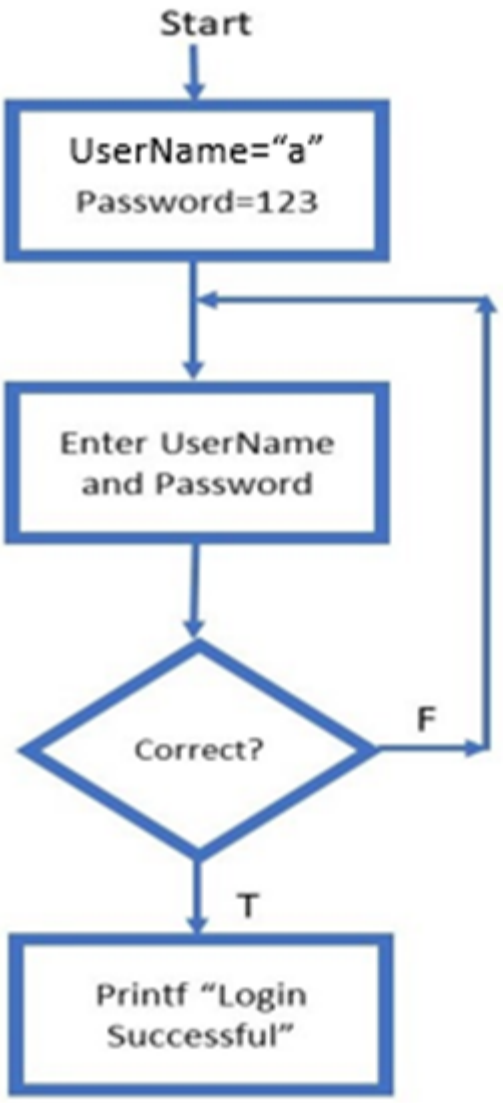
   a)

```c
#include<stdio.h>

void main()
{
    int i=1;
    int k=1;
    while (i<=5)
    {
       printf("%d",i);
       if (k==5){
            printf("Lab4");
            }
        i++;
        k=k+2;
    }
}
```

   b)

```c
#include<stdio.h>

int main(){
    int i,k;
    for (i=0, k=0; (i< 5 && k < 3); i++, k++)
    {
        ;
    }
    printf("%d\n",i);
    printf("%d\n",k);

    return 0;

}
```

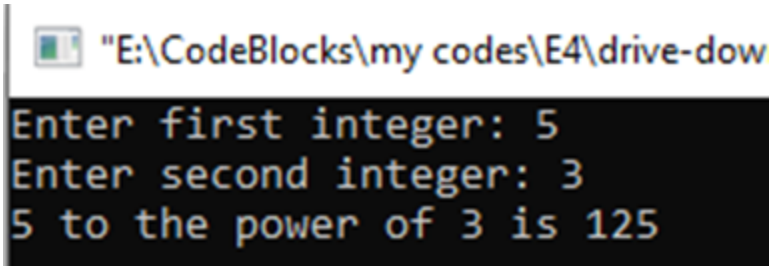
2. Write a C program for any login application. Convert the given flowchart of login algorithm into C language.



3(a) Complete the following code to get the output as shown below. The code computes the power of an integer.

```
1  #include<stdio.h>
2
3  int main( void )
4  {
5       int x,y;
6       int i = 1;
7       int power = 1;
8
9       printf( "Enter first integer: " );
10      scanf( "%d", &x ); // read value for x from user
11      printf( "Enter second integer: " );
12      scanf( "%d", &y ); // read value for y from user
13
14      while ( )
15      {
16          power *= x;
17
18      }
19
20      printf(  ); // display power
21 } // end main function
```

OUTPUT



(b) Rewrite the above code to get the output as shown above using a FOR-loop.

4 (a). Write a program to print an “inverted” pyramid of 3 rows as shown below. Use nested FOR-loops to complete this task.

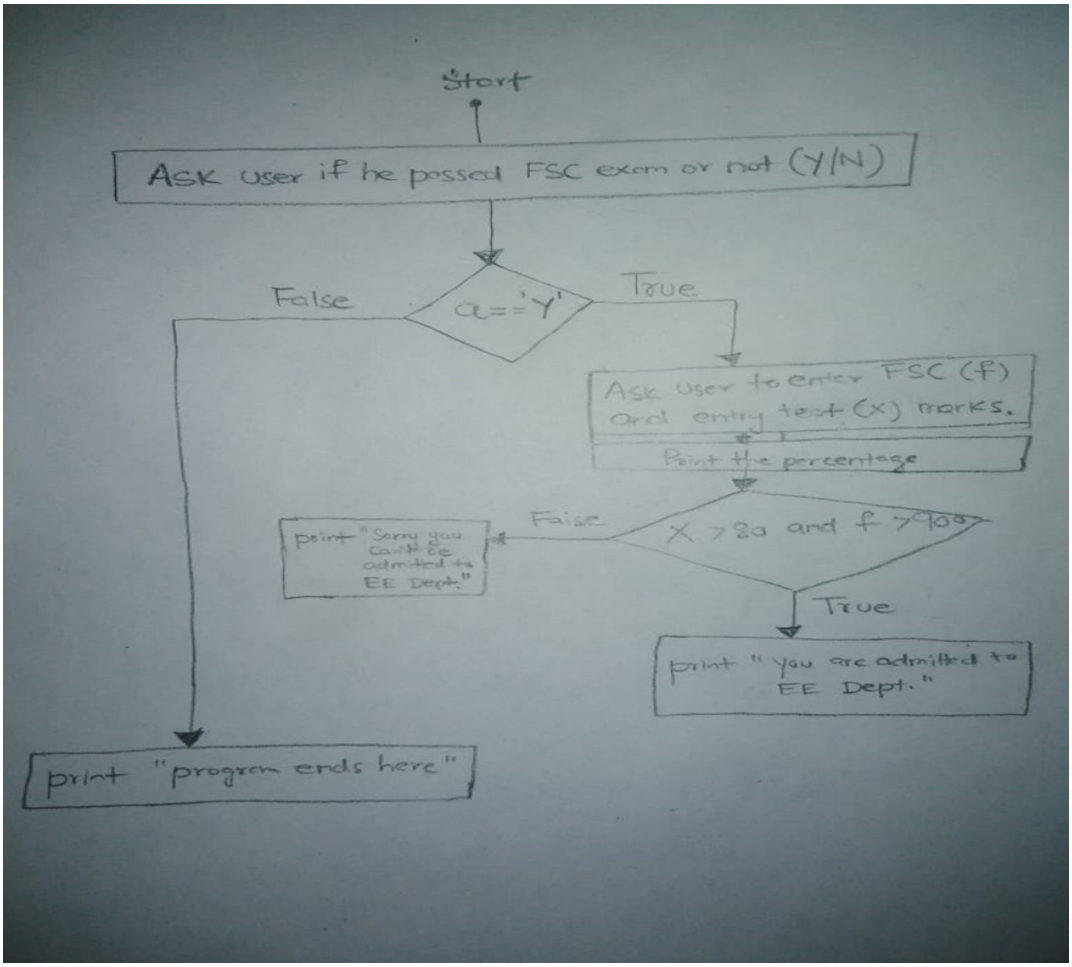
(b) Write a program to print a pyramid of 3 rows as shown below. Use nested FOR-loops to complete this task.

(c) Combine and modify the codes written in part a and b to print the “hour glass” of 5 rows as shown below.

(d) Draw a flowchart for the codes in part a, b and c of this question.

5. Write a program to get the output exactly as shown below by following the flowchart given below. Admission to EE Department happens when the percentage of FSC and entry test marks is greater than 80 and the Fsc marks is greater than 900. Note: Both the outputs belong to the same code. The percentage is calculated by considering 70% of the FSc and 30% of the entry test marks.

FLOWCHART



OUTPUT 1 **OUTPUT 2**

```
C:\Users\UET\Downloads\Mycodes-20210317
Have you passed FSc Exams?(Y/N):Y
FSc Marks?:902
Entry Test Marks?:318
Your percentage is 81.250000
You are admitted to EE Dept
```

```
C:\Users\UET\Downloads\Mycodes-20210317T080
Have you passed FSc Exams?(Y/N):N
program ends here
```

Experiment # 5:

Implementation of Functions in C

1. Complete the following program to get the output as shown below. Do not change the code written in the main body. Hint: Four functions have to be defined properly to get the output.

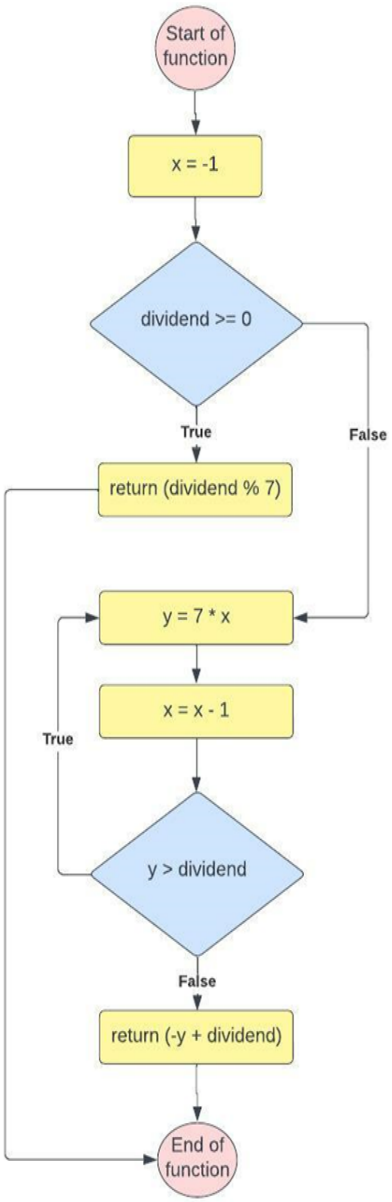
```c
int main(void)
{
  int s;
  s = add(1, 2);
  printf("The sum is %d\n", s);
  diff(1, 2);
  printf("The product of 1 and 2 is %d\n", prod());
  div();
  return 0;
}
```

```
rehan@DESKTOP-44H03JG:~/e5$ c_style_check.txt e5t1.c
rehan@DESKTOP-44H03JG:~/e5$ gcc -Wall -Wstrict-prototypes -ansi -g -pedantic e5t1.c -o E5t1
rehan@DESKTOP-44H03JG:~/e5$ ./E5t1
The sum is 3
The difference is -1
The product of 1 and 2 is 2
1/2 = 0.500000
```

2. Complete the following program to calculate and display the modulo (remainder) of two integers i.e. the dividend and the divisor. The flow chart to find the mod is given below. The program should work correctly for any values of the dividend and the divisor.

```c
int main(void)
{
  int dividend, divisor;

  printf("Enter dividend: ");
  scanf("%d", &dividend);
  printf("Enter divisor: ");
  scanf("%d", &divisor);
  printf("The modulo is %d\n", calculate_modulo( dividend, divisor ) );
  return 0;
}
```

```
rehan@DESKTOP-44H03JG:~$ c_style_check.txt e5t2.c
rehan@DESKTOP-44H03JG:~$ gcc -Wall -Wstrict-prototypes -ansi -pedantic -g e5t2.c -o E5t2
rehan@DESKTOP-44H03JG:~$ ./E5t2
Enter dividend: 16
Enter divisor: 7
The modulo is 2
rehan@DESKTOP-44H03JG:~$ ./E5t2
Enter dividend: -11
Enter divisor: 7
The modulo is 3
rehan@DESKTOP-44H03JG:~$ ./E5t2
Enter dividend: -24
Enter divisor: 7
The modulo is 4
```



3. Write a program to print the name of the day at which a particular date occurs. Use the function *Print_Day()* which takes day number, month number, and year as arguments. The function should then print **"The day is <day>"** corresponding to the given date. For example, Wednesday is the day on March 1, 2000. Use Zeller's Congruence to find the day. The algorithm is also given below. Use the function calculate_modulo(dividend, divisor) defined in task 2 to find the modulo (remainder).

```c
int main(void)
{
  int q, m, y;

  printf("Enter date: ");
  scanf("%d%d%d", &q, &m, &y);
  printf("The day is ");
  Print_Day(q, m, y);
  return 0;
}
```

```
rehan@DESKTOP-44H03JG:~/e5$ gcc e5t3.c -o E5t3
rehan@DESKTOP-44H03JG:~/e5$ ./E5t3
Enter date: 2 2 2022
The day is Wednesday
rehan@DESKTOP-44H03JG:~/e5$ ./E5t3
Enter date: 28 1 2022
The day is Friday
rehan@DESKTOP-44H03JG:~/e5$ ./E5t3
Enter date: 15 2 2016
The day is Monday
rehan@DESKTOP-44H03JG:~/e5$ ./E5t3
Enter date: 25 5 2022
The day is Wednesday
```

$$h = \left(q + \left\lfloor \frac{13(m+1)}{5} \right\rfloor + K + \left\lfloor \frac{K}{4} \right\rfloor + \left\lfloor \frac{J}{4} \right\rfloor - 2J\right) \bmod 7,$$

where

- h is the day of the week (0 = Saturday, 1 = Sunday, 2 = Monday, ..., 6 = Friday)
- q is the day of the month
- m is the month (3 = March, 4 = April, 5 = May, ..., 14 = February)
- K the year of the century ($year \bmod 100$).

- J is the zero-based century (actually $\lfloor year/100 \rfloor$) For example, the zero-based centuries for 1995 and 2000 are 19 and 20 respectively

- $\lfloor \ldots \rfloor$ is the floor function or integer part
- mod is the modulo operation or remainder after division

In this algorithm January and February are counted as months 13 and 14 of the previous year. E.g. if it is 2 February 2010, the algorithm counts the date as the second day of the fourteenth month of 2009 (02/14/2009 in DD/MM/YYYY format)

Examples [edit]

For 1 January 2000, the date would be treated as the 13th month of 1999, so the values would be:

$q = 1$
$m = 13$
$K = 99$
$J = 19$

So the formula evaluates as $(1 + 36 + 99 + 24 + 4 - 38) \bmod 7 = 126 \bmod 7 = 0 = \text{Saturday}$.

(The 36 comes from $(13 + 1) \times 13/5 = 182/5$, truncated to an integer.)

However, for 1 March 2000, the date is treated as the 3rd month of 2000, so the values become

$q = 1$
$m = 3$
$K = 0$
$J = 20$

so the formula evaluates as $(1 + 10 + 0 + 0 + 5 - 40) \bmod 7 = -24 \bmod 7 = 4 = \text{Wednesday}$.

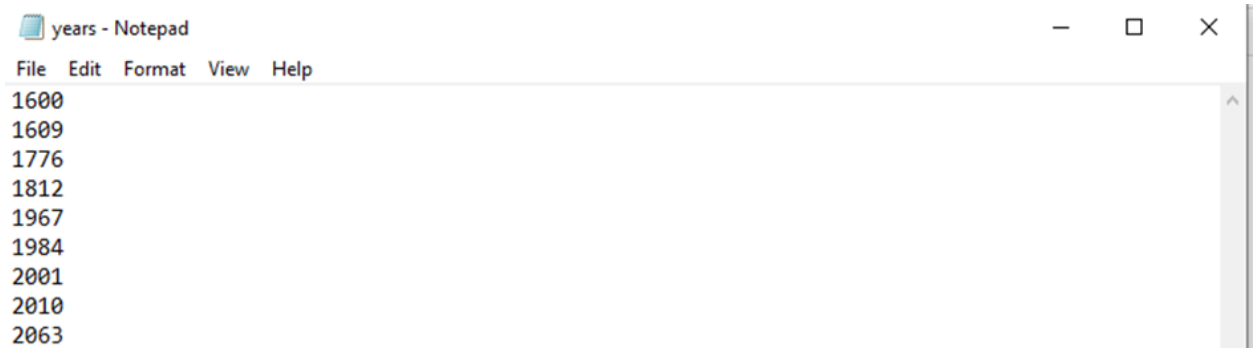
4. Write a program to find the date (Month and day number) at which Easter event occurs. The pseudo code is given below. Define and use the function calculate_Easter_date(year) to implement the pseudocode. Your program should take "year" as input from years.in file. Your program should store output to the easter_dates.out file as shown below. The command *./E5t4 < years.in > easter_dates.out* will do these two tasks for you. Hint: In step E8, return a positive day number if N > 31 else return a negative day number .

Pseudocode

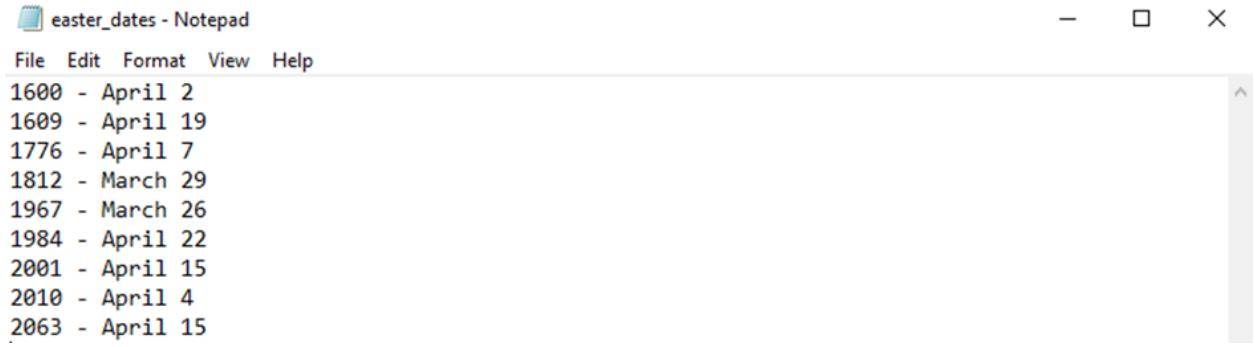
```
GIVEN:   Y: the year for which the date of Easter is to be determined.
FIND:    The date (month and day) of Easter

STEP E1: Set G to (Y mod 19) + 1.
         [G is the "golden year" in the 19-year Metonic cycle.]
STEP E2: Set C to (Y / 100) + 1. [C is the century]
STEP E3: Set X to (3C / 4) - 12. [X is the skipped leap years.]
         Set Z to ((8C + 5) / 25) - 5.
         [Z is a correction factor for the moon's orbit.]
STEP E4: Set D to (5Y / 4) - X - 10.
         [March ((-D) mod 7 + 7) is a Sunday.]
STEP E5: Set E to (11G + 20 + Z - X) mod 30.
         If E is 25 and G is greater than 11 or if E is 24,
         increment E.
         [E is the "epact" which specifies when a full moon occurs.]
STEP E6: Set N to 44 - E.  [March N is a "calendar full moon".]
         If N is less than 21 then add 30 to N.
STEP E7: Set N to N + 7 - ((D + N) mod 7).
         [N is a Sunday after full moon.]
STEP E8: If N > 31 the date is APRIL (N - 31),
         otherwise the date is MARCH N.
```

```
rehan@DESKTOP-44H03JG:~$ c_style_check.txt e5t4.c
rehan@DESKTOP-44H03JG:~$ gcc -Wall -Wstrict-prototypes -ansi -pedantic -g e5t4.c -o E5t4
rehan@DESKTOP-44H03JG:~$ ./E5t4 < years.in > easter_dates.out
```

Input File


years - Notepad
File Edit Format View Help
1600
1609
1776
1812
1967
1984
2001
2010
2063

Output File


easter_dates - Notepad
File Edit Format View Help
1600 - April 2
1609 - April 19
1776 - April 7
1812 - March 29
1967 - March 26
1984 - April 22
2001 - April 15
2010 - April 4
2063 - April 15

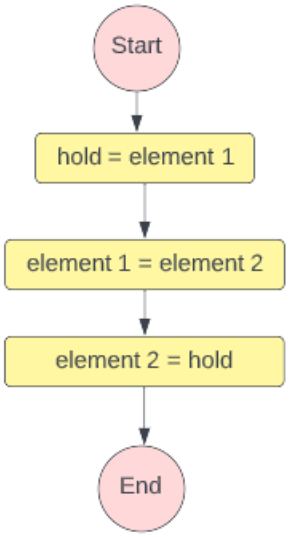
Experiment # 6:

Implementation of Arrays in C

1. Complete the following program which swaps any two elements of an array **a** and then prints the resultant array. Define and use the function **swap()** to swap two elements. This function should take, as arguments, the array **a,** and the **indices** of the two elements to be swapped. Define and use the function **print_array()** to print the whole array **a**. This function should take the array **a,** and the number of elements of the **a** as arguments. The flowchart to swap two elements is given below.

```c
int main(void)
{
  int a[9] = { 5, 9, -2, 150, -95, 23, 2, 5, 80 };

  swap(a, 3, 5);
  print_array(a, 9);

  return 0;
}
```

```
rehan@DESKTOP-44H03JG:~/e6$ gcc -Wall -Wstrict-prototypes -ansi -pedantic e6t1.c -o E6t1
rehan@DESKTOP-44H03JG:~/e6$ c_style_check.txt e6t1.c
rehan@DESKTOP-44H03JG:~/e6$ ./E6t1
The elements of the array are: 5, 9, -2, 23, -95, 150, 2, 5, 80,
```



2. Complete the following program to implement **Minimum Element Sort** algorithm to sort integers in ascending order. The pseudo code of the algorithm is given below.

Pseudocode

Let `array` be the array of integers, and `num_elements` be the total number of elements in `array`.

1. Start with `start = 0` (for the index of the zeroth element).
2. Set `smallest = start` (smallest stores the index of the smallest element encountered so far).
3. Run through a loop with the variable `index` going from `start` to `num_elements`
4. If `array[index] < array[smallest]`, set `smallest = index`.
5. Once the loop ends, swap `array[start]` and `array[smallest]` (moving the smallest element found to the beginning of the array you searched).
6. Increment `start`, and if `start < num_elements`, go back to step 2. **Do not** use a `goto` statement, however; use another loop.
7. If `start >= num_elements` then you're done and the array is sorted.

```c
int main(void)
{
  int array[9] = { 5, 9, -2, 150, -95, 23, 2, 5, 80 };
  return 0;
}
```

```
rehan@DESKTOP-44H03JG:~/e6$ gcc -Wall -Wstrict-prototypes -ansi -pedantic e6t2.c -o E6t2
rehan@DESKTOP-44H03JG:~/e6$ c_style_check.txt e6t2.c
rehan@DESKTOP-44H03JG:~/e6$ ./E6t2
The array elements after sorting are: -95, -2, 2, 5, 5, 9, 23, 80, 150,
```

3. Write a program to read the command-line arguments and store them in an array named **array.** You should define and use a function **transfer()** to transfer all the relevant items from array **argv[]** to array **array[]** as shown below. This function should take 3 arguments i.e. the array **array[]**, array **argv[]** and number of elements of array **argv[]**.

```c
int main( ? )
{

  int num_elements = ?;
  int array[32];

  transfer(array, argv, num_elements);
  print_array(array, num_elements);
  return 0;
}
```

```
rehan@DESKTOP-44H03JG:~/e6$ gcc -Wall -Wstrict-prototypes -ansi -pedantic e6t3.c -o E6t3
rehan@DESKTOP-44H03JG:~/e6$ c_style_check.txt e6t3.c
rehan@DESKTOP-44H03JG:~/e6$ ./E6t3 455 143 200
The array elements are: 455, 143, 200,
```

4. Write a program to implement **Bubble Sort** algorithm to sort integers in ascending order. The code to implement Bubble Sort is given below. Define and use a function **transfer()** to transfer relevant elements from array **argv** to array **array**. Define and use a function **BubbleSort()** to implement the Bubble Sort algorithm. This function should take the array to be sorted and number of elements of that array as arguments. Also define and use a function **swap()** to swap two elements of an array. The output is shown below.

```
int main(?)
{

  int num_elements = ?;
  int array[32];

  transfer(array, argv, num_elements);
  BubbleSort(array, num_elements);
  print_array(array, num_elements);
  return 0;
}
```

Bubble Sort Code

```
do
  {
    s = 0;
    for( index = 1; index < n; index++)
      {
        if( x[index-1] > x[index] )
          {
            swap(x, index, index - 1);
            s = 1;
          }
      }
  }while( s != 0 );
```

```
rehan@DESKTOP-44H03JG:~/e6$ c_style_check.txt e6t4.c
rehan@DESKTOP-44H03JG:~/e6$ gcc -Wall -Wstrict-prototypes -ansi -pedantic e6t4.c -o E6t4
rehan@DESKTOP-44H03JG:~/e6$ ./E6t4 5 9 -2 150 -95 23 2 5 80
The array elements are: -95, -2, 2, 5, 5, 9, 23, 80, 150,
```

5. Complete the following program which uses the user-defined function **TransferSortAndPrint()** to transfer 5 integers entered by the user in the command line to an array named **array**. The array **array** is then
    a. sorted using minimum element sort algorithm if the command line argument does not contain any optional argument.
    b. sorted using bubble sort algorithm if the command line argument contains the optional argument **-b** which can **occur anywhere** in the command line after **./E6t5**.

    At the end, the sorted array is to be printed using the user-defined **print_array()** function. Hint: Using the built-in function atoi(), one can detect an integer or non-integer element present in array argv[].

```
#define MAX 32

int main(int argc, char *argv[])
{
  TransferSortAndPrint(argc, argv, 5);
  return 0;
}

void TransferSortAndPrint(int argc, char *argv[], int c)
{
   int i, x = 0;
   char SortAlgo = 'm';
   int array[MAX];

   for ( i = 1; i < argc; i++ )
     {
       if( strcmp( argv[i], "-b") == 0 )
         {
           SortAlgo = 'b';
         }
       else if( ? ) /*else if the element is an integer.*/
         {
           array[x] = ?
           x++;
         }
     }

  if( SortAlgo == 'm' )
   {
     MinimumElementSort(array, c);
   }
  else
   {
     BubbleSort(array, c);
   }

   print_array(array, c);

}
```

```
rehan@DESKTOP-44H03JG:~/e6$ gcc -Wall -Wstrict-prototypes -ansi -g -pedantic e6t5.c -o E6t5
rehan@DESKTOP-44H03JG:~/e6$ ./E6t5 -3 -7 0 3 1
Minimum Element Sort:
-7
-3
0
1
3
rehan@DESKTOP-44H03JG:~/e6$ ./E6t5 -3 -7 0 3 1 -b
Bubble Sort:
-7
-3
0
1
3
rehan@DESKTOP-44H03JG:~/e6$ ./E6t5 -3 -7 -b 0 3 1
Bubble Sort:
-7
-3
0
1
3
rehan@DESKTOP-44H03JG:~/e6$ ./E6t5 -b -3 -7 0 3 1
Bubble Sort:
-7
-3
0
1
3
```

Experiment # 7:

Implementation of Pointers in C - Part 1

1. Complete the following program which stores the addresses of **x** and **y** in two pointers **i** and **j**. Use these pointers to add the values present in **x** and **y**.

```
int main(void)
{
  int x = 1;
  int y = 2;
```

```
rehan@DESKTOP-44H03JG:~/e7$ gcc -Wall -Wstrict-prototypes -ansi -g -pedantic e7t1.c -o E7t1
rehan@DESKTOP-44H03JG:~/e7$ c_style_check.txt e7t1.c
rehan@DESKTOP-44H03JG:~/e7$ ./E7t1
The sum is 3
```

2. Complete the program in question 1 which uses user-defined function **add()** to add the values of **x** and **y** and store the sum in **x**. The function **add()** should take two arguments i.e. the addresses of **x** and **y and should return nothing back**. The sum should be printed out by the main function.

3. Modify the program in question 2 so that the function **add()** prints the sum at the output without storing the sum into another variable.

4. Complete the following program so that the output is obtained as shown below.

```
int main(void)
{
  int i = 1;
  printf("The address of i is %p\n", foo1(&i));
  printf("The value of i is %d\n"  , foo2(&i));
  return 0;
}
```

```
rehan@DESKTOP-44H03JG:~/e7$ ./E7t4
The address of i is 0x7ffc67275644
The value of i is 1
```

Experiment # 8:

Implementation of Pointers in C - Part 2

1. Write a program to print the address of each element of the array shown below. Use the concept of **field width** to introduce spaces at the output. What is the difference between two consecutive addresses? Why is it so? **Note: The addresses given below might be different from the ones displayed at your output terminal**.

```
#include <stdio.h>
int main(void)
{
  int x[5] = {1, 2, 3, 4, 5};
  int i;
  printf("12345678901234567890123456789012\n");
  printf("%7s%8s%17s\n", "Element", "Value", "Address");

  for( i = 0; i < 5; i++ )
    {
      printf(?);
    }

  return 0;
}
```

```
rehan@DESKTOP-44H03JG:~/e8$ ./E8t1
12345678901234567890123456789012
Element   Value           Address
   x[0]       1   0x7ffe014f5520
   x[1]       2   0x7ffe014f5524
   x[2]       3   0x7ffe014f5528
   x[3]       4   0x7ffe014f552c
   x[4]       5   0x7ffe014f5530
```

2. Complete the following program which sums up all the elements of the array. Use pointer **x** and the pointer-offset notation, i.e. ***(x + offset)** where offset is an integer value, to access each element of the array **x**.

```
int main(void)
{
  int x[5] = { 1, 2, 3, 4, 5 };
```

```
rehan@DESKTOP-44H03JG:~/e8$ c_style_check.txt e8t2.c
rehan@DESKTOP-44H03JG:~/e8$ ./E8t2
The sum is 15
```

3. Complete the following program which adds each element of the two arrays **x** and **y** and then stores the sum in a third array **z**. Use **pointer-offset notation** to access and store the array elements.

```
int main(void)
{
  int x[5] = { 1, 2, 3, 4, 5};
  int y[5] = { 1, 2, 3, 4, 5};
```

```
rehan@DESKTOP-44H03JG:~/e8$ gcc -Wall -Wstrict-prototypes -ansi -g -pedantic e8t3.c -o E8t3
rehan@DESKTOP-44H03JG:~/e8$ ./E8t3
z[5] = {2,4,6,8,10}
```

Experiment # 9:

Implementation of Dynamic Memory Allocation in C

1. Create an array having max number of elements equal to 5 using **malloc()**. Store and print the integers of the array as shown below**. Don't forget to free the memory.**

```
rehan@DESKTOP-44H03JG:~/e9$ gcc -Wall -Wstrict-prototypes -ansi -pedantic e9t1.c -o E9t1
rehan@DESKTOP-44H03JG:~/e9$ ./E9t1
j[0] = 0
j[1] = 1
j[2] = 2
j[3] = 3
j[4] = 4
```

2. Re-do the above task by writing and using function **foo()** which uses a pointer j to return the address given by **malloc()**. Also use pointer-offset notation to store and print the integers stored in the array. **Don't forget to free the memory.**

```
rehan@DESKTOP-44H03JG:~/e9$ gcc -Wall -Wstrict-prototypes -ansi -pedantic e9t2.c -o E9t2
rehan@DESKTOP-44H03JG:~/e9$ ./E9t2
*(x + 0) = 0
*(x + 1) = 1
*(x + 2) = 2
*(x + 3) = 3
*(x + 4) = 4
```

Incomplete code

```
____ foo (____)
{
  _________
  _________

  return j;
}
```

3. Write a program the takes the max number of elements and the elements of an array from the command line. All the elements in the command line should be stored into an array **x** and then be printed out as shown below. In the pic below, 5 represents max no of elements of an array. Use **calloc()** to allocate memory. Use a function **foo()** for memory allocation as done in question 2. **Don't forget to free the memory.**

```
rehan@DESKTOP-44H03JG:~/e9$ gcc -Wall -Wstrict-prototypes -ansi -pedantic e9t3.c -o E9t3
rehan@DESKTOP-44H03JG:~/e9$ ./E9t3 5 10 4 1 3 6
x[0] = 10
x[1] = 4
x[2] = 1
x[3] = 3
x[4] = 6
```

4. Write a program which displays a pattern of characters **'*'** and **'.'** randomly. Store the pattern in an array **x** created using dynamic memory allocation. Then print out the elements of the array. The program should read the number of characters of the pattern from the command line.

```
rehan@DESKTOP-44H03JG:~/e9$ gcc e9t4.c -o E9t4
rehan@DESKTOP-44H03JG:~/e9$ ./E9t4 15
****..**..***.*
rehan@DESKTOP-44H03JG:~/e9$ ./E9t4 10
.*.**..*.*
rehan@DESKTOP-44H03JG:~/e9$ ./E9t4 5
..*.*
```

Experiment # 10:

Implementation of Structures in C

1. Write a program which uses the structure **struct point** having two members of type int. Create and initialize the two points (variables) p1 and p2 of type struct point and find and print the distance between the two points.
2. Re-do the above problem using a pointer **j** to **p1** and another pointer k to **p2**. Use pointers **j** and **k** to initialize **p1** and **p2**, to calculate distance and to print the distance.
3. Re-do question 1 using function **foo()** which takes two arguments **p1** and **p2** and calculates the distance. The distance should then be returned back and printed at the output.
4. Re-do question 2 using function **foo()** which takes three arguments j, k and a pointer to variable **dist**. The function **foo()** should calculate and store the distance in **dist** using pointers **j**, **k** and **a**.

Experiment # 11:

CS 11 C track: Assignment 1

Table of Contents

Goals
Getting set up
Language concepts covered this week
Suggested Reading
Programs to write
To hand in

Goals

In this assignment you will learn the basics of compiling programs and the most fundamental aspects of the C language.

Getting set up

We are providing you with instructions for setting up a virtual machine (VM) running Ubuntu Linux in which you should write all of your code. (See this page for information on how to set up the VM.) You are not required to do this (the setup page also describes some alternatives) but we expect that you will be able to set up a Unix-like environment (either a Mac, a Linux VM, or a Windows Linux environment like WSL or Cygwin), and use this environment when you run and test your programs.

You will also need to know basic Linux commands (particularly terminal commands). If you've taken CS 1, you should already have this knowledge, but if not, you can consult any online tutorial on Linux terminal use.

Language concepts covered this week

- basic input/output using `printf` and `scanf`
- conditionals (`if` statements)
- loops (`for` statements)
- numeric types and conversions (`int` to `double`)
- C strings
- preprocessor directives (`#include`)
- using standard libraries
- the `main()` function
- compiling code using `gcc`

Suggested Reading

- Darnell and Margolis, chapter 3.
- K&R, chapter 1 (all) and chapter 7 (pp. 153-159). Some of the material on `scanf` presupposes an understanding of pointers. Since you haven't seen pointers yet, I've included an "aside on scanf" below to help you out.
- If you don't understand the C compilation process, take a moment and read this page now. Also review lecture 1.

Programs to write

- `hello1`

Write a program called `hello1` to print `hello, world!` to the terminal. Use the `printf` function to do the printing. Make sure there is a newline at the end of the message. Compile it using the command:

```
$ gcc -Wall -Wstrict-prototypes -ansi -pedantic hello1.c -o hello1
```

(`$` is the unix shell prompt.) `gcc` is the GNU C Compiler, whose job it is to convert the file `hello1.c` (called *source code*; this is the file you create) into a binary executable. `hello1` is the name of the binary program; the `-o` option tells the compiler that the next argument is the name of the output file. Don't worry about the `-Wall`; it turns on compiler warnings so that the compiler will warn you about anything it considers dubious but which is still legal. It's a good habit to use `-Wall` whenever you use gcc. Note that although almost all C compilers have some option for enabling or suppressing warnings, there is no standard command-line option for these, so `-Wall` will only work for gcc. Make sure that your `main` function returns 0 or the compiler will issue a warning. The options `-Wstrict-prototypes -ansi -pedantic` ensure that your program is ANSI-compliant and that your prototypes have been declared correctly. Don't worry about this for now, but do use it; it will make your life much easier later on.

- `hello2`

Modify `hello1.c` so that the program (now called `hello2`, corresponding to the source code file `hello2.c`) prints a prompt string (*e.g.* `Enter your name:`), after which you enter your name and it prints `hello, <your name>!` to the terminal (with your name substituted for `<your name>`, of course). **Make sure your program prints a prompt string before reading the input** (a lot of people forget to do this). The name you enter should be a single word only (say, your first name). Use the library function `scanf` to read the string from standard input (also known as `stdin`) and print to standard output (also known as `stdout`). `stdin` and `stdout` both represent input and output from the terminal (as opposed to, say, a file).

Aside on `scanf`: `scanf` should be invoked as follows:

```c
char s[100];  /* N.B. strings are arrays of chars in C */
/* ... maybe some intervening code ... */
scanf("%99s", s);
```

When run, the program will pause while executing the `scanf` until you enter a string and hit return, or until 99 characters have been entered (whichever comes first). [1] Then it will continue. The character array `s` will then contain the string you entered, and can be passed to `printf`. I will discuss this in more detail in later lectures. Note that `scanf` used as above will also ignore anything past the first whitespace character (space or tab). There are ways to get around this, but they aren't important now. That's why your name should just be a single word.

- `hello3`

Modify `hello2` so that the program prints a prompt, you enter your name, the computer generates a *single* random number `n` between 1 and 10 and prints that many messages. The format of each message that you will print is: `<n>: hello, <your name>!` or: `<n>: hi there, <your name>!`, where `<n>` is the random number you generated and `<your name>` is, well, your name. Print the first message when `<n>` (**not** the loop index variable) is even and the second message when `<n>` is odd. **All the messages for a single run of the program will thus be identical.**

I repeat: **All the messages for a single run of the program are identical.** *Pay attention to this!* Every term, some students print a different message depending on whether the loop index variable is odd or even instead of n, or don't print the number n at all. If you do this, you'll lose marks.

By "loop index variable" I mean *e.g.* `i` in: `for (i = 0; i < n; i++) ....`

Use the `rand` library function to generate the random numbers; this function is found in the `stdlib.h` header file, so be sure to add:

```c
#include <stdlib.h>
```

to the top of your program source code. You will also need to "seed" the random number generator to get it started; the best way to do this is to include this line:

```c
srand(time(0));
```

at the beginning of the program. `srand` is also in `stdlib.h`, but `time` is in the `time.h` header file, so you should `#include` that as well. Be aware that there is not always 100% standardization on the locations of functions in header files among different operating systems. You should also look at the man pages for `rand` and `srand`, which will contain useful information. To do this, type: `man 3 rand` at the unix prompt (it's in section 3 of the online manuals; just `man rand` will not work, because there is another `rand` that has a man page).

A slightly tricky part of this task is converting from the return value of `rand()`, which returns an int between 0 and `RAND_MAX` (which is a large integer constant defined in `<stdlib.h>`) into a random number between 1 and 10. There's more than one way to do this.

ASIDE ON NUMERIC CONVERSIONS: You can convert an integer to a real number (double or float) as follows:

```c
int i = 10;
double d = (double)i;  /* Convert int to double. */
float f = (float)f;    /* Convert int to float.  */
```

Recall that doubles are double-precision while floats are single-precision. Similarly, you can convert real numbers (doubles or floats) to integers as follows:

```c
double d = 12.3;
int i = (int)d;    /* Convert double to int. */
```

In this case, the conversion to integers throws away everything to the right of the decimal point (the fractional part). I'll talk about this more in lecture 2.

To hand in

The `hello1.c`, `hello2.c`, and `hello3.c` programs. Before you hand them in, make sure that you run the style checking program on all of them to catch obvious style mistakes. See the style guide for more information on this.

1. The reason you can't enter 100 characters with this code will be explained in a later lecture.

Experiment # 12:

CS 11 C track: Assignment 2

Table of Contents

Goals
Language concepts covered this week
Other concepts covered this week
Reading
Operators and operator precedence in C
Program to write
  Description of the algorithm
  Explanation of the algorithm
  Description of the program
Testing the program
Other things to do
Supporting files
To hand in
  For extra credit...
References

Goals

In this assignment you will write a more substantial program and learn some new language constructs.

Language concepts covered this week

- operators
- functions
- function prototypes
- more on loops
- comments

Other concepts covered this week

- the `make` program and `Makefile`s
- I/O redirection (Unix)
- test scripts

Reading

Read this page to familiarize yourself with the `make` program and `Makefiles`.

Also take a look at the C style guide. Starting from this assignment, we'll be more picky about style issues.

Operators and operator precedence in C

An "operator" is a character or character sequence that has a special syntactic meaning to the C compiler. Most operators are *binary*, which means that they are found sandwiched between two values. An example is the addition operator, `+`, which is found in expressions such as `1 + 2`. Some operators are *unary*, such as the bitwise-NOT operator (`~`). One operator is actually *trinary* (the dreaded `?:` operator), but we won't discuss it here. Compared to most languages, C has a very large number of operators and a correspondingly large number of operator precedence levels (15 of them to be precise; see table 2-1 in K&R if you're curious). Operator precedence levels determine how to interpret expressions with multiple operators. For instance,

```c
a = b + c * d + e;
```

is interpreted as being

```c
a = b + (c * d) + e;
```

because multiplication has higher precedence than addition. If we want it to be interpreted differently, we need to use parentheses, *e.g.*

```c
a = (b + c) * (d + e);
```

Note that the `=` sign is also an operator (the assignment operator). It has very low precedence, so we don't need to use parentheses around the arithmetic expression to the right of the `=` sign. Also note that (confusingly) equality testing uses the `==` operator, not the `=` operator.

We **do not** want you to memorize the operator precedence table! Instead, simply use these three rules:

- Multiplication and division have precedence over addition and subtraction;
- All assignment operators (except for `++` and `--`) have extremely low precedence;
- Put parentheses around everything else where there is any possibility of confusion.

Also note that C has a lot of shortcut assignment operators:

- Operators of the form `<op>=` *e.g.* `+=`, `-=`, `*=`, `/=`, `%=`, *etc.* These all have this meaning:

```c
x <op>= y;
```

means:

```c
x = x <op> y;
```

for some operator `<op>`.

- The `++` and `--` increment/decrement operators.

Used judiciously, they often result in more concise and understandable code. An annoying fact is that the `++` and `--` operators have a very *high* precedence, whereas all the other assignment operators have the same precedence as `=` does.

Program to write

You will write a program called `easter` that will compute the day of the year on which Easter falls, given the year.

Description of the algorithm

This algorithm is taken from Donald Knuth's famous book *The Art of Computer Programming* (see the references below).

```
GIVEN:   Y: the year for which the date of Easter is to be determined.
FIND:    The date (month and day) of Easter

STEP E1: Set G to (Y mod 19) + 1.
         [G is the "golden year" in the 19-year Metonic cycle.]
STEP E2: Set C to (Y / 100) + 1. [C is the century]
STEP E3: Set X to (3C / 4) - 12. [X is the skipped leap years.]
         Set Z to ((8C + 5) / 25) - 5.
         [Z is a correction factor for the moon's orbit.]
STEP E4: Set D to (5Y / 4) - X - 10.
         [March ((-D) mod 7 + 7) is a Sunday.]
STEP E5: Set E to (11G + 20 + Z - X) mod 30.
         If E is 25 and G is greater than 11 or if E is 24,
         increment E.
         [E is the "epact" which specifies when a full moon occurs.]
STEP E6: Set N to 44 - E.  [March N is a "calendar full moon".]
         If N is less than 21 then add 30 to N.
STEP E7: Set N to N + 7 - ((D + N) mod 7).
         [N is a Sunday after full moon.]
STEP E8: If N > 31 the date is APRIL (N - 31),
         otherwise the date is MARCH N.
```

Note 1

All divisions in this algorithm are integer divisions, which means that any fractional remainders are thrown away. Also, the comment `[March ((-D) mod 7 + 7) is a Sunday]` is technically only true for years after 1752, because there was an 11-day correction applied to the calendar in September of 1752. You don't need to mention this in your comments, since it doesn't affect the Easter computation.

Note 2

We will be adding another step to make the algorithm work nicely with our C program; see the description of the program below for more details.

Note 3

Just because the great Don Knuth wrote this algorithm this way doesn't mean that it's written in a nice or easy-to-understand way. In particular, the use of single characters as variable names is usually a very bad idea (because single characters don't have any meaning to the person reading the code), and we don't want you to do that in this program. Knuth was trying to describe the algorithm in as short a space as possible; you don't have that restriction.

Explanation of the algorithm

Ever wonder what those monks did during the Dark Ages, all secluded away in their distant mountaintop monasteries and things? Well, it turns out that they were busy calculating the date of Easter. See, even back then, there wasn't much point in spending any effort on calculating the dates of holidays like Christmas, which as everyone knows, is on the same day each year. That also went for holidays which have become a bit more obscure, like Assumption (August 15th).

But the trouble with Easter is that it has to fall on Sunday. I mean, if you don't have that, all the other non-fixed holidays get all screwed up. Who ever heard of having Ash Wednesday on a Saturday, or Good Friday on Thursday? If the Christian church had gone and made a foolish mistake like that, they'd have been the laughingstock of all the other major religions everywhere.

So the Church leaders hemmed and hawwed and finally defined Easter to fall on the first Sunday after the first full moon after the vernal equinox.

I guess that edict must not have been too well-received, or something, because they then went on to define the vernal equinox as March 21st, which simplified matters quite a bit, since the astronomers of the time weren't really sure that they were up to the task of finding the date of the real vernal equinox for any given year other than the current one, and often not even that. So far so good.

The tricky part all comes from this business about the full moon. The astronomers of the time weren't too great at predicting that either, though usually they could get it right to within a reasonable amount, if you didn't want a prediction that was too far into the future. Since the Church really needed to be able to predict the date of Easter more than a few days in advance, it went with the best full-moon-prediction algorithm available, and defined "first full moon after the vernal equinox" in terms of that. This is called the Paschal Full Moon, and it's where all the wacky "epacts" and "Metonic cycles" come from.

So what's a Metonic Cycle?

A Metonic cycle is 19 years.

The reason for the number 19 is the following, little-known fact: if you look up in the sky on January 1 and see a full moon, then look again on the same day precisely 19 years later, you'll see another full moon. In the meantime, there will have been 234 other full moons, but none of them will have occurred on January 1st.

What the ancient astronomers didn't realize, and what makes the formula slightly inaccurate, is that the moon only really goes around the earth about 234.997 times in 19 years, instead of exactly 235 times. Still, it's pretty close — and without computers, or even slide rules, or even pencils, you were happy enough to use that nice, convenient 19-year figure, and not worry too much about some 0.003-cycle inaccuracy that you didn't really have the time or instruments to measure correctly anyway.

Okay, how about this Golden Number business then?

It's just a name people used for how many years into the Metonic cycle you were. Say you're walking down the street in Medieval Europe, and someone asks you what the Golden Number was. Just think back to when the last 19-year Metonic cycle started, and start counting from there. If this is the first year of the cycle, it means that the Golden Number is 1; if it's the 5th, the Golden number is 5; and so on.

Okay, so what's this "Epact" thing?

In the Gregorian calendar, the Epact is just the age of the moon at the beginning of the year. No, the age of the moon is not five billion years — not here, anyway. Back in those days, when you talked about the age of the moon, you meant the number of days since the moon was "new". So if there was a new moon on January 1st of this year, the Epact is zero (because the moon is new, *i.e.* zero days "old"); if the moon was new three days before, the Epact is three; and so on.

When Easter was first introduced, the calculation for the Epact was very simple — since the phases of the moon repeated themselves every 19 years, or close enough, the Epact was really easy to calculate from the Golden Number. Of course, this was the same calendar system that had one leap year every 4 years, which turned out to be too many, so the farmers ended up planting the fields at the wrong times, and life just started to suck.

Pope Gregory Makes Things More Complicated

You may already know about the changes Pope Gregory XIII made in 1582 with respect to leap years. No more of this "one leap year every four years" business like that Julius guy said. Nowadays, you get one leap year every four years *unless* the current year is a multiple of 100, in which case you don't — ***unless*** the current year is *also* a multiple of 400, in which case you do anyway. That's why 2000 was a leap year, even though 1900 wasn't (I'm sure many of you were bothered by this at the turn of the millenium).

Well, it turns out that the *other* thing Pope Gregory did, while he was at it, was to fix this Metonic Cycle-based Easter formula which, quite frankly, had a few bugs in it — like the fact that Easter kept moving around, bumping into other holidays, occuring at the wrong time of year, and generally making a nuisance of itself.

Unfortunately, Pope Gregory had not taken CS 11. So instead of throwing out the old, poorly-designed code and building a new design from scratch, he just patched up the old version of the program (this is common even in modern times). While he was at it, he changed the definition of Epact slightly. Don't worry about it, though — the definition above is the new, correct, Gregorian version.

This is why you'll see Knuth calculating the Epact in terms of the Golden Number, and then applying a "correction" of sorts afterwards: Gregory defined the Epact, and therefore Easter, in terms of the old definition with the Metonic cycles in it. Knuth is just the messenger here.

So what is this thing with "Z" and the moon's orbit?

It's just the "correction" factor which the Pope introduced (and Knuth later simplified) to account for the fact that the moon doesn't really orbit the earth exactly 235 times in 19 years. It's analogous to the "correction factor" he introduced in the leap years — the new formula is based on the old one, is reasonably simple for people who don't like fractions, is also kind of arbitrary in some sense, and comes out much closer to reality, but still isn't perfect.

What about all the rest of that stuff?

Ah, well, you wouldn't want me to make this too easy, would you? Our hope is that, after this brief introduction, that code up there will not seem quite so mysterious, and that you may, in fact, be able to figure out, if not exactly what's going on, at least most of the stuff that's happening in there.

Description of the program

Write a program that reads a series of years from a text file and prints out the date of Easter on all those years, as follows.

Running the program

When the program is written, use Unix input/output redirection to handle input from and output to files (if you don't know about this, here (http://en.wikipedia.org/wiki/Redirection_(Unix)) is a decent tutorial). In other words, invoke the program like this:

```
$ easter < infile > outfile
```

where `$` is the terminal prompt (so you don't type that in). **Note:** this is Unix shell (terminal) syntax, **not** C syntax! The `< infile` part means to take the input from the file called `infile` instead of from the keyboard, and the `> outfile` part means to send the output to the file called `outfile` instead of printing it to the terminal. See below for more details on this.

This will *not* work on Windows (unless you are running Linux using the Windows Subsystem for Linux or using Cygwin) but it will work on Linux and MacOS. We recommend you set up and use a virtual machine as described elsewhere for all the assignments in this track.

You can't leave out the `<` and `>` characters *i.e.* don't do this:

```
$ easter infile outfile
```

That won't work unless you make the C code more complicated. Also, we want you to do it so that it works with the `<` and `>` characters.

`infile` is a file containing a list of years (one per line) *e.g.*

```
1994
1995
1998
```

and `outfile` will become a list of year/date pairs, *e.g.*

```
1994 - April 3
1995 - April 16
1998 - April 12
```

Handling input from files and output to files directly from your C program is possible, but it's a little bit more complicated, so we won't bother with it now (it involves functions like `fopen()`, `fclose()` and `fscanf()`; look them up if you're curious). Instead, you just have to read from standard input (*i.e.* the terminal; use `scanf()` like in the previous assignment) and write to standard output (using `printf()`). Unix-derived operating systems (including Linux and MacOS) will convert this to reading from a file and writing to a file if you use the `<` and `>` symbols in the command line as we showed above:

```
$ easter < infile > outfile
```

Technically, what this does is bind standard input to the file `infile` and bind standard output to the file `outfile` just for this one invocation of the `easter` program, so that when in your program you read from standard input (using `scanf()`) you're really reading from `infile`, and when you write to standard output (using `printf()`) you're really writing to `outfile`. [1]

Easter computation

The program should include a function called `calculate_Easter_date` (yes, that exact name) which takes an integer argument (the year) and returns an integer representing the date. The month should be indicated by the sign of the integer return value: negative means March and positive means April. The absolute value of the integer represents the day of the month. So April 10 would be represented as the integer 10, while March 23 would be represented as the integer -23. **This is not part of the Knuth algorithm!** You have to convert from the value that Knuth's algorithm gives you to the value in this representation (which is quite easy).

Returning the date this way is an egregious hack. There are much better ways to return multiple values from a C function, which will be described in later assignments.

The allowable years are in the range 1582 to 39999; if the input is outside of this range, the `calculate_Easter_date` function should return 0. When the main program sees this return value, it should print an error message to `stderr` (NOT to `stdout`; use `fprintf` instead of `printf` for this), and then continue with the loop.

Don't print the error message inside the `calculate_Easter_date` function; do it inside the `main` function. This is good design; the `calculate_Easter_date` function should only be concerned with calculating the Easter date; printing error messages are not its responsibility. In general, you should design functions to do one thing and only one thing; it'll make your programs much more elegant and much easier to debug.

Input/output

In the `main()` function, use a call to `scanf` to read in each the input line from `stdin`. Note that to read an integer value using `scanf`, you need to use the `"%d"` format string (where "d" means "decimal"). Store the return value of `scanf` in a variable (yes, `scanf` does return a value, but it isn't used very often). This return value will be equal to the integer constant `EOF` ("end of file") when there is no more input. `EOF` is defined in the header file `<stdio.h>`, just like `printf` and `scanf`. You will find the `break` statement (K&R pp. 64-65) to be useful in your loop. Use a `while` loop that loops forever *e.g.* `while (1) { ... }` until an `EOF` is encountered, and then `break` out of the loop. The `main()` function should call the `calculate_Easter_date()` function for each line of input. If `calculate_Easter_date()` returns 0 (because of a range error), print an error message to `stderr` and keep going.

Make sure you have declared function prototypes at the top of your file before you define the functions. Although this isn't strictly necessary here, it's a good habit to get into. Prototypes allow you to reference functions before they are defined, which allows you to program without having to worry about what order your functions are defined in. (However, you do not have to write a prototype for the `main` function.)

Commenting

Comment your code liberally, especially the Easter algorithm itself. Very few programmers write too many comments; most write way too few. Remember, your task is to write a program which (a) does what it's supposed to, and (b) is clearly understandable. You are free to copy Knuth's algorithm verbatim if you like, but make sure you add comments explaining what each step of the algorithm does. Also, your version of the algorithm should contain better variable names than the ones Knuth uses. If you use one-letter variable names like Knuth does, you'll lose a lot of marks. Instead, use variable names that are words or phrases that are descriptive of the meaning of the variable.

For functions, **put a comment before the function that states what the function does, what the arguments mean, and what the return value means.** These are the most important kinds of comments you'll ever write, because they are what will allow other people to use your code.

Also, put a comment at the top of the file explaining what the program does as a whole.

We will be grading your program not only on how well it performs its task, but on how easy it is to read and understand. Make sure you keep this in mind as you do this assignment. And don't think this is only an exercise for this assignment — future assignments have to have the same standard of commenting.

Testing the program

We are supplying you with a simple `Makefile`. Download it into your `lab2` directory as a file called `Makefile` (it should be called that by default; just don't change the name). Running `make` will create the `easter` program. Running `make test` will run a simple test of the program and report whether it is correct or not with respect to the inputs. This is an example of a "test script". Test scripts are critically important in producing correctly-working code. Some programmers even advocate writing test scripts and test cases for functions before writing the actual code to be tested. Running `make check` will run the style checker on your code.

Other things to do

Write a `clean` target for the Makefile. This target will remove the `easter` program, all object files, and all files generated by the program when you type `make clean`. It must **not** remove the input file used to test the program, your source code file, the test script, or the correct output file.

Use the Unix command `rm -f` (*i.e.* the `rm` program with the `-f` optional argument) to remove your files. Normally, `rm` complains if you try to remove a nonexistent file, but with the `-f` optional argument it won't. Note also that `rm -f` can take multiple filename arguments.

Try the program on an input file that intentionally contains years that are outside the correct range. Send the output to a file as usual. The error messages should be printed on the terminal, not put into the file. This shows you the difference between printing to `stdout` (which `printf()` does) and printing to `stderr` (which `fprintf()` does if you tell it to).

Supporting files

Make sure you download all of these files into your `lab2` directory (except possibly for the style checker, which ideally you should already have put into your `~/bin` directory).

- The Makefile.

Be sure that all the command lines in the Makefile start with tabs, or they will not work. Please do *not* try to copy and paste the file from a web browser window. Instead, use the "Save Page As" function of your browser (which is probably under the File menu). Alternatively, you can select "Save Link As" while right-clicking on the link to the `Makefile`.

- The test script.

  In order to get this to work, you have to do `chmod +x run_test` after downloading this file, in order to make it executable. Make sure that the resulting file name is `run_test` and not (for instance) `run_test.txt`, or it won't work!.
- The input file, for testing.
- The correct output file, for testing.
- The style checker.

To hand in

The program `easter.c` and the completed `Makefile`. We will run the program to see if it passes the test script.

For extra credit...

Have your program use "Zeller's Congruence" to verify that the date it is printing really does fall on a Sunday. Use `assert` to signal an error in case it doesn't. Get the definition of Zeller's Congruence by doing an internet search. Type `man assert` to learn more about the `assert` macro. Assert is a very valuable and under-appreciated debugging aid, which we will meet again in later assignments.

References

- K&R, chapters 2, 3, and 4.

- Donald Knuth, *The Art of Computer Programming, vol. 1: Fundamental Algorithms.*

  The Art of Computer Programming is a (so far) four-volume set (although more volumes are being written) which is a definitive treatment on computer algorithms written by Donald Knuth. The books are usually referred to simply as "Knuth vol. 1", *etc.* They are extremely dense and not really suitable for beginners, but they are good if you need to look up an algorithm and learn more about it. Knuth virtually invented the field of mathematical analysis of computer algorithms, and is still going strong. [2]

1. If you think this is kind of cool, then you're right.
2. His Christmas lectures (which you can watch on YouTube) are particularly awesome.

Experiment # 13:

CS 11 C track: Assignment 3

Table of Contents

Goals
Language concepts covered this week
Other concepts covered this week
Reading
Arrays in C
Program to write
The Sample Output
Command-line arguments
Converting strings to integers
The minimum element sort algorithm
Adding command-line options
More on command-line options
Avoiding magic numbers
Commenting
Testing your program
The joy of `assert`
Using `assert` in your program
Running the test script
Supporting files
To hand in
References

Goals

In this assignment you will learn about C arrays, C strings and how to make your program interact with the Unix (*e.g.* Linux but also MacOS) command line. You will then write a simple sorting program. Finally, you will learn some useful strategies for testing code.

Language concepts covered this week

- arrays
- strings
- command-line argument processing

Other concepts covered this week

- `Makefile`s again
- Using `assert` for debugging
- Test scripts again

Reading

Please read this page on command-line argument processing.

Arrays in C

Arrays in C are declared as follows:

```c
int foo[10];  /* Declares an array called 'foo' with space for ten ints. */
```

and accessed as follows:

```c
int i;
...
i = foo[4];  /* This gets the element from 'foo' at index 4. */
```

Note that arrays in C start at element 0, *not* element 1. Therefore, in this case, the last element in the array would be `foo[9]`. Note also that array elements are not initialized to be anything, so they should be assumed to hold garbage (arbitrary values) until you assign a value to them. Also, you need to realize that if you try to access an element "off the end" of the array (*e.g.* the 100th element of the ten-element array `foo` in the code above), you will get no compiler warnings, but the program will probably crash when it runs.

This is part of C's no-error-checking I-assume-you-know-what-you-want philosophy of programming.

Arrays can also be two-dimensional, three-dimensional, *etc.* The syntax is analogous:

```c
int foo[10][5];
int i;

...

i = foo[4][2];
```

In addition, arrays can be *initialized* when they are declared:

```c
int foo[5] = { 1, 2, 3, 4, 5 };

int bar[2][3] =
{
  { 1, 2, 3 },
  { 4, 5, 6 }
};
```

You will need to use array initialization in the next assignment. Finally, you can pass arrays to a function like you would pass a normal variable:

```c
void munge(int array[])
{
   /* code that uses the values in array[] */
}

/* more code... */

int main(void)
{
    int stuff[10];

    /* more code... */

    /* Pass the 'stuff' array to the function 'munge'. */
    munge(stuff);

    /* {etc} */
}
```

However, when you do this, you should be aware that you are *not* passing a copy of the array to the function but the array itself. That means that if you modify the array in the function it will remain modified when you return from the function. This will be useful in the assignment below. The reason for this behavior is due to the fact that C arrays are actually represented as pointers; we'll cover this in later lectures and assignments.

Program to write

Write a program called `sorter` which behaves as follows:

- If there are no command-line arguments at all when the program is run, the program should print out instructions on its use (a "usage message"; see this page). There should only be one usage message, and it must follow the standard conventions (see the link). Note that the optional command-line arguments (see below) must be included as part of the usage message.
- The program will be able to accept up to 32 numbers (integers) on the command line.
- If there are more than 32 numbers on the command line, or no numbers at all, the program should print out the usage message and exit.
- If the optional command-line arguments `-b` or `-q` are found **anywhere** in the command line, change the behavior of the program as described below.
- If any of the command-line arguments to the program are not integers or one of the two optional command-line arguments, your program's response is undefined — it can do anything. (*i.e.* you shouldn't worry about having to handle anything but integer arguments or the two command-line options). The way to deal with this is as follows: for each argument, first check to see if it's one of the command-line options. If so, proceed accordingly. If not, assume it represents an integer and convert it using the `atoi()` function (see below).
- Sort the numbers using either the minimum element sort or the bubble sort algorithm (see below). **Do not use a global array to hold the integers**; use a locally-defined array in `main` and pass the array to the sorting function. Define separate functions for both sorting algorithms. Use `assert` (see below) to check your sorting function for correctness.
- Print out the numbers from smallest to largest, one per line.

The Sample Output

If the above doesn't make sense, this is what your program should look like (bold is what you would type, and `$` is the Unix prompt; yours may be different):

```
$ sorter 5 9 -2 150 -95 23 2 5 80
-95
-2
2
5
5
9
23
80
150
$ sorter
usage: sorter [-b] [-q] number1 [number2 ... ] (maximum 32 numbers)
```

Command-line arguments

Your program begins in the `main` function, which up until now has looked like this:

```c
int main(void)
{
   /* your code here */
   return 0;
}
```

but will now look like this:

```c
int main(int argc, char *argv[])
{
   /* your code here */
   return 0;
}
```

So let's take the first line apart. There are a few parts to this:

`int`

Declares that this function returns an integer.

`main`

Declares this function's name, `main`. Recall that C programs always begin executing in the `main` function.

`int argc`

`argc` is equal to the number of elements of `argv`.

`char *argv[]`

Okay, this one's a bit trickier. The second argument, `argv`, is an array of "`char *`"s. `char *` is C's way of handling character strings, which are represented as arrays of characters where the last character is ASCII character 0 (often written as `'\0'` and sometimes called the "nul" character [1]). This will make more sense when we discuss pointers. `argv` contains one string for each of the command line arguments that your program is run with. More on this below.

To give an example of this, let's take the command line from the example above:

```
$ sorter 5 9 -2 150 -95 23 2 5 80
```

This would produce the following values in `argc` and `argv`:

```
argc        10
argv[0]    "sorter"
argv[1]    "5"
argv[2]    "9"
argv[3]    "-2"
argv[4]    "150"
argv[5]    "-95"
argv[6]    "23"
argv[7]    "2"
argv[8]    "5"
argv[9]    "80"
```

Note that `argv[0]` holds the name of your program, and that the first user-supplied argument is in `argv[1]`. **Remember this!** (One thing this implies is that `argc` is 1 greater than the number of user-supplied arguments.) Notice also that a command-line argument of `5` is **not** an integer; it's a string that can be converted into an integer. Which leads us to the next topic.

Converting strings to integers

Okay, so how do we turn these `argv` strings into integer values? Well, there's this handy function called `atoi` ("**a**scii **to** **i**nteger"). For example, `atoi("5")` equals 5. In order to use `atoi`, you need to put the following line at the top of your program:

```c
#include <stdlib.h>
```

`atoi` is a pretty dumb function; if you pass it a bogus value it'll just return 0 instead of signalling any kind of error. That means that you need to check for the optional command-line arguments `-b` and `-q` before trying to convert a command-line argument to an `int`.

The minimum element sort algorithm

Alright, so what is this "minimum element sort" algorithm? The basic idea is that the smallest element in the array will be the zeroth element in the sorted array, the second-smallest will be the first element, *etc.* Here's how it works:

Let `array` be the array of integers, and `num_elements` be the total number of elements in `array`.

1. Start with `start = 0` (for the index of the zeroth element).
2. Set `smallest = start` (smallest stores the index of the smallest element encountered so far).
3. Run through a loop with the variable `index` going from `start` to `num_elements`
4. If `array[index] < array[smallest]`, set `smallest = index`.
5. Once the loop ends, swap `array[start]` and `array[smallest]` (moving the smallest element found to the beginning of the array you searched).
6. Increment `start`, and if `start < num_elements`, go back to step 2. **Do not** use a `goto` statement, however; use another loop.
7. If `start >= num_elements` then you're done and the array is sorted.

Adding command-line options

You will add the ability to process two optional command-line arguments (usually called "command-line options") to your program. These options will be `-b` and `-q`. To test for these, you'll need to be able to compare strings. The function `strcmp(str1, str2)` returns `0` if `str1` and `str2` are the *same* and nonzero otherwise. So to check if the first argument is `-b`, you would have to do something like:

```c
if (strcmp(argv[1],"-b") == 0)
{
    /* put stuff here */
}
```

To use the `strcmp` function, you need to put the following line with the other `#include`s at the top of your file:

```c
#include <string.h>
```

WARNING! Do **not** do this:

```c
if (argv[1] == "-b")  /* WRONG! */
{
    /* put stuff here */
}
```

This doesn't work; you can't compare strings using the `==` operator. [2] The reason for this will be made clear when we talk about pointers.

You should add information on any command-line options you allow to the usage information that is printed out when there are no arguments. If you don't, you will lose marks.

If the user supplies one of the command-line options, you must *not* treat it as part of the list of integers. Also, **you must *not* assume that the command-line options are going to be entered before the numbers**; it should be possible to enter them anywhere after the program name (and the test script will check for this). Finally, **it's legal to enter the command-line options more than once**, although this won't make any difference after the first time. **You don't need complicated code to achieve this!** You can process all of the command-line arguments in a single pass through the `argv` array. Many students write absurdly complicated code to handle the command-line arguments, and it's just a waste of effort. In other words, it's much easier than you probably think it is.

One thing you **do** have to make sure of is that you aren't adding values into the array of numbers at indices that are too large. This array can only be 32 elements long at most, so if you try to assign to indices 32 or greater, it's an error even if it seems to work properly.

It should be clear why indices greater than 32 are off-limits, but why do you think it's illegal to assign to index 32 exactly?

To keep things simple, you are only required to handle integers or the two specific command-line options `-b` and `-q`, and any command-line argument (not counting the program name) that isn't `-b` or `-q` can be assumed to be an integer.

Here are the command-line options and what they mean:

- The `-b` option means that you should sort using a "bubble sort" instead of a minimum element sort algorithm.

Bubble sort, like minimum element sort, is not a very efficient algorithm. Much more efficient algorithms (such as quicksort) exist, but they are best left until after you've had some experience with recursion, which is coming up next week ☺. Also, the C standard library has a `qsort` (quicksort) library function, but to use it you need to understand function pointers, which we won't cover in this track (although it's not that hard).

- The `-q` option suppresses the output (*i.e.* nothing gets printed). Why would we want to do this? See the next section.

Information on different sorting algorithms, including bubble sort, can be found all over the web, in algorithms textbooks, *etc.* The Wikipedia page on bubble sort (https://en.wikipedia.org/wiki/Bubble_sort) is a good reference.

More on command-line options

Here are some guidelines (which we expect you to follow) for implementing the command-line option processing in this program. They will make it much easier to get a working program that passes the test script.

- First off, **do not change the elements in the `argv` array**. There is no need to. Some people try to shuffle values in the `argv` array, as if you need to pass `argv` to the sorting functions. You don't. What you need to do is to create a brand new array that can hold all the numbers in the `argv` array. Altering the `argv` array is legal, but it's poor programming style.
- Second, this new array has to be large enough to hold all the numbers on the command line, and no larger. Given what you all know now, this means that it should be 32 elements long. You may not use all of them (that's fine), but they will be there in case someone enters a command line with 32 numbers. Later, I'll show you how to create arrays that have exactly the right number of elements, based on the input. Also, you aren't allowed to do this:

```c
int n = 10;
int array[n];  /* Invalid */
```

This isn't legal C [3], so if you do this, you'll get a compiler warning, and our policy is to not allow programs that generate compiler warnings. In addition, you can't just use a humongous array, like this:

```c
int array[1000];
```

This kind of thing is just programming to make the test script work, usually for the wrong reason. So 32 is as big an array as we'll allow.

- Third, you have to check if there are **more** than 32 numbers in the command line, which means 32 values that are neither the program name (`argv[0]`) nor `-b` nor `-q` . If there are more than 32 numbers, the program should exit with a usage message and return a non-zero value from `main()` . The test script will check for this. In addition, make sure that your program doesn't at any point add an element to the numbers array at an invalid index *i.e.* an index larger than 31. Even if the program passes the test script, this is a serious error, because you're writing into memory that isn't part of the array.
- Fourth, the `-q` or `-b` arguments can occur *anywhere* on the command line except in `argv[0]` (you should know why they can't occur in `argv[0]`), and there can even be more than one `-b` or `-q` arguments in the command line. Multiple `-b` or `-q` arguments don't do anything beyond what single ones do. The test script checks for this too.
- Finally, despite all these guidelines, this is actually a pretty easy problem; you can do everything in a single `for` loop. If your solution is very complicated, you are almost certainly doing something wrong. If that's the case, don't be shy about asking for help from the TAs or the instructor!

Avoiding magic numbers

A "magic number" is a number that is just plopped into a program with no context, especially if a number like this is repeated several times in a program. Usually, a magic number is some kind of unidentified parameter of the program.

In this assignment, the number `32` hard-coded into the program would be a magic number. Magic numbers are almost always bad, for two reasons:

- The significance of the number is unknown, or must be inferred from the code it's embedded in. That makes the code harder to understand.
- If you decide to change it to a different value, you have to change it in multiple places, and it's easy to forget one, leading to hard-to-find bugs.

It's extremely easy to avoid using magic numbers; just use the `#define` preprocessor instruction to define a meaningful symbolic name for the magic number, and **only** use the symbolic name in the program. That way, if you want to change the number's value, you only have to do it in one place (by changing the `#define` statement).

We will take marks off for using magic numbers in this assignment and in all subsequent assignments, so make sure you don't use them. **Specifically, the number "32" should only occur *once* in your entire program, inside a `#define` statement!**

Commenting

Don't forget to write good comments! From now on, we'll expect your commenting to be top-notch, unless (as in some later assignments) we supply you with a code template that already has the comments (in which case we'll expect that you've read and understood the comments, and that your code is consistent with them).

Specifically, we want to see:

- A comment at the top of the source code file explaining what the program does.
- A comment before each function (except for `main`) which explains what the function does, what its arguments mean, and what its return value represents (if it has a return value).
- Comments inside functions explaining how the code works, unless it's totally obvious. For this assignment, for instance, you'll want to explain how your sorting functions work. It's better to put this inside the function than before the function, because knowledge of how it works isn't usually important for using the function (*i.e.* it's not part of the function's interface, just its implementation).

You may think this is too much work; it isn't. The amount of time it takes you to write good comments is usually only a fraction of the time it takes you to write good code. In fact, many programmers advocate writing comments **before** writing code; the comments then serve as a kind of "scaffolding" around which you build your program. Writing the comments first forces you to think clearly about what you want to achieve, instead of just randomly writing code and stopping when it seems to work.

Testing your program

Many programmers can write working code, but fewer programmers test their code systematically to see if it actually *does* work. It's important to learn different strategies for doing this, because it can not only improve the quality of your code, but also the quality of your life. If you don't believe me, that's because you haven't spent enough four-hour+ sessions trying to track down a single bug caused by a typo somewhere in your program. [4] What we want to do here is see how we can make it happen less often.

The joy of `assert`

Using `assert` is probably the single easiest way to improve the quality of your code. The idea behind `assert` is to make your program *self-checking*. Let's say you've just coded up an incredibly hairy algorithm which will balance the budget, cure cancer, send mankind to the stars and do other wonderful things. How do you know that your code doesn't have a bug? And if it does have a bug, how do you find it?

Well, there are probably points in your program where you expect certain things to be true. For instance, in the sorting program described above, after you've completed the sort, you expect that the array of integers is, in fact, sorted. If you could check this right after you did the sort, then if your sorting algorithm ever fails on any input, you will know about it right away. `assert` makes this easy. Using `assert` looks like this:

```c
int i;

/* your algorithm goes here... */

assert(i == 10);  /* assert that the integer variable i is equal to 10. */
```

In the above case, if the program reaches the `assert` statement and `i` is equal to `10`, the program continues on executing; the `assert` statement has no effect. However, if `i` is not equal to `10`, the program aborts (exits), telling you the exact line in the file where the error occurred. "Whoa!", you're probably saying, "that's a bit extreme, isn't it?" Well, the point is to use `assert` for situations where you *know* something has to be true assuming that your algorithm is correct. If it isn't true, your program has failed, and all bets are off. (In actual fact, it's fairly easy to write custom versions of `assert` that don't abort but just print a nasty message and continue.) The `assert` statement is referred to as an *assertion*, logically enough.

You might also be asking: why don't I just write this:

```c
int i;

/* your algorithm goes here... */

if (i != 10)
{
    abort();  /* or exit(), or some similar function */
}
```

instead of using that `assert` function? Well, aside from the fact that `assert` is more concise, and that `assert` prints out line numbers where the assertion failed, the big advantage of assertions is that they can be *switched off*. Even though assertions don't usually slow your code down appreciably, there is some cost, which goes up the more assertions you use. Let's say you've used assertions to debug your code, and you reach a point where you are highly confident that your algorithm is correct. At this point you may want your algorithm to run as fast as possible, and not waste time checking assertions that you are sure will always be true. In other words, you want to leave the assertions out in the compiled code, but you don't want to remove the lines with the `assert` statements in them in case you modify the code later and need to debug it again.

It turns out that by adding the magic word `-DNDEBUG` as an argument to `gcc` when you compile the C code, the assertions will be removed from the code before compilation, so your code will run at maximum speed.

Using `assert` in your program

To use `assert`, make sure you add this to the top of your file:

```c
#include <assert.h>
```

In your sort function(s), write this right at the end of the function:

```c
/* Check that the array is sorted correctly. */

for (i = 1; i < num_elements; i++)
{
    assert(array[i] >= array[i-1]);
}
```

where `num_elements` is the number of elements in the array and `array` is the name of the array. Notice that here we're putting the `assert` statement in a loop. This bit of code is technically called a *postcondition*; a postcondition is what is required to be true after a function has completed. [5] Here, the postcondition will cause an assertion failure if the array isn't sorted at the end of the sorting function, which is what we want.

Notice how the postcondition is just three lines of trivial code, as compared to the sort routines themselves, which will probably be considerably longer and trickier to write. This illustrates the general principle that it's much easier to check if something is right than to make it right in the first place. Assertions are easy to write; use them!

NOTE: It may seem wasteful to write the same assertion code at the end of both sorting functions, but that's where I want you to write it, and not *e.g.* in the `main()` function. That's because you will eventually be writing larger programs which span many files, and functions you write will be used outside of the file they are written in. It's important for functions to be self-contained as much as possible, and if the assertion checking is built in to a sorting function, then anyone using that function will automatically get the benefit of the assertion checking.

Running the test script

Now you get to know why you added the `-q` option to your program. The reason for the `-q` option is to make the program only print output if the assertion fails. We can use this to write a test script to check the program.

We are supplying you with a Makefile and a test script for your program. The test script is written in the Python (http://www.python.org) language, which you remember from CS 1. Python is available on all Linux systems. What you should do is to download the test script into the same directory that your assignment 3 C code is located in. (Use the "save page as" feature of your browser; don't try to cut and paste the code into a text editor, because it's very probable that it won't work properly if you cut and paste it.) When the file is in the correct location, make it executable by typing:

```
$ chmod +x run_test
```

where `$` is the terminal prompt.

What the test script does is to generate a series of random inputs to the program and run the program with the `-q` option. The test script is invoked by typing

```
$ make test
```

at the terminal prompt. The test script will tell you what it's doing as it proceeds. If your sort functions ever sort incorrectly, you will see the output of `assert`, telling you what line the failure occurred at. The test script will run the program using the minimum element sort as well as the bubble sort. If something else goes wrong (*e.g.* your command-line processing is faulty, or a core dump occurs), the test script will report an error, along with the program invocation that caused it. It will not tell you exactly what error occurred, but by re-running the erroneous invocation (by cutting and pasting into your terminal window) you should be able to figure it out yourself. If all goes well, the test script will report success.

Note that if there is a bug in your assertion code itself, all bets are off. However, as mentioned above, writing correct assertions is pretty trivial (especially since we've done it for you :-)).

Supporting files

- The Makefile.

  Be sure that all the command lines in the Makefile start with tabs (which they will if you save the link directly), or they will not work.

- The test script, called `run_test`.

  Once again, in order to get this to work, you have to do

  ```
  $ chmod +x run_test
  ```

  after downloading this file, in order to make it executable. From now on, we'll assume you'll remember this (and the `Makefile` tab rule above) for future assignments.

- The style checker. To invoke the style checker you can just type

  ```
  $ make check
  ```

  at the prompt (assuming that the Makefile is in the same directory).

To hand in

The `sorter.c` file. We will check to see that it compiles, passes the style checker and passes the test script.

References

- Darnell and Margolis, chapter 7.
- K&R, chapters 1 and 5.
- Any algorithms textbook *e.g.* Sedgewick, *Algorithms in C*.
- The Wikipedia page on bubble sort (https://en.wikipedia.org/wiki/Bubble_sort).

---

1. not to be confused with the `NULL` pointer, which is something completely different
2. It's not a syntax error, but it won't do what you want.
3. Technically, it isn't legal "ANSI C". It *is* legal in the more recent "C99" standard, but we are using the most portable form of C in this track, which is ANSI C.
4. Don't laugh; it happens all the time, and it happens more with C than with any other computer language (except maybe C++).
5. Some computer languages, such as Eiffel, have very sophisticated systems for checking postconditions (as well as *preconditions* and various kinds of *invariants*); these languages make it easier to write correct code than C does.

Experiment # 14:

CS 11 C track: Assignment 4

Table of Contents

Goals
Language concepts covered this week
Reading (optional)
On recursion
Program to write
  Description of the game
  Description of the program
  An Example
Supporting files
Testing your program
Hints
  Solving the problem
  Various other hints
  Coding style hints
To hand in
References

Goals

In this assignment you will write a program to play a solitaire game, as well as learn about recursion.

Language concepts covered this week

- arrays
- recursion

Reading (optional)

Read chapter 7 of Darnell and Margolis, and/or chapter 5 of K&R (especially section 5.3).

On recursion

In most programming languages, including C, it is legal for a function to call itself. This is known as *recursion*. Those who haven't seen recursively-defined functions before often find these functions confusing, as if some rule is being broken. In fact, each invocation of the function is a separate entity and contains its own arguments, local variables, and return value. Here is a simple program including a recursive function to calculate factorials:

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

/* prototype */
int factorial(int n);

int factorial(int n)
{
    assert(n >= 0);

    if (n == 0)
    {
        return 1;
    }
    else
    {
        /* Recursive call to the factorial function: */
        return (n * factorial(n - 1));
    }
}

int main(int argc, char *argv[])
{
    int n, f;

    if (argc != 2)
    {
        fprintf(stderr, "usage: %s n\n", argv[0]);
        exit(1);
    }

    n = atoi(argv[1]);
    f = factorial(n);
    printf("factorial of %d = %d\n", n, f);

    return 0;
}
```

Copy this to a file, name it `factorial.c`, and compile it with:

```
$ gcc -Wall -Wstrict-prototypes -ansi -pedantic factorial.c -o factorial
```

where `$` is the terminal prompt as usual. Run the program with various values. What happens when the input is greater than 12? Greater than 20? What happens when the input is less than zero? Why do think that might be the case? *NOTE*: You don't need to answer this in your submission; it's just to get you thinking.

A very important feature of a recursively-defined function is that there must be a *base case* to which all invocations of the function eventually reduce. In this example, the base case occurs when the argument to the `factorial` function is zero (giving the answer 1). The most common mistake in recursive functions is to omit the base case (or one of the base cases), leading to an infinite loop. It is also important to check that each recursive call of the function has an argument which is closer to the base case than the original argument (here, `n - 1` is less than `n`).

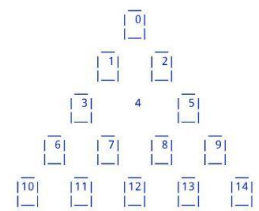
This example is somewhat contrived, because it is easy to calculate factorials without using recursion. However, for many problems, the recursive solution is much easier to come up with than any other solution. This assignment is an example of this.

Program to write

You will write a program to solve the *Triangle Game* which is described below.

Description of the game

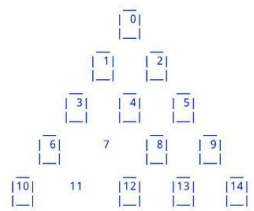
The Triangle Game is a solitaire (one player) board game played on a triangular board with fifteen equally-spaced holes in it. The diagram below shows how the holes are arranged and numbered.



Initially, fourteen of the holes have pegs in them, while hole 4 starts out empty. We denote holes with pegs in them by drawing a box around the number.

A peg can move by jumping over an adjacent peg which is then removed, just like in checkers. However, unlike in checkers, it is okay to jump horizontally as well as diagonally (vertical jumps are not allowed). Also unlike checkers, you can't just move pegs around; the only way to move a peg is to jump over an adjacent peg into an empty space.

For instance, starting from the initial board configuration, we could take peg 11, jump it over peg 7, and land in space 4. Peg 7 is removed, leaving us with the board configuration shown here:



We could continue by taking peg 9, jumping over peg 8, and landing in space 7. Notice that there is one fewer peg after every move, which means that the longest possible game has thirteen moves, removing thirteen pegs, which leaves only one peg remaining.

The object of the game is to make the game last as long as possible (thirteen moves). It might also be nice if you could get the final peg to end up in position 4, thereby preserving a sense of harmony, but as it turns out, that isn't possible. (In fact, the final peg will always be in position 12.)

It's convenient to use the term "board" to refer to a particular arrangement of pegs — for instance, we would say that the two diagrams above show two different boards. A move is said to be legal for a particular board if it complies with the rules described above. For instance, the move "jump 9 over 8 landing on 7" is a legal move for the second board shown above, but it is not a legal move for the initial board. The set of possible moves is just the set of all moves which are legal in at least one conceivable arrangement of pegs. So "jump 9 over 8 landing on 7" is a possible move. However, "jump 9 over 8 landing on 14" is not a possible move, because it is never legal.

Description of the program

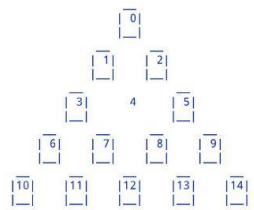
Your program should do the following:

- Get an initial board from the user. The board will be represented as a one-dimensional array of integers. At any location in the array, a value of 0 means that there is no peg at that location and 1 means that there is a peg at that location.
- Find out if there are a set of moves which lead to only having a single peg left on the board.
  - If no such set of moves exist, report this fact and exit.
  - If at least one set of moves exist, print out the board positions that the moves go through, starting with a single peg, and ending with the user-supplied board. (In other words, print out the solution, but in reverse order.)

In order to make this easier, and to allow you to concentrate on the problem at hand, we are supplying you with the input/output functions for the program. These functions are:

```
void triangle_print(int board[])
```

This function takes a `board` (an array of 15 integers), and prints it out in a visually-appealing way. For example, the starting board for this assignment gets printed as:



```
void triangle_input(int board[])
```

This function asks the user to create a triangle game starting position, which is put into the `board`.

To use these handy functions (and you really should use them), save the following files to the same place as your program will go:

- triangle_routines.h
- triangle_routines.c

You're not required to understand how they work, but by all means look through them if you're curious.

An Example

triangle_example.c is a short example of the use of these functions. Save it into the same directory as `triangle_routines.c`, and compile and run it like this:

```
$ gcc -Wall -o triangle_example triangle_example.c triangle_routines.c
$ triangle_example
```

When you run it will simply ask you for a board and print out the board you entered.

A couple of things to notice in the example:

- You need to have the following line with your other `#include`. Note the quotation marks — that means that the preprocessor won't look in the standard location for the header file (which is `/usr/include` on Linux and MacOS) but instead will look in the current directory.

```
#include "triangle_routines.h"
```

- Compile your program like this:

```
gcc -Wall -o triangle_game triangle_game.c triangle_routines.c
```

  Alternatively, you can use the Makefile we supply. Type `make triangle_example` to make the example program, and `make triangle_game` for the game itself. If you have any trouble with this, ask your instructor or TA.

We strongly encourage you to use this example as a starting point for your program.

Supporting files

All of these should be downloaded to your `lab4` directory, except for the style checker, which you should already have in your `~/bin` directory if you've followed the instructions in the style guide.

- The Makefile
- triangle_routines.h
- triangle_routines.c
- triangle_example.c
- The style checker
- A test input file

Testing your program

You can test your program by running it directly from the command line, or by typing `make test`. This will use the `test_input` file as the input to your program. This file loads all the pegs except the central one, and the resulting board has a solution, so it should work.

Hints

This is a more difficult program than the previous ones in this track, so here are some hints.

Solving the problem

Your board-solving function should take one argument: an array representing a board. It should return `1` (true) if the board can be solved, and `0` (false) if the board cannot be solved. Other than that, it should not change the board. It's OK if it changes the board temporarily, as long as it changes it back to its original state before returning.

To check if the board is solvable, it's easiest if you use a recursive algorithm. One way to use recursion to solve a board would be to proceed as follows:

- First, identify the base case. What kind of board can you tell immediately is solvable? (*Hint*: what kind of board is *already* solved?)
- Second, let's say that the base case doesn't apply. Then you can make each legal move, recursively check if the resulting board (board #2) is solvable, and then unmake the move you just made. If the recursive call told you that board #2 was solvable, then you know that your original board is also solvable because you can make a single move, convert it into board #2, and solve that. If this is the case, you're done and you can return. If not, keep trying more moves.
- If you never find a successful first move, then the board is unsolvable.

In order to know what moves are allowed, you need to come up with a representation of the move. One way is to represent a move as an ordered triple of numbers. For instance, in the initial position we can take a peg at position 11, jump over position 7 and land in position 4. This can be represented as the triple: 11, 7, 4. Similarly, we can enumerate all possible moves (there are 36 of them), and stuff them into a global array:

```c
#define NMOVES 36

int moves[NMOVES][3] =
{
  {0, 1, 3},
  {3, 1, 0},
  {1, 3, 6},
  {6, 3, 1},
  /* ...  (lots more, total of 36) */
  {12, 13, 14},
  {14, 13, 12}
};
```

This uses C's array initialization syntax (see lecture 3, or Darnell and Margolis sections 7.3 and 7.11, or K&R sections 4.9 and 5.7). Then you can iterate through the array, looking for moves which are legal given the current board position. When you find one, make the move and recursively call the function on the resulting board. If the move does not lead to a solution, un-make it and continue. Once there is only a single peg on the board, you have found a solution.

If there's a solution, you're required to also print out the solution in reverse order (*i.e.* starting with a board with only one peg and growing by one peg each time until the starting board is printed). This seems really hard but in fact is really easy. All you need to do is (in the board-solving function) to print out the board before any place where you return 1 (a true value *i.e.* success). Because of the way the recursive function works, this will end up printing out the successful solution in reverse order. Normally, it's bad design to print out something in a function whose main job is to do something else (like solve a board in this case); however, here we're mainly using the board printing as a debugging aid, so it's OK.

Don't print the board in any function except for the `solve` function. If you think you have to print it somewhere else as well, keep thinking.

Various other hints

Here are the function prototypes we used for our solution, along with comments stating what they do:

```c
/* Return the number of pegs on the board. */
int npegs(int board[]);

/* Return 1 if the move is valid on this board, otherwise return 0. */
int valid_move(int board[], int move[]);

/* Make this move on this board. */
void make_move(int board[], int move[]);

/* Unmake this move on this board. */
void unmake_move(int board[], int move[]);

/*
 * Solve the game starting from this board.  Return 1 if the game can
 * be solved; otherwise return 0.  Do not permanently alter the board passed
 * in. Once a solution is found, print the boards making up the solution in
 * reverse order.
 */
int solve(int board[]);
```

Note that the `move[]` arguments in the above function prototypes don't refer to the entire 2-dimensional moves array, but to a single move *i.e.* a one-dimensional array of 3 integers.

The `solve` function is a bit tricky (it's the recursive one); the rest are quite straightforward. When we say "do not permanently alter the board passed in" what we mean is that the function can change the board, but it should undo the change before it exits. When you think about it, this makes sense; all the function really has to do is to figure out whether the board is solvable or not (which doesn't require the board to be altered when it exits) and to print out the solution if it finds one (ditto). The `unmake_move()` function will be useful to ensure that the board doesn't get permanently altered before the `solve` function exits.

Even after you find a solution, make sure you unmake the board to its original state so it hasn't changed when the `solve` function returns.

As we mentioned above, the fact that the `solve` function prints the solution after it finds it is unusual, and would normally be considered bad design. Normally we would store the solution somewhere, and print it in another function. That can be done, but it's quite a bit harder. To make this easier, you just have to print the solution immediately, and you don't have to store any old boards anywhere. That's also the reason for the reverse order; it makes it possible to print the boards without having to store them. To be absolutely clear, by "reverse order" we mean that the solved board (with one peg) is the first thing you print, followed by the previous board (with only two pegs), and so on up to the full board, with *e.g.* 14 filled pegs and one empty peg. Of course, use the `triangle_print()` function to print the board. Make sure you print the original board!

We recommend that you concentrate on solving the board first, and only after that worry about printing the boards in the solution. It won't require much extra code.

You should also realize that it's OK to use one-dimensional array components of a two-dimensional array. So the first move in the array is moves[0], which in the above example stands for the array `{0, 1, 3}`. A lot of students copy the move to a new array, but that isn't necessary.

Coding style hints

Don't use any global variables for this program except for the two-dimensional array of moves! It's OK for the moves array to be a global variable because it's really a global constant; you never change it. Global variables that get changed are occasionally useful too, but more often they just make your program much harder to debug, so normal practice is to avoid using them if possible. None of the assignments in the C track require global variables other than this one, and this one's only global variable is the moves array.

Don't use magic numbers! Don't put numbers like 36 or 15 in your code; use `#define` to create symbolic names for them and use the symbolic name instead. That's good programming style.

To hand in

Your completed program `triangle_game.c`.

References

- Darnell and Margolis, chapter 7.

- K&R, chapters 4 and 5.