```c
/*
Filename: tutorial.c
Author:                 Kevan Thompson
Date Created:   August 28, 2012
Date Update:    August 30, 2012

Description:

        This is a very basic tutorial on C. If you have any
        questions about any keywords, functions, or operators
        vast amounts of information can be found easily
        online.

        A note of programming style. Every program you write
        should begin with a programmers block similar to this one.
        Every function should have a description similar to the
        ones used in this program. You should also attempt to
        layout your code in a similar manner to this program.
        Properly indenting your code isn't really optional.
        It makes code legible and easier to debug. Also
        don't forget to comment your code. This isn't just
        nitpicking by your professor/TA's. If you follow this
        advice it will greatly help you (and your group
        members) to understand and debug your code.

        To compile use the following command:

        gcc -lm -o tutorial tutorial.c


*/

//----------------------------------------------------------
//INCLUDES
//----------------------------------------------------------

//This is where you tell the compiler to
//include other libraries in your program

#include <stdio.h> //scanf(), printf()
#include <math.h> //log10(), ceil()
#include <stdlib.h> //malloc(), free()

//Remember if you make your own header files to use
// " " and not < >

//----------------------------------------------------------
//DEFINES
//----------------------------------------------------------

//This is where you can define preprocessor directives
//At compile time, the compiler will replace these
//place holders in your code with the actual value

#define TRUE 1==1
#define FALSE !TRUE

//----------------------------------------------------------
//STRUCTS
//----------------------------------------------------------

        //This is where we can define structs

        typedef struct{
                int real;
        int imag;
        } num;


//----------------------------------------------------------
//FUNCTION PROTOTPYES
//----------------------------------------------------------

//This tells the compiler how to use your functions
num addcomplex(num a, num b);
void addcomplexPtr(num * a, num b);

//----------------------------------------------------------
//MAIN
//----------------------------------------------------------

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

        //The first thing we do when writing any function
        //is define our variables.

        //Basic Variables:
        char a; //This is a char. It holds a single ASCII character, which is
                        //also 1 byte
        int b;  //This is an int. The size of an int depends on the compiler, and
                        //the architecture of the system. It is usually 4 bytes
        //We can instantiate more than one variable in a line and also
        //provide an initial value
        int ans, root, c, i = 0;
        char count, last;
        //Arrays:
        int arr[50];    //This is a basic array of ints.
                                        //We can access each member with an
                                        //offset starting at zero.
        int arr1[] = {0,1,2,3}; //This creates and initialized an array
                                                        //with values 0,1,2,3

        //In C strings are stored as arrays of characters
        //To denote the end of a string we use a special character
        //called the string delimiter \0
        char str [] = {'H','e','l','l','o',' ','W','o','r','l','d','\0'};


        //Pointers and Structs:
        int *ptr;       //This is a pointer to an int. The value stored in ptr is the
                                //address of an int. In this case ptr isn't initialize
d
                                //so it doesn't point to anything
        num num1, num2, num3;           //This is a struct of type num.
        num * numPtr;   //This is a pointer to a struct of type num
        //There will be additional examples of pointers later

        //Basic I/O in C
        //In C++ we often use cin, and cout to communicate to the user
        //(using the standard in/standard out streams) in C we can use
        //printf(), and scanf()

        //Example
        printf("Hello World \n"); //the character /n is used to add a new line.
        //We can also print variables
        printf("The third value off array arr1 is %d \n",arr1[3]);
        printf("%s\n",str);

        printf("Please enter a non-negative integer: ");
        scanf("%d",&root); //Note the use of the & to pass an address
        ans =(int) ceil(log10((double)root)/log10((double)2));
        printf("It takes %d bits to store %d values \n\n",ans,root);

        //For additional practice try writing a simple program that
        //prints out witty comments to a user, or displays some ascii art.

        //Basic Operations
        //These include + (add), -(subtract), *(multiply), /(divide), %(modulo)
        printf("Basic Operations:\n");
        //Assignment operator:
        a = 1;
        b = 3;
        //Addition:
        c = a + b; //Basic addition
        printf("%d = %d + %d \n",c,a,b);

        //Subtraction:
        c = a - b; //Basic subtraction
        printf("%d = %d - %d \n",c,a,b);

        //Multiplication:
        c = a * b; //Basic multiplication
        printf("%d = %d x %d \n",c,a,b);

        //Division:
        c = b / a; //This is basic division. The quotient of b/a is assigned to c
        printf("%d = %d / %d \n",c,b,a);

        //Modulo:
        c = a % b;      //This is the same as division. Except the remainder of b/a
                                //is assigned to c instead of the quotient
                                //The modulo operator can be extremely useful
                                //ie when implementing a circular buffer

        printf("%d = %d %% %d \n",c,a,b);       //Note the %% to print a single %
                                                                //this
 is needed because % is a
                                                                //spec
ial character

        //Bitwise Operators
        //These include << (left shift), >> (Right Shift) =, & (and), | (or)
        printf("Bitwise Operations: \n");
        //Shift left
        c = a << b;     //This shifts the bits of a and assigns the result to c
        printf("%d = %d << %d \n",c,a,b);
        //Note each shift is equivalent to a multiplication by 2


        //Shift right
        c = b >> a;     //This shifts the bits of a and assigns the result to c
        printf("%d = %d >> %d \n",c,b,a);
        //Note each shift is equivalent to a division by 2

        //Logical AND
        //Example
        a = 0xFFFF;     //Sometimes it is useful to as sign a value in hex
                                //we can do this using 0x
        //Lets set bits 0, 3, and 5 to zero
        b = 0xFFD6;
        c = a & b;
        //To make it easier to see lets print these variables as hex values
        printf("%04x = %04x & %04x \n",c,a,b);
                //Logical AND
        //Example
        a = 0;
        //Lets set bits 1, 2, and 4 to zero
        b = 0x16;
        c = a | b;
        printf("%04x = %04x | %04x \n",c,a,b);

        //Pointers, Arrays, and Structs
        printf("Pointers, Arrays, and structs:\n");

        //We can treat array indexes like regular variables
        arr1[1] = arr1[2] + arr1[3];
        printf("%d = %d + %d \n",arr1[1],arr1[2],arr1[3]);
        //Remember it's up to you to make sure you don't try to
        //access an elemet outside the array


        //Pointers are similar to arrays in that they point to
        //variables infact the name of an array is actually
        //just a pointer to the first element in the array

        //lets point our pointer at something
        ptr = &b; //Remember in this context & means address
        printf("b = %d, ptr = %d \n", b, *ptr);

        //In order to change the variable pointed to by the pointer
        // we need to use the dereference operator (*)
        *ptr = 1234;
        printf("b = %d, ptr = %d \n", b, *ptr);
        //notice that b changed value. This is because ptr points to b
        //and not a copy of b.

        //We can also treat a pointer like an array
        ptr = arr1;
        printf("*(ptr + 2) = %d, ptr[2] = %d \n", *(ptr+2), ptr[2]);

        //For additional pratice try to implement a circular buffer using just
        //an array, and 2 pointers or indexes

        //For more advanced topic look up multi dimensional arrays [][]
        //and how to use pointer to pointers

        //Structs are like objects in C++ except that they don't have methods
        // Alternativly you can think of them like variables within a variable
        //Remember the struct we defined before? It has to members:
        //      typedef struct{
        //      int real;
    //    int imag;
        //} num;
        //You can access these members using the dot operator

        num1.real = 1;
        num1.imag = 1;
        printf("num1.real = %d, num1.img = %d \n", num1.real, num1.imag);

        num2.real = 2;
        num2.imag = 2;
        printf("num2.real = %d, num2.img = %d \n", num2.real, num2.imag);


        //We can also make a pointer to a struct
        numPtr = &num1;
        //now we need to dereference the point. However the dot operator
        //has a higher priority than the dereference operator
        (*numPtr).real = 2;
        //However this is a pain in the butt. So instead we can use
```

```c
        //the -> operator
        numPtr->imag = 2;
        printf("num1.real = %d, num1.img = %d \n", numPtr->real, numPtr->imag);

        //Pointers and functions
        printf("Pointers and Functions:\n");
        //When we pass values to a function we can either pass a copy of a
        //variable or we can pass a pointer to a variable. This is analogous
        //to you asking for a cup of coffee. I can either get you the coffee,
        //in which case my coffee machine is safe if you spill the cup.
        //Or I can tell you where the coffee machine is and risk you
        //damaging it. When you pass a ptr the function can and
        //probably will change the value in the variable. Destroying
        //what was originally there.

        //Lets look at some examples using our structs. This first
        //function does not take pointers for inputs.
        num3 = addcomplex(num1, num2);

        printf("%d = %d + %d \n",num3.real, num1.real, num2.real);
        printf("%d = %d + %d \n",num3.imag, num1.imag, num2.imag);
        //Notice how the values in num1 and num 2 didn't change
        //Now try a different function that takes a pointer
        //as an arguement
        addcomplexPtr(&num1, num2); //remember we can use & to pass an address
        printf("num1.real = %d, num1.img = %d \n", num1.real, num1.imag);
        printf("num2.real = %d, num2.img = %d \n", num2.real, num2.imag);
        //Notice how the answer was returned in num1!

        //For additional pratice try to make a struct that stores the values
        //polar form. And create new functions that use this struct.
        //Useful sin, cos, and tan functions can be found in math.h

        //Control Statements (if,else,while,for,case)
        printf("Control Statements:\n");
        //So far our program has executed every piece of code that we've written
        //But sometimes we want code to execute under certain conditions.
        //Or we want code to execute multiple times

        //The most basic control statment is if. We can use it to execute code
        //under certain coditions using control operators:
        // == (equal), != (not equal), > (greater than), < (less than),
        // >= (greater than and equal), <= (greater than and equal)

        //And using logical operators:
        // && (logical and), || (logical or), ! (not)

        //Remember the value we took in for our scanf example.
        //Let's see if it's even or odd

        if((root%2) == 0){
                printf("The value you entered was even! \n");
        }else{
                printf("The value you entered was odd! \n");
        }
        //We can combine many if/elses together to form an
        //if-else chain:

        /*
                if(something){

                }else if(something else){

                }else{
		}
	*/
	//You can also put an if within an if to make a nested
	//if statement

	/*
		if(something){
			if(something else){

			}
		}
	*/

	//Sometimes a long if-else chain is inefficient
	//and it is better to use a case statement:

	switch(root%2){
		case 0:
			printf("The value you entered was even! \n");
		break;
		case 1:
			printf("The value you entered was odd! \n");
		break;
		default:
			printf("The value you entered was even! \n");
		break;
	}

	//For additional practice write a simple program that
	//takes in a resister value, and prints out its colour
	//code.

	//If we want to rerun a peice of code multiple times
	//we can use while, do, and for loops

	//The while loop is the simplest of these loops
	//Note: if the conidition isn't initially met
	//then the code never executes

	/*

		while(some condition){
			execute some code
		}
	*/

	// do-while loops are similar, except the code always
	//executes atleast once.

	/*
		do{
			execute some code
		}while(some condition);
	*/

	//For loops are slightly more complex
	//Like while loops they include a condition statement
	//but they also include a vaiable declaration,
	//and some form of operation

	//Example, lets use a for loop to find the max size of a
	//char

	for(count = 0; count >= 0; count++){
		last = count;
```

```c
        }
        printf("The maximum value of a char is: %d \n",last);

        //For additional practice try this again with an int
        //How long will it take to run? Should you change its
        //starting value so it'll finish in your life time?

        //Dynamic Memory Allocation
        printf("Dynamic Memory Allocation:\n");
        //When we instatiate a variable at the beginning of our
        //program, it's size in memory is fixed. But sometimes
        //we don't know how much memory we'll need until
        //runtime. Or we might have a data structure that
        //grows in size. In this case we can use Dynamic
        //Memory Allocation

        //To allocate memory we can use the function malloc()
        //Lets use malloc to create an array of 4 ints
        ptr = (int *) malloc(sizeof(int)*4);
        //malloc returns 0 if there is an error
        if(ptr != 0){
                ptr[0] = 0;
                ptr[1] = 1;
                ptr[2] = 2;
                ptr[3] = 3;
                for(i = 0; i < 4; i++){
                        printf("ptr[%d] = %d \n",i,ptr[i]);
                }

                //we are done we can de-allocate it using free
                free(ptr);
        }else{
                printf("Error could not allocate memory!! \n");
        }

        //For additional practice try to impliment a
        //linked list in C.

        //Command Line Arguements
        printf("Command Line Arguements:\n");
        //Try running this program again as follows:
        // ./tutorial Hello World
        //Congratulations you've now passed an arguement to your
        //program using the command line.

        //Did you notice the paramaters beside main?
        //int main(int argc, char *argv[]){
        //These are used for command line Arguements
        //argc tells you how many were passed to the program
        //and argv is an array of strings that contains the
        //values passed.

        //lets print them out
        for(i = 0; i < argc; i++){
                printf("Arg %d is %s \n",i,argv[i]);
        }
        //Notice that the first arguement is the name of the program

}

//Functions:
//If we want to reuse code we can always right our
//own functions. Remember while a function can take in
//any number of arguements. It can only return one.
```

```
//---------------------------------------------------------
//Function Name: addcomplex()
//Author: Kevan Thompson
//Date Created:
//Date Update:
//Inputs: num a, num b
//Outputs: num c
//Purpose: Adds complex number a to complex number b
//                      and returns the result in c
//---------------------------------------------------------

num addcomplex(num a, num b){
        num c;
        c.real = a.real + b.real;
        c.imag = a.imag + b.imag;
        return(c);
}

//---------------------------------------------------------
//Function Name: addcomplexPtr()
//Author: Kevan Thompson
//Date Created:
//Date Update:
//Inputs: num *a, num b
//Outputs:
//Purpose: Adds complex number a to complex number b
//                      and returns the result in a
//---------------------------------------------------------

void addcomplexPtr(num * a, num b){

        a->real = a->real + b.real;
        a->imag = a->imag + b.imag;
}
```