

## Laboratory 2


# Conditional Statements: The `if` and `switch` Statements

### 2.1 Overview

This laboratory will introduce you to the `if` and `switch` statements. Both allow your program to make decisions. You will learn how to form *relational expressions* which evaluate to be either *true* or *false* and allow `if` statements to work. You will find that `switch` statements are useful when there are lots of exact options.

This lab will also introduce you to two ways in which you can get input from the user of your program (even if that's you!). The `scanf` function allows you to get a whole line of input from the user, and is useful for obtaining input made up of lots of key presses, like numbers. The `getch` function allows you to get a single key press from the user and is useful for more interactive input. You will use both of these functions a great deal more in later labs as well.

### 2.2 Getting Input from the User

We will begin by using the `scanf` function to obtain input from the user of your program. Change the target to "lab2" and open it in the IDE. 

The "lab2" program will ask the user for an integer number. After the user has typed in a number and pressed the "Enter" key, the program tells the user what number they entered. Try building and executing the "lab2" project.

The source code for "lab2.c" is shown below.

```
/*
 * A program to demonstrate the use of the scanf function
 * C Programming laboratory 2
 */

/* This line allows the compiler to understand both the
 * printf and scanf functions
 */
#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 the number that the user entered */
printf("The number you entered was %d\n", number_entered);

return 0;
}
```

The program uses the `printf` function that you met in the last laboratory to display a message to the user:

```
printf("Enter an integer number: ");
```

It then uses the `scanf` function to get a single integer value from the user, and to put the information into the `number_entered` variable:

```
scanf("%d", &number_entered);
```

The first argument of the `scanf` function is at least one place holder enclosed in double quotes, exactly the same as they are used in the `printf` function. In a similar way to `printf`, for every place holder in the quotes there should be a variable listed as part of the function call. Usually you will only use one place holder and one variable with `scanf`.

You should notice that the variable has an *ampersand*, a “&” character, before it. This is not a mistake! When you use `scanf` you should put an ampersand before each variable name. The exact reason for this is quite complicated and you will learn about later in the course. Until then you will have to remember that whenever you use the `scanf` function to get numbers or single characters from the user **you must put an ampersand (&) before the variable name.**

**Function Reference: `scanf` — Obtains a line of input from the user**

```
[variable = ] scanf(format_string [, &data_item]n);
```

e.g.

```
scanf("%d", &an_int_variable);
scanf("%c", &a_char_variable);
number_of_matches = scanf("%lf", &a_double_variable);
```

The `scanf` function obtains typed input from the user. It lets the user type in text and then waits for them to press 'Enter'. It then takes the text that they have entered and attempts to match it to the place holders specified as part of the *format\_string*.

It takes the following arguments:

- *format\_string* contains the place holders that `scanf` will try to match input to enclosed in double quotes "..."
- *data\_item* — these are variables which will be set to the values recovered from the user's input

`scanf` uses the same place holder system as `printf`. As a reminder, the most useful place holders are:

- `%d` for integer (`int`) variables;
- `%c` for single characters (`char`);
- `%lf` real valued (`double`) variables.

The place holders are matched with the comma separated *data\_item* variables *in order*.

The *return value* of `scanf` (the value that is assigned to *variable*) is the number of place holders that were successfully matched.

`scanf` is defined in `stdio.h`.

**Exercise 2.1: Changing the Variable Type**

Alter the "lab2" program so that it accepts real valued numbers from the user, not just integers. You will need to change the type of the `number_entered` variable and the place holders in both the `scanf` and `printf` functions. Rebuild and execute the program to check that it works.

## 2.3 Making a Decision: Conditional Statements

We are now going to use a conditional statement to make a decision about which part of your program will execute depending on the number that the user enters. We will do this using an `if` statement.



Change your “lab2” program to remove the second `printf` line (the one that displays the value back to the user), and replace it with the following lines:

```
if (number_entered > 10)
    printf("That number is bigger than ten\n");
```



Try building and executing the program. After you have entered the number, it should tell you whether the number is bigger than 10.

When you are satisfied that it works (you may have to run it a few times), add another two lines after the `if` part to add an `else` condition, like this:

```
if (number_entered > 10)
    printf("That number is bigger than ten\n");
else
    printf("That number is smaller than ten or equal to ten\n");
```



Build your program and execute it a few times, to test it with different numbers.

### Exercise 2.2: Understanding Conditionals

Use the debugger to try stepping through the program you have just written with the `if` statement in it. Where does the flow of execution (the yellow arrow) go:

- for numbers greater than ten;
- for numbers less than ten.

When the yellow arrow reaches a `scanf` statement it will wait until you enter a number before it will continue. To enter the number you must make sure that the window in which your program is actually executing (it will have a black background) is at the front.

An `if` statement makes a decision based on the part that is in brackets. The part in brackets is called a *relational expression*. It compares things using *relational operators* (such as greater than, `>`, and less than, `<`) to obtain an answer that is *true* or *false*. The words `if` and `else` are treated specially by C; it uses them to recognise that you want to do something conditionally. Words that are treated specially in this way are called *keywords*, because they are treated specially they cannot be used for variable names. You will meet many more keywords as part of this course.

## Syntax: Relational Operators and Expressions

< > == != <= >=

Relational operators allow you to compare things such as the value of variables and numbers or characters. A simple relational expression compares two things and uses one relational operator:

```
item1 relational_operator item2
```

The relational operator can be any one of the following:

- < (less than)
- > (greater than)
- == (equal to)
- != (not equal to)
- <= (less than or equal to)
- >= (greater than or equal to)

The result of a relational expression is therefore *true* or *false*. (We say the expression *evaluates* to *true* or *false*).

So, for example, let's construct a relational expression which tests the value of a variable called `test` against the number 5. If `test` has the value 6 then:

- `test < 5` will evaluate to *false*
- `test > 5` will evaluate to *true*
- `test == 5` will evaluate to *false*
- `test != 5` will evaluate to *true*

If `test` has the value 5 then:

- `test < 5` will evaluate to *false*
- `test > 5` will evaluate to *false*
- `test == 5` will evaluate to *true*
- `test != 5` will evaluate to *false*
- `test <= 5` will evaluate to *true*
- `test >= 5` will evaluate to *true*

It is very important to notice that the operator for testing whether two things are the same is **not** `"=`". This is the assignment operator, which we saw in the previous lab. The operator for testing if two things are the same is the double equals `"=="`, this is **different to the assignment operator**. Don't get them confused. Sometimes the compiler won't pick the error up and your program will malfunction in peculiar ways!

**Syntax: if statements**

```
if (relational_expression) statement1; [else statement2;]
```

The `if` statement allows you to execute a piece of your program conditionally. The `if` statement guards a single statement (*statement1*) of a program and only executes it if the *relational\_expression* evaluates to *true*. Optionally, an `if` statement can be followed by an `else` section, which guards a second statement (*statement2*). This second statement only executes if the *relational\_expression* evaluates to *false*.

You have already seen how to add an `else` section to an `if` statement. You can produce a more complicated (and more useful) structure by making the statement that the `else` part guards (*statement2* in the above syntax box) another `if` statement. This allows you to connect `if` statements together, for example:

```
if (number_entered < 5)
    ...
else if (number_entered > 5)
    ...
else if (number_entered == 5)
    ...
else
    ...
```

You will need to understand how this works in order to be able to complete the next exercise. Ask if you need any help.

**Exercise 2.3: Using Relational Operators**

Alter the “lab2” program so that after it accepts the number from the user it prints either

```
That number is less than or equal to zero
```

or

```
That number is greater than or equal to ten
```

or

```
That number is between one and nine
```

appropriately.

If you don’t know how to answer this question ask one of the demonstrators for help.

## 2.4 Grouping Statements into Compound Statements

Try adding a second `printf` statement to the final `else` condition of the program you have just written. The code should look something a bit like this:



```
if (...)
    ...
else if (...)
    ...
else
    printf(...);
    printf("A printf statement I have just added\n");
```

What happens when you execute it?

You should find that the second `printf` statement is *always* executed. Compare the code with the syntax box on `if` statements. You should notice that `if` and `else` clauses (a *clause* is a part of a statement) only guard **one** statement. If you want the `if` statement to guard more than one statement you must group statements together into a *compound statement*. A compound statement replaces any statement like this:

```
statement;
```

**including the semicolon** with a set of statements, grouped together by braces (curly brackets). Like this:

```
{
    statement1;
    statement2;
}
```

You can replace **any** statement in C with a compound statement. This is useful for making an `if` statement guard a set of statements. Returning to our example, it should be rewritten like this:

```
if (...)
    ...
else if (...)
    ...
else
{
    printf(...);
    printf("A printf statement I have just added\n");
}
```

### Exercise 2.4: Adding Compound Statements

Add an extra `printf` statement to every branch of your `if ... else`. You will need to use a compound statement for each branch. Check that it works as you would expect when you execute it.

Compound statements are very useful for `if` statements. You will find them essential in the next lab when we look at repeating and iterating sets of statements grouped together into block statements.

### Syntax: Compound Statements

```
{ ... }
```

A compound statement is a way of grouping statements together to replace a single statement with many statements. Any statement like this:

```
statement;
```

can be replaced by lots of statements like this

```
{
    statement1;
    statement2;
    ...
}
```

This is useful for making the `if` or `else` clauses of an `if` statement guard multiple statements.

## 2.5 Making More Complex Decisions: Logical Operators

Student *x* wants to go to the bar (which is only open after 7:00pm) but she will only go if she has some money left. We could try and code this decision in C, it might look a bit like this:

```
if ((time >= 19:00) and (bank_balance > cost_of_one_drink))
{
    go_to_bar();
    finish_c_assignment();
}
else
    finish_c_assignment();
```

Please note: this is not real C! This kind of more complicated decision is common in everyday life. To handle this kind of thing in programming code requires us to combine several decisions to form one complex decision.

The relational expressions we have looked at so far make a single decision on the basis of a comparison between two values (which could be variables). Quite often we need to be able to combine comparisons together with an *and* or an *or*. These operations combine expressions which can already be evaluated to the logical values *true* or *false*, in C they are known as *logical operators*. In C, an *and* is represented by the symbol `&&` (an ampersand **twice**) and an *or* is represented by `||` (a vertical bar **twice**). There is an extra logical operator for *not*, which turns a *true* into a *false* or a *false* into a *true*. The *not* operator is an exclamation mark (`!`). You may need to use parentheses (brackets) with the *not* operator to ensure that the operator is applied to the correct part of the expression.



## Syntax: Logical Operators

```
relational_expression && relational_expression  
relational_expression || relational_expression  
!(relational_expression)
```

Logical operators allow relational expressions to be connected together to form larger relational expressions. There are three logical operators which form expressions which may be evaluated to *true* or *false* depending on the truth value of the expressions they are connecting.

- **and** (&&) evaluates to *true* only if *both* the relational expressions it connects evaluate to *true*.
- **or** (||) evaluates to *true* if *both* either (or both) of the relational expressions it connects evaluate to *true*.
- **not** (!) evaluates to *true* only if the expression it is modifying evaluates to *false*.

So, for example, if we have two variables: `test1` which is 5 and `test2` which is 21 then:

- `((test1 < 10) && (test2 > 30))` is *false*
- `((test1 < 10) || (test2 > 30))` is *true*
- `((test1 < 10) && (test2 > 20))` is *true*
- `((test1 < 10) || (test2 > 20))` is *true*
- `((test1 < 1) && (test2 > 30))` is *false*
- `((test1 < 1) || (test2 > 30))` is *false*
- `!(test1 < 1)` is *true*
- `!(test1 < 1) && !(test2 > 30)` is *true*

Make sure you understand these before you go on to the next exercise. If you need help, just ask.

## Exercise 2.5: Logical Operators

From previous exercises you should have a program that takes a numeric input from the user and distinguishes between numbers that are less than or equal to zero, numbers that are greater than or equal to ten and numbers in between. In each case it should print a different response to the user from a number of `printf` statements.

Add to this program to allow the user to input two different numbers (you should prompt them twice). Use `printf` statements to show that the program can distinguish between the conditions:

- when either number is less than 0;
- when both numbers are less than 0;
- when the first number is greater than 10 and the other is not greater than ten.

The program should only need to display **one** of these messages (you may need to structure your `if...else` statements carefully to get it to do this).

## 2.6 Dealing with Many Options

Sometimes you may need to make a decision on a single variable where there are lots of possible options. You could deal with this by stringing together lots and lots of `if` statements, but C provides a special statement to deal with this situation, called `switch`. The `switch` statement allows you to specify a number of cases, which are the options that you are interested in. There is also an special case for 'all other options', labelled `default`.



The "lab2a" project demonstrates the `switch` statement in a very simple way. Change the target to "lab2a" and open it in the IDE.

Now, build and execute it.

The source code for "lab2a.c" is shown below.

```
/*
 * A program to demonstrate the use of the switch statement
 * C Programming laboratory 2
 */

#include <stdio.h>

int main(void)
{
    int number_entered;

    /* Output some text to the user */
    printf("Enter an integer number between 1 and 9: ");

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

    /* Display the number that the user entered */
    /* But display it as English text */
    printf("The number you entered was ");

    /* This switch statement decides between lots of options */
    switch (number_entered)
    {
    case 1:
        printf("one\n");
        break;

    case 2:
        printf("two\n");
        break;

    case 3:
        printf("three\n");
        break;

    case 4:
        printf("four\n");
        break;

    case 5:
        printf("five\n");
        break;

    case 6:
```

```
        printf("six\n");
        break;

    case 7:
        printf("seven\n");
        break;

    case 8:
        printf("eight\n");
        break;

    case 9:
        printf("nine\n");
        break;

    default:
        printf("not between one and nine\n");
    }

    return 0;
}
```

Let us look at the `switch` statement in detail. It begins with the keyword `switch` followed by the name of the variable that we wish to ‘switch’ on, in parentheses. Like this:

```
switch (number_entered)
```

We then describe a number of *cases*, which are different options for the value of the variable given in the `switch` part. All these cases are enclosed in a set of braces, like a compound statement. The `switch` statement looks for the first `case` clause that matches the value of the variable. When it finds it, it starts executing from that point onwards. Let’s take the example where the variable `number_entered` has the value 4. The `switch` looks down its list of `case` clauses until it finds the one that say `case 4:`, it then starts executing at that line, so the next thing it executes is the line:

```
    printf("four\n");
```

The next line says

```
        break;
```

which tells it to stop executing the code inside the `switch` statement, and to continue after the closing brace `}`. You should also notice that there is a special `case` clause called `default`. This gets executed if none of the other `case` statements before it match. Because the `switch` statement attempts to match `case` clauses in order, the `default` clause should always go last.

### Syntax: `switch` statements

```
switch(variable) { [case_clause]n [default_clause] }
```

The `switch` statement allows you to make a choice between a number of options. Each option is a different value of the *variable* specified at the start of the statement. Each possible value of interest is described using a *case* clause. A *case* clause has the following syntax:

```
case value:  
    ...
```

A `switch` statement finds the first *case* that matches and continues to execute from that point onwards. **It will even continue into the next case.** To stop it executing at the end of the code for each *case* it is common to include a `break` statement. The `break` statement stops any more of the `switch` statement from executing. A *case* clause with a `break` statement looks like this:

```
case value:  
    ...  
    break;
```

A *default* clause is like a *case* clause except that it matches *anything*. This is commonly used at the end (after all the *case* clauses) to match anything that hasn't already been matched. Because the *default* clause goes last, it does not need a `break` statement.

### Exercise 2.6: Flow of Execution in `switch` Statements

Try using the debugger to step through the `switch` statement in the “lab2a” example. Watch where the flow of execution goes. What do you think will happen if you remove some of the `break` statements? When you have finished stepping through, remove some of the `break` statements, rebuild the project and try stepping through it again. Does it do what you expected?

If you do not understand the program's behaviour ask one of the demonstrators.

## 2.7 Getting a Single Character from the User

The programs in this laboratory have used the `scanf` function to get input from the user. As you have seen, the `scanf` function collects input and allows the program to continue **only after the user has pressed enter**. Sometimes you want to make a program that is more interactive than this. In this section we will use the `getch` function to collect a single key press from the user.

The `getch` function waits for the user to press a single key and then *returns* the value of the key that the user pressed back to the program as a character. The `getch` function is described in `conio.h`, so to use it you must add the line

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

to the top of your source file.

You would use `getch` in the following way:

```
int key_entered;
```

```
key_entered = getch();
```

`getch` will wait for a key press and then the assignment (=) will copy the value of the key (as a character) into the `key_entered` variable. So, if you use the bit of code above, you have the value of the key press in a variable. Now you need to know how use the value.

You can describe the value of a character in C by enclosing the single character in *single quotation marks*. Like this:

```
if (key_entered == 'r')
    printf("You pressed the R key\n");
```

### Function Reference: `getch` — Obtains a single key press from the user

```
[character = ]getch();
```

The `getch` function waits for a key press on the keyboard and then returns the value of the key that was pressed as an integer character code.

To find out what key was pressed then you should assign the result of the function call to an integer variable (one of type `int`)

```
int_variable = getch();
```

There are some special cases which relate to keys on your keyboard which are not straightforward letters. The best example is the arrow keys. In these cases when you call `getch` and the user presses an arrow key it will return zero. You will then need to call `getch` a second time. This time `getch` **will not wait for a key press**. It will return immediately with the value of an arrow key. In case a user presses a non-character key, `getch` should always be properly used as follows:

```
int_variable = getch();
if (int_variable == 0)
    int_variable = getch();
```

You should see from this code that `getch` is called twice if the value from the first `getch` was zero. In this case the value of the key can be tested to see if it matches one of many useful values, including the arrow key values which are shown below

Key	Value
←	75
↑	72
→	77
↓	80
'Enter'	13

In some cases, the `getch` function may return the number 224, instead of zero, to indicate that an extended key has been pressed. It is usually good practice to check to see if a call to `getch` returns *either zero or 224*.

`getch` is defined in `conio.h` and also `graphics.lib.h`.

### Exercise 2.7: Using `getch` and `switch`

Adapt the “lab2a” program to use `getch` instead of `scanf`. The program should prompt the user for a single letter which is the first letter of a month. The program should then display the months that begin with that letter. You should use a `switch` statement for this. The program should not care whether you use upper or lower case, for example, pressing either ‘F’ or ‘f’ should result in the program displaying “February”.

Hint: You could use multiple `case` clauses to handle the upper and lower case letters. For example:

```
case 'f':  
case 'F':  
    ...  
    break;
```

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

## 2.8 Graphics Option



We are going to alter the graphics program you were working on in the last laboratory to incorporate some of the new techniques you have just learned.

Close any project you have open at the moment and change the target to “graphics1”. Open “graphics1.c” for editing.

Your code from last lab should draw a stick person on the screen. You are going to add to this code so that the user can decide the position of the stick person and the colour that it is drawn in.

### Exercise 2.8: Adding User Control of Position to the Stick Person

When your program starts the user should be able to type in a number which specifies the horizontal location of the stick person on the screen. The user should be able to choose any location **in the left half of the screen**. You should use an **if statement** to check that the number is sensible, i.e. the user should not be able to specify a location that results in any part of the stick person being cut off by the left-hand edge of the window. The user should also not be able to specify a location that is in the right-hand half of the window. If the user does specify an invalid location, you should display an error message and **not** show the graphics window with the stick person in it. The process of ensuring that input is suitable for your program is known as *input validation*.

For this exercise to work you will have to make sure that your stick person is not too big. You might like to draw in a line representing ground level below the stick person’s feet.

You should now have a working program that allows the user to control the horizontal position of the stick person, within bounds which you have specified.

### Exercise 2.9: Adding User Control of Colour of the Stick Person

The next step is to allow the user to choose the colour of the stick person. You should do this by showing the user a menu (a list) of possible colours and **allow them to choose the colour by pressing a single letter** (usually the one that corresponds to the first letter of the colour, e.g. 'R' for red). This choice should be case-insensitive, i.e. 'r' and 'R' should be treated the same. You will probably want to use a `switch` statement for this. You should use the `default` clause to watch for invalid input from the user. If the user presses an invalid key you should display an error message and **not** show the graphics window with the stick person in it.

The circle which makes up the head of the stick person is currently empty. We can use an alternative to the graphics function `filled_circle` to draw a filled circle with colour.

#### Function Reference: `filled_circle` — Draws a filled circle in the graphics window

```
filled_circle(xpos, ypos, radius, fillcolour);
```

The `circle` function draws a filled circle in the graphics window in the colour specified by `fillcolour`. The circle will have its centre point at the position specified by `xpos` and `ypos` which are coordinates from the top-left corner of the graphics window, in pixels. It will have a radius as specified by `radius`.

For example

```
filled_circle(200, 300, 20, BLUE);
```

will draw a blue filled circle with its centre at (200, 300) with a radius of 20 pixels.

`filled_circle` is defined in `graphics_lib.h`.

At this point it is worth mentioning a few more useful graphics functions.

#### Function Reference: `ellipse` — Draws an ellipse in the graphics window

```
ellipse(xpos, ypos, xradius, yradius, thickness);
```

The `ellipse` function draws an outlined ellipse in the graphics window in the current colour. The ellipse will have its centre point at the position specified by `xpos` and `ypos` which are coordinates from the top-left corner of the graphics window, in pixels. It will have a radii as specified by `xradius` and `yradius`, in pixels. The sum of the radii is a constant and equal to the length of the ellipse's major axis. The thickness of the ellipse perimeter is specified by `thickness`.

For example

```
ellipse(200, 300, 20, 10, 5);
```

will draw an ellipse with its centre at (200, 300) with an `xradius` of 20 pixels, a `yradius` of 10 and line thickness of 5.

`ellipse` is defined in `graphics_lib.h`.

You can also draw a filled ellipse with the function

### Function Reference: `filled_ellipse` — Draws a filled ellipse in the graphics window

```
filled_ellipse(xpos, ypos, xradius, yradius, fillcolour);
```

The `filled_ellipse` function draws a filled ellipse in the graphics window in the colour specified by `fillcolour`. The ellipse will have its centre point at the position specified by `xpos` and `ypos` which are coordinates from the top-left corner of the graphics window, in pixels. It will have a radii as specified by `xradius` and `yradius`.

For example

```
filled_ellipse(200, 300, 10, 20, BLUE);
```

will draw a blue filled ellipse with its centre at (200,300) with an xradius of 10 pixels and a yradius of 20 pixels.

`filled_ellipse` is defined in `graphics_lib.h`.

### Exercise 2.10: Using `filled_circle` and `filled_ellipse`

Add to your program to ensure the stick person's head is filled with the same colour as the colour used to draw the body.

Another useful graphics functions provided in `graphics_lib.h` is the arc drawing function defined below.

### Function Reference: `arc` — Draws a circular arc in the graphics window

```
arc(xpos, ypos, radius, angle_start, angle_end, thickness);
```

The `arc` function draws an arc of a circle of radius `radius` in the graphics window in the current pen colour and with a line thickness specified by `thickness`. The arc is drawn clockwise starting at angle (in degrees) specified by `angle_start` and ending at angle `angle_end`. The angles may be negative and are real-numbers (as opposed to integers). An angle zero is equivalent to the horizontal axis. Note that the end angle is *relative* to the first (i.e. it draws an arc by *adding* the end angle to the first).

For example

```
arc(200, 340, 10, -90, 270, 2);
```

will draw a circular arc with its centre at (200, 340) with a radius of 10 pixels. It will begin at 90 degrees *above* the horizontal and draw clockwise until the final angle is 180 (-90+270).



## 2.9 Music Option



We are going to alter the MIDI program you were working on in the last laboratory to incorporate some of the new techniques you have just learned. Close any project you have open at the moment and change the target to “music1”. Open “music1.c” in the editor.

Your code from last lab should play a melody with some chords on a separate channel (and a different instrument). You are going to add to this program to allow the user to choose the instruments for the melody and backing chords. You will also allow them to choose the key that the piece will be played in.

### Exercise 2.11: Adding User Control of Instruments

When your program starts the user should be able to type in a number which specifies the instrument that will be used to play the melody. This should be a number from the General MIDI specification (i.e. a number from 1 to 128). You should use an `if` statement to check that the number is in this range. If the user does specify an invalid instrument number, you should display an error message and **not** play the tune. The process of ensuring that input is suitable for your program is known as *input validation*.

The program should do the same for the instrument which plays the chords, i.e. the user should be able to choose an instrument for both before the piece is played.

You should now have a working program that allows the user to control the instruments that are used for both the melody and the chords, within bounds which you have specified.

For the next step you will need to make sure that all of the notes in the tune are specified as an offset from a variable whose value specifies the key. For example:

```
int key;

/*
 * The key of this piece is C. All notes are specified
 * with reference to middle C
 */

midi_start();

key = 60;

/* Turn a note on */
midi_note(key + 3, 1, 64);

pause(1000);

/* Turn the note off */
midi_note(key + 3, 1, 0);

midi_close();
```

Ask a demonstrator for help if you do not understand how to do this.

### Exercise 2.12: Adding User Control of Key

The next step is to allow the user to choose the key in which the piece is played. You should do this by showing the user a menu (a list) of possible keys. Next to each choice you should indicate which keyboard key the user will need to press in order to select the musical key for the piece.

You may like to choose keys on the keyboard whose layout most closely resembles a standard piano-style keyboard.

This choice should be case-insensitive, i.e. 'r' and 'R' should be treated the same. You will probably want to use a `switch` statement for this. You should use the `default` clause to watch for invalid input from the user. If the user presses an invalid key you should display an error message and **not** play the tune.

## 2.10 Summary

Now that you have completed this lab you should have the ability to write conditional statements in C. You should have learnt about, and be able to use:

- `if` statements;
- `switch` statements.

You should be able to form relational expressions, including quite complex containing with logical operators.

You should understand `switch` statements and the way in which the flow of execution passes through them, including the reason for `break` statements.

You should also be able to obtain input from the user of your program. You should know how to use:

- `scanf` to get a complete line of input from the user at one time (until the user presses 'Enter');
- `getch` to obtain a single key press.

In the music and graphics options you should have used these functions to provide various options to the user of your program. You should have used conditional statements to process the options and to make sure that none of the input is invalid.