Laboratory 4

Functions

4.1 Overview

This laboratory will give you the capability to create your own functions, and help you better understand how existing functions are defined and called. You will learn about:

- how to define functions of your own;
- how to specify parameters so that you can pass values to your own functions;
- how to allow your functions to return a value back to the place where they were called from;
- how and, most importantly, why you should use functions to split up your programs.

You will also be introduced to the `fflush` function, which you will use to make sure that `scanf` handles user input better.

Some standard C mathematical functions are introduced, which you might find useful in your own programs.

In the graphics and music options you will make use of functions to make your programs more readable and more powerful. In the graphics option you will add the capability for the user to choose the launch angle for the object. In the music option you will be introduced to the `random_number` program which you will use to try and create a program which improvises.

4.2 Creating New Functions

In most of the example programs you have seen and created so far, you have used functions a great deal. Most of the time you have used (*called*) existing functions such as `printf` and `getch`. However, every program you have written has contained exactly one function of your own: `main`. In this lab you will see how to write other functions, which you can call in exactly the same way as `printf`, `getch` and others.

Open the "clab" project and make sure the target is "lab4" project and open "lab4.c" in the editor. This project will introduce two new concepts:

- creating new functions of your own, and calling them;
- using the `fflush` function to handle bad input from the user when using the `scanf` function.

The source code of "lab4.c" is:

```
/*
 *  A program to demonstrate function creation
 *  C Programming laboratory 4
 */

#include <stdio.h>

/*
 * display_welcome function - displays a welcome message
 *
 * The display_welcome function takes no parameters and
 * does not return anything.
 */
void display_welcome(void)
{
    printf("Welcome to the squaring numbers program\n\n");
}

/*
 * square function - squares an integer number
 *
 * The square function takes a single integer
 * parameter.  It returns an integer which is the square
 * of the value of the parameter it was given.
 */
int square(int value)
{
    int squared_value;

    squared_value = value * value;

    return squared_value;
}

/*
 * The main function - the program starts executing here
 *
 * The main function takes no parameters and returns an
 * integer.
 */
```

```
int main(void)
{
    int num_placeholders_matched;
    int number_entered, squared_number;

    /* Call the display_welcome function */
    /* This will display a welcome message */
    display_welcome();

    /* Obtain a number from the user */
    printf("Please enter an integer number: ");

    /* Loop for as long as the user gives us values */
    /* that aren't integers */
    do
    {
        /* Call scanf and use the return value to find out */
        /* how many placeholders were matched */
        num_placeholders_matched = scanf("%d", &number_entered);

        /* The integer placeholder was not matched */
        if (num_placeholders_matched < 1)
        {
            /* Clear out whatever the user typed */
            fflush(stdin);

            /* Tell the user to try again, loop will go again */
            printf("That was not an integer. Please try again: ");
        }
    }
    while (num_placeholders_matched < 1);

    /* Square the number using the square function */
    squared_number = square(number_entered);

    /* Display the result */
    printf("%d squared is %d\n", number_entered, squared_number);

    return 0;
}
```

All the programs you have seen so far have started with the `main` function. This program is no different, it's just that there are two other functions included in the source code before the `main` function. These two functions are called `display_welcome` and `square`.

The first thing that the `main` function does is to call the `display_welcome` function, which it does with this line:

```
    display_welcome();
```

You will notice that no arguments are specified as part of this function call (there is nothing between the parentheses).

The next part of the `main` function obtains an integer from the user using the `scanf` function. It is more complicated than the way you have used `scanf` before because it checks to see whether the user actually entered a valid integer.

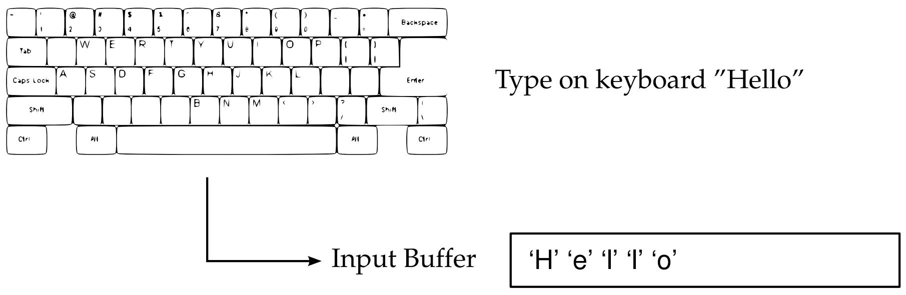
When the user types in a response the `scanf` checks what they typed against the placeholder, it could either be

- a valid integer, in which case the `scanf` function has a return value of 1;
- something else, in which case the `scanf` function has a return value of zero;

This return is the number of placeholders that were matched with the text the user entered. The return value is assigned to the variable called `num_placeholders_matched`.

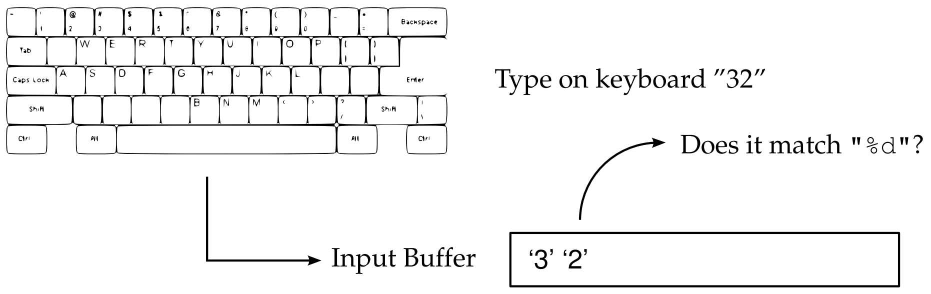
But what happens if the user doesn't type in a valid integer? To answer this question we must look at the way `scanf` works.

When you type in characters on the keyboard they get stored in a special variable in the `stdio` library. This variable is called an *input buffer*.

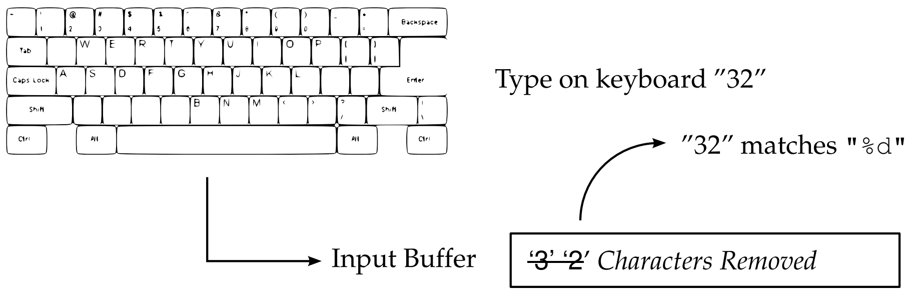


When you call `scanf` the first time the input buffer is empty, because you haven't typed anything yet. Because the buffer is empty, `scanf` waits for you to type in characters and press 'Enter'.

When you have pressed enter it checks to see what is in the input buffer. It then tries to interpret the characters in the input buffer, in away that matches the placeholder you specified. So for example. Say you called `scanf` and you wanted an integer, so you specified a placeholder of `%d`.

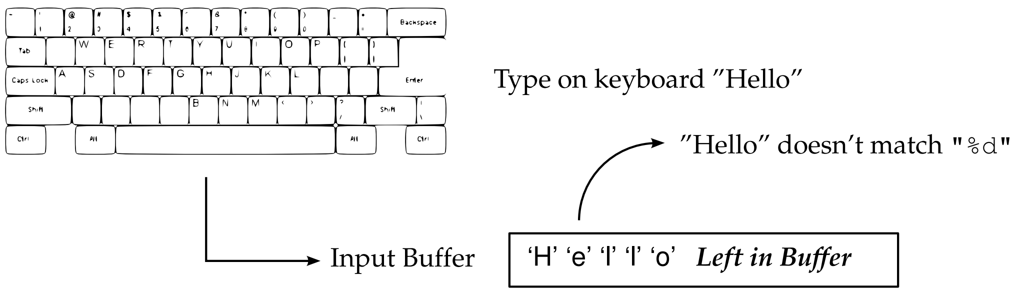


If the characters in the input buffer match the type that is required by the placeholder then it takes them out of the input buffer.



The `scanf` call will then have a return value of 1, because it matched one placeholder.

If, on the other hand, the text does not match the placeholder, `scanf` **leaves the characters in the input buffer**.


Type on keyboard "Hello"
"Hello" doesn't match "%d"
Input Buffer
'H' 'e' 'l' 'l' 'o' Left in Buffer

This is a problem! If we call `scanf` again, the first thing it will do is to check the input buffer. It will find some text there already so it **will not wait for the user to type in more text**.

To get the next call to `scanf` to work properly, we have to clear out, or *flush* the input buffer. We do that by calling the function `fflush`, like this:

```
fflush(stdin);
```

This tells the `stdio` library to clear out (*flush*) whatever it has in its input buffer. The name `stdin` is the name of input buffer which receives input from the keyboard. `stdin` is short for *standard input* and the variable is defined in `stdio.h`. There are other buffers that are used in this way. In general these buffer variables are called *streams*.

Function Reference: `fflush` — Flushes a stream

`fflush(`*`stream`*`)`

e.g.

```
fflush(stdin);
```

Clears a buffer identified by *`stream`* that is being used for input or output. The most common usage of `fflush` is to make sure that the input buffer is emptied after a `scanf` call which had unmatched placeholders. In this case the return value of `scanf` would be tested to determine how many placeholders were matched. If `scanf` did not manage to match all of its placeholders than there will still be characters left in the input buffer. In this situation `fflush` should be called to remove unmatched input from the buffer.

There are three predefined streams which may be used with `fflush`:

- `stdin` — the standard input stream, usually the keyboard;
- `stdout` — the standard output stream, usually the screen;
- `stderr` — the standard output stream especially for error messages – this is also usually the screen.

`fflush`, `stdin`, `stdout` and `stderr` are defined in `stdio.h`

The next line in the `main` function (after the end of the `while` loop) is another function call:

```
result = square(n);
```

This line calls the `square` function passing it the value of the argument `n`. The `square` function has a return value which is equal to the square of the value of its argument, in this case the return value is assigned to the `result` variable.

The code before the `main` function defines the `display_welcome` and `squared` functions. The following code defines the `display_welcome` function:

```
void display_welcome(void)
{
    printf("Welcome to the squaring numbers program\n\n");
}
```

A C function is made up of two parts:

- the first line is the function *head* and identifies the function. We will study this part in more detail in the next section.
- the function *body* which is a set of statements enclosed in braces, in exactly the same way as a compound statement.

The following code defines the `square` function:

```
int square(int value)
{
    int squared_value;

    squared_value = value * value;

    return squared_value;
}
```

Exercise 4.1: Investigating Functions

Step through the "lab4" program in the debugger. Make sure you use the **step into** (shift F7 key) feature to move the execution point into the functions when they are called.

- What happens to the variables defined in `main` when the execution point is inside the body of `display_welcome`?
- What about when the execution point is inside the body of `squared`?
- What variables *are* available when the execution point is inside these functions?

If you do not understand how to step into the `display_welcome` and `square` functions, ask one of the demonstrators.

You should find that any variables defined in the body of the `main` function are not available in other functions. When the point of execution is in the `square` function, there is one variable available (`squared_value`). Variables defined inside the body of a function are known as *local variables* because they are *local* to the body of the function. The section of the program in which a variable is valid (i.e. the function body that contains it) is known as the variable's *scope*.

Exercise 4.2: `scanf` Input Validation

The code which surrounds the `scanf` function (the code inside the `while` loop) validates the user input to make sure it is of the correct type. Try running the "lab4" program and typing in text instead of a number. What happens?

Try stepping through the program in the debugger and give the program the same input. Make sure you understand how the program works before continuing. If you need any help, ask one of the demonstrators.

4.3 Anatomy of a Function

We are now going to look at how a function head is constructed so that you can define functions of your own. There were two functions defined in "lab4": `display_welcome` and `square`. The function head for `display_welcome` looked like this:

```
void display_welcome(void)
```

`void` is a type, like `int` and `double`, except that its main use is in function declarations. In this context `void` means *nothing*. You can see from this line that function heads are made up of three parts:

```
type_of_return_value function_name ( parameter_list )
```

We do not need the `display_welcome` to return a value, so we specify its *return type* as `void`. We do not need any extra information for the function to work i.e. when the function is called there is no need to specify any arguments. So we put `void` in the parentheses after the function name.

When the `main` function called the `display_welcome` function, it did it like this:

```
    display_welcome();
```

You can see from this line that there was no return value to assign to variable, and no arguments were specified. This matches the function head for `display_welcome`.

The `square` function is a bit different. The function head looked like this:

```
int square(int value)
```

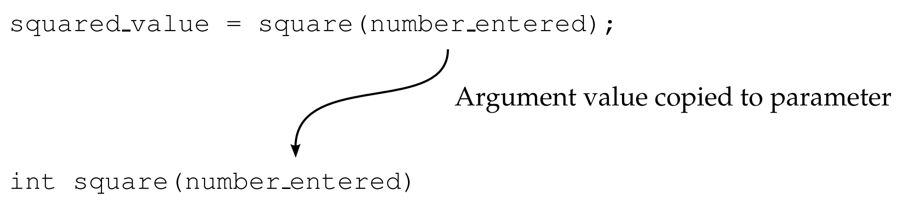
This function head declares the `square` function to return a value of type `int` and to require it to be called with one argument of type `int`. When the argument reaches the function it is given a name: `value`. It is then called a *parameter*. Parameters are very like variables, they have a type and a name. You can declare many parameters for a function in a list separated by commas. Each parameter in the list must have its own type. Parameters also have the same scope as a local variable declared in that function. You can only use the parameter as a variable inside the function body.

When a function is called there must be an argument for each parameter in the function head. The values of the arguments specified in a function call are copied into the parameters before the execution point enters the function body. The values are copied in order i.e. each argument is matched with a parameter, in order.

When the `square` function was called, it looked like this:

```
    squared_number = square(number_entered);
```

Before the execution point (represented by the yellow arrow in the debugger) moves into the `square` function, the value of the `number_entered` variable is copied into the `value` parameter specified in the `square` function head.



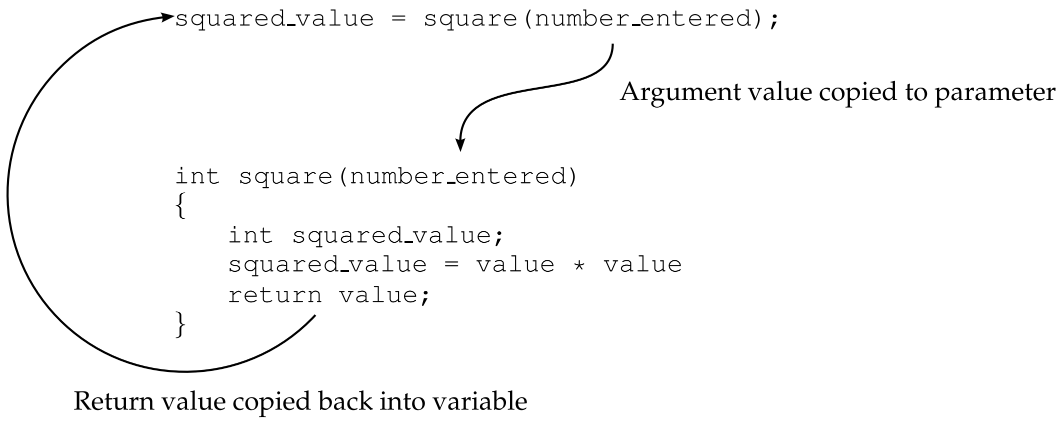
```
squared_value = square(number_entered);

int square(number_entered)
```

When the `square` function has finished everything it needs to do it uses the `return` statement, specifying a value to return. The type of this value **must** match the return type specified in the function head. `square` uses the `return` statement to return the value of an integer variable like this:

```
    return squared_value;
```

The value of the `squared_value` variable is returned to `main` and assigned to the `squared_number` variable.



```
squared_value = square(number_entered);

int square(number_entered)
{
    int squared_value;
    squared_value = value * value
    return value;
}
```

Argument value copied to parameter
Return value copied back into variable

When a function defines a return type, like in the case of the `square` function or the `getch` function, this does **not** mean that you must assign the result to a variable. You don't **have to**. For example:

```
key = getch();
```

calls the `getch` function, which waits for a key press, and when the function returns it assigns the result (the code for the key that was pressed) to the `key` variable. You could also call `getch` like this:

```
getch();
```

In this case, the `getch` function is still called, and it still returns a value, but the compiler throws that value away because you have not assigned it to a variable. In this case the `getch` function will still wait for a key press and return the value of key that was pressed. However, that value will be discarded because no assignment was specified.

Syntax: Function Definitions

```
return_type function_name ( parameter_list or void )
{
    ...
}
```

A function definition declares the name of a function (*`function_name`*) and associates with it a *`return_type`* (which may be `void`) and a *`parameter_list`* (which may also be specified as `void`). Specifying the *`return_type`* as `void` indicates that the function will not be returning a value. Specifying `void` in place of the *`parameter_list`* indicates that the function should not be called with any arguments.

A *`parameter_list`* has the following syntax:

type name[, *type name*]$^{\mathbf{n}}$

For example:

```
int param1, double param2
int param1, int param2, int param3
```

Each parameter has a name and a type. Note that even if the types of subsequent parameters are the same **they must still be specified explicitly**.

Exercise 4.3: Investigating Function Types

Alter "lab4.c" to use `double`s instead of integer values. You will need to think about:

- the `square` function head;
- the variables in `main`;
- the placeholders used by the `scanf` and `printf` functions;
- the message displayed to the user.

Compile and run your program to check that it works. Make sure that it works with non-integer values and that it still handles invalid text input correctly.

Exercise 4.4: Transferring Code into a Function

You are now going to move the input code (the `while` loop with all of its contents) into a separate function. You should call the function `get_double`.

Make sure you can answer the following questions before you start typing:

- What should the return type of the function be?
- How many parameters does the function need?
- How do you want to be able to call the function?

You should end up with a program with four functions (including `main`). The `main` function should be quite short. In general, it is good programming practice to try and keep your `main` function short as it makes your program more readable. Build and execute your program and test it several times to make sure that it works.

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

4.4 Functions: Why Bother?

Why did you move the input handling code into a separate function in exercise 4.4? One answer is that it made a very important part of your program, the `main` function, more readable. Readability is something that you as a programmer should be very worried about. As a general rule, programs are read about 10 times more often than they are written. When you come to write bigger programs, you will need to be able to glance back over code that you have written previously and still understand how it works.

If a program is being designed and written by more than one person, as is usually the case in industry, the problem becomes even greater. One software engineer's code may be read by many others who need to understand what it does easily and quickly. Software may also be used or updated by people other than those who wrote it. This is especially common in industry and can be a real problem.

So, making code more readable is one good reason for functions. However, perhaps the most important reason for functions is that once you've turned a piece of code into a function you can use it as a building block. Just as you have been doing with `printf`, for example. Functions do two things which make them powerful building blocks:

- they hide all the complicated workings of the program and present a simple interface in the form of a function call. You do not have to know how a function works in order to be able to use it. In software engineering, this is called *encapsulation*.

- they allow you to use parts of your program many times without having to re-type the same bits of code. Logically enough, this is called *code reuse*.

If you choose the names of you functions wisely then what they do should be fairly clear just from looking at a function call. This will make your programs easier for you to write and debug. Carefully crafted functions have the potential to be used more than once in a program, which saves you effort. Or you could copy and paste a function out of one program and into another if you think it will be useful.

As an example, the `get_double` function you have just written is very useful. It gets a `double` value from a user and does some input validation. There are likely to be many times in the following labs, and in the assignments, when you need to get a value in this way. You might find it useful to reuse the function by copying it and pasting it into a different program. You can only do this so easily because it is a separate function.

4.5 Positioning Functions in a File

In the example program you have just modified, the functions `display_welcome`, `square` and `get_double` were defined *before* the `main` function.

Exercise 4.5: Changing the Order of Functions in a File

Move the `get_double` function so that it is defined underneath `main`. (You could use "Edit" → "Cut" and "Edit" → "Paste" for this). Try building the project. What errors/warnings do you get? Why do you think there is a problem with putting functions in this order?

When a compiler is turning your source code into machine code, it reads it in order. When it reaches a function call it will try and associate it with a function that it already knows something about. If it doesn't know anything about the function yet it has two options:

- make some assumptions about the information it doesn't know and continue assuming that the assumptions are correct;
- throw its hands up in the air and stop.

In most cases compilers choose the first of these options (even though the second would cause fewer problems). If the compiler subsequently finds some actual information about the function it usually discovers that the information it assumed was wrong, which causes another problem.

The only information the compiler needs to know about a function to allow you to call it is:

- what arguments the function takes and what their types should be;
- what the return type of the function is.

These two pieces of information are both contained in the function head.

C provides a way to declare *just the head* of a function so that you can use it before you define the body of the function. This allows you to put the function definition (the combination of the head and the body) below the place where it is called in the source code. Declaring just the head of a function on its own is called a *prototype*. It should be exactly the same as the head of the function, **but with a semi-colon at the end**. It should be placed outside of any other function.

For example, a prototype for the `square` function would look like this:

```
int square(int value);
```

Note the semicolon on the end of the line. Because the compiler only needs to know the types of parameters in the parameter list (their names don't really matter at this stage) you can miss the names out, like this:

```
int square(int);
```

A prototype like this would usually be placed somewhere near the top of the file, just after any `#include` lines.

Exercise 4.6: Using Prototypes to Change the Order of Your Functions

Declare prototypes for all of your functions, except `main`. Move the order of functions in your file so that `main` is above the other functions. You might like to try and put the functions in some kind of logical order.

If you don't understand how to do this, or if you are still getting errors or warnings when you try to build your code, ask for help.

In the same way that a prototype is all the compiler needs to know about a function, a prototype is also all the programmer needs to know about a function. From this point on the laboratory scripts will use prototypes to define functions in function reference boxes.

Function Reference: Change to Function Boxes

From this point on, functions will be described using their prototype like this:

```
int getch(void);
```

Rather than using the syntax-type method used up to this point, which has looked like this:

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

4.6 The Maths Function Library

So far, on this course, you have been introduced to a number of different functions that are part of the C standard library. This have been functions defined in `stdio.h` and `conio.h`. You are now going to use some mathematical functions, defined in `math.h`.

You have already written your own function for squaring numbers. `math.h` defines a square root function called `sqrt`. It also defines trigonometric functions (`sin`, `cos`, `tan`), inverse trigonometric functions (`asin`, `acos`, `atan`) and many more. You will find full documentation for all of the available mathematical functions in a book on C, or on the internet. Some suggestions for reading are given in Appendix **??**, you will also have been given some suggestions for reading in the lectures. This section introduces some useful maths functions that you may need during the rest of the course.

Function Reference: `sqrt` — Calculates the square root of a number

```
double sqrt(double);
```

The square root function takes one argument, a value of type `double`, and returns another `double` value which is equal to the square root of the argument.

`sqrt` is defined in `math.h`

Function Reference: `pow` — Raises one number to the power of another number

```
double pow(double x, double y);
```

The `pow` function raises the first argument to the power of the second argument and returns the result (i.e. x^y). For example:

```
result = pow(2, 3);
```

would assign the value 8 to the variable `result`.

`pow` is defined in `math.h`

Function Reference: `sin,cos,tan` — Trigonometric functions

```
double sin(double);
double cos(double);
double tan(double);
```

The `sin`, `cos` and `tan` functions take a single argument of type `double`, which is treated as an angle **in radians** (2π radians = 360°). Each function returns a `double` value:

- `sin` returns the sine of the angle;
- `cos` returns the cosine of the angle;
- `tan` returns the tangent of the angle.

`sin`, `cos` and `tan` are defined in `math.h`

Function Reference: `asin,acos,atan` — Inverse Trigonometric functions

```
double asin(double);
double acos(double);
double atan(double);
```

The `asin`, `acos` and `atan` functions take a single argument of type `double`. In the case of `asin` and `acos` this value should be in the range -1 to 1. The return value of each of these functions is an angle **in radians** (2π radians = 360°) represented as a value of type `double`.

- `asin(x)` returns $\sin^{-1}(x)$ which will be in the range $-\pi/2$ to $\pi/2$;
- `acos(x)` returns $\cos^{-1}(x)$ which will be in the range $-\pi/2$ to $\pi/2$;
- `asin(x)` returns $\tan^{-1}(x)$ which will be in the range $-\pi$ to π;

`asin`, `acos` and `atan` are defined in `math.h`

Exercise 4.7: Calculating the Length of a Triangle's Hypotenuse

Alter the "lab4" program to prompt the user for two values. You should call your `get_double` function twice to do this. Treat these two numbers as the lengths of two sides (the opposite and adjacent sides) of a right-angled triangle. Use the `square` and `sqrt` functions to calculate the length of the hypotenuse using the Pythagorean Theorem.

Remember to `#include` the `math.h` header file.

4.7 Graphics Option

4.7.1 Creating More Manageable Code

Open the "graphics1" program you have been working on in previous laboratories. Your graphics program should operate in the following way:

- The user is asked for an initial velocity, which they type in.
- The stick person appears on the screen and the user gets the opportunity to move the stick person left and right using the arrow keys.
- When the user presses 'Enter' the stick person throws an object which is drawn on the screen.
- The user then has another two goes at this.

Your program is now getting quite large. To make it more convenient you should now split in into multiple functions.

Exercise 4.8: Splitting Your Program into Functions

Define suitable functions for:

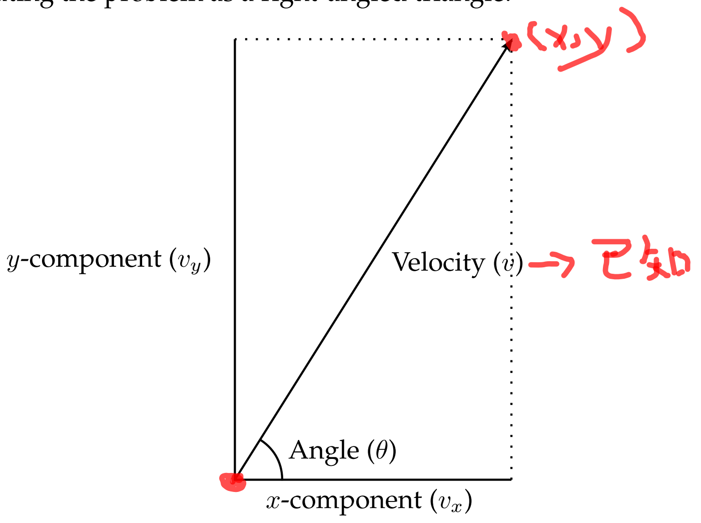
- drawing the stick person at any location on the screen. This function should have two parameters which specify the x- and y-coordinate at which the stick person will be drawn. It should also take a parameter which specifies the colour in which the stick person should be drawn.
- drawing the projectile path. This function should have two parameters which specify the starting point for the projectile in screen coordinates.

Place these functions under the `main` function in your source file. You will need to define a prototype for each function.

If you do not know how to do this, ask for help.

4.7.2 Allowing Control of the Launch Angle

Currently the stick person always throws the object at 45° because the horizontal and vertical velocities are equal. You are now going to change the program so that the user can choose the launch angle of the object between 0° and 90°. The velocity that the user specifies should be treated as the initial velocity of the object. This velocity needs to be *resolved* into horizontal and vertical *components*. These components can be calculated by treating the problem as a right-angled triangle:



The x- and y-components can therefore be calculated using the sine and cosine of the launch angle, θ:

$$v_x = v\cos(\theta) \quad (4.1)$$
$$v_y = v\sin(\theta) \quad (4.2)$$

Exercise 4.9: Allowing the User to Choose the Launch Angle

Alter your program to allow the user to type in the launch angle, as well as the initial velocity, before the graphics window appears. The user should be able to specify an angle between 0° and 90°, but not outside that range. Use the `sin` and `cos` functions from the maths library to resolve the velocity into horizontal and vertical components.

Remember that the maths library functions expect the angle to be **in radians**. You will need to convert the angle from degrees into radians.

4.7.3 Making Your Program More Visually Appealing

This section introduces three new graphics functions that you might find useful:

- the `cleardevice` function clears any drawing you have done from the graphics window, just as if you had re-opened the window;
- the `setbkcolor` function which sets the current background colour;
- the `outtextxy` function allows you to put text in the graphics window.

Function Reference: `cleardevice` — Clears the contents of the graphics window

```
void cleardevice(void);
```

The `cleardevice` function clears the graphics window of any contents. It fills the screen with the current background colour (previously set using `setbkcolor`), and sets the current location (which is used for things like `lineto`) to (0,0).

`cleardevice` is defined in `graphics_lib.h`.

Function Reference: `setbkcolor` — Sets the background colour

```
void setbkcolor(int colour);
```

The `setbkcolor` function sets the current background colour. It takes one argument which is the integer number which identifies the colour. You can use the same colour names as for `setcolor`, which are listed in the function box on page 25.

`setbkcolor` is defined in `graphics_lib.h`.

Function Reference: `initfont` — Initializes the font for the outtextxy function

```
void initfont(void);
```

The `initfont` function sets up the font type used by outtextxy. It has no arguments. It uses the font defined in

```
 data/fixed_font.tga
```

. This font definition (and others) are in the “data” folder that is inside the clab folder.

`initfont` is defined in `graphics_lib.h`.

Function Reference: `outtextxy` — Draws text in the graphics window

`void outtextxy(int x, int y, text);`

The `outtextxy` function places the *`text`* in graphics window at the location specified by `x` and `y`. *`text`* should be enclosed in double quotes, in the same way as with `printf`. Unlike `printf` however, you can't use placeholders with `outtextxy`. The text is drawn in the current colour.

For example:

```
initfont();
outtextxy(100, 250, "Press Enter to Continue");
```

would display the text **Press Enter to Continue** starting at location (100, 250).

Important: You will notice that the prototype given above for the `outtextxy` function does not have a type for the third argument (*`text`*). This is because we have not yet studied how to specify the type for variables containing text. We will cover this in the next laboratory.

`outtextxy` is defined in `graphics_lib.h`.

Exercise 4.10: Using the New Graphics Functions

Now is your opportunity to use the new graphics functions (`cleardevice`, `setbkcolor` and `outtextxy` (together with `initfont`) to add to your graphics program. You could:

- use `cleardevice` to clear the graphics window in between each of the user's throws instead of opening and closing the window;
- use `outtextxy` to display helpful text for the user;
- change the background colour (to something other than black) for the scene using `setbkcolor` and `cleardevice`.

Add as much to your program as you can, using functions appropriately.

4.8 Music Option

Open your "music2" program that have been working on in previous laboratories. Your program should play a simple piece constructed of whole tone scales with appropriate drone tones and dynamics. The piece should play until the user presses a key. By now your code will be getting quite large so the first thing we will do is to split it into functions to make the program more readable and to allow code reuse.

Exercise 4.11: Splitting Your Program into Functions

Define a function that produces a number of whole tone scales. The function should have parameters for:

- the number of scales to play;
- the pitch of the starting note in the scale;
- the number of notes in each scale;
- the velocity to use for the scales.

You can add other parameters if you want. Place this function under the `main` function in your source file. You will need to define a prototype for the function.

Currently your program repeats the same sequence of scales until the user presses a key. We can make a music program more interesting by using randomness to try to create a program which improvises. C contains standard functions for producing random numbers, but they are not easy to use. The music library contains a much simpler random number function called `random_number`.

Function Reference: `random_number` — Generates a random number

```
int random_number(int lower_range, int upper_range);
```

The `random_number` function returns a random number which could be anywhere in the range `lower_range` to `upper_range`, including the values `lower_range` and `upper_range`. For example:

```
pitch = random_number(60, 71);
```

will assign a value to `pitch` that could be 60, 71 or any integer value in-between.

`random_number` is defined in `amio_lib.h`.

Exercise 4.12: Improvising with Whole Tone Scales

Using the function you have just written which plays whole tone scales, try to use the `random_number` function to make your program improvise pieces of music based on whole tone scales. Drone tones and dynamics should also be improvised. How can you make it sound reasonably musical?

All the improvisation your program does at the moment is based on whole tone scales. Whole tone scales have an interesting effect, but their use can become quite constraining. You now have the chance to add to your program to make it insert passages of freer improvised melody in-between sections of whole tone scales. This will all be based on random numbers.

Here are some things to think about:

- you might like to control the lengths of the free and whole tone scales sections to enforce some kind of structure with recognisable phrasing;

- you could use a random number to decide whether you are about to improvise a phrase based on whole tone scales, or completely random notes;
- you should use random numbers to allow the program to make choices about pitch and note length (for example), but you might want to place certain constraints on what it can do to make sure the results still sound musical.
- you will have to be careful with the improvisation between the whole tone scale phrases to ensure that the piece does not become too disjointed.

Exercise 4.13: Adding to Your Improvisation Program

Create a separate function which chooses a note at random and plays it. You might like to call it `random_note`. You should use this function to create the free phrases. You will need to think about what parameters the function needs so that you can control the random notes it produces. You might like to control (and have parameters for):

- the range in which the pitch can lie;
- the velocity of the note;
- the length of the note;
- whether there is a rest before the note.

Improvising in this way is hard, but see how musical you can get it to sound. You might find that it is easier to produce a convincing piece if the tempo is kept fairly slow (e.g. *adagio*).

4.9 Summary

In this laboratory you should have learned why functions are useful and how to create functions of your own. You should understand how to specify parameters and return values, and how to declare function prototypes so that you can choose the order in which functions are declared.

Function prototypes will be used in all function boxes from now on.

You should understand how to use the `fflush` function to control the way `scanf` works and to ensure that user input is validated correctly. In the second part of the lab you were introduced to some more mathematical functions which you might find useful.

In the graphics option you had to put your knowledge of functions into practice, and you improved your program to allow the user to choose the launch angle for the projectile. In the music option you were introduced to the `random_number` function which you used to try to produce a program which improvises, and found out how hard it is to get it to sound musical!