englinks

MNTC 313 Midterm Workbook

2020-2021

Written by Carson Cook and Benjamin Beggs

Edited by Nayana Menon

Comments

Comments are used to explain what your code does when it is not obvious. They are little notes that are not actually executed. In C, anything on the same line and to the right of '//' is a comment and won't be executed. Also, you can comment large chunks with '/*' and '*/'. Anything between the /* and */ is commented out, even when they are on different lines.

```
//this will not be executed
/*neither will
this*/
```

Printing

In order to show output from a program, **printf()** will output words and variable values.

```
printf("Hey there! The year is %d. Welcome to Queen's, sci %d!", 2017, 21);
//the '%d' means that an integer is going to be subbed in that place, which
//is provided by the 2017 and 21
//output: Hey there! The year is 2017. Welcome to Queen's, Sci' 21!
```

Any number of values can be added to *printf(),* as long as there is a matching specifier that a value will be subbed in. Multiple values are subbed in for indicators in the order they appear.

Specifier	Type	Example Value
%d	Integer	23
%i	Integer	23
%0.xf	Decimal value with x decimal places	23.45 (%0.2f)
%e	Scientific notation	2.3e+1
%%	Char – a single '%'	%

Question 1: Predict the output

```
int k;
float x = 13.7;
k = x;
printf("%d, %0.1f\n", k, x);
k = k/3;
x = k*3;
printf("%d, %0.1f", k, x);
```

Preprocessor Directives

#include

This command includes code from other files; it essentially copy pastes the code once the program is compiled.

```
#include <file.h>  //the '<>' means the file comes with the regular library
//from C
#include "file.h"  //the "" means the file is a custom one
```

Custom files must be in the same directory as the file including it. These files are called header files.

#define

This command is similar to creating a variable. You can assign any value (even a function!) to a given word:

```
#define SIZE 6 //in your code you can use SIZE in place of a 6 - it is
//constant and cannot be changed
```

Errors

Syntax

Syntax errors are essentially typos. These errors will stop your code from compiling, usually with a message about the issue and where it is, making these errors easier to find and fix. For example, typing 'fucntion' instead of 'function', or forgetting to add a closing '}' after an opening one.

Logic

Logic errors are much harder to fix. These errors won't stop a programming from executing, but at some point the program will exhibit weird behaviour. There won't even be a message (unless the error causes the program to crash), so you have to go through the code manually. To fix logic errors, you need to look at the program line by line and realize what is actually being executed. You cannot assume any area works as expected and you must be very granular in your analysis. A great way to avoid doing this is to explain what each line of code does to an inanimate object; this way you will understand exactly what each line does. These are also called bugs.

Common errors

Syntax

- After *#include* < > or *#define SIZE 6*, do not put a ';'
- Everything is case sensitive. *Function* is different from *function*
- There is usually no ';' before a '{', even if the '{' is on a new line
- If a function has a non-void return type, it must return something
- Functions cannot have the same name as a variable in the same scope

Logic

- When comparing equality, use '==' not '='. '=' will assign a value and always result in true
- ^ does not computer a power. To find 2^3, use *pow(2,3);*. You will need to do *#include <math.h>*
- & and | perform different operations than && and ||
- The first element in an array is at position 0
- Ensure your *while* loop condition will change. Often a counter is forgotten and it loops forever
- Often *int* and *float* conversions are missed. If you want 5/2 to been seen as 2.5, you must do *float x = (float)5/(float)2;* OR *float x = 5.0/2.0;*
- Order of operations in a chain of logical operators gives preference to brackets first, then AND, then OR. For example, *3<4 || 2==0 && 5>4* is really *3<4 || (2==0 && 5>4)* which is *true || false*

Variables

A variable holds data that can be referenced using the name of the variable and can be changed at any time. It's similar to algebra: y=3x+7. In programming, that would simply be:

```
int x = 6; //x was given a value, the x's value can be referenced
int z; //z was not given a value, but because it hasn't been referenced yet,
//that is fine
int y = 3*x+7; //x's value was referenced
```

One thing you might notice however is that unlike algebra, a variable must be given a value before that value can be used. If not, then a random value will be used, or the program will crash. At best, unexpected behaviour will happen, unless you are VERY sure in what you're doing.

Note: In MATLAB, you do not need to declare variable types, only the variable name and its value. Thus, if you wanted x to have a value of 6, you would only need to declare `x = 6`, not `int x = 6`.

Types

Type	Explanation	Example Value
int	Any integer value, between –(2^15)+1 and 2^15-1 .	4, 5, -1023
long	Any integer value between –(2^31)+1 and 2^31-1.	4, 5, -1023
float	A decimal value between –(2^15)+1 and 2^15-1.	5.24, -12.532
double	A decimal value between –(2^31)+1 and 2^31-1.	5.24, -12.532
char	A character. Denoted by ' '. Every character has a corresponding integer value: www.asciitable.com	1, a, 97
boolean	A Boolean, or bool, is a true or false value. In the standard C library, there is no actual Boolean type; a true value is represented by a 1 and false is 0.	true/false, or 1/0
unsigned int	Any positive integer value, between 0 and 2^16-1. "unsigned" can also be applied to char, short, and long.	4, 5

Typedef

A typedef is used to make a short form for a variable type.

```
unsigned int x; //fine, but lots of typing if using unsigned int often
typedef unsigned int uint; //unsigned int is its own type, but now uint can
//be used instead
uint y; //same as the first line, but less letters!
```

Type conversion

Mixing variable types can be dangerous as unpredictable results can happen. For example, assigning a float value to an integer value will cause the decimals to be 'chopped off'. When mixing data types, types that use smaller memory than an integer (for example a char) get converted into integers. If a type being mixed takes more memory than an integer, the following hierarchy is used:

int<unsigned int< long< unsigned long< long long < unsigned long long < float< double < long double

Types can also be changed by casting: '**int** x = (**int**)43.25;'. Here, 43.25 will be converted into an integer, so x will hold the value 43. Be careful when using this, as not all conversions will work.

Question 2: Predict the values

```
int result = 18/5;
```
result =

```
float result = 18.0/5;
```
result =

```
int i = 17;
char c = 'c'; //ASCII value of c = 99
int sum;
sum = i+c;
```
sum =

```
int a;
float b;
a = 2;
b = (float)a/3;
```
b =

```
int a, b, c, d;
float e;
a = 5;
b = 3;
c = 1;
d = a*b + c/b + b;
e = (float)c/b +b;
```
d =

e =

Arrays

Arrays are used to store multiple objects of the same type, so you can store related values in one variable. For example, one character is not as useful as a whole sentence, but you don't want to declare so many char variables. This is commonly called a string, which in C is just an array of char variables. An important note is that the position of the first element in an array is at 0, NOT 1.

```
int array[3] = {1,2,3}; //declares an array of integers with 3 values, set to
//be 1, 2 and 3
int arr[3]; //declares an array of 5 integers, but you haven't set the values
//yet (automatically set to 0)
arr[0] = 1; //sets first position to have value 1
arr[1] = 2;
arr[2] = 3;
//array and arr currently are the exact same!
arr[3] = 4; //ERROR - there is no arr[3]!

//(Although the line above might work sometimes, its undefined behaviour,
//unreliable, and will likely be marked wrong on an exam)
```

2-D arrays are very similar to 1-D arrays, except it can be thought of as more like a matrix or a table. Each row holds its own array.

	values[x][0]	values[x][1]	values[x][2]
values[0][x]	1	4	6
values[1][x]	5	7	9
values[2][x]	2	8	0

```
int values[3][3] = {{1,4,6},{5,7,9},{2,8,0}}; //declaring above values matrix
//values[0] is the array {1,4,6} (you can't actually access it, this is just
//theoretical)
int arr[2][2];
arr[0][0] = 4; //must specify both positions when assigning
arr[1][2] = 5; //etc.
```

Operators

Arithmetic Operators

Arithmetic operators are used to change number values. They follow BEDMAS order of operations.

Operator	Explanation	Example

+	Adding two numbers, two variables or a number and a variable	int y = 1 + 2; //y = 3
		int y = x + z;
		int y = x + 1;

-	Subtracting two numbers, two variables or a number and a variable	int y = 3 - 1; //y = 2
		int y = z – x;
		int y = x - 1;

/	Dividing two numbers, two variables or a number and a variable	int y = 9/3; //y = 3
		float y = x/z;
		float y = x/2

*	Multiplying two numbers, two variables or a number and a variable	int y = 5 * 4; //y = 20
		int y = x * z;
		int y = 5 * x;

%	Finding the remainder of a division. For example, 9/4 has a remainder of 2, so 9%4=2	int y = 9 % 3; //y = 0
		int y = x % z;
		int y = x % 3;

Assignment Operators

Operator	Explanation	Example

=	Sets the variable on the left to be the value calculated on the right.	y = 3;
		y = x;
		y = y + 3; //y becomes the previous value of y plus 3

++	Sets the variable to increase by one	++y; //essentially y = y + 1
		y++; //same as above

--	Sets the variable to decrease by one	--y; //essentially y = y -1;
		y--; //same as above

+=	Adds the value onto previous value of variable, then assigns variable to that new value
	Note: MATLAB does not have this operator. In MATLAB, you would have to type out `y = y+5` in order to achieve the same result. The same applies to the above operators
		y+=5; //same as y = y + 5
		y-=5; //y = y – 5
		y *=5; //y = y * 5
		//etc.

Relational

Relational operators compute a Boolean result.

Operator	Explanation	Example
==	True if the values are equal	5==5; *//true* 4==5; *//false* x==y; *//true if x and y have same value* x==5; *//true if x has value of 5*
!=	True if the values are not equal **Note:** In MATLAB this operator is ~=.	5!=5; *//false* 5!=4; *//true* x!=y; *//true if x and y have different values* x!=5; *//true if x has a value other than* 5
>	True if left side is bigger than right side	5>4; *//true* 4>5; *//false* x>y; *//true if x has larger value than y* x>5; *//true if x has larger value than 5*
>=	True if left side is bigger or equal to the right side	
<	True if left side is smaller than right side	
<=	True if left side is smaller than or equal to the right side	

The '==' operator works for almost all variable types, but the others usually need to be used in with number types such as integers or floats.

Logical

Logical operators are used to combine Boolean values into a larger expression that also outputs a Boolean. These operators can be chained together themselves as well.

Operator	Explanation	Example
||	OR. If either value is true, the result is true.	(**true**||**false**) *//true* (**false**||**true**) *//true* (**true**||**true**) *//true* (**false**||**false**) *//false*
&&	AND. If both values are true, the result is true.	(**true** && **false**) *//false* (**false** && **true**) *//false* (**false** && **false**) *//false* (**true** && **true**) *//true*
!	NOT. Reverses the Boolean value. **Note:** In MATLAB this operator is ~.	!(**true** && **false**) *//true* !(**false** && **true**) *//true* !(**false** && **false**) *//true* !(**true** && **true**) *//false*

Conditionals

Conditional statements are used to execute code only when a give Boolean expression evaluates to true.

if Statements

```
if (x==y)  //code between '{' and '}' only executes if the condition between
//'(' and ')' is true
{
      printf("hi"); //so, hi is only printed if x has the same value as y
}
```

Question 3: Predict the output of the code below

```
if (5>4)
{
printf("5 > 4");
}
if (5<4)
{
      printf("5<4");
}
```

Along with **if** statements are **else if** and **else** statements. These require a leading *if* statement. If the first statement is false, then the *else if* statement is evaluated, if that one is also false then the next *else if* is evaluated and so on. An *else* statement will execute only if all above *if/else if* statements evaluated to false. These are called if chains.

```
if (x==y)
{
      printf("hi");
}
else if (x==4)
{
      printf("hello");
}
else
{
      printf("howdy");
}
//if x==y, we will ONLY see 'hi'. If x!=y, if x==4, we will see 'hello'. If
//x!=y AND x!=4, we will see 'howdy'
```

You can also nest **if** chains:

```
if (x==y)
{
      printf("hi");
      if (x==4)
      {
            printf("hello");
      }
}
//if x==y we will see 'hi'. If x==y AND x==4, we will see 'hi' AND 'hello'
```

In MATLAB, the commands for conditional statements are the same, but no parentheses **( )** or curly brackets **{ }** are required. Statements that are chained use **elseif** instead of **else if**.

Switch Statements

Switch statements can be used in place of a bunch of if chains. A variable is used and matched to a value in a case statement. If the values match, the code underneath that case is executed. Usually, cases end their code with a break statement; as otherwise, the next case's code is also run. There is a special case called default which is executed if no other cases match the given variable's value.

```
switch (x)
{
      case 1:
            printf(“hey’);
//no break statement, so case 2 will execute if case 1 is triggered
      case 2:
            printf(“hi”);
            break; //avoids executing below code if this case is triggered
      case 3:
            printf(“howdy”);
break;
      default: //gets executed if x matches no cases
            printf(“hello”);
}
//if x is 1, we see ‘hey’ AND ‘hi’, if x is 2 we just see ‘hi’, if x is 3 we
//just see ‘howdy’
//if x is NOT 1, 2 OR 3, we see ‘hello’
```

If chains and switch statements can be interchanged. For example, here is the above switch statement in the form of an if chain.

```
if (x==1)  //replaces case with no break statement
{
      printf(“hey”);
      printf(“hi”);
}
if (x==2) //replaces first case with a break statement
{
      printf(“hi”);
}
else if (x==3) //replaces cases that are separated by break statements
{
      printf(“howdy”);
}
else //replaces the default case
{
      printf(“hello”);
}
```

In MATLAB, the commands for switch case statements are the same, but like if-else statements, no parentheses or curly brackets are required. There is also no colon **:** after the **case** command and no **break** command after each case statement. The final statement also uses **otherwise** instead of **default.**

Loops

Loops are used to rerun code while a given condition is true.

int arr[5];
//left of first ';' is the variable used to count the number of loops
//between the two ';' is the condition for when the loop stops
//right of second ';' is the operation to do on the loop variable
for (**int** i = 0; i<5; i++) *//i starts at 0, then will be 1, then 2…when i<5 is false (i equals 5), loop will stop*
{
	arr[i] = i;
}
//result is arr is {0,1,2,3,4}

Question 4: Predict the output of the code below

```
int nums[5] = {1,7,5,9,6};
int highsum = 0;
int lowsum = 0;
for(int j=0; j<5; j++)
{
	if(nums[j]<=5)
	{
		lowsum += nums[j];
	}

	else
	{
		highsum += nums[j];
	}

}
	printf("low sum = %d", lowsum);
	printf("high sum = %d", highsum);
```

While Loop

```
int arr[5];
int i = 1;
while (i<5) //if the condition here is true, the code executes until the
//condition is false
{
	arr[i] = i;
	i++; //must change a variable in the condition, otherwise will have an
	//infinite loop!
}
```

For Loop

```
int arr[5];

for(int i =1; i<5; i++){ //make an int called i and start it at one. As long
//as i is <5, increment i by one at the end of the for loop

	arr[i] = i;

}
```

For and **while** loops are always interchangeable. For example, the above **for** loop and the above **while** loop have the same result. (note that *i* will be 5 after the while loop but 4 after the for loop)

In MATLAB, for loops again do not have parentheses or curly brackets, but the loop will end with an **end** command, and the counter is also different. The structure of a for loop counter is `i = initial:increment:final`. While loops are the same with the exception of the counter (since they do not use a counter). Do-while loops do not exist in MATLAB.

Do-While Loops

Do-while loops are very similar to while loops, the only difference being in a do-while, the code is guaranteed to run at least once. As a result, the do-while loop is not interchangeable with the other two kinds of loops.

```
int i = 1;
do
{
	printf("%d,",i); //will get 1, 2
	i++;
}
while (i<3);
do
{
	printf("hi");
}
while (i<-1); //i is 3, so this won't repeat, but because it's a do-while
//loop, we will see one "hi"
```

Continue Statements

Continue statements will make the code go back up to the top of the loop, but at the next iteration. So, for example, with a for loop at *i = 2*, a continue statement would make the loop start again at *i = 3*.

```
int arr[5] = {7,7,7,7,7};
for (int i = 0; i<5; i++) //i starts at 0, then will be 1, then 2…when i<5 is
//false (i equals 5), loop will stop
{
	if (i==2)
	{
		continue; //goes back to top of loop with next i value
	}
	arr[i] = i;
}
```

Question 5: What is the resultant array?

Break Statements

Break statements will take the code out of the loop, ending the loop's execution.

```
int arr[5] = {1,1,1,1,1};
for (int i = 0; i<5; i++) //i starts at 0, then will be 1, then 2…when i<5 is
false (i equals 5), loop will stop
{
	if (i==2)
	{
		break; //ends loop
	}
	arr[i] = i+5;
}
```

Question 6: What is the resultant array (arr)?

Question 7: Write a Program

Write a program that calculates values of the polynomial $y = 3x^3 - 5x^2 + 100x - 17$ at x = 0, x=0.2, x=0.4, … , x=4.0 (that is, for values of x separated by 0.2 between a start value of 0 and an end value of 4.0). The y-values that are between -17 and +60, inclusive, are considered mid-range; the values that are greater than 60 are considered high and the values that are lower than -17 are considered low. Your program should determine and then display how many of the x-values in [0,4] evaluate to high y-values, how many evaluate to mid-range y-values and how many evaluate to low y-values.

Functions

Functions perform tasks. Tasks are the outcome of a set of operations. For example, every line of code below is a single operation:

```
int a=2;                                                    //SNIPPET
int b=3;
int c=a*b;
printf("%d ", c );
```

These operations cumulatively perform the *task* of printing the product of variables *a* and *b* to the command line. To reduce code repetition and allow users to define the values of these variables, this task can be written into a **function**, declared outside the *main* body of the program:

```
void productPrint(int a, int b);                            //SNIPPET


void productPrint(int a, int b)
{
int c=a*b;
printf("%d \n", c );
}
```

This function, labelled *productPrint,* can now be called with one line of code in the *main* program body to perform the aforementioned task. The section next to the function label indicate the *arguments* taken by the function. These are the values of the *a* and *b* variables the function will use throughout its operation. For example:

```
int main()                                                  //SNIPPET
{
productPrint(2,3);
productPrint(3,3);
productPrint(1,2);
}
```

Will produce an output on the command line of:

```
6

9

2
```

A function is commonly used to *return* a value to another variable for use elsewhere in the program. The previous *productPrint()* function had a data return type of *void*, seen in the declaration of: `void productPrint(int a, int b)` . This means the function performed operations but did *not* return data. We often desire data returns though, an example of which is given below:

```
#include <stdio.h>                                                    //PROGRAM
#include <stdlib.h>


//Function declaration
int productCalculation(int a, int b); //This function takes 2 integers and
returns their product

int main()
{
	int productOne = productCalculation(1,2);

	int productTwo = productCalculation(2,3);

	//Note we are now using the returns from our first two executions of
	//productCalculation to compute productThree

	int producthree = productCalculation(productOne, productTwo);

	printf("%d\n", productThree);

	return 0;
}

//Function definition, notice the return type int before the function name
int productCalculation(int a, int b)
{
	int product=(a*b);
	return product;
}
```

This produces a final output on the command line of 10. Because functions deal with variables, it is important to understand variable *scope* in the context of functions.

AN IMPORTANT NOTE ON ARRAYS PART #1: Functions cannot return arrays. The method of using a function to modify array values will be discussed in the upcoming *Pass By Reference* section.

Scope

The *scope* of a variable defines where the variable can be accessed within a program. This needs to be understood so that variables and functions are declared in the proper order in a program, otherwise errors will be produced. All variables are either:

- **Local variables.** Declared inside a function, accessible only to that function.
- **Global variables.** Declared outside a function, accessible anywhere in the program *beneath* its declaration statement.

For example, building on the previous program example:

```
#include <stdio.h>                                                    //PROGRAM
#include <stdlib.h>

int globalVar = 10; //Declared outside a function, above the rest of the
                    //program, this variable is thereby accessible anywhere
                    //in the program

int productCalculation(int a, int b);

int main()
{
      int product = productCalculation(globalVar, globalVar);

      printf("%d\n", product);

      return 0;

}


int productCalculation(int a, int b)
{
      int product=a*b;
      return product;
}
```

This will successfully produce a command line output of: `100`.

Contrast this with the below examples, all of which will produce errors or misfunction due to *improper variable scope*:

Issue: globalVar is declared *below* its use in main(), it is outside its scope

```
int pCalc(int a, int b);      //SNIPPET

int main()
{
int product = pCalc(gVar, gVar);
printf("%d\n", product);
return 0;
}

int gVar = 10;

int pCalc(int a, int b)
{
      int product=a*b;
      return product;
}
```

Issue: pCalc() is declared *below* its use in main(), it is outside its scope

```
int gVar = 10;                //SNIPPET

int main()
{
int product = pCalc(gVar, gVar);
printf("%d\n", product);
return 0;
}

int pCalc(int a, int b);

int pCalc(int a, int b)
{
      int product=a*b;
      return product;
}
```

Issue: pCalc is *void*, and its product variable is local to the function

```
int gVar = 10;                //SNIPPET
int pCalc(int a, int b);

int main()
{
pCalc(gVar, gVar);
printf("%d\n", product);
return 0;
}

void pCalc(int a, int b)
{
      int product=a*b;
}
```

Issue: The product variable modified in pCalc() is local to pCalc, it is outside main()'s scope

```
int gVar = 10;                //SNIPPET
int pCalc(int a, int b, int product);

int main()
{
int product;
pCalc(gVar, gVar, product);
printf("%d\n", product);
return 0;
}

void pCalc(int a, int b, int product)
{
      int product=(a*b);
}
```

Reading and Writing Files

There are multiple ways to read and write data from text files in C. Generally, when reading this data, we deal with **char**, **float** and **int** data types. In your course, you mostly deal with **string** (group of chars), **float** and **int** variable types, where **float** and **int** are for numeric data and **strings** are for text data.

Reading Files

When reading files, there is a set of key steps to follow:

1. Set file name
2. Open the file
3. Read the data line by line
4. Set up a line input stream
5. Extract string values from the lines of data
6. Convert the string into a numeric format (int or float)

We set the file name using a string type variable or keyboard input. When using a string type variable, we create a string variable with a name and set it equal to the name of our text file. An example of this is shown below.

```
string name = "datafile.txt";
```

We can also use the **cout** and **cin** commands to collect user input for the file name.

```
string name;
cout << "Enter the name of the file: ";
cin << name;
```

This will store the name the user inputs as in our string variable as the file name.

When opening the file, we use an input file stream variable. This is called using the **ifstream** command.

```
ifstream inputStream(name);
```

The variable inputStream is now an input file stream variable that we can use to access the data in the text file. We can now read a line of text data into a **string** variable using the **getline** command. We can then set up a line stream variable that parses (breaks down) the line of data using the **stringstream** command. This makes it possible for us to extract separate parts of the data afterwards.

```
string newline;
getline(inputStream, newline);
stringstream lineStream(newline);
```

We will now extract our parsed data into a string variable using the **getline** command (this would typically be done in a loop to get each parsed part of the line). We add a delimiter (something that separates the data; sometimes a comma) to the getline command so the program knows when to move on to the next line).

```
string dataValue;
getline(lineStream, dataValue, ',');
```

Finally, we convert our data into our **int** or **float** variable type so that we can use it later in the program. This is done using the **istringstream** command and the extraction operator (>>).

```
int x;
istringstream(dataValue) >> x;
```

Writing Files

Writing text files also involves a series of standard steps:

1. Set file name
2. Open the file to write to it
3. Write the text into the file (typically done with a loop)
4. Close the file

Setting the file name is the same as what was done in order to read the file, as shown above. We open the file to write to it using the **FILE*** object and **fopen** command using the file name and a flag which indicates the action being taken. In this case the flag would be a "w" for write.

*We must convert the file name into a C string in order to use the **fopen** command. We do this by appending c_str() onto our file name, as shown below.*

```
FILE* output = fopen(name.c_str(), "w");
```

We then write the data to the file using the **fprintf** command and close the file using the **fclose** command.

```
fprintf(output, "The result is %d", x);
fclose(output);
```

Workbook Problems

All below problems sourced from last-years MT2 EngLinks booklet.

1. **Predict the output of the below code:**

```
int arr[6];
for(int i = 0; i < 6; i++)
{
	arr[i] = i+i;
}
for(int i=0; i < 6; i+=3)
{
	printf("array[%d] = %d\n", i, arr[i+1]);
}
```

Expected output:

2. **Predict the output of the below code:**

```
int matrix[4][4] = {{1, 2, 3, 4},
                    {5, 6, 7, 8},
                    {9, 10, 11, 12},
                    {13, 14, 15, 16}};
for(int i = 1; i < 4; i = i + 2)
{
	for(int j = 0; j < 4; j = j + 2);
	{
		printf("%d\n", matrix[i][j]);
	}
}
```

Expected output:

3. **Write the below function:**

Professor Smarts takes liquid level measurements of a chemical process. She takes each measurement twice and then performs some analysis on her readings. Help Professor Smarts by writing a function getLevels that has the following parameters:

- Two arrays of floats, **list1** and **list2**, that store the two lists of measurements
- An array of floats **low** that will hold the lower of the two measurements from **list1** and **list2**
- an integer **n** that indicates the size of the arrays
- An integer **low1** that is passed by reference (described below)

The function **getLevels** should work as follows:

- Compare each element of **list1** to the corresponding element of **list2** and put the lower (or equal) value in the array **lows** in the same index).
- Count how many times a lower value comes from **list1** and store that in **low1**.
- Count the number of times that the value at a given index of **list1** was within 0.01 of the value of the element at the same index of **list2**. The function should **return** this value.

Assume that this type definition has been given at the top of the program

typedef float Float1D[100];

Assume that in the **int main**, the list of measurements are stored in arrays **levels1[100]** and **levels2[100]**, the list of low values will be stored in array **lowVals[100]**, and the number of times a low value comes from **levels1** will be stored in a variable called **numLow1**.

Possible solution:

```
int getLevels(Float1D list1, Float1D list2, Float1D lowVals, int n, int
*low1)
{
	(*low1) = 0;
	int close = 0;
	for(int i = 0; i < n; i++)
	{
		if(list2[i] < list1[i]) //check for a lower value from list 2
		{
			lowVals[i] = list2[i];
		}
		else //if list2 doesn't have lower value, that means list1 has
		//lower or equal value
		{
			lowVals[i]=list1[i];
			(*low1)++;
		}
		if(abs(list1[i] - list2[i]) < 0.01) //check for a close
		//value…abs(x) gets absolute value of x
		{
			close++;
		}
	}
	return close;
}
```

4. Write the below function:

Professors for a new course will be using the following marking scheme:

	If a student gets 50% or more on final exam	If student gets less than 50% on final exam
Final exam	50	65
Project	35	20
Quiz	15	15

Final exam marks were recorded out of 100, project marks out of 30 and quiz marks out of 4. Write a function getMarks() that will calculate the final mark of each student. The function has the following parameters: one-dimensional arrays exam, project and quiz, which store the marks for each of the N students, and one-dimensional array finalMark which will store the calculated percentage marks for each of the N students, where N is no more than 10,000. You may assume that N is a symbolic constant that has been defined at the top of the program and that a type definition

typedef float *Grades1D[10000];*

has been given at the top of the program. Your function will also return the number of students whose final mark is less than 50%.

Assume that in the main function, the list of grades for exams, projects, and quizzes have been stored in arrays *studentExam[N], studentProj[N], studentQuiz[N*], and that you will store the final marks in array *studentMark[N].*

```
int getMarks(Grades1D exam, Grades1D project, Grades1D quiz, Grades1D
finalMark)
{
	int failed = 0;
	for(int i = 0; i < N; i++)
	{
		if(exam[i] < 50) //check which marking scheme they are in
		{
			//calculate ith students' mark based on scheme equation
			finalMark[i] = exam[i]*0.65 + project[i]*0.2 +
				quiz[i]*0.15;
			failed++;
		}
		else //only 2 marking schemes
		{
			//calculate ith students' mark based on scheme equation
			finalMark[i] = exam[i]*0.5 * project[i]*0.35 +
				quiz[i]*0.15;
		}
		return failed;
	}
}
```

5. Complete the code:

Daily maximum temperatures in degrees centigrade for February 2014 in Kingston were recorded, as follows:

1.9, 1.4, -4.1, -5.8, -7.2, -7, -8.3, -7, -8, -7.2, -12.6, -8.4, -4.3, -0.1, -18, -9.2, -11.1, -1.6, -3.7, 2.1, 4.8, 3.8, 0.7, -6.2, -7.4, -8.4, -8.2, -12.6

The values are in order by date from Feb. 1 to Feb. 28.

The temperatures are entered in a one-dimensional array, **max_temps**, in a C program. Complete the program to do the following:

A. Calculate the average temperature, avg, in int main, where avg is obtained by summing over all temperature values (Ti, i=0,…,N-1) and dividing by the total number of values (N)
B. Display the average value
C. Find the median temperature value: The median value of an array, is the value in the middle of the sorted array; for example the median of {3, 5, 6, 9, 12} is 6 and the median of {1.2, 1.8, 3.6, 4.2, 4.9, 6.4} is the average of the two values in the middle: (3.6+4.2)/2 = 3.9. To find the median value of **max_temps**:
    - copy the array **max_temps** into a second array **sorted_temmps**; pass **sorted_temps** to the function **sortArray** to sort the temperature values from minimum to maximum (use selection sort or bubble sort)
    - In main(), find the median value from the sorted array; include code to find the median if the number of values, N, in the array is odd or even, where:
        i. if N is odd, find the middle index of the array and get he value at that position
        ii. If N is even, find the two middle indices and average the values in those positions
D. Display the median temperature

Complete the skeleton code given on the following two pages.

```
#define N 28
typedef float Float1D[N];

void sortArray(Float1D temps);
void swap(int ind_1, int ind_2, Float1D list);

void main()
{
      //variable declarations - declare any other variables used in the
      //program
      float max_temps[] = {1.9, 1.4, -4.1, -5.8, -7.2, -7, -8.3, -7,
                           -8, -7.2, -12.6, -8.4, -4.3, -0.1, -18,
                           -9.2, -11.1,-1.6, -3.7, 2.1, 4.8, 3.8,
                           0.7, -6.2, -7.4, -8.4, -8.2, -12.6}
      float sorted_temps[N], total = 0;
      float average;
      float median;
      int middle_element;

      //calculate and display the average temperature
      for(int i = 0; i < N; i++)
      {


      }



      //copy the values in max_temps into sorted_temps and call sortArray
      for(int j = 0; j < N; j++)

      {

      }

      // find the median value if N is even or odd and display the median
      //value on the LCD screen
      if(         )
      {

      }
      else
      {


      }

      printf( "Median: %0.2f", median);
}
```