Laboratory 3

Repetition and Iteration — `do`, `while` and `for`

3.1 Overview

This laboratory will introduce you to three ways of repeating parts of your program:

- the `while` loop repeats a part of your program whilst (and providing) a condition is true;
- the `do...while` loop also repeats a part of your program whilst a condition is true, but it will always execute that part of your program at least once;
- the `for` loop is a convenient way to repeat a part of your program with a built in counter.

You should learn how to use these repetition structures, and which is the most appropriate for a given situation.

The graphics and music options allow you to use these new structures for carrying out repetitive tasks that would be difficult without loop structures.

3.2 Repeating Statements Many Times

It is often necessary to repeat parts of your program many times. Sometimes you will know exactly how many times this will happen, and other times you will not. In C, structures that repeat many statements are often referred to as *repetition structures*, *loop structures* or just *loops*.

The most basic way to repeat a single statement lots of times is to use a `while` loop. A `while` loop executes a single statement over and over again as long as a relational expression (just like the ones you used in `if` statements) is *true*. `while` loops look like this:

```
while (relational_expression)
    statement;
```

Just like with `if` statements you will often want to put lots of statements inside a `while`. To do this you should use a compound statement in place of the single statement underneath the `while` clause. This is what has been done in the "lab3" example. Open the "clab" project, change the target to lab3, and open the source code "lab3.c". Now try building and executing it.

The source code for "lab3.c" is shown below.

```
/*
 *  A program to demonstrate the use of the while statement
 *  C Programming laboratory 3
 */

#include <stdio.h>

int main(void)
{
    /* Declare a variable to store an integer number */
    int number_entered;

    /* Output some text to the user */
    printf("Enter an integer number: ");

    /* Wait for the user to enter a number and hit enter */
    /* Store the number in the number_entered variable */
    scanf("%d", &number_entered);

    /* Display all the numbers from the one entered */
    /* up to (and including) the number 10 */
    while (number_entered <= 10)
    {
        printf("%d\n", number_entered);
        number_entered = number_entered + 1;
    }

    return 0;
}
```

You should find that this program asks you for a number and then displays all the numbers from the one you entered up to, and including, the number ten. If you enter a number greater than ten it doesn't display anything.

You will notice that in the example the `while` clause has a compound statement underneath, rather than a single statement. This allows us to repeat more than one statement.

Try stepping through the example program. Do you see how the execution point jumps backwards inside the compound statement underneath the `while` clause?

Make sure you understand how the example works before continuing. If you have any problems ask for help.

Syntax: The `while` Loop

`while (`*`relational_expression`*`)` *`statement`*`;`

The `while` structure allows you to repeat a single *`statement`* many times. The `while` statement will execute for as long as the *`relational_expression`* is *true*. If you want to repeat (or *loop*) many statements, rather than just one, *`statement`* (**including the semicolon**) should be replaced by a compound statement (just as with `if`) to give:

```
while (relational_expression)
{
    ...
}
```

Exercise 3.1: Using a `while` Loop

Alter the “lab3” program to display a sequence of numbers which starts with the number that the user entered, but which doubles with every number it displays. It should stop when it reaches or exceeds the square of the number the user entered.

Hint: you may need to introduce another integer variable.

Sometimes you know that you want to execute the statements inside a loop *at least once*. We can do this with a `while` loop by using a `do` clause. The `do` clause goes at the beginning of the loop and tells C that we are starting a loop, but that we don’t want it to check the condition until the end. It works like this:

```
do
{
    ...
}
while (relational_expression);
```

A couple of really important things to note:

- the loop will execute **while** the `relational_expression` is *true*, **not until** it is *true*, this is a very easy mistake to make;
- there **is** a semicolon at the end of the `while` clause in a `do...while` loop.

You will probably find that `do...while` loops are more useful than plain ordinary `while` loops, because you usually want a part of your program to execute at least once.

Syntax: The `do...while` Structure

```
do statement; while (relational_expression);
```

The `do...while` loop is very similar to a `while` loop (see above), except that the condition that is checked to determine whether the loop should continue (`relational_expression`) is checked at the *bottom* of the loop rather than the top. This means that the statement inside the loop will always execute at least once. As in the case of the `while` loop, it is common to replace `statement` (and its semicolon) with a compound statement.

Exercise 3.2: Using `do...while` for Validation

In the last laboratory you used `if` statements for user input validation (i.e. checking that user input was valid). If the user entered an invalid number your program simply ended. It would be better if it told the user that the number was invalid.

Edit the code you have produced in the first exercise of this lab to prevent the user typing in a number greater than 100 or less than 1. If the user enters an invalid number the program should tell them why the number they entered was not accepted. It should then ask them for a number again. It should repeat the process of asking for a number until the user gives a valid input.

You should find that a `do...while` loop is very useful for this purpose.

3.3 Repeating Statements and Counting

There are many occasions when it is useful to have a loop which is controlled by a variable which counts the number of loops or *iterations*. The `for` loop exists in C for this purpose. The code below is from the project “lab3a”. Change the target to “lab3a and open “lab3a.c” in the editor. You will probbaly notice that “lab3a” is nearly identical to “lab3” except that it uses a `for` loop instead of a `while` loop. It also declares an extra variable, `count`, to use as a counter in the `for` loop.

Try compiling and stepping through “lab3a”.

```
/*
 *  A program to demonstrate the use of the for statement
 *  C Programming laboratory 3
 */

#include <stdio.h>

int main(void)
{
    /* Declare a variable to store an integer number */
    int number_entered;

    /* Declare a counter variable */
    int count;

    /* Output some text to the user */
    printf("Enter an integer number: ");

    /* Wait for the user to enter a number and hit enter */
    /* Store the number in the number_entered variable */
    scanf("%d", &number_entered);

    /* Display all the numbers from the one entered */
    /* up to (and including) the number 10 */
    for (count = number_entered; count <= 10; count++)
        printf("%d\n", count);

    return 0;
}
```

At the beginning of a `for` loop, after the keyword `for`, is a set of parentheses with three statements inside. You can understand how these statements work by noticing that any `for` loop can be written as a `while` loop. In general, the `for` loop:

```
for (statement1; statement2; statement3)
   statement_to_loop;
```

can be written as an equivalent `while` loop like this

```
statement1;
while (statement2)
{
   statement_to_loop;
   statement3;
}
```

To help this make sense, lets look at the example in "lab3a". The `for` loop is:

```
for (count = number_entered; count <= 10; count++)
   printf("%d\n", count);
```

We can translate this directly to a `while` loop. It then becomes:

```
count = number_entered;
while (count <= 10)
{
   printf("%d\n", count);
   count++;
}
```

which is very similar to the kinds of `while` loops that you have already seen and created for yourself, except for the line

```
count++;
```

This line uses an operator you have not seen before called the *post-increment* operator. This is one of a number of *shorthand* operators in C. The shorthand operators just make it easier to type and describe commonly used operations. The post-increment operator:

```
variable++;
```

is equivalent to:

```
variable = variable + 1;
```

i.e. the post-increment operator simply adds one to a variable. There is another way of writing the 'add one to a variable' operation: the *pre-increment* operator. The pre-increment operator is used like this:

```
++variable;
```

When used on its own like this there **is no difference** between the pre- and post-increment operators. The difference between the two increment operators only becomes clear when you try and use an increment operator **at the same time** as doing something else.

Exercise 3.3: Understanding Pre- and Post-Increment Operators

Temporarily add the following two lines of code to the end of the “lab3a” source code

```
count = 5;
printf("count = %d\n", count++);
```

Rebuild the project and try stepping through the code. What value is displayed on the screen? What is the value of `count` after the `printf` line has executed?

Replace the post-increment operator with a pre-increment operator so that the lines read

```
count = 5;
printf("count = %d\n", ++count);
```

How is that different? You can remove these lines of code from the “lab3a” project when you have finished investigating them.

The increment operators are often useful for writing more compact C code. They are most often used in `for` loops, but can be used in many other places. You should take great care when using shorthand operators as sometimes they make the purpose of code a lot less clear when it is read.

Exercise 3.4: A Simple `for` Loop

Edit “lab3a” to create a program which always counts up from zero, up to the number that the user entered. You should only need to make very small changes to the “lab3a” code. If you do not understand how to do this, ask one of the demonstrators for help.

Syntax: Shorthand Operators

`++ -- += -= *= /=`

C specifies a set of shorthand operators that allow you to write more compact C code. They are often especially useful in loops.

The most useful of these operators in loops are the pre- and post-increment and decrement operators:

```
variable++;
++variable;
variable--;
--variable;
```

The increment operator, `++`, is equivalent to adding one to a variable. The decrement operator, `--`, is equivalent to subtracting one from a variable. The value of a post-incremented expression is the value of the variable *before* it is incremented. The value of a pre-incremented expression is the value of the variable *after* it is incremented. The same applies to the decrement operator.

Other shorthand operators are more straight forward

- `variable1 += variable2` is equivalent to `variable1 = variable1 + variable2`
- `variable1 -= variable2` is equivalent to `variable1 = variable1 - variable2`
- `variable1 *= variable2` is equivalent to `variable1 = variable1 * variable2`
- `variable1 /= variable2` is equivalent to `variable1 = variable1 / variable2`

So, for example

```
count += 5;
```

is directly equivalent to

```
count = count + 5;
```

Exercise 3.5: Using a `for` Loop: A More Advanced Problem

Alter the "lab3a" source code so that, after the user has entered a number, the program responds by displaying the factorial ($x!$) of the number that the user entered. You should find that a `for` loop is the best way to do this. Make sure your code still works if the user enters a number less than 1.

Hint: A factorial is the product of an integer and all positive, integers below it. e.g. $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

If you don't understand how to do this ask for help.

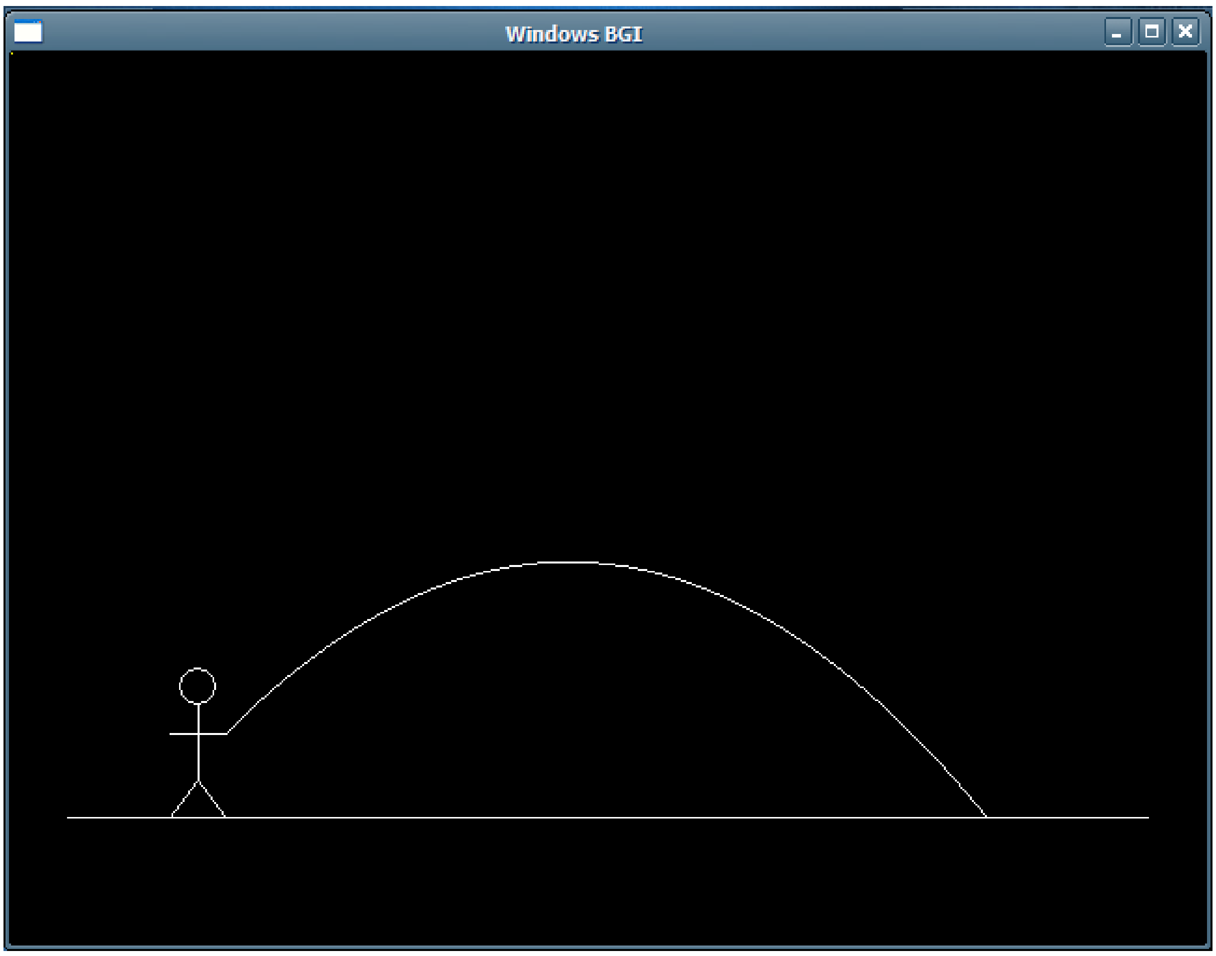
3.4 Deciding What Kind of Loop to Use

There are no hard and fast rules as to what kind of loop you should use in a given situation. Here are some guidelines to help you to make a decision when designing and coding your own programs. You should refer back to this list in later labs, whenever you are not sure what type of loop to use.

- In general, the `for` loop tends to be the neatest and easiest to read loop when you have some kind of counter variable. Even if the counter is not simply counting up (it could be counting down, or in steps of 2 etc.) The `for` loop is also an obvious choice when you know, in advance, how many times the loop should be executed.
- If your code does not neatly fit into a `for` loop, you probably want a `while` loop. If the condition at the top of the `while` loop is not met the first time, the loop will not execute at all. Quite often this behaviour is exactly what you want.
- You should use `do...while` loop when you think a `while` loop is appropriate and you are certain that you always want the loop to execute *at least once*.

3.5 Graphics Option

We are going to add to your "graphics1" program to allow your stick person to throw an object. The graphical display will show the path of the object. It should look something like this



First we will have a brief look at the mathematics we need to describe the path of an object. Once we have the maths we will write some C to carry out the required calculations and plot the results.

We will begin by assuming that there is no air resistance. The initial velocity of the object (as it comes out of the stick person's hand) can be resolved into horizontal and vertical velocities. These act along the x- and y-axes respectively. The only force acting in our simple world is that of gravity. Therefore, there are no forces acting in the x-direction and the horizontal velocity will remain constant. There is one force in the y-direction: gravity. The position (in graphics coordinates) of our object is:

$$P_x = P_{x0} + V_x t \tag{3.1}$$
$$P_y = P_{y0} - V_y t + (gt^2)/2 \tag{3.2}$$

where

- (P_x, P_y) is the current position of the object;
- (P_{x0}, P_{y0}) was the initial position of the object;
- V_x was the velocity with which the object was thrown, in the x-direction in ms^{-1};
- V_y was the velocity with which the object was thrown, in the y-direction in ms^{-1};
- g is the gravitational pull (9.81ms^{-2});
- t is the time in seconds.

For simplicity we will treat each pixel in the graphics window to be a metre wide, just for the purpose of these equations. The exact scale relationship between the graphics window and reality is not important.

We must now think about how these equations should be translated into a C program. The best way to get a good graphical output is to find the height of the projectile (in graphics y-coordinates) for every x-position. For example, the following code would work

```
time = (pos_x - initial_pos_x) / vel_x;
pos_y = initial_pos_y - (vel_y * time) + (gravity * time * time)/2;
```

You should be able to see that the first line works out what the current time must be (since the throw), using the current x-position. This line is a simple rearrangement of equation 3.1. Make sure that you understand this rearrangement before continuing. The second line is simply equation 3.2. To keep accuracy in all of these calculations we must declare some of the variables as being of `double` type (i.e. they may contain real-valued numbers).

We will be drawing the path of the object in the graphics window. Rather than try and draw a curved line, we will construct a curve out of lots of very small straight lines. There are two useful graphics functions which we are going to use for drawing the path of the object. The first allows us to give the graphics system a 'current location'. The function call

```
moveto(100, 300);
```

does not draw anything in the graphics window. It simply tells the graphics system that we wish to set the location (100, 300) to be our 'current location'. We can then use the `lineto` function. The function call

```
lineto(200, 350, 3);
```

draws a line in the graphics window from wherever the 'current location' is to the coordinates specified, in this case the coordinates (200, 350). The line thickness used is given by the third parameter. It then sets the 'current location' to be (200, 350) ready for us to use `lineto` all over again.

Function Reference: `moveto` — Sets the current graphics location

`moveto(`*`x_position`*`, `*`y_position`*`)`

e.g.

```
    moveto(100, 200);
```

The graphics system keeps track of a 'current position', which will be used by functions like `lineto`. The `moveto` function allows you to set this 'current position'. When you call the `moveto` the 'current position' is set to the value (*`x_position, y_position`*) in graphics window coordinates. Both of these values should be integers.

`moveto` is defined in `graphics_lib.h`.

Function Reference: `lineto` — Draws a line from the current location to the specified location

`lineto(`*`x_position`*`, `*`y_position`*`, `*`thickness`*`)`

e.g.

```
    lineto(150, 250, 5);
```

The graphics system keeps track of a 'current position', which can be set by the `moveto` function. The `lineto` function draws a line of a specified thickness in the graphics window from the coordinates specified by the 'current position' to the coordinates (*`x_position, y_position`*). These values must be integers. Once the `lineto` function has drawn the line it sets the 'current position' to be the end of the line it has drawn (i.e. to coordinates (*`x_position, y_position`*)). The next time you call `lineto` it will draw from this point.

`lineto` is defined in `graphics_lib.h`.

If you use the C code we have just looked at to calculate the y-position of the object for a given x-position, we can use the `lineto` function for joining these points together with graphical lines. In a moment you are going to do this, but first a couple of hints.

You should use a loop to work through all the x-coordinates that you need to. Before the loop you should use `moveto` to set the current location, you can then use `lineto` inside the loop to draw all the lines that will make up the path of the object. A suitable `do...while` loop would look something like this

```
moveto(initial_pos_x, initial_pos_y);

do
{
    time = (pos_x - initial_pos_x) / vel_x;
    pos_y = (int)(initial_pos_y - (vel_y * time) + (gravity * time * time)/2);
    lineto(pos_x, pos_y);
    pos_x++;
}
while (pos_x ??);
```

You will notice that part of the `while` condition from this loop is missing (where the `???` is). You will have to think about this for yourself. You should also be aware that `moveto` and `lineto` functions are expecting integer numbers (of type `int`) as arguments. If your variables are of type `double` they have to be converted by putting `(int)` before the variables that are arguments to these functions. The conversion between `double` and `int` types is built in to C and is called a *type cast*. You can see a type cast being carried out in the assignment statement for *`pos_y`*.

Exercise 3.6: Drawing the Projectile Path

Using the information and hints you have been given, add to your "graphics1" project to draw the path of an object thrown from the stick person's hand. The object path must not go underground. If you use the standard value for gravity (9.81ms^{-2}) you might find that an initial velocity of 60ms^{-1} is suitable for both x- and y-directions. The code you produce should work whatever horizontal location the user chooses for the stick person. You should ask the user for the initial velocity of the object. The same initial velocity should be used for both vertical and horizontal components.

Hint: If you really get stuck, have a look at the source code "graphics2.c" . It should help you.

If you still do not understand how to do this, ask one of the demonstrators for help.

Exercise 3.7: Allowing the Stick Person to Move

Use a `getch` function call to wait for the user to press the 'Enter' key before drawing the path of the object. This should happen *after* the stick person has appeared in the graphics window (you may need to look back at the script for laboratory 2 to find the key value for 'Enter').

Remove the option at the beginning of the program for the user to choose the horizontal location of the stick person. Let the user move the stick person (before they hit 'Enter') using the left and right arrow keys. To make the stick person appear to move you should either draw over the top of the stick person with another stick person made up of black lines, or more simply draw a filled rectangle (see below) that covers over the stick man. This will erase the stick person from the screen. You can then redraw the stick person in a new position using coloured lines.

Here is the function that will draw filled rectangles:

Function Reference: `filled_rectangle` — Draws a filled rectangle in the graphics window

`filled_rectangle(`*`xupperleft, yupperleft, xlowerright, ylowerright,fillcolour`*`);`

The `filled_rectangle` function draws a rectangle in the colour specified by `fillcolour`. The rectangle will have its upper left and lower left points at the position specified by *`xupperleft`*, *`yupperleft`*, *`xlowerright`*, *`ylowerright`* respectively. For example

```
    filled_rectangle(10, 10, 40, 60, LIGHTCYAN);
```

will draw a filled rectangle in a light cyan colour with its upper left coordinates (10, 10) and lower right coordinates (40, 60).

`filled_rectangle` is defined in `graphics_lib.h`.

For reference there is also an unfilled rectangle drawing function defined below:

Function Reference: `rectangle` — Draws an outlined rectangle in the graphics window

```
rectangle(xupperleft, yupperleft, xlowerright, ylowerright,thickness);
```

The `rectangle` function draws a rectangle in the default colour (set by the setcolor function) in the graphics window with perimeter line thickness specified by `thickness`. The rectangle will have its upper left and lower left points at the position specified by *`xupperleft`*, *`yupperleft`*, *`xlowerright`*, *`ylowerright`* respectively. For example

```
    rectangle(10, 10, 40, 60, 5);
```

will draw an outlined rectangle with its upper left coordinates (10, 10) and lower right coordinates (40, 60) using a pen thickness of 5 pixels.

`rectangle` is defined in `graphics_lib.h`.

Exercise 3.8: Giving the User Three Goes

The program you have now should allow the user to choose the initial velocity of the object. It should then display the stick person and allow them to use the arrow keys to move the stick person. The program then waits for the user to press ‘Enter’ before it plots the path of the projectile. After the object has been thrown the program waits for a key press and then exits. You should alter this process so that after this last key press the program allows the user to have another go, providing they have not already had three goes. You should close the graphics window and re-open it for each go.

Hint: You will have to put almost all of the code you already have inside another `for` loop to get this to work.

3.6 Music Option

3.6.1 Simple Scales

Repetition is very important in music. We are going to use loops to generate simple scales and use them to attempt to construct music in a minimalist style, similar to that used by composers such as Steve Reich, Terry Riley and Philip Glass. The program “music2” plays one octave of a chromatic scale starting on middle-C. Change the target to “music2” and open the source code “music2.c” in the editor.

The source code for “music2.c” is shown below.

```
/*
 * A program to demonstrate a simple chromatic scale
 * C Programming laboratory 3
 */

#include <midi_lib.h>

int main(void)
{
/* Declare integer variables for specifying a note */
int pitch, channel, velocity, offset;

/* Set the pitch variable to 60, which is middle C */
pitch = 60;

/* We will play the note on MIDI channel 1 */
channel = 1;

/* The note will have a medium velocity (volume) */
velocity = 64;

    /* initialize the midi functions */
    midi_start();

/* Play an octave's worth of chromatic scale */
for (offset = 0; offset <= 12; offset++)
{
/* Start playing a note */
midi_note(pitch + offset, channel, velocity);

/* Wait, so that we can hear the note playing */
pause(400);

/* Turn the note off */
midi_note(pitch + offset, channel, 0);
}

    /* close down all midi functions */
    midi_close();

return 0;
}
```

Try compiling and running the program. Does it sound how you expected it to sound? Make sure you understand the `for` loop before continuing.

Exercise 3.9: Adding to the Chromatic Example

Add to the “music2” code to make it complete two octaves of ascending chromatic scale before descending back down (chromatically) to the starting note.

3.6.2 Whole Tone Scales

A whole tone scale ascends in steps of whole tones, in MIDI note numbers this is increments of two. Composers like Debussy used whole tone scales to create a smooth, dreamy effect, which is often used in films to denote misty or underwater scenes.

Exercise 3.10: Playing a Whole Tone Scale

Alter the code you have just created to play two octaves of ascending then descending whole tone scale.

Exercise 3.11: Using Whole Tone Scales

By altering the code you have just created, produce a program which plays:

- four notes of a whole tone scale (ascending), four times; followed by
- five notes of the same whole tone scale, (ascending from the same starting note) four times; followed by
- six notes of the same whole tone scale (in a similar manner as previously), four times; followed by
- seven notes of the same whole tone scale, four times.

Finally the last whole tone scale should be played in descending order to complete the piece.

You will need to put one `for` loop inside another to get this to work. Putting one programming structure inside another is called *nesting*.

If you do not understand how to do this, ask one of the demonstrators for help.

3.6.3 Adding to Your Whole Tone Scale Piece

You can hear the patterns in this piece but it probably doesn't sound very musical. One reason for this is that there is no properly defined rhythmic structure; there is no sense of where the bar-lines might be.

Exercise 3.12: Making the Piece Sound More Musical

Without destroying your nested `for` loop structure, alter your code so that each ascending scale takes the same amount of time, no matter how many notes are going to be played. You will need to calculate the note durations inside the loop.

What other changes can you make to try and make the piece more musical? Can you make the second and fourth scale in each set quieter than the first and the third (without changing the `for` loop structure)?

Exercise 3.13: Adding Drone-Tones for Depth

The next step is to add some depth to your piece by allowing one or more drone-tones to play in the background. You might like to change the tone that is used during the piece. You could do this once or even at the beginning of each set of scales.

You might choose to play the tones on a different channel and instrument to the scales.

3.6.4 Playing Until Told to Stop

The `kbhit` function allows you to find out whether the user has pressed a key on the keyboard. You use it like this

```
int_variable = kbhit();
```

If the *`int_variable`* has the value 0 after this function call then no key has been pressed. You can use the `kbhit` function to repeat a part of your program until a key is pressed. For example the following `do...while` loop will execute the statements inside it until a key is pressed.

```
do
{
    ...
}
while (kbhit() == 0);
```

It is very important to note that because the `kbhit` function is called at the end of every loop to check if a key has been pressed, this loop will always execute **a whole number of times**.

Function Reference: `kbhit` — Checks to see if a key has been pressed

```
[int_variable =] kbhit();
```

The `kbhit` function tells you whether a key has been pressed. It will return the value 1 if a key has been pressed, or the value 0 if no key has been pressed. This value will be assigned to the *int_variable* if it is present. The `kbhit` function does not wait for a key press. You can retrieve a number identifying what key was pressed using the getch() if `kbhit` returned 1.

`kbhit` is defined in `conio.h`

To use `kbhit` you should make sure that the line

```
    #include <conio.h>
```

is at the top of your source file.

Exercise 3.14: Repeating Until a Key Is Pressed

Alter your program to play the four sets of ascending whole tone scales repeatedly until the user presses a key. When the key is pressed the program should finish its current group of four sets of scales and then finish with the descending scale. You might like to add to the code which executes after the key is pressed to give the piece more of a finale.

3.7 Summary

Now that you have completed this lab you should have the ability to use three kinds of repetition structure:

- `while` loops;
- `do...while` loops;
- `for` loops.

You should also be able to decide which of these types of loop is appropriate in a given situation.

If you completed the graphics option you should have made some major changes to your graphics program which allow your stick person to throw an object. The path of the object should appear in the graphics window. The user should be able to move the stick person left and right using the arrow keys and should have three goes at throwing an object.

If you completed the music option you should have investigated how to use loops to generate scales. You should have put together a piece built on repetition of simple whole tone scales with drone tones in the background. You should also have learned how to detect when the user has pressed a key and used this to allow the user to choose when your program should stop.