Grab Bag: Solutions

Pass-by-reference

Concepts

1. Pass-by-reference is used when we need to change the value of a variable inside of a function, and have that change maintained after the function ends. It can be used, for example, to return multiple values from a function. Although we don't touch on it in this course, arrays and file streams are **always** passed by value by default, due to the internals of how C++ handles those things.
2. There are indeed two ways to handle this. In the first, I could return the sum (or product) and pass in the other quantity by reference. In the second, I could pass them both by reference and make the function `void`. I will demonstrate both.

```
int sumProductOnePass(int a, int b, int &prod)
{
  int sum = a + b;
  prod = a*b;
  return sum;
}
void sumProductBothPass(int a, int b, int & prod, int &sum)
{
  sum = a + b;
  prod = a*b;
}
```

3. Pass-by-reference should not be used if you do not intend to change the value of the variable you are passing into the function. One of the key downsides to pass-by-reference is that you will be unable to type cast. For example, the code below is **invalid** and will not compile:

```
void addFive(int &a)
{
  a = a + 5; //so far so good, this will work fine
}
int main()
{
  double dbl = 5.0;
  addFive(dbl); //can't do it!  dbl is a double, but the function needs
  //and integer!
  cout << dbl;
  return 0;
}
```

This is not the case if the function were re-written as a pass-by-value with a return statement, like so:

```
int addFiveGood(int a) //pass a by value!
{
  return a + 5; //a does not get modified
}
int main()
{
  double dbl = 5.0;
  addFiveGood(dbl); //now we are OK.  The double gets cast to an int.
  //and integer!
  cout << dbl;
  return 0;
}
```

Question 1

Both functions are *supposed* to swap two integers. However, `swapOne` has all of its arguments passed in by value. As a result, those values get changed **only in the scope of the function**. When we print out the result of the first function in main, we will see that `a = 4` and `b = 3`. Their values have not changed. On the other hand, `swapTwo` passes in the integers by reference. What we will see when we print them out is `a = 3` and `b = 4`.

Question 2

Since I need this one function to return both the parallel and series combination of the resistors, I will need to pass at least one of those things in by reference. I'll choose to pass them both in by reference, though, and have a void function. Therefore, I need four values to be input to the function: the two integers R1, R2, and two doubles, ReqSeries, and ReqPar.

```
void comboResistors(int R1, int R2, double& ReqSeries, double& ReqPar)
{
  ReqSeries = R1 + R2;
  ReqPar = 1.0*(R1*R2)/(R1 + R2);
}
```

Some good tests for this function: R1 = R2, R1 = 0, R2 = 0.

Question 3

I'm going to over-write the values of the vector given, since this will otherwise be a function that requires 7 parameters!

```
void scalarMult(double alph, double &x1, double &x2, double &x3)
{
  x1 = alph*x1;
  x2 = alph*x2;
  x3 = alph*x2;
}
```

Question 4

Recall that the angle between two vectors is given by:

$$\theta = \cos^{-1}\frac{\vec{v1}\cdot\vec{v2}}{|\vec{v1}||\vec{v2}|}$$

So, I'll need the following function:

```
double dotAng(double v1x, double v1y, double v2x, double v2y, double &dot)
{
  dot = v1x*v2x + v1y*v2y;
  double v1Mag = sqrt(v1x*v1x + v1y*v1y);
  double v2Mag = sqrt(v2x*v2x + v2y*v2y);
  double thet = acos(dot/(v1Mag * v2Mag));
  return thet;
}
```

Notice that I chose to return the angle and passed in the dot product by reference. You may have chosen to do smething different.

2D Arrays

Concepts

1. Setting the elements of a 2D array to zero is the same as doing so for a 1D array:

```
int twoDArray[5][5] = {0};
```

2. It has 5 rows and 6 columns.
3. This requires a nested for loop:

```
int twoDArray[2][2] = {0}; //always initialize!
for(int i = 0; i < 2; i++)
{
  for(int j = 0; j < 2; j++)
  {
    //just red in directly
    cin >> twoDArray[i][j];
  }
}
```

4. I'm going to re-use the code above, so let's assume that the array twoDArray is already initialized. Then, I can do thing like use setw, which I intend to do:

```
for(int i = 0; i < 2; i++)
{
  for(int j = 0; j < 2; j++)
  {
    cout << setw(15) << twoDArray[i][j] << " ";
  }
  cout << endl;
}
```

Applications

Question 1

We need to use nested loops for this question. I'm also going to use constants for the array size, so that it becomes easier to write the testing code by reducing the size of the constants.

```
const int N_PLAYERS = 5;
const int N_ROUNDS = 10;

int score[N_ROUNDS][N_PLAYERS] = {0};

//now get the users to enter their score every round
for(int i = 0; i < N_ROUNDS; i++)
{
  cout << "Enter your scores for round: " << i + 1 << endl;
  for(int j = 0; j < N_PLAYERS; j++)
  {
    cout << "Player " << j + 1 << ": ";
    cin >> score[i][j];
  }
}

//Now we need to see who won, and again we need to loop.
int maxScore = 0;
int winner = 0; //the array index of the player who won

//Notice that the for loop is different than the one above.  We are
//looping over players this time, not rounds.
for(int i = 0; i < N_PLAYERS; i++)
{
  int playerScore = 0;
  for(int j = 0; j < N_ROUNDS; j++)
  {
    playerScore += score[j][i];
  }
  //find the max.  Note: we are assuming there are no ties!
  if(playerScore >= maxScore)
  {
    maxScore = playerScore;
    winner = i; //player i so far has the highest score.
  }
}

cout << "Player " << winner + 1 << " Has won!";
```

Question 2

Interestingly enough, this question is **nearly** identical to the one above. The only difference is that we need to drop the lowest grade. I'm going to re-use a lot of code from above, with some changes to variable names so that everything is more clear.

```
const int N_STUDENTS = 10;
const int N_QUIZZES = 5;

int grade[N_QUIZZES][N_STUDENTS] = {0};

//now get the users to enter their score every round
for(int i = 0; i < N_QUIZZES; i++)
{
  cout << "Enter your grades for quiz: " << i + 1 << endl;
  for(int j = 0; j < N_STUDENTS; j++)
{
  cout << "Student " << j + 1 << ": ";
  cin >> grade[i][j];
}
}

//now we need to compute averages and output them
for(int i = 0; i < N_STUDENTS; i++)
{
  int avg = 0;
  int minGrade = grade[0][i];
  for(int j = 0; j < N_QUIZZES; j++)
  {
    avg += grade[j][i];

    //check for the minimum, but don't do anything yet
    if(minGrade > grade[j][i])
    {
      minGrade = grade[j][i];
    }
}

//remove the lowest score now
avg -= minGrade;

//output the average
cout << "Student " << i << " avg: " << avg/(N_QUIZZES - 1.0) << endl;
}
```

Robot C

Question 1

We should be using the encoders here to measure distance, since they will be very accurate. We are assuming that the robot will drive perfectly straight, although this is not usually how it works out in reality. However, the assumption is fine to make on a quiz or an exam. My approach is as follows:

- The room is rectangular and if I drive forward I know I am going to hit a wall. So I'll drive forward until I hit the wall, then stop. I'll use the touch sensor to detect the wall.
- I will then rotate 180 degrees. Since I don't have a robot with me, I'm going to make an assumption: it takes 1 second to rotate 180 degrees at 40 percent power. If that's not actually true, it's OK - on an exam the assumption is fine since on a real robot we just need to test it once or twice to get the exact value, so 1s will be a placeholder for now. I'm also not backing up slightly, for simplicity.
- I will then drive forward until I hit the other wall, and record the distance.
- I'll then turn 90 degrees (assume: 0.5s at 40 pecent power), and repeat the above process.

  ```
  sensorType[S1] = sensorTOUCH;
  float length = 0;
  float width = 0;
  for(int i = 0; i < 2; i++)
  {
  motor[motorA] = 100;
  motor[motorC] = 100; //drive forward
  while(sensorValue[S1] == 0){} //until you hit the wall
  motor[motorA] = 40;
  motor[motorC] = -40;
  wait1Msec(1000); //turn 180 degrees
  motor[motorA] = 0;
  motor[motorC] = 0; //not really needed, but just for completeness
  nMotorEncoder[motorA] = 0; //reset the encoder
  motor[motorC] = 40; //drive forward again.
  while(sensorValue[S1] == 0){} //until you hit the wall.
  motor[motorA] = 0;
  motor[motorC] = 0;
  //convert the distance
  if(i == 0)
  {
  length = 2PI1.5nMotorEncoder[motorA]/360.0;
  }
  else
  {
  width = 2PI1.5nMotorEncoder[motorA]/360.0;
  ```

```
}
//turn 90 degrees.
motor[motorA] = 40;
motor[motorC] = -40;
wait1Msec(500);
//restart. After the first time through, we are facing the other walls
//but don't know where we are in the room again, so we need to
//first find the wall, then measure its length.
}
//print it out
displayString(1,"%f, %f", length, width);
```

Question 2

We can do this without using arrays, and we will handle all of it in task main. Ultimately, I need to:

1. Read the sonar sensor in a for loop
2. If the measurement is valid, add it to a running sum and increment the number of good measurements
3. If, at the end of the loop, I have **any** good measurements whatsoever, compute the average measurement.

```
task main()
{
  SensorType[S1] = sensorSONAR;

  int meas = 0; //my current measurement
  float dist = -1; //the default
  int numGoodPoints = 0;

  for(int i = 0; i < 10; i++)
  {
    meas = SensorValue[S1];
    if(meas != 255 && meas != 0)
    {
      numGoodPoints++;
      dist += meas;
    }
  }

  //now compute the average, assuming any of the measurements were OK
  if(numGoodPoints != 0)
  {
    dist = dist/numGoodPoints;
  }

  //let's display it to see
  displayString(1,"%f",dist);

}
```

Question 3

My process is the following:

1. Move forward until the touch sensor is pressed.
2. Back up slightly and turn 90 degrees. I'm going to assume that a turn of 90 degrees is accomplished by moving at 50% power for 750mS. This type of assumption is valid on an exam. In reality, you'd just test it until it worked.
3. Drive until we do not see the box anymore on SONAR.
4. Turn 90 degrees in the opposite direction.
5. Drive until we **do** see the box again. At this point, reset the encoder.
6. Drive until we **do not** see the box anymore. Record the encoder value as the length of the box.
7. Repeat steps 4 through 6, this time recording the width.

```
task main()
{
  SensorType[S1] = sensorSONAR;
  SensorType[S2] = sensorTOUCH;

  motor[motorA] = motor[motorC] = 50;
  while(SensorValue[S2] == 0){} //wait until we hit the box

  //back up, turn, and drive
  motor[motorA] = motor[motorC] = -50;
  wait1Msec(500); //just back up a bit
  float length = 0;
  float width = 0;
  for(int i = 0; i < 2; i++)
  {
    motor[motorA] = 50; //turn
    wait1Msec(750);
    //drive until we do not see the box anymore
    motor[motorC] = 50;
    while(SensorValue[S1] < 255){}

    //turn again, this time in the opposite direction.  DO NOT reset
    //the encoders yet, the turn will affect them.
    motor[motorA] = -50;
    wait1Msec(750);
    //move forward again until we see the box
    motor[motorA] = 50;

    while(SensorValue[S1] == 255){}

    //reset the encoder
    nMotorEncoder[motorA] = 0;

    //wait until we no longer see the box
    while(SensorValue[S1] < 255){}

    //record the length or the width
    if(i == 0)
    {
      length = nMotorEncoder[motorA] * 1.5 * 2 * PI / 360.0;
    }
    else
    {
      width = nMotorEncoder[motorA] * 1.5 * 2 * PI / 360.0;
    }

    //loop again!
  }
  //let's display them
  displayString
}
```