THE UNIVERSITY *of York*
Department of Electronics

Computer Programming Using C

Laboratory Scripts 2007–2008

Peter Mendham
with
Dr Andy Hunt, Dr Julian Miller,
Andy Pomfret and Dr Steve Smith

September, 2007

Contents

Function Reference

Syntax Reference

Introduction

Overview

This section gives a brief introduction to the structure of the C Programming Laboratory course, and the way in which these scripts are written.

The Structure of the Lab Course

There is a C programming laboratory once a week, every week from 2 to 10, in both the Autumn and Spring terms. There are four laboratories, weeks 2 to 5, in the Summer term. There is a laboratory script for each lab in the Autumn and Spring terms. The content of the summer term labs will be clear when you come to do them. You can work through each lab at your own pace, however, you will be expected to complete one laboratory's script before the beginning of the next lab. If you have not finished a script by the end of a lab you may have to complete it in your own time.

Each lab contains two optional sections:

- music option, intended for Music Technology students;
- graphics option, intended for all other students.

You must choose one of these options. If you have time, you may find it fun to attempt parts of the other option. The C programming course is assessed through two assignments: one in the middle of the course, and one at the end. For each assignment there will also be two options: music and graphics. If you have managed to learn techniques from the other option it will make your assignments all the more impressive!

How These Scripts are Written: Structure and Conventions

These scripts are intended to be worked through in order, however, some sections are highlighted to make it easier for you to refer back to important information later in the course. The normal text of the script will contain both information and instructions for the tasks you should carry out. You must make sure you read the scripts carefully, especially before asking for help — you may have missed something important.

Sections where you need to think carefully about what you are doing are shown as exercises and appear like this.

Exercise 0.1: An Example Exercise

Exercises appear in boxes like this one.

Sometimes you need to do something but the instructions are not given in an exercise box. Usually this is because the action you need to carry out is fairly simple. To make sure these parts of the lab script are easy to find, they have a picture of two gears next to them in the margin.

Like this.

There are two other kinds of special box:

- *syntax boxes* which tell you information about how to form valid statements in the C language; syntax boxes look like this

Syntax: Some Syntax

```
the syntax goes here...
```

. . . and the description of the syntax goes here.

- *function boxes* which tell you how to use C language functions; function boxes look very similar to syntax boxes:

Function Reference: `A Function` — which does something

```
the function goes here...
```

. . . and the description of the function goes here.

Don't worry that you don't know what a function is yet — this will become clear when you start the first lab!

The descriptions of both syntax and functions use some special symbols. You should try and remember what these mean, ready for when you meet your first syntax (or function) box:

- if something appears in square brackets, [`like this`], then it means that it is *optional*;
- if something appears in square brackets with a small 'n' above and to the right of it, [`like this`]^{n}, then it means that it is optional and can be *repeated* as many times as you want;

Don't worry if this does not seem clear now, just remember that when you meet the first syntax (or function) box you can look back at this section to help you understand the information.

Before You Begin

Overview

This section contains details of things you will need to do in order to prepare for writing and compiling C programs. This has been put in a separate section to make it easier to refer to as you work your way through the lab scripts.

The first two tasks must be completed before you can start with anything else, **but they should only ever need doing once**:

- mapping your N Drive;
- creating a folder called "clab" for your work.

The next three tasks are things you will need to do quite a lot. The instructions are in this section, rather in the main lab scripts, so that you can refer to them:

- copying an existing project from the N drive;
- opening the integrated development environment (IDE);
- opening an existing project in the IDE.

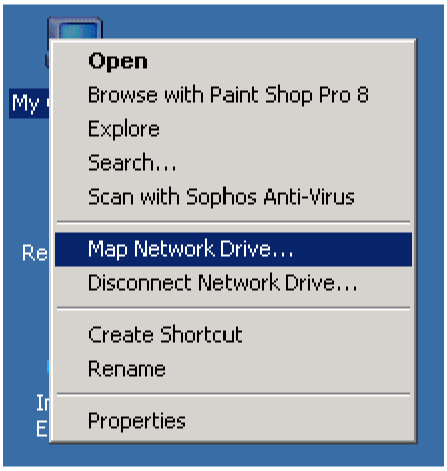
The lab scripts will tell you when you need to copy a project and which project you should copy.

Mapping Your N Drive

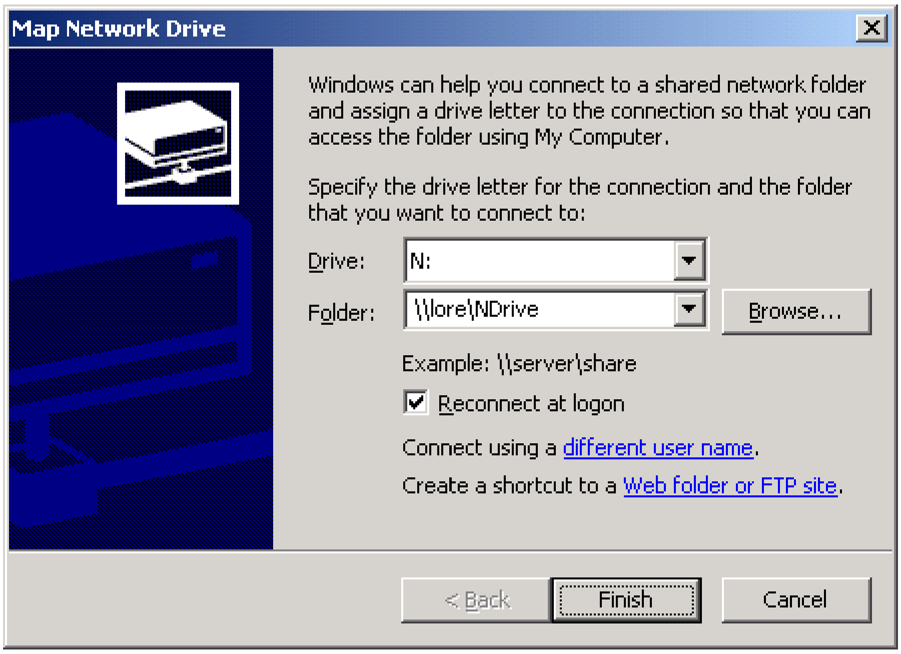
Configuring or *mapping* a drive provides a network connection to one of the Department of Electronics servers and gives you access to all of the material needed for this course. You should only need to complete this once.

- Using the right-hand mouse button, click on the "My Computer" icon on the desktop.

- You should get a menu which looks like the one shown below.


My
Re
In
E
Open
Browse with Paint Shop Pro 8
Explore
Search...
Scan with Sophos Anti-Virus
Map Network Drive...
Disconnect Network Drive...
Create Shortcut
Rename
Properties

- Select the item which says “Map Network Drive...”. A dialog box will appear. Fill in the details exactly as shown in below and then click “Finish”.



Map Network Drive

Windows can help you connect to a shared network folder and assign a drive letter to the connection so that you can access the folder using My Computer.

Specify the drive letter for the connection and the folder that you want to connect to:

Drive: N:

Folder: \\lore\NDrive

Example: \\server\share

☑ Reconnect at logon

Connect using a different user name.

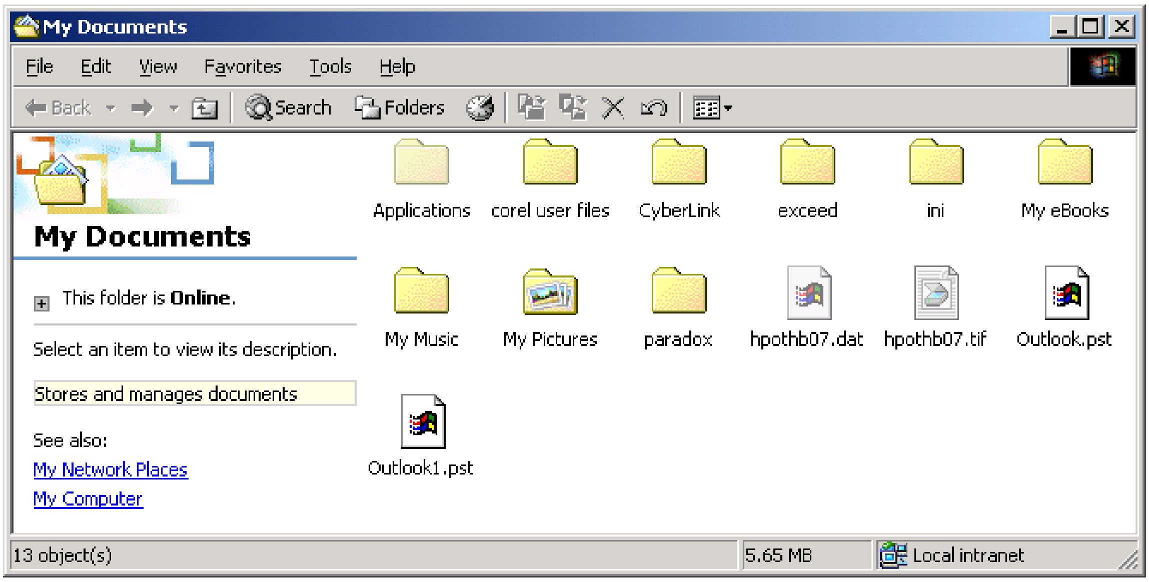
Create a shortcut to a Web folder or FTP site.

Creating a Folder For Your Work

It is a good idea to have a folder that will be used for all of your work during the C Programming course. You will only need to create this folder once.

- Double click on the “My Documents” icon on your desktop.

- A window like the following should appear.



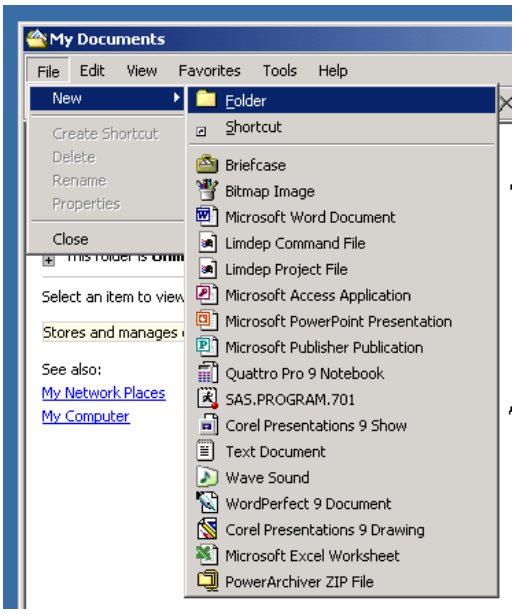
This folder is **Online**.

Select an item to view its description.

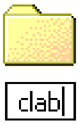
Stores and manages documents

See also:

- From the menu select “File” → “New” → ”Folder”. This will create a new folder on your home drive. The folder will be called “New Folder”.



- Type to rename the folder, call it “clab”.



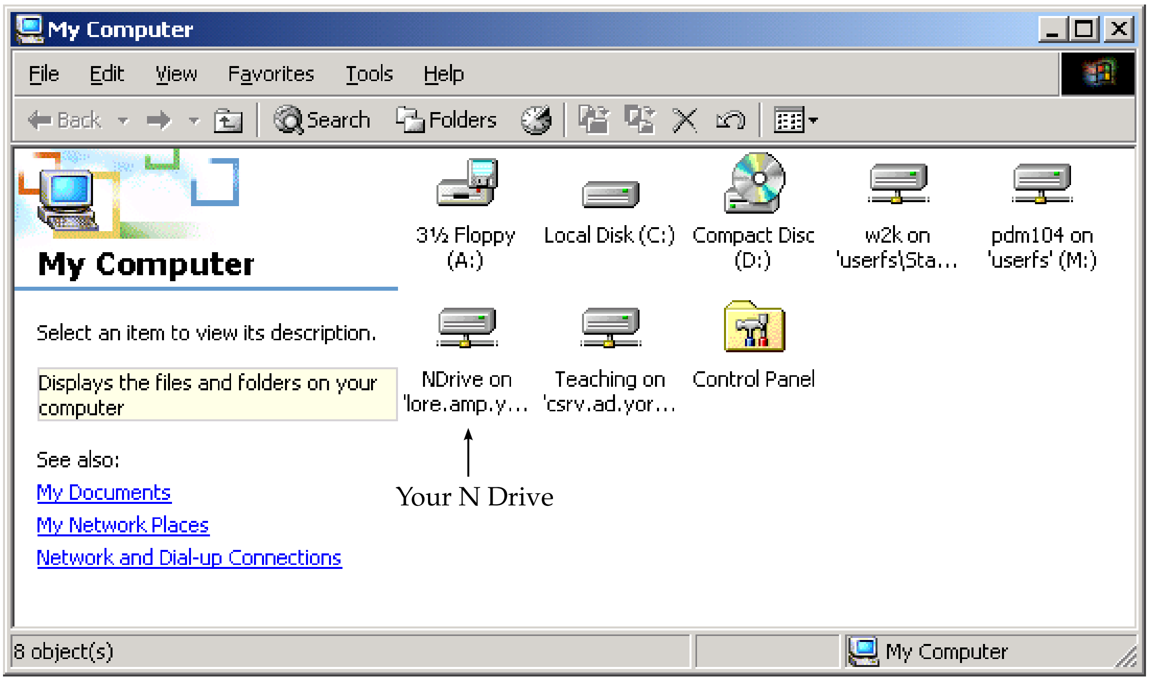
This is where you will keep all your projects for the C laboratory course.

Copying a Project

Each of the labs in the course will use one or more projects that have already been created for you. They are stored on the N drive but they must be copied into your “clab” folder before you can use them. This will need doing each time you want to begin work on a new project.

- Double click on the “My Computer” icon on your desktop

- A window like the following should appear.

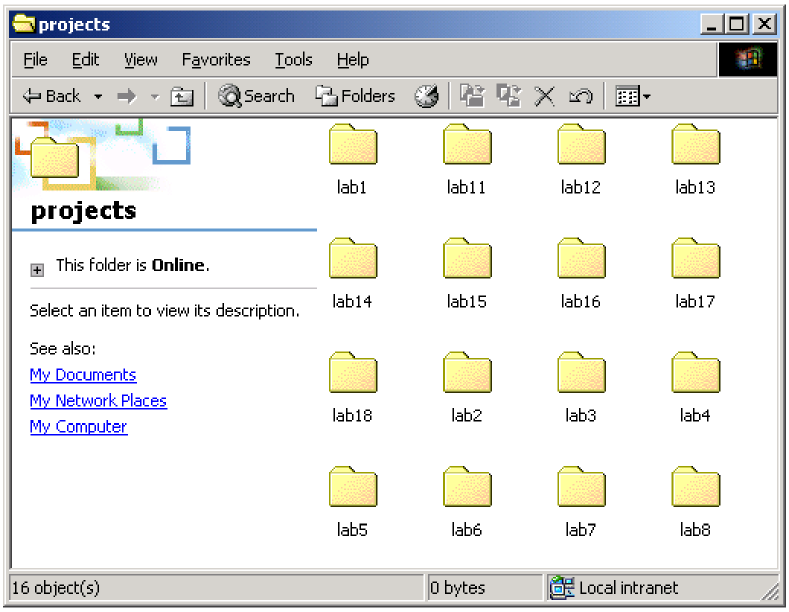


Select an item to view its description.

Displays the files and folders on your computer

- Double click on the icon for the N drive to view its contents. Then double click on the following folders to navigate your way to the projects folder: “course” → “elec” → “examples” → “c” → ”projects”. You should have a window like the following.

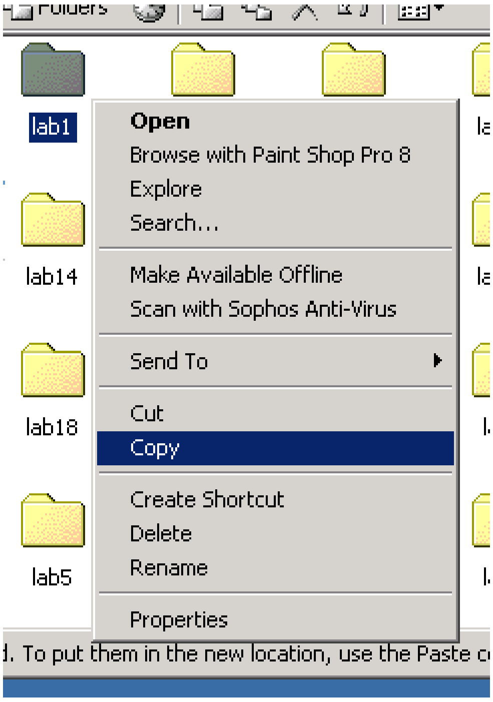
  (As an aside, this navigation is often written as a *path* in the following way: `N:\course\elec\examples\c\projects`)



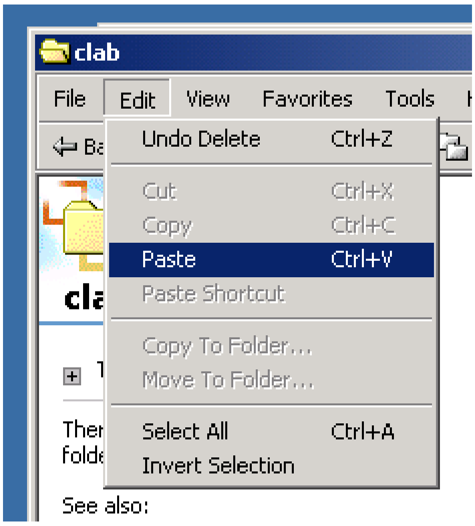
This folder is **Online**.

Select an item to view its description.

- Using the right-hand mouse button, click the folder that corresponds to the project you want to copy. In this example we are copying the project “lab1”. Select the “Copy” item from the menu.



- You can close the projects folder windows now if you want to. Double click the “My Documents” icon on the desktop. When a window appears, double click the “clab” folder that you created to view its contents. Click on the “Edit” menu, and select “Paste”.

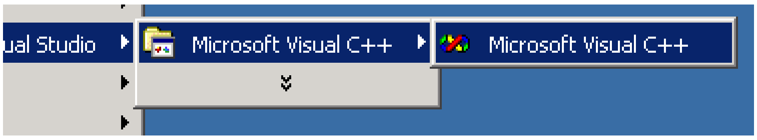


- The folder containing the project will have been copied into your “clab” folder.

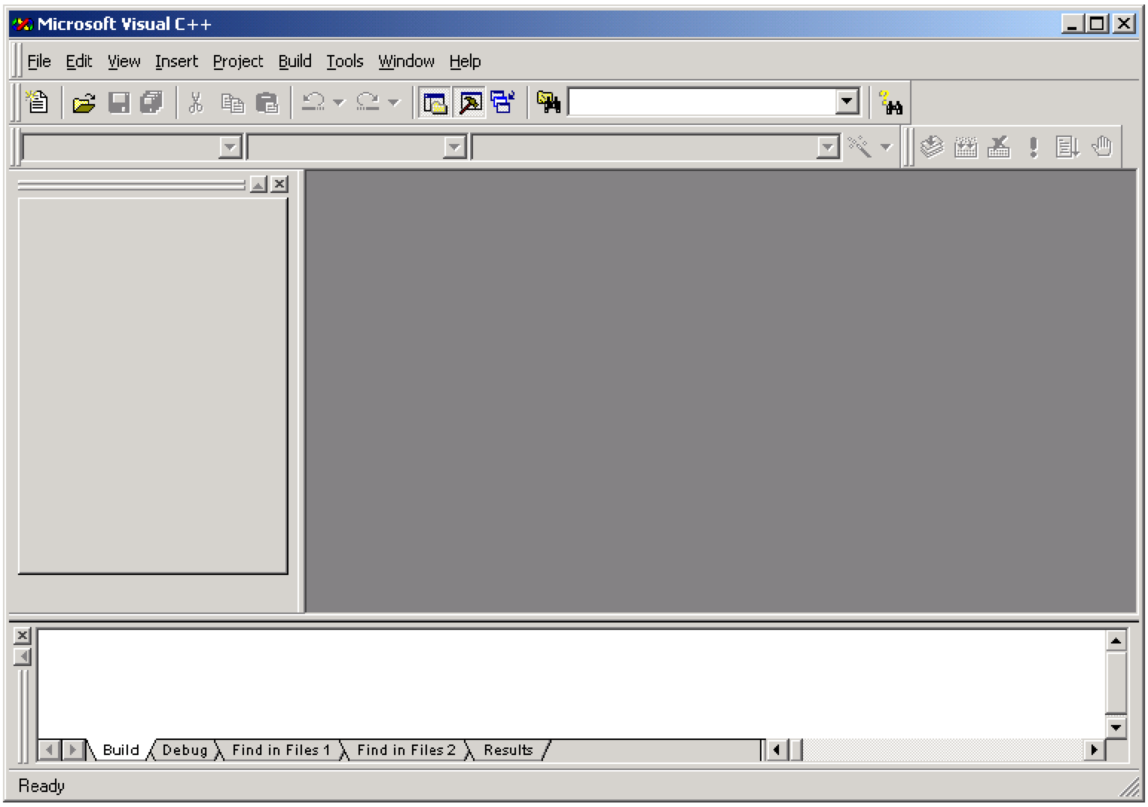
Opening the Integrated Development Environment (IDE)

You will do all your programming using Microsoft Visual Studio which is an integrated development environment, or IDE. You will need to open the IDE every time you want to work on a C program.

- Click on the “Start” button on the task bar and follow the menus “Programs” → “Programming” → ”Microsoft Visual Studio” → “Microsoft Visual C++” and then select the icon for “Microsoft Visual C++”.



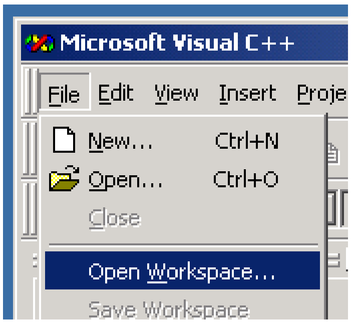
- The IDE window should look like this.



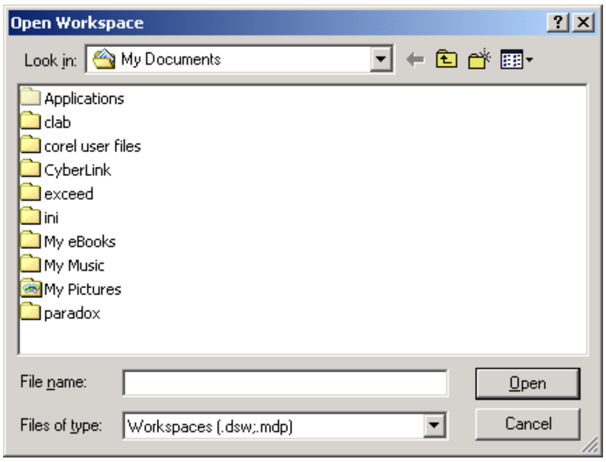
Opening a Project in the IDE

To begin or continue work on a project you have copied you must open the project in the IDE, each project is contained in a *workspace*.

- With the IDE open, click on "File" → "Open Workspace..."



- You should now be viewing the contents of "My Documents".



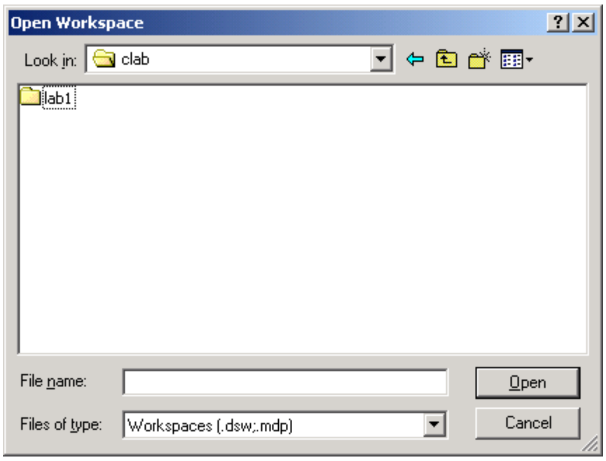
Look in: My Documents

Applications
clab
corel user files
CyberLink
exceed
ini
My eBooks
My Music
My Pictures
paradox

File name:

Files of type: Workspaces (.dsw;.mdp)

- Double click on the "clab" folder to view its contents.



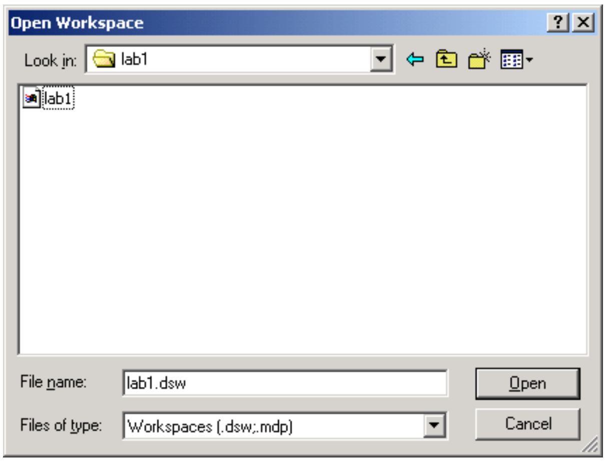
Look in: clab

lab1

File name:

Files of type: Workspaces (.dsw;.mdp)

- Double click the folder of the project you want to open, in this example we will be loading the project "lab1".



Look in: lab1

lab1

File name: lab1.dsw

Files of type: Workspaces (.dsw;.mdp)

- Select the workspace for the project. This will be a file with the same name as the project, in this example "lab1". Then click the "Open" button to open the workspace.

Summary

If you have read this section in full then you should have learned:

- how to map you N drive;
- how to create a folder for your work.

which are things you should only have to do once.
You should also have learned how to carry out tasks you will need to do in every laboratory:

- how to copy a project;
- how to open the IDE;
- how to open a project in the IDE.

All these tasks will be an essential part of the first lab.

Part I

Autumn Term Laboratories

Laboratory 1

A Simple C Program in Microsoft Visual Studio

1.1 Overview

In this first laboratory you will be introduced to the Microsoft Visual Studio integrated development environment (IDE). You will be using this software to write and test C programs.

This lab will also introduce the following concepts:

- the structure of a simple C program;
- variables, their names and types;
- how to do simple mathematics in C;
- how to call functions;
- how to use the `printf` function.

You should understand all of these concepts by the end of the lab. You will also have been introduced to either the graphics or the music facilities.

1.2 Setting Up

Before you begin this lab you should have read the “Introduction” and “Before You Begin” sections.

Before we can begin the lab, it is essential that you have carried out the first two tasks in the section “Before You Begin”. These are:

- mapping your N drive;
- creating a folder called “clab” for your work.

For this lab we will be using the project “lab1”. Following the instructions in the section “Before You Begin”:

- copy the project to your “clab” folder;
- open the IDE;
- open the project in the IDE.

If you have any questions about any of this steps just ask the demonstrators.

1.3 Introducing the IDE

This laboratory course uses Microsoft Visual C++ Version 6.0. Although the software supports programming in the C++ language as well as the C language, we will just be using C. We will be using Microsoft Visual Studio to create programs using Microsoft Visual C++. This software is called an *Integrated Development Environment* or IDE because it combines things that, many years ago, used to be separate. They are:

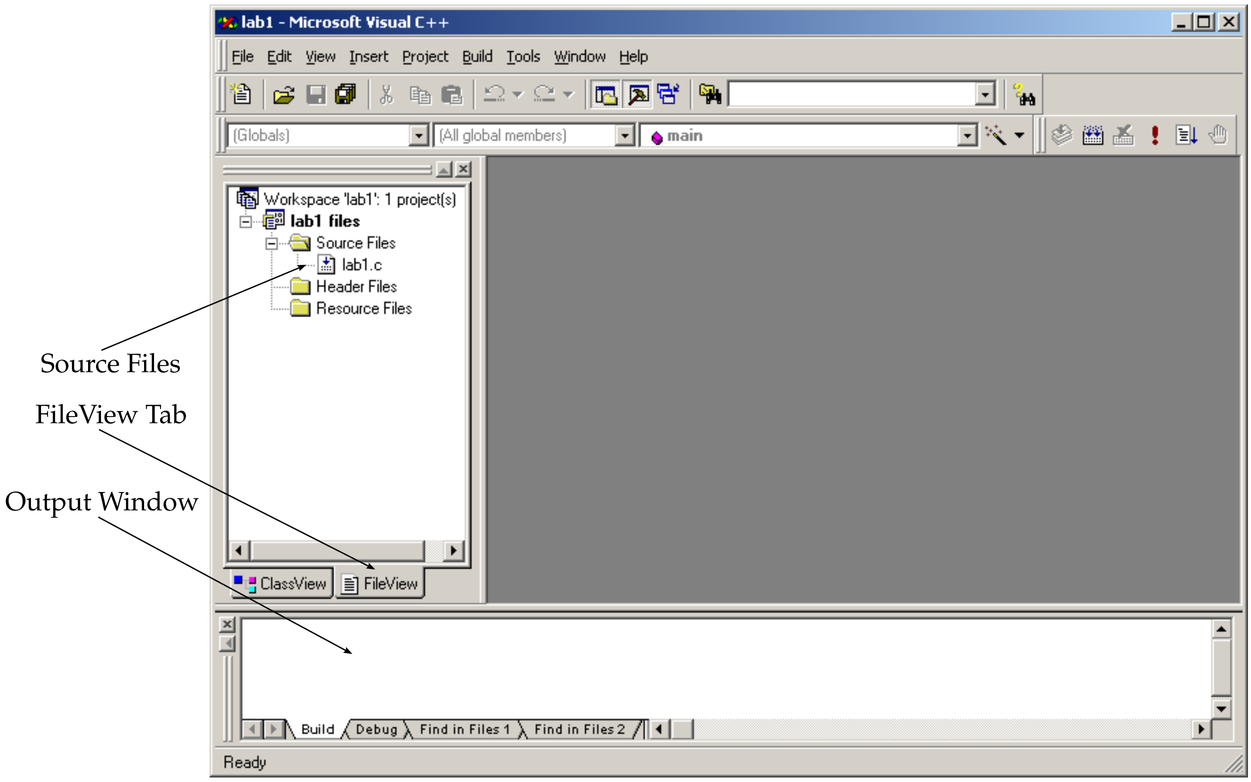
- an **editor** which allows you to edit the text files that contain the C language which will become your program (these are often known as *source files*);
- a **compiler** which takes the source files and translates them from text in the C language to *machine code* which the computer can understand and execute. All programs on your computer are in machine code.

This is all you need to create a program in C, but an IDE also usually contains:

- a **debugger** which allows you to test out your programs in a very controlled way and diagnose problems with them.

We will be making a great deal of use of all of these features, right from the start.

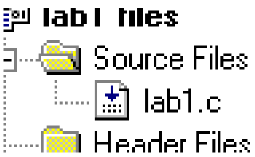
The Visual Studio IDE you have in front of you should look something like this.



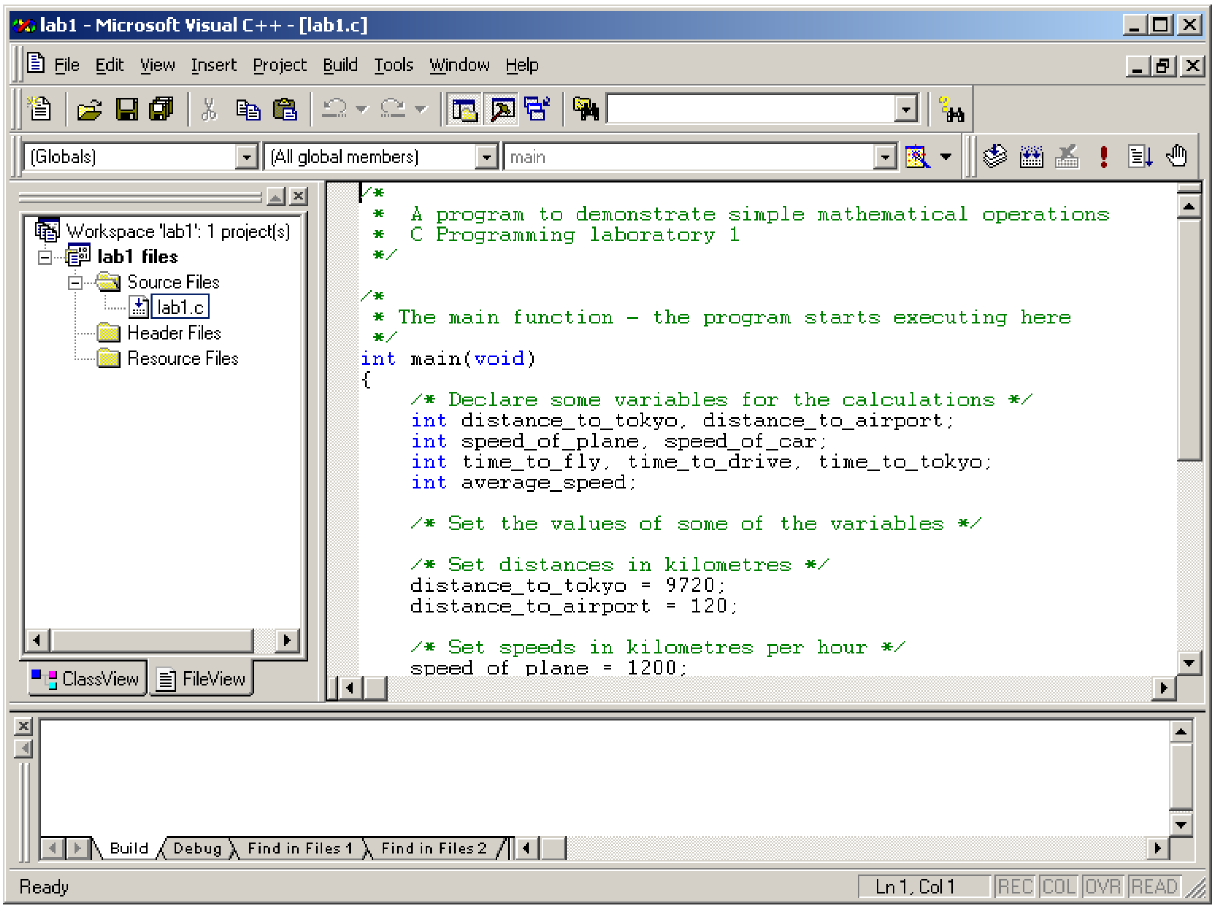
The C language source files that make up the program are shown in the left pane. In this example there is only one source file, however, projects may contain many source files. If you can't see a source file list similar to the one shown above, click the FileView tab. When we perform tasks such as compiling or debugging programs, the IDE will sometimes give you messages, such as errors. These will be shown in the output window. If the IDE behaves in a way that you didn't expect, this pane is a good place to look at first.

1.4 Looking at Your First C Program

Double click on the icon for the source file "lab1.c".



The C language *source code* will appear in the editor window. This should look like this.



```
/*
 *  A program to demonstrate simple mathematical operations
 *  C Programming laboratory 1
 */

/*
 * The main function - the program starts executing here
 */
int main(void)
{
    /* Declare some variables for the calculations */
    int distance_to_tokyo, distance_to_airport;
    int speed_of_plane, speed_of_car;
    int time_to_fly, time_to_drive, time_to_tokyo;
    int average_speed;

    /* Set the values of some of the variables */

    /* Set distances in kilometres */
    distance_to_tokyo = 9720;
    distance_to_airport = 120;

    /* Set speeds in kilometres per hour */
    speed_of_plane = 1200;
```

The source code for this program is printed below. This is the same as the code in the “lab1” project.

```
/*
*  A program to demonstrate simple mathematical operations
*  C Programming laboratory 1
*/

/*
* The main function - the program starts executing here
*/
int main(void)
{
    /* Declare some variables for the calculations */
    int distance_to_tokyo, distance_to_airport;
    int speed_of_plane, speed_of_car;
    int time_to_fly, time_to_drive, time_to_tokyo;
    int average_speed;

    /* Set the values of some of the variables */

    /* Set distances in kilometres */
    distance_to_tokyo = 9720;
    distance_to_airport = 120;

    /* Set speeds in kilometres per hour */
    speed_of_plane = 1200;
    speed_of_car = 100;

    /* Calculate time taken to get to Tokyo */

    time_to_fly = (distance_to_tokyo - distance_to_airport) / speed_of_plane;

    time_to_drive = distance_to_airport / speed_of_car;

    time_to_tokyo = time_to_fly + time_to_drive;

    /* Calculate the average speed */

    average_speed = distance_to_tokyo / time_to_tokyo;

    return 0;
}
```

We will now go through this code, a line or two at a time. Make sure that you understand it before continuing.

The first important thing to notice is that everything between `/*` and `*/` is a *comment*. This means that it is ignored.

Syntax: Comments

```
/* ... */
```

Comments are used for adding text to a source file which will be ignored by the compiler. This is very useful for adding notes which will remind you about what a particular part of your program is supposed to do. In Microsoft Visual Studio when you type a comment the editor will understand and colour it green. The different colour helps you see easily which parts of your program are comments are which are not.

It is good programming practice to comment your programs well. A good rule of thumb when writing comments is to write them for someone who understands C but doesn't understand how your program works.

The program begins with the line:

```
int main(void)
```

This states that all of the C code between the two *braces* (or *curly brackets*), { `...` }, is part of a *function* called `main`. Every C program has a `main` function, which is the place where the program starts executing. At the moment, the `main` function is the only function in this program. Later on in the lab you will use, or *call*, some other functions that are often used in C. In laboratory 5 you will learn to create your own functions.

The next part of the program lists the *variables* we are going to use in this function:

```
    int distance_to_tokyo, distance_to_airport;
    int speed_of_plane, speed_of_car;
    int time_to_fly, time_to_drive, time_to_tokyo;
    int average_speed;
```

Variables are used in programs in much the same way that `x`, `y` and `z` (for example) are used in algebra, to represent values. The computer needs us to tell it what variables we will be using *before we use them*. To do that we need to tell it what the variable is called and what kind of information it will hold. This is called the variable *type*. In C terminology we *declare* the variable and its type.

Syntax: Variable Declarations

type name[, *name*]^{n} ;

e.g.

```
    int an_integer_variable;
    double a_real_variable;
    int one_int_variable, another_int_variable;
```

Variables are placeholders for values, just like in algebra. In C you must declare a variable before you can use it. To declare it you must give it a name and a type. The types you will use the most in C are:

- `int` for integer numbers, both positive and negative such as 500, 2 and -20;
- `char` for single characters, such as 'd' and '!';
- `double` for decimal or *real* numbers, both positive and negative, such as 0.05, 3.141 and -2.718.

Variable names can be as short as a single letter or very long indeed. It helps to give variables descriptive names to help you remember what they are for. A variable name, or *identifier*, can contain any of the following characters:

- upper and lower case letters, a-z, A-Z;
- the digits 0-9;
- the underscore character '_ '.

Variable names may not start with a digit. Variable names are *case sensitive*, that is, upper and lower case letters are treated as being different.

Some examples of valid variable names are:

- `distance_to_tokyo`
- `i`
- `anotherCounter`

Some examples of invalid variable names are

- `2me2you` (it starts with a number)
- `dot.dot` (it contains an invalid character)

Notice that there is a semi-colon (;) at the end of the line. Every C program is split into *statements* in a very similar way that English is split into sentences. Every sentence in English ends with a full stop (or period). Every statement in C ends with a semi-colon.

The next two lines of code are:

```
distance_to_tokyo = 9720;
distance_to_airport = 120;
```

This statement is called an *assignment*. It copies whatever is on the right-hand side of the equals sign ('=') to whatever is on the left-hand side. In other words, the variable on the left is *assigned* the value on the right. The item on the left-hand side must be a variable. An assignment is very different to the way an equals sign is used in mathematics. An equals sign in maths declares a relationship between the left- and the right-hand sides e.g. $x + y = z$ in maths tells you that $x + y$ must be equivalent to z. The equals sign in C copies information. The first line of code above copies the value 9720 into the variable `distance_to_tokyo`.

This means you can write statements which make a lot of sense in C, but no sense in maths, such as:

```
        someCounter = someCounter + 1;
```

Which means that the variable `someCounter` will be one greater in value after this line than it was before. (Because you take the value of `someCounter`, add one to it and copy the result back into `someCounter`).

The next few lines carry out some mathematical operations. All the lines that do maths are also assignments. They carry out a calculation and assign the result to a variable. For example the line:

```
time_to_drive = distance_to_airport / speed_of_car;
```

takes whatever the value of the variable `distance_to_airport` is, it then divides is by whatever the value of the variable `speed_of_car` is and copies the result into the variable `time_to_drive`.

Syntax: Mathematical Operators

```
+ - * / ( )
```

The example source code "lab1.c" contains a few of the mathematical operators that C supports. The standard ones are:

- `+` (addition);
- `-` (subtraction);
- `*` (multiplication);
- `/` (division).

When there are multiple operators on a line some basic rules are followed: operators are looked at by the C compiler from right-to-left and a special order is followed. This order is known as *operator precedence* and means that multiply and divide are considered 'more important' than add and subtract. For example

```
        6 + 4 / 2
```

gives a result of 8. This is the same as the way that they are used in mathematics. Quite often you want to clarify what you mean, you can use parentheses (brackets), '`(`' and '`)`', for this purpose. For example

```
        (6 + 4) / 2
```

gives a result of 5.

The last line of the `main` function, before the closing brace (}) is:

```
return 0;
```

This stops the computer from executing any more of the `main` function, and, because the `main` function was the first function to run, there is nothing to run afterwards. So the program ends. The value 0 is called a *return value*. We will see a little more about this later in this lab, and a lot more about it when you come to write your own functions other than `main`.

Exercise 1.1: Understanding "lab1.c"

Before we continue you should make sure that you understand what is happening in the C program source code.

- What do you think the program is trying to do?
- Repeat the calculations with a calculator. What results do you get? Write them down, as we will need them very shortly.
- When you compile and run the program, what do you think it will appear to do?

If you can't answer these questions, ask one of the demonstrators for help.

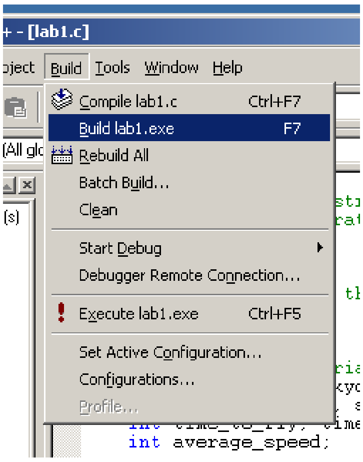
1.5 Compiling, Executing and Debugging a C Program

We are now going to use the IDE to compile, execute (run) and debug the C program we have just been looking at. First we will look at how to compile a program using the IDE.

Compiling a program happens in two stages:

- **compiling** where each source file is translated into an intermediate format called an *object file*;
- **linking** where all the object files are linked together to form a single machine code file — an *executable*.

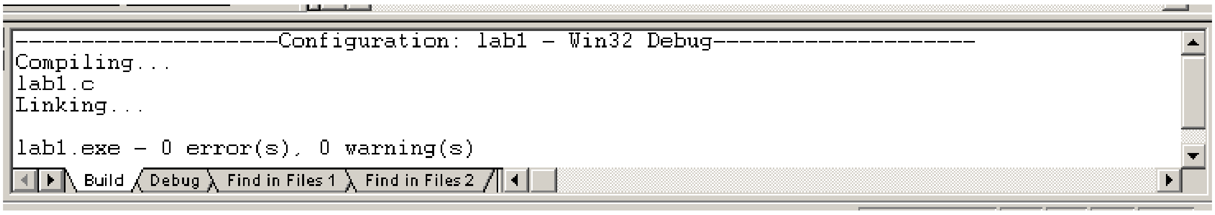
On Microsoft Windows all executables end with the letters ".exe". The commands for instructing the IDE to compile your project are on the "Build" menu.



Let us look at the top three menu items:

- “Compile lab1.c” will compile the current file (“lab1.c”) to produce an object file;
- “Build lab1.exe” will do any compiling that needs to be done of any files in the whole project, then it will link the files together to produce the executable “lab1.exe”;
- “Rebuild All” forces the compiler to re-compile everything from scratch even if it thinks it is not necessary.

We will be using the “Build” item the most often. Select this item now. The output window will show the messages generated by the compiler. It should look like this.



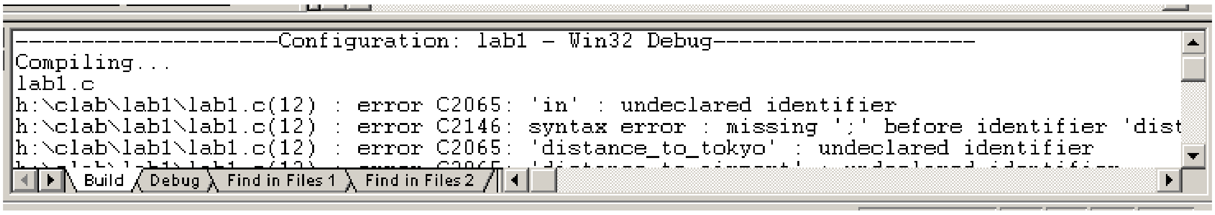
```
--------------------Configuration: lab1 - Win32 Debug--------------------
Compiling...
lab1.c
Linking...

lab1.exe - 0 error(s), 0 warning(s)
```

You can see that the IDE ran the compiler, which produced an object file, and the linker, which produced an executable. It also tells you that there were no problems during this process i.e. there were no errors and no warnings.

Errors and warnings occur when there are problems with the C source code:

- **errors** are when the source code is not written in proper C language syntax and the compiler doesn’t understand, the compiler will give an *error message* to try to help you to understand the problem;
- **warnings** are when the source code is written in valid C but the compiler thinks you might have made a mistake. It may be that you have mistyped something or the flow of your program is logically wrong.

Only errors will stop the compiler and linker from producing an executable. Warnings will not stop it, but you should still pay a great deal of attention to them! Errors and warnings appear in the output window like this. (This is just an example. You shouldn’t be seeing anything like this at the moment).



```
--------------------Configuration: lab1 - Win32 Debug--------------------
Compiling...
lab1.c
h:\clab\lab1\lab1.c(12) : error C2065: 'in' : undeclared identifier
h:\clab\lab1\lab1.c(12) : error C2146: syntax error : missing ';' before identifier 'dist
h:\clab\lab1\lab1.c(12) : error C2065: 'distance_to_tokyo' : undeclared identifier
```

Each line in this output is an error. Looking at the first line, the text `h:\clab\lab1\lab1.c` is the name of the file the compiler was working on. `(12)` means that the error was on line 12. `error C2065` is the the error number, which you can usually ignore. All the remaining text is the error message. If you cannot understand an error message ask the demonstrators for help.

We have now compiled our program. There are two ways in which we can run it:

- find the executable file, and double click on it to run it;
- ask the IDE to run it for us.

The executable file is in a subfolder of the project, which has the following path:

```
h:\clab\lab1\debug\lab1.exe
```

You can look for it if you like. The easier way is to use the "Execute lab1.exe" option on the "Build" menu. Click on that option now.

You should find that a new window appears which is black and contains says "`Press any key to continue`". That means that your program has run and has already finished. Very unimpressive so far!

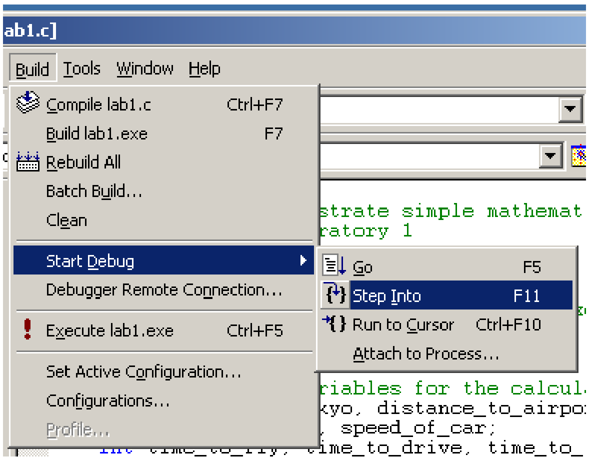
You should have noticed from the program code that there are no statements telling the computer to display any information to the screen. So it doesn't. When you are writing a C program the computer will not do anything that you haven't told it to do. Sometimes that is useful, other times, less so.

In order to find out whether your program is working or not, and whether you understand what it is trying to do, we are going to look inside the program whilst it is running. This process is called *debugging* as it is usually used to try and find errors, or *bugs*, in your program and remove them.

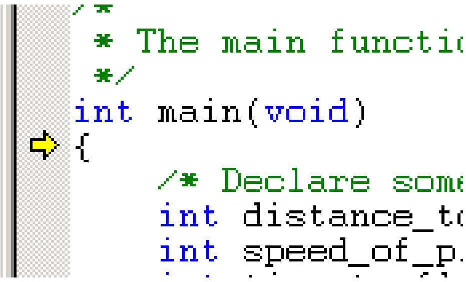
The debugger is built in to the IDE. We will be using two of its most important features:

- **stepping**, which allows you to execute your program a line at a time, while the editor shows you which line will be executed next;
- **inspecting**, which allows you to look inside variables to find out what their values are, whilst the program is running.

To begin stepping through your program, select the menu item "Build" → "Start Debug" → "Step Into".

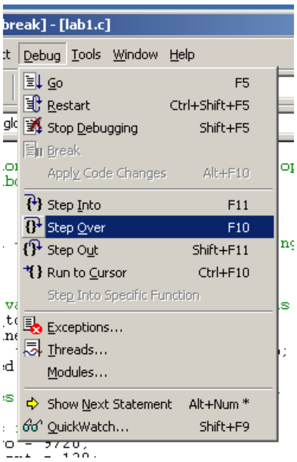


Your program will begin executing and will pause, just before it reaches the first line. The editor will show where it is up to with a small yellow arrow.



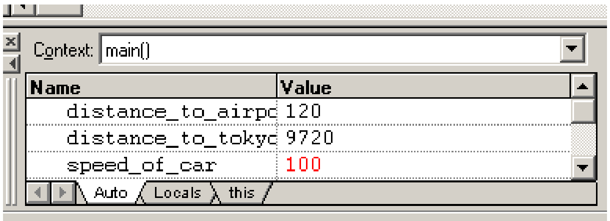
```
/*
 * The main functi
 */
int main(void)
{
    /* Declare some
    int distance_t
    int speed_of_p
```

You can now *step* through your code a line at a time by using the "Step Over" item on the "Debug" menu, which will have appeared once you started the program running.



You can also use the F10 key on your keyboard. This will move the yellow arrow down the left-hand side of your source code, a line at a time. The debugger will skip lines that it cannot execute, such as comments and variable declarations.

When you instructed the IDE to begin debugging it changed the layout of the main window to contain the *watch window*.



Context: main()

Name	Value
distance_to_airpc	120
distance_to_tokyc	9720
speed_of_car	100

This shows you what the contents of variables are at the current point in the program. As you step through the program you should be able to see the values change in this window. The "Auto" tab inside the watch window (the one that is displayed by default) shows you the variables that have changed most recently. Sometimes a variable that you want to see the value of is not on this list. By clicking on the "Locals" tab, the watch window will show a list of all variables that are *local* to the current function (the `main` function in this case). In the case of the current program, the "Locals" tab will show a list of all variables and their values.

When you reach the last line of the program, **do not continue to step**. The debugger will start to try and execute code that is not part of your program, which can get very confusing. Choose the "Go" option from the "Debug" menu to allow the program to terminate.

Exercise 1.2: Using the Debugger

Use the debugger to step through the program. Check the watch window after each time you step and check that you understand what has changed and why.

- Do you understand what the program is doing?
- How do the calculations compare with the results you obtained with a calculator?
- If the results are different, why are they different?

If you can't answer these questions, ask one of the demonstrators for help

1.6 Displaying Text on the Screen

We are now going to add a statement to the program which calls a function to put some text on the screen. The function we are going to use is called `printf`. To be able to use the function the compiler needs to have some information about the function which is contained in a separate file, called a *header file*. You can tell the compiler to look in the header file that is needed for `printf` by putting the following line at the top of your program (before the `main` function).

```
#include <stdio.h>
```

This means "include information from the header file called `stdio.h`". It will allow the compiler to understand the `printf` function we are about to use.

After the line:

```
    time_to_tokyo = time_to_fly + time_to_drive;
```

Type in a new line:

```
    printf("It takes %d hours to get to Tokyo\n", time_to_tokyo);
```

Make sure you get the direction of the backslash (\) correct and that you don't miss the semicolon off the end of the line. Be careful that you don't change the case of the letters that you type from the ones printed here. C **always** treats lowercase letters (like 'a', 'b' and 'c') differently from uppercase letters (like 'A', 'B' and 'C').

Build the executable again. You shouldn't have any errors or warnings. If you do, ask the demonstrators for help.

Use the menu item "Build" → "Execute lab1.exe" to execute the new program. You should see a window containing the following text.

```
    It takes 9 hours to get to Tokyo
    Press any key to continue
```

Syntax: Function Calls

[*variable* =]*name*([*argument* [, *argument*]^n]);

To call a function you must first know its name, and whether or not it needs any information when you call it. Pieces of information that you give to a function when you call it are called *arguments*. Arguments are enclosed in a set of parentheses (brackets) and are separated by commas. For example, the function call

```
    printf("It takes %d hours to get to Tokyo\n", time_to_tokyo);
```

calls the `printf` function and *passes* two arguments. The first argument is

```
    "It takes %d hours to get to Tokyo\n"
```

and the second argument is

```
    time_to_tokyo
```

Some functions give, or *return*, a result after they have been called. We will meet some examples of these types of functions in later labs.

Function Reference: `printf` — Displays text on the screen

`printf(`*`format_string`* [, *`data_item`*]^{n} `);`

e.g.

```
    printf("Hello, World!\n");
    printf("Display the value of an int variable: %d\n", an_int);
    printf("Display the value of a double variable: %lf\n", a_double);
```

The `printf` function displays text and data on the screen.

It takes the following arguments:

- *`format_string`* is the text to be printed on the screen enclosed in double quotes `"..."`
- *`data_item`* these are variables whose values will be printed on the screen

In order to specify where in the text the values of variables are to be printed, special characters are used in *`format_string`*. These are known as *place holders* because they reserve space for these values to be printed. All `printf` place holders begin with a percent character (%). The most common place holders are:

- `%d` for integer (`int`) variables. You can remember this as `d` for **d**ecimal.
- `%lf` real valued (`double`) variables.
- `%c` for single characters (`char`).

The place holders are matched with the comma separated *`data_item`* arguments *in order*. For example:

```
printf("Result 1 = %d, result 2 = %d\n", 6, 7);
```

will display

```
Result 1 = 6, result 2 = 7
```

The `printf` function is defined in the header file `stdio.h`, therefore, to use `printf` you must make sure that the line

```
#include <stdio.h>
```

appears near the top of your source file. This allows the compiler to understand and find the `printf` function.

The `printf` *`format_string`* can also contain special sequences of characters such as `\n` which begins a new line and `\t` which displays a tab character. The backslash (\) functions as a special character causing the next character to be treated specially. This is called an *escape sequence*. If you want to specify a backslash in C you have to type the escape sequence for a backslash, which is two backslashes (\\).

Exercise 1.3: Using the `double` Variable Type

You should have noticed that because the program uses integer variable types it produces an answer of 9 hours, when the correct answer is 9.2 hours. To get the correct answer you should use the `double` variable type.

- Change the variable declarations so that the type of all variables is now `double` rather than `int`.
- Change the format specifier in the `printf` function call to be `%lf` rather than `%d`.

You should now be able to rebuild your program and execute it.

Exercise 1.4: Calculating the Proportion of Time Spent in the Air

Your program should now display the fact that it takes 9.2 hours to get to Tokyo. You are going to add some code of your own to display the percentage of the total time which is spent on the aeroplane. It should also display the percentage of time spent in the car (obviously the sum of the two should be 100).

Your program should display something like:

```
x percent of time is spent in the aeroplane
x percent of time is spent in the car
```

with the *x*s replaced by the results of the calculations you have added.

Some hints:

- you will need some more variables. Choose their names carefully;
- you will need to add some calculations of your own. Remember that every statement in C ends with a semicolon;
- you will need to add two `printf` lines to display the percentage results.

Ask the demonstrators for help if you do not understand how to do this.

Exercise 1.5: The Return Code

If you step through your program in the debugger, or run it in the debugger (but not if you choose the “Execute” command), after your program has finished the output window contains the text

```
The program 'H:\clab\lab1\Debug\lab1.exe' has exited
with code 0 (0x0).
```

If you change the last line of your program so that `return 0;` becomes `return 20;` (for example), what happens to the message in the output window after your program has finished running?

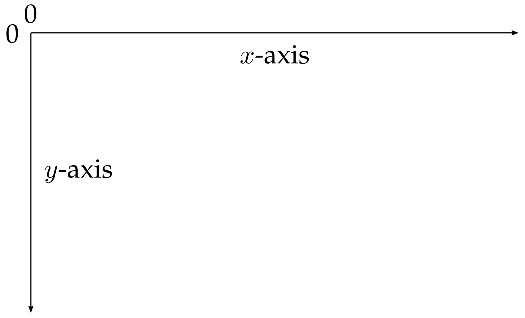
(You can change this number to be whatever you like but it must be an integer).

1.7 Graphics Option

You should only begin this section if you have chosen to do the graphics option (i.e. if you are a non-Music Technology student), or if you have chosen to do the music option but have already completed that section. The section for the music option begins after this one (Section 1.8 on page 32).

As an example of how to use C we are going to investigate graphical output. Although this is not strictly part of the C language, the principles are important for all C programming. Producing graphical output in Windows is quite complicated so this laboratory course uses a set of functions which are explicitly intended for creating graphics easily on Windows. This set of functions is an example of a *software library*. The graphics library functions create a separate window to draw in.

Graphics are displayed in the graphics window by colouring the tiny dots that make up the display. These tiny dots are called *pixels*. Using the graphics functions you can control the colour of any pixel inside the graphics window. Each pixel in the window has a location which can be described using an x- and y-coordinate, much like on a graph. The difference from most graphs is that the origin of the x- and y-axes is in the **top left** of the window.



You will be able to choose the size of the graphics window, up to the size of the screen.

1.7.1 A Simple Graphics Program

If you have a project open at the moment, the first thing you must do is close it. Select “File” → “Close Workspace” to do this. You should then take a copy of the project “graphics1”, and open it in the IDE. Check back to the “Before You Begin” section if you cannot remember how to do this.

The “graphics1” program behaves quite simply at the moment. It displays the graphics window, which has a black background, and draws a red circle on it. When a key on the keyboard is pressed, the window closes. Try building and executing “graphics1”.

The source code for “graphics1.c” is shown below.

```
/*
*  A program to demonstrate simple graphical operations
*  C Programming laboratory 1
*/

/*  This line allows the compiler to understand the
*  graphics functions
*/
#include "graphics_lib.h"

/*
* The main function - the program starts executing here
*/
int main(void)
{
    /* Declare two variables for the x and y positions */
    int x_position, y_position;
    /* Open a graphics window */
    /* Make it 640 pixels wide by 480 pixels high */
    initwindow(640, 480);

    /* Set up some coordinates */
    x_position = 100;
    y_position = 340;

    /* Set the current drawing colour to be red */
    setcolor(RED);

    /* Draw a circle at the coordinates */
    /* Give it a radius of 10 pixels */
    circle(x_position, y_position, 10);

    /* Wait for a key press */
    getch();

    /* Close the graphics window */
    closegraph();

    return 0;
}
```

We will go through this program a few lines at a time. Make sure you understand it before continuing.

At the top of the source file, after the initial comment, is the line

```
#include "graphics_lib.h"
```

This allows the compiler to understand the graphics functions. This works in the same way as the `#include <stdio.h>` line you needed to put into your code to allow the compiler to understand the `printf` function.

The first line of the main function declares two integer variables:

```
    int x_position, y_position;
```

These variables are declared in exactly the same way as in the first example.

The next line calls a graphics function to display the graphics window:

```
    initwindow(640, 480);
```

Calling the `initwindow` function in this way produces a window 640 pixels wide (the x-direction) and 480 pixels high (the y-direction).

Function Reference: `initwindow` — Creates a graphics window of a specified size

```
initwindow(width, height);
```

e.g.

```
    initwindow(640, 480);
```

The `initwindow` function creates the graphics window to allow you to begin drawing in it. The graphics window will be created with a drawing area of the specified dimensions, i.e. it will be *`width`* pixels wide and *`height`* pixels high.

When you have finished with the graphics window you should close it with the `closegraph` function.

`initwindow` is defined in `graphics_lib.h`.

The next two lines set the coordinate variables to the point at which we are going to draw the circle:

```
    x_position = 100;
    y_position = 340;
```

This position will be the centre of the circle.

Before we do any drawing, we set the colour that we will be using to draw with:

```
    setcolor(RED);
```

Calling the `setcolor` function does not produce any output on the screen. It changes the colour that will be used for future graphics functions. It is the equivalent of picking up a pen of a specific colour.

Function Reference: `setcolor` — Sets the current drawing colour

`setcolor(`*`colour_number`*`);`

The `setcolor` function determines the colour that is used for future graphics drawing operations. The colour is set using the *`colour_number`* argument which is an integer from 0 to 15. You can just pass it a number but the graphics system defines some colour names to make it easier to use:

0	`BLACK`	8	`DARKGRAY`
1	`BLUE`	9	`LIGHTBLUE`
2	`GREEN`	10	`LIGHTGREEN`
3	`CYAN`	11	`LIGHTCYAN`
4	`RED`	12	`LIGHTRED`
5	`MAGENTA`	13	`LIGHTMAGENTA`
6	`BROWN`	14	`YELLOW`
7	`LIGHTGRAY`	15	`WHITE`

For example, to set the drawing colour to yellow you could either write

```
    setcolor(14);
```

or

```
    setcolor(YELLOW);
```

`setcolor` is defined in `graphics_lib.h`.

The next line actually draws the circle on the screen:

```
    circle(x_position, y_position, 10);
```

The circle has its centre at (`x_position`, `y_position`) and has a radius of 10 pixels.

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

`circle(`*`xpos`*`, `*`ypos`*`, `*`radius`*`);`

The `circle` function draws a circle in the graphics window in the current colour. 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`*, in pixels.

For example

```
    circle(200, 300, 20);
```

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

`circle` is defined in `graphics_lib.h`.

The next line calls a function which waits for a key press:

```
    getch();
```

This simply has the effect of leaving the graphics window on the screen while the computer waits for a key press. When the user presses a key, the program moves on, and closes the graphics window. Without this line the graphics window would disappear before you had a chance to see it. We will be seeing more of the `getch` function in later labs.

Function Reference: `getch` — Waits for a key press from the keyboard

[*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.

It is easy to use as a function that waits for a key press, when you don't care about which key the user has pressed. For example

```
    getch();
```

If you want to find out what key was pressed then you can assign the result of the function call to a variable

int_variable `= getch();`

We will see more of this in later labs.

`getch` is defined in `conio.h` and also `graphics_lib.h`.

After the user has pressed a key, the program will continue to the next line:

```
    closegraph();
```

which closes the graphics window.

Function Reference: `closegraph` — Closes the graphics window

```
closegraph();
```

The `closegraph` function closes the graphics window. It should be called before your program ends if you opened a graphics window.

`closegraph` is defined in `graphics_lib.h`.

Exercise 1.6: A First Look at Graphics

Before we continue you should make sure that you understand what is happening in this graphics example.

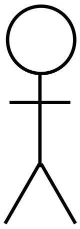
Some things to investigate for yourself:

- try changing the colour that is used to plot the circle;
- try moving the position in which the circle is drawn by changing the coordinates;
- try changing the size of the circle.

After you have made a change remember to rebuild the program and execute it to see the results.

1.7.2 Some Simple Drawing

The next thing you will do is to draw a very simple stick person using a circle and four lines. Hopefully it will look a bit like this:



You have already seen the `circle` function. The only other function you need to know about to draw the stick person is the `line` function.

Function Reference: `line` — Draws a line in the graphics window

`line(`*`start_x`*`, `*`start_y`*`, `*`end_x`*`, `*`end_y`*`);`

e.g.

```
    line(100, 150, 200, 250);
```

The `line` function draws a line in the graphics window using the current colour. The line is defined by its starting point and its ending point. The arguments *`start_x`* and *`start_y`* define the x- and y-coordinates of the starting point respectively. The arguments *`end_x`* and *`end_y`* define the x- and y-coordinates of the ending point respectively.

`line` is defined in `graphics_lib.h`.

Exercise 1.7: Drawing the Stick Person

Change the program to place the circle in the right place for you to draw your stick person. Add four `line` function calls to draw the lines to make up the stick person's body, arms and legs. This may take a lot of experimentation!

It helps if you draw it out on paper first and try and work out what coordinates you will need. Remember that the origin of the x- and y-axes is in the top left of the window.

Plot the lines using the `x_position` and `y_position` variables. For example:

```
line(x_position, y_position + 10, x_position, y_position + 50);
```

This means that changing the `x_position` and `y_position` variables moves the location of the whole stick person, not just the circle.

Ask the demonstrators for help if don't know how to complete this exercise.

1.8 Music Option

You should only begin this section if you have chosen to do the music option (i.e. if you are a Music Technology student), or if you have chosen to do the graphics option but have already completed that section.

As an example of how to use C we are going to investigate how to get the computer to make sounds. Although this is not strictly part of the C language, the principles are important for all C programming.

We will be using the computer's *MIDI* facilities to produce sound. MIDI stands for Musical Instrument Digital Interface and is a way of connecting electronic musical devices together such as keyboards, synthesisers and computers. Elsewhere in your Music Technology course you will learn about it in much greater musical and technical detail. For now, we will just be using it to get the computer to play sounds.

Producing MIDI sound output in Windows is quite complicated so this laboratory course uses a set of functions which are explicitly intended for creating sounds easily on Windows. This set of functions is an example of a *software library*.

1.8.1 A Simple Music Program

If you have a project open at the moment, the first thing you must do is close it. Select "File" → "Close Workspace" to do this. You should then take a copy of the project "music1", and open it in the IDE. Check back to the "Before You Begin" section if you cannot remember how to do this.

The "music1" program behaves quite simply at the moment. It plays a single note, a middle C, for a second and then stops. Try building and executing "music1".

The source code for "music1.c" is shown below.

```
/*
*  A program to demonstrate simple musical operations
*  C Programming laboratory 1
*/

/*  This line allows the compiler to understand the
*  midi functions
*/
#include "midi_lib.h"
```

```
/*
* The main function - the program starts executing here
*/
int main(void)
{
    /* Declare integer variables for specifying a note */
    int pitch, channel, velocity;

    /* 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;

    /* Start playing a middle C at moderate volume */
    midi_note(pitch, channel, velocity);

    /* Wait, for 1 second, so that we can hear the note playing */
    pause(1000);

    /* Turn the note off by setting its volume to 0 */
    midi_note(pitch, channel, 0);

    return 0;
}
```

We will go through this program a few lines at a time. Make sure you understand it before continuing.

At the top of the source file, after the initial comment, is the line

```
#include "midi_lib.h"
```

This allows the compiler to understand the MIDI (music) functions. This works in the same way as the `#include <stdio.h>` line you needed to put into your code to allow the compiler to understand the `printf` function.

The next few lines set the values of integer variables which will control the note that the computer plays:

```
    pitch = 60;
    channel = 1;
    velocity = 64;
```

We will see what these numbers mean in a moment when we look at the `midi_note` function.

The next line starts the computer playing a sound — it turns a note on:

```
    midi_note(pitch, channel, velocity);
```

The `midi_note` function is perhaps the most useful function you will use for creating sounds. It is used to play a note of a certain *pitch* on a certain *channel*. MIDI allows 16 different channels, each of which can be set up to sound like a different instrument. Each channel can have notes playing on it at the same time. You can therefore have up to 16 different instruments (e.g. violin, piano, synth etc.) each on its own MIDI channel. The channels are numbered 1 to 16.

The musical pitch is specified by a number. Each semi-tone (each note on a piano-like keyboard) has its own number. Middle C, for example, has the number 60. The C# immediately above middle C has the number 61, the D above that has the number 62, and so on. There are 12 semitones in an octave, so the octave below middle C has the number 48 (60 - 12 = 48), and the octave above has the number 72. MIDI can produce notes in a range a few octaves wider than a standard piano: the lowest note number is 0, the highest is 127.

The other thing you must specify when turning on a note is the *velocity*. This is equivalent to how hard (or fast — hence the term velocity) the key on a piano (or synthesiser) would have been pressed to produce a note of this volume. Velocity is therefore a specification of the 'loudness' or 'power' in a note. It ranges from 127 (the loudest) down to 0 (slient). In fact, MIDI uses a velocity of 0 to turn a note *off*. You will see this in a moment where we instruct the computer to turn the note off.

Function Reference: `midi_note` — Starts or stops a MIDI note playing

`midi_note(`*`pitch`*`, `*`channel`*`, `*`velocity`*`);`

The `midi_note` function sends a signal to the computer's MIDI device allowing you to turn notes on and off. It has three arguments:

- *pitch* The pitch of the note as an integer value in the range 0 to 127. Middle C has the value 60.
- *channel* The MIDI channel you wish to use to play the note as an integer number. There are 16 MIDI channels numbered 1 to 16, each one can be set up to use a different instrument and notes can be playing on all channels at once.
- *velocity* The velocity (related to volume) of the note to be played as an integer value. This argument can range from 127 (loudest) to 0 (silent). Setting the velocity of a note to zero is used to turn the note off.

So for example, to play middle C on channel 1 at moderate volume you could write:

```
    midi_note(60, 1, 64);
```

which would turn the note on. To turn the note off you would write:

```
    midi_note(60, 1, 0);
```

A note that is not turned off will continue to play forever, possibly even after your program has ended!

`midi_note` is defined in `midi_lib.h`.

If we turned the note off again straight away it would play for such a short time that you would not hear it. To make sure you can hear it we make the computer wait, using the `pause` function:

```
    pause(1000);
```

You tell the `pause` function how long to wait for (in milliseconds) and it does not allow your program to continue until that time has elapsed. The `pause` therefore controls how long the note is played for: its *duration*. In this case the note will play for one second (1000ms = 1 second).

Function Reference: `pause` — Waits for a specified amount of time

`pause(`*`duration`*`);`

The `pause` function causes the computer to wait before continuing. The amount of time the computer waits for is determined by the *`duration`* argument, which is a integer number, and specifies the amount of time the computer should wait for in one thousandths of a second (milliseconds).

For example:

```
pause(500);
```

will cause the computer to wait for half a second (500 milliseconds) before continuing.

`pause` is defined in `midi_lib.h`.

After the program has waited for 1 second, the note is turned off:

```
midi_note(pitch, channel, 0);
```

Notice that the correct pitch and channel must be specified so that the computer knows which note to turn off.

Exercise 1.8: Experimenting with MIDI

Experiment with changing the values relating to the note that is played in the “music1” example. Try altering:

- the pitch;
- the velocity;
- the duration (the `pause` function argument).

After you have made a change remember to rebuild the program and execute it to see the results.

When you are happy with creating different notes and durations try to get the computer to play a set of notes, one after the other. Build up a set of notes to form a tune. You can use the “Copy” and “Paste” commands on the “Edit” menu in the IDE to copy sections of code multiple times. This should save you having to type the same sections again and again!

If you don’t know how to do this, ask the demonstrators for help.

1.8.2 Playing More than One Note at Once

If you turn two notes on, one after the other, they will both play at once. For example:

```
midi_note(pitch, channel, velocity);
midi_note(pitch + 3, channel, velocity);
```

and after a pause, turn the same notes off:

```
midi_note(pitch, channel, 0);
midi_note(pitch + 3, channel, 0);
```

By choosing the correct notes you can create a chord. For example, you could turn on a major triad with the following code.

```
/* Turn on the tonic */
midi_note(pitch, channel, velocity);

/* Turn on the major third */
midi_note(pitch + 4, channel, velocity);

/* Turn on the perfect fifth */
midi_note(pitch + 7, channel, velocity);
```

You would need to turn all the notes off again afterwards.

Exercise 1.9: Creating Chords

Change your tune so that it ends on an appropriate major triad. Experiment with ending on other chords, can you produce:

- a minor chord?
- a dominant 7th?
- a diminished chord?
- a minor 7th added 9th chord?

Some of these may not be the most appropriate for your tune, but try them anyway!

1.8.3 Changing the Instrument

You can change the MIDI instrument that is being used on a given channel using the `program_change` function. For example:

```
program_change(1, 57);
```

sets MIDI channel 1 to use instrument number 57, which is usually a trumpet. The exact correspondence between instrument numbers and instrument sounds (or *voices*) is dependent on the sound card being used.

Function Reference: `program_change` — Changes the current instrument on a MIDI channel

```
program_change(channel, voice);
```

e.g.

```
    program_change(1, 51);
```

The `program_change` function changes the active instrument on the channel specified by *`channel`*. *`channel`* should be an integer number from 1 to 16 (the same as used in the `midi_note` function). The instrument is selected using the *`voice`* argument. This should be an integer number between 1 and 128. Each number corresponds to a different instrument. The instrument that a given number corresponds to is defined by the General MIDI specification, which most MIDI devices adhere to. Some common instruments and their voice numbers are:

1	Acoustic Grand Piano	43	Cello
7	Harpsichord	49	String Ensemble
10	Glockenspiel	51	Synth Strings
13	Marimba	57	Trumpet
17	Drawbar Organ	58	Trombone
27	Electric Jazz Guitar	66	Alto Sax
28	Electric Clean Guitar	67	Tenor Sax
33	Acoustic Bass	74	Flute
41	Violin	119	Synth Drum

Check MIDI documentation (for example on the internet) for more information.

`program_change` is defined in `midi_lib.h`.

Exercise 1.10: Changing the Program

Experiment with using `program_change` to change the voice that is being used to play your tune.

Exercise 1.11: Creating both Melody and Chords

Use the `program_change` function to set up two different voices on two different MIDI channels. Add some simple chords to your tune (just a few will do) which should play *at the same time* as the melody, but on a different channel.

If you don't know how to do this, ask the demonstrators for help.

1.9 Summary

Now you have completed this lab you should have some understanding of the basic structure of a C program. You should also:

- understand how to declare variables and choose their type;

- be able to carry out some simple mathematics in C;
- understand what a function call is and be able to call some simple functions such as `printf`.

If you chose the graphics option then you should understand the basic structure of a graphics program including how to display and close the graphics window. You should also be able to draw circles and lines in many colours.

If you chose the music option you should be able to instruct the computer to play notes of any valid pitch and on any instrument. You should have learned that, with MIDI, notes need turning on and off, and that the time between the two is the note's duration. You should also have learned how to play chords.

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. Copy the project "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 it in laboratory 12, next term. 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 the one of the demonstrators 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, ask one of the demonstrators.

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 `case`s, 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. Copy the project, 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 open the “graphics1” project you were working on.

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 should 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 the graphics function `floodfill` to fill this circle with colour. The `floodfill` function 'pours' colour onto the screen at a specified location. The colour 'spreads out' until it reaches a line of a colour that you specify. If it doesn't meet a line with the colour you specify it will fill the whole screen. You can control the pattern and the colour that is used for the fill with the `setfillstyle` function. This is very like the `setcolor` function, it sets the style and colour to be used by all `floodfill` function calls.

Function Reference: `setfillstyle` — Sets the graphical style used for fill operations

```
setfillstyle(pattern_number, colour_number);
```

e.g.

```
    setfillstyle(SOLID_FILL, RED);
```

The `setfillstyle` function determines the pattern and colour that is used for all future `floodfill` operations. The pattern is set using the *`pattern_number`* argument. The colour is set using the *`colour_number`*. The *`pattern_number`* argument should be an integer number from 0 to 12, while *`colour_number`* should be an integer from 0 to 15. You can just pass it a number for either but the graphics systems defines some pattern names to make it easier to use:

0	`EMPTY_FILL`	7	`HATCH_FILL`
1	`SOLID_FILL`	8	`XHATCH_FILL`
2	`LINE_FILL`	9	`INTERLEAVE_FILL`
3	`LTSLASH_FILL`	10	`WIDE_DOT_FILL`
4	`SLASH_FILL`	11	`CLOSE_DOT_FILL`
5	`BKSLASH_FILL`	12	`USER_FILL`
6	`LTBKSLASH_FILL`		

For *`colour_number`* you can use any of the names defined for the `setcolor` function (see function box on page 29).

The most common pattern (and perhaps the most useful) is `SOLID_FILL`.

`setfillstyle` is defined in `graphics_lib.h`.

Function Reference: `floodfill` — Fills an area with a coloured pattern

```
floodfill(x, y, boundary_colour);
```

The `floodfill` function begins filling the screen with colour, working outwards in all directions from the location specified by the coordinates (*x*, *y*). The fill spreads in all directions until it reaches a line of colour *boundary_colour*, when it stops. This colour serves as a perimeter for the fill.

The fill takes place in whatever the current colour is, and whatever the current style is. These are set by a call to the `setfillstyle` function.

```
setfillstyle(SOLID_FILL, RED);
floodfill(100, 100, WHITE);
```

will fill the screen with red, start from location (100, 100) and stopping wherever a white line is encountered.

`floodfill` is defined in `graphics_lib.h`.

Exercise 2.10: Using `floodfill`

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. You might like to try out different fill styles.

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 open the "music1" project you were working on.

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
 */
key = 60;

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

pause(1000);

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

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.

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. Copy the “lab3” project, then 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 one of the demonstrators.

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”. If you open the project you should see that it 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 one of the demonstrators for help.

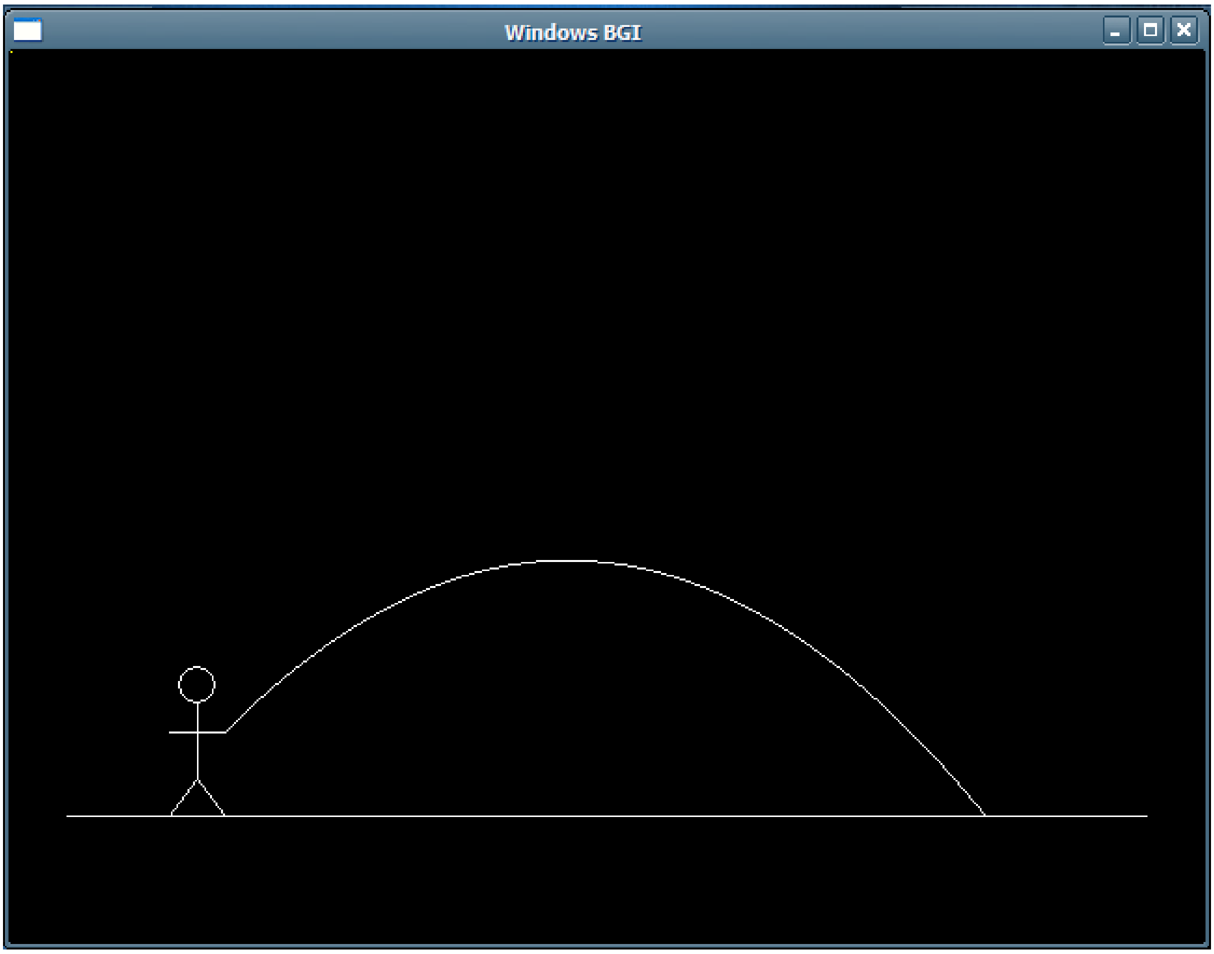
3.4 Deciding What Kind of Loop to Use

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" project 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);
```

draws a line in the graphics window from wherever the ‘current location’ is to the coordinates specified, in this case the coordinates (200, 350). 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`*`)`

e.g.

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

The graphics system keeps track of a 'current position', which can be set by the `moveto` function. The `lineto` function draws a line 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 in the "graphics2" project. 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 draw over the top of the stick person with another stick person made up of black lines. This will erase the stick person from the screen. You can then redraw the stick person in a new position using coloured lines.

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 project “music2” plays one octave of a chromatic scale starting on middle-C. Copy the project and open it.

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;

    /* 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);
    }

    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.

Laboratory 4

The C Preprocessor and Useful Debugging Techniques

4.1 Overview

This laboratory covers a number of different topics that are useful to know about when constructing, and particularly, troubleshooting (debugging) your C programs.

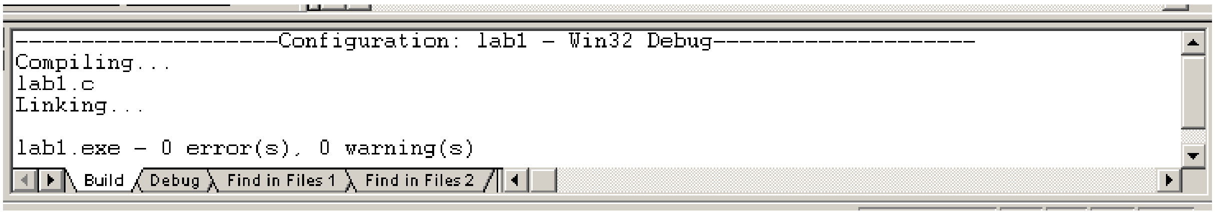
We will look at:

- a part of the compiler called the *preprocessor*, which deals with all of the lines in your code that begin with a hash (or sharp) symbol (#);
- how to use preprocessor commands to stop parts of your program from compiling;
- how to use conditional compilation for debugging;
- setting *breakpoints* and using them for debugging your program;
- finding the executable file that corresponds to your program and running it without the IDE.

Some of the tasks you do will be different depending on whether you are following the graphics or music option, but there are no special sections for the different options in this laboratory.

4.2 Stages of Compilation

When you build a project, the output window shows messages generated by the compiler. If everything goes well, the output window looks like this:



```
--------------------Configuration: lab1 - Win32 Debug--------------------
Compiling...
lab1.c
Linking...

lab1.exe - 0 error(s), 0 warning(s)
```

From this window you can see the two major stages of compiling your programs (this is a reminder from the first lab):

- **compiling**, which produces an intermediate file (called an *object file*) for every source file in your project;

- **linking**, which combines all of the object files with *library files* which contain the code for pre-existing functions such as `printf` to form an executable file: your program.

We are going to look at the compiling stage in a little bit more depth.

What we have been called the compiling stage can actually be split into two major parts:

- the **preprocessor** which processes the text in your source file, before it is passed on to...
- the **compiler** which does the work of translating your text written in C syntax into machine code object files.

The preprocessor takes, as input, a text file, which contains the C program that you have written. It produces, as output, another version of that text file, still in text, but expanded and altered slightly. For example, an important job that the preprocessor usually does is to remove all of your comments from the file so that the compiler never sees them.

The next few sections of this laboratory are going to focus on the preprocessor, to understand what it does and to be able to use it to our advantage.

4.3 Introducing the C Preprocessor

Most of the time the C preprocessor silently goes about its work every time you build your program, for example stripping out comments as mentioned above. Sometimes you might want to communicate with the preprocessor directly. This is done by including preprocessor commands, or *directives*, in your program. All preprocessor directives begin with the hash (or sharp) symbol, #. Preprocessor directives are **not** statements, so they are not followed by a semicolon.

You have already used one preprocessor directive many times: `#include`. For example:

```
#include <stdio.h>
```

You might have noticed that there are two kinds of `#include` directive:

```
#include <filename>
```

and

```
#include "filename"
```

We will see what the difference is between these forms in a moment. In common with most preprocessor directives, the `#include` directive does not do anything very complicated. When the preprocessor sees a `#include` directive it looks for the file that have specified, for example `stdio.h`. When it finds the file it simply takes all of the text out of the file an inserts it into your source file in place of the `#include` directive. It does not try and process anything in the file, or try to understand it in any way. That is the compiler's job. The difference between enclosing the filename in angle brackets (like this `<stdio.h>`) and quotation marks (like this `"stdio.h"`) is in where the preprocessor looks to find the file you have specified.

- If the filename is enclosed in quotation marks then the preprocessor looks for the file in the current folder (where your source file is stored) before looking in any places that you have specified.
- If the filename is enclosed in angle brackets then the compiler assumes it is a standard library header file. Standard library header files describe functions that are part of the the C *standard library*: functions that are specified as part of the C language. This includes functions like `printf` and `scanf` in `stdio.h`, and `getch` in `conio.h`. It does not include any of the graphics or music functions.

So the preprocessor handles a #include by finding the file (looking where the angle brackets or quotation marks tell it to look) and substituting the *entire* contents of the file for the #include line.

Syntax: The #include Directive

#include <*filename*> or #include "*filename*"

The #include preprocessor directive is completed as part of the preprocessor, before the compiler is run. The directive identifies a file (*filename*), which is usually a header file, but can be anything. The directive takes the text from the header file and inserts it in place of the #include directive.

The angle brackets and quotation marks determine where the preprocessor looks for the file. As a general rule, angle brackets are for header files that are part of the C standard library (such as stdio.h) and quotation marks are for header files that are not (such as the graphics and music libraries). With most modern compilers (including the one you are using), using quotation marks instead of angle brackets (or vice-versa) will not stop your program from working.

4.4 Using the Preprocessor for Text Substitution

A lot of Windows programs, like Microsoft Word, have the facility to search the text of a document to find a particular bit of text that you specify. This is called "Find". You may also have come across the facility (also in Word) to find bits of text and automatically replace them with another bit of text that you specify. This is usually called "Replace" and is an example of *text substitution*.

The C preprocessor has a facility for doing text substitution for you. It allows you to define keywords of your own which will get substituted for (usually more complicated) bits of text, which you specify. This is done using the #define preprocessor directive.

Copy and open the "lab4" project. This is a very simple example of how to use the #define directive for text substitution. The source code for "lab4.c" is shown below.

```
/*
 *  A program to demonstrate the use of the #define directive
 *  C Programming laboratory 4
 */

/*  These lines allow the compiler to understand the
 *  printf and getch functions
 */
#include <stdio.h>
#include <conio.h>

/* An example use of the #define directive */
/* The preprocessor will replace HELLO_STRING, */
/* wherever it appears, with "Hello, World!\n" */
#define HELLO_MESSAGE "Hello, World!\n"

/* A numeric example, wherever PI appears */
/* it will be replaced by 3.14159265 */
#define PI 3.14159265

int main(void)
{
    /* Print a message to the screen */
    printf(HELLO_MESSAGE);
    /* Display pi */
    printf("Pi = %lf (roughly)\n", PI);

    /* Wait for a key press */
    getch();

    return 0;
}
```

If you build and execute the program you should find that it displays the following text on the screen:

```
Hello, World!
Pi = 3.141593 (roughly)
```

Remember that the preprocessor runs through the source code *before* it gets to the compiler. So the code that the compiler sees looks a bit like this:

```
...
int main(void)
{
    printf("Hello, World!\n");

    printf("Pi = %lf (roughly)\n", 3.14159265);

    getch();

    return 0;
}
```

Notice how the code was changed by the preprocessor.

Like the `#include` directive, a `#define` is also very simple. It is structured like this:

```
#define tag replacement
```

The `#define` directive tells the preprocessor: "from now on, everywhere you see the text *`tag`*, replace it with the text *`replacement`*."

Syntax: The `#define` Directive

```
#define tag replacement
```

Starting from where the `#define` directive is written the preprocessor replaces an occurrence of the text *`tag`* with the text *`replacement`*.

For example, the directive:

```
#define E_CONST 2.71828
```

replaces any occurrence of the text `E_CONST` with the text `2.71828`.

It is generally considered good programming practice to use UPPERCASE names for `#define` tags to visually distinguish them from variable and function names.

`#define` directives are used to define replacement tags for two main reasons:

- To make your program more readable. For example:

```
circle_area = PI * radius * radius;
```

is a lot easier (and quicker) to read and understand than:

```
circle_area = 3.141592653 * radius * radius;
```

- To make your program easier to change. For example, it is common to use a `#define` directive for your program's version number. This allows you to put a `#define` directive clearly at the beginning of the source code, like this:

  ```
  #define PROGRAM_VERSION 2.3
  ```

  Somewhere in your source code, perhaps when the program starts, you might write:

  ```
  printf("Welcome to myprog Version %lf\n", PROGRAM_VERSION);
  ```

  When you make a change to your program, you don't have to go looking for all the places where you display the version number, you simply change the number in the `#define` directive at the top of the source file.

Exercise 4.1: Using the `#define` Directive

Add to the "lab4" example program to:

- prompt the user for a number, the radius of a circle;
- store the number in variable of type `double`;
- calculate the area of a circle with the radius they entered using `PI`;
- display the result.

Exercise 4.2: Problems with `#define` Directives

You are now going to change the `#define PI` directive so that it is **wrong**. Change the line to be something like this:

```
#define PI error 3.14159265 error
```

This means that the text that will be substituted for the tag `PI` will be `error 3.14159265 error`, which is obviously not going to be valid C! Try compiling your program.

The compiler will give you an error message. Which line does the compiler think the error is on? Why is this?

If you don't understand this, ask one of the demonstrators to explain it to you. Remember to put the `#define` directive back to how was before you continue.

Choose one of the following two exercises depending on which option you have been doing.

Exercise 4.3: Adding a `#define` Directive: Graphics Option

Open the graphics program you have been working on, which should be called “graphics1”. The horizontal location of the stick person is controlled by the user. The vertical location is fixed. Use `#define` directives to define tags for:

- the width and height of the graphics window. You could call these something like `WINDOW_HEIGHT` and `WINDOW_WIDTH`.
- the vertical location of the stick person. You could call this something like `PERSON_POS_Y`.
- the key number for the ‘Enter’ key (13). You could call this something like `ENTER_KEY`.

Have a look to see what other parts of your program you think would be better substituted for a `#define` tag. Good programming practice says that you should consider using `#define` tags whenever you have a fixed value in your program that you might want to change at compilation time (like the window size).

Exercise 4.4: Adding a `#define` Directive: Music Option

Open the music program you have been working on, which should be called “music2”. Add a `program_change` function call for each channel you are using. The instrument number used should be defined as a `#define` tag. For example, the tag `PIANO` might be defined to be replaced by the number 1, like this

```
#define PIANO 1
```

Once you have done this add one or more `#define` directives like the following:

```
#define FIRST_CHANNEL_INSTRUMENT PIANO
```

You would then change the `program_change` function call to be:

```
program_change(1, FIRST_CHANNEL_INSTRUMENT);
```

Notice how you are using one `#define` tag to define another. This technique can be very useful in making your program easier to read, understand and change.

Have a look to see what other parts of your program you think would be better substituted for a `#define` tag. Good programming practice says that you should consider using `#define` tags whenever you have a fixed value in your program that you might want to change at compilation time (like instruments, maybe even the musical key).

4.5 Using `#define` Directives for Conditional Compilation

In laboratory 2 we saw how to get parts of your program to execute only under certain conditions using an `if` statement. There is a similar preprocessor directive that allows you to conditionally *compile* parts of your program under certain conditions. Logically, the directive is called `#if`.

Close whatever project you have open. Copy and open the “lab4a” project. You should see that the code is very similar to the example we looked at in the first laboratory. It has some additional lines at the end to display the results, it also has this bit of code:

```
#if defined(DEBUG)

    printf("time_to_fly = %d, time_to_drive = %d\n", time_to_fly, time_to_drive);

#endif
```

What do you think this will do? Try building and executing the program to find out.

Exercise 4.5: Investigating Conditional Compilation (1)

Make the compiler ignore the line:

```
#define DEBUG
```

by enclosing it in a comment (enclose it between `/*` and `*/`). Build and execute the program. What has happened?

You should have seen that the code inside `#if...#endif` directives is only compiled under certain conditions.

The `#if` directive allows you to set conditions under which parts of your program will be removed by the preprocessor and therefore ignored by the compiler. For example:

```
#if 0
    ...
#endif
```

will **never** compile the code between the `#if` and `#endif` directives because 0 is treated as *false*. On the other hand, in this example:

```
#if 1
    ...
#endif
```

the code between the `#if` and `#endif` directives will **always** be compiled because a non-zero value is treated as *true*. It is not common to use `#if` directives with a constant value after them like this. It is more common to use them like this:

```
    #define ADD_EXTRA_FUNCTIONALITY 0
        ...
    #if ADD_EXTRA_FUNCTIONALITY
        ...
    #endif
```

This means that the compilation of the section of code between the `#if` and `#endif` directives is dependent on the value that the `ADD_EXTRA_FUNCTIONALITY` tag has defined as its replacement. In the case above, the code between the `#if` and `#endif` directives would **not** be compiled.

Rather than test the value of a `#define`d tag, it is more common to test whether a tag has been `#define`d *at all*. You have already seen how to do this in the example program.

Syntax: The #if and #endif Directives

`#if` *`condition`* `...#endif`

The #if directive tells the preprocessor to remove all the parts of your program until the next #endif directive, only if the *condition* is *false*. When testing the condition, a value of 0 is interpreted as *false*; a non-zero value is interpreted to be *true*.

The most common way to use the #if directive is with the special preprocessor syntax defined(*tag*). This tests to see if the specified tag has been previously specified as part of a #define directive. It is used in the following way:

```
#if defined(SOME_TAG)
    ...
#endif
```

You can form more complicated tests using the logical operators &&, || and !. The *not* operator (!) is perhaps the most useful. For example:

```
#if !defined(SOME_TAG)
    ...
#endif
```

In this case the guarded code would only be compiled if the tag SOME_TAG is **not** #defined.

There are two very useful abbreviations for these simple operations:

#ifdef SOME_TAG	is short for	#if defined(SOME_TAG)
#ifndef SOME_TAG	is short for	#if !defined(SOME_TAG)

Exercise 4.6: Investigating Conditional Compilation (2)

Change the #if directive so that the code is compiled only if DEBUG is **not** defined. You will need to use the information in the syntax box about the #if and #endif directives to do this.

#if directives are very useful for including code (such as printf statements) which you only want to use for finding problems in your code (debugging). It means you can add lots of extra statements to your program and stop them from being seen by the compiler by removing a single #define line from the top. Even better, if you find similar problems again, you can add the lines back into your program again by putting just one #define line back.

Using #if directives for including special lines for debugging is a very useful technique in situations where stepping through your code is tedious. For example, inside a loop you could add an if statement that displays the value after the tenth iteration. The code could be removed from the final program by ensuring that you do not define a DEBUG tag.

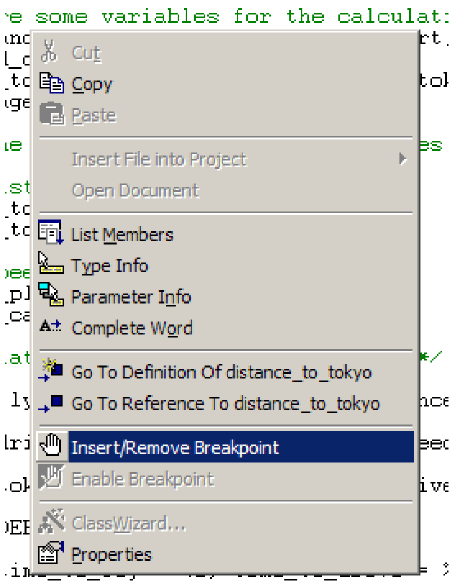
4.6 Using Breakpoints for Debugging

Right from the beginning of this course we have used the IDE's debugging facility for stepping through your programs. If you are confident that all of the code at the beginning of you program works then stepping through it to get to the bit you are worried about can be very boring. If you have loops with a large number of iterations then it could take all day!

Fortunately, the debugger has a facility for you to be able to specify a line in your source code that you want to be able to stop on. To do this you mark a line in your code with a *breakpoint*.

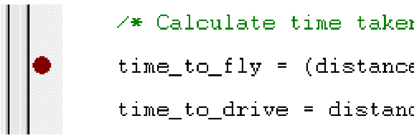
Open a project you have been working on in previous labs. Choose something with a fairly substantial amount of code in it; a project from your option (either graphics or music) would be a good idea.

Pick a line of source code that will execute (i.e. not a comment) and click on it with the right-hand mouse button. You should see as menu like this one:



Select the option "Insert/Remove Breakpoint".

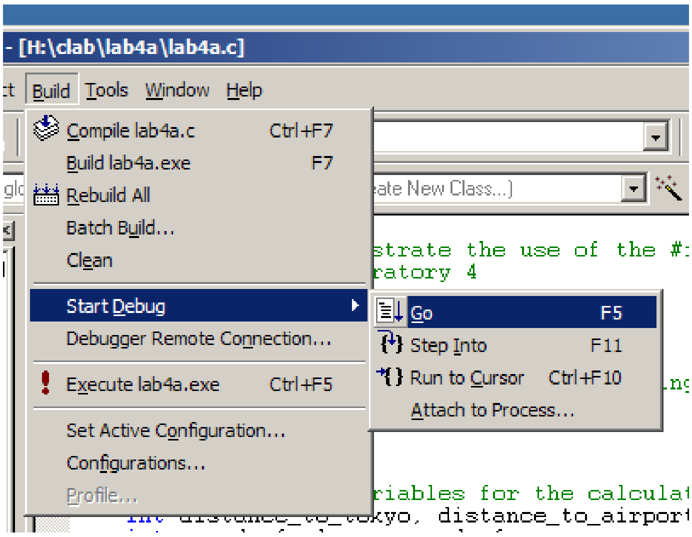
You should find that the line in the source code that you clicked on now has a small red dot at the left-hand side of the line, like this:



```
/* Calculate time taken
time_to_fly = (distance
time_to_drive = distanc
```

This indicates that the line in the source code has a breakpoint on it.

Start the program running in the debugger by selecting "Build" → "Start Debug" → "Go":



You should see that the program starts running, until it reaches the breakpoint. It then stops the program, but you have the chance to continue the program from where it is using the step commands or to continue running using the "Go" option on the "Debug" menu. When the debugger stops on a breakpoint, it is exactly as if you had used the step commands to get there, just a lot quicker!

You can remove a breakpoint by clicking on the line of source code with the breakpoint on it using the right-hand mouse button. Then choose the "Remove Breakpoint" option from the menu.

Exercise 4.7: Trying Out the Breakpoint Feature

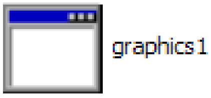
Now is your opportunity to try out breakpoints. Try setting them, removing them and using them for debugging. Breakpoints are very useful and you will probably find them invaluable when debugging your programs for your assessments.

4.7 Locating the Programs You Have Written

All the programs you have produced so far do not have to be run in the IDE. In fact, they do not need the C source code or the C compiler to run once you have built them. If you sent your program to a friend you would only have to send them one file: the executable. Just to prove this point you are now going to try executing some of your programs without using the IDE.

First close the IDE; we won't be using it now. Now we have to find the executable file that was built by the compiler and linker from your C source code.

Open up "My Documents" from the desktop, and double click on the "clab" folder that you created in the first lab. You should now see one folder for each project you have worked on. Double click on the folder for the last option project you were working on (either "graphics1" or "music2" depending on which option you are doing). You should see some files and a folder called "Debug". Double click on the "Debug" folder to view its contents. You will see one file which has the same name as the project. It will have an icon a bit like this:



(This example is for the "graphics1" project). This is the executable that was produced when you built the "graphics1" or "music2" project. If you double click on it, the program will run. Try it.

The other files in this folder are intermediate files, including the object files which were produced after the compilation stage and used by the linker to produce your executable.

Exercise 4.8: Running Your Programs from Executables

Prove that your some of your other programs can be executed this way by finding their executable files and double clicking on them.

It is important to note that if you were to give your program to someone else to try out, **this is the only file they would need**. This executable file is exactly the same type of file that executes when you run any program on your computer. You could copy this file onto any medium, like a floppy disk or a CD, or even make it available over the internet. Anyone with access to just this file could run your program.

4.8 Summary

Now that you have finished this laboratory you should understand the basic functions of the C preprocessor including the `#include`, `#define`, `#if` and `#endif` directives.

You should be able to use `#define` directives, both on their own, and in combination with `#if` and `#endif` directives for debugging.

You should have seen what a breakpoint does, and understand how to use them to debug your programs.

Finally, you should have seen where to find the executable file that corresponds to each of the programs you have written.

Laboratory 5

Functions

5.1 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.

5.2 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.

Copy the “lab5” project and open it in the IDE. 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 to “lab5.c” is shown below.

```
/*
 *  A program to demonstrate function creation
 *  C Programming laboratory 5
 */
```

```
#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.

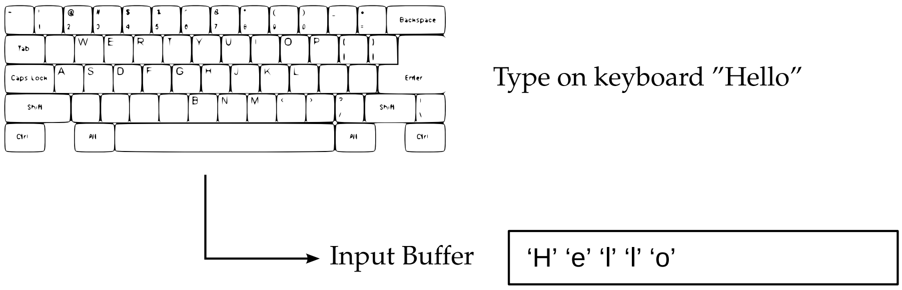
When the user types in a response the `scanf` checks what they typed against the placeholder, it could either be

- 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`.

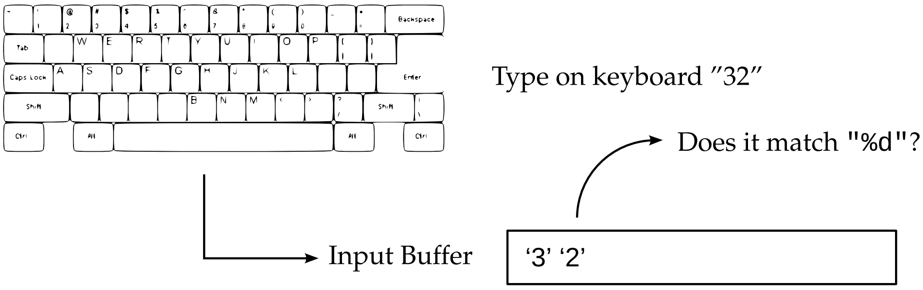
But what happens if the user doesn't type in a valid integer? To answer this question we must look at the way `scanf` works.

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*.

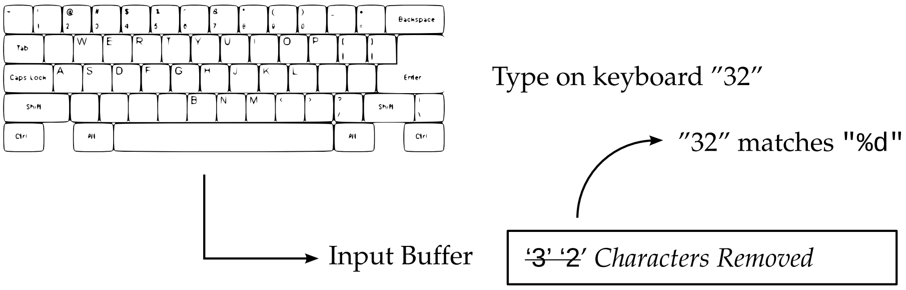


When you call `scanf` the first time the input buffer is empty, because you haven't typed anything yet. Because the buffer is empty, `scanf` waits for you to type in characters and press 'Enter'.

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`.

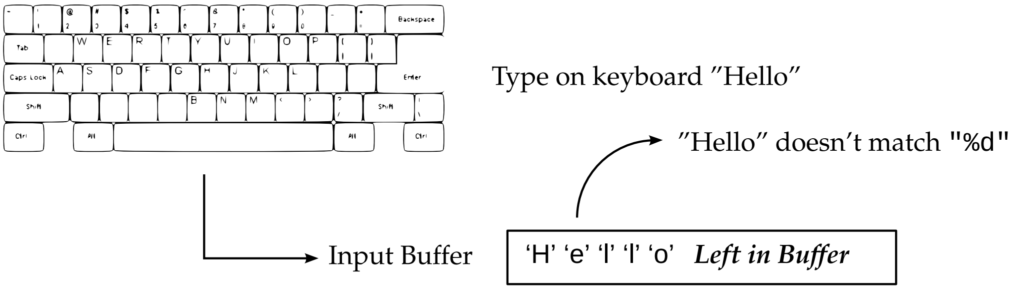


If the characters in the input buffer match the type that is required by the placeholder then it takes them out of the input buffer.



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 5.1: Investigating Functions

Step through the "lab5" program in the debugger. Make sure you use the **step into** (the F11 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 5.2: `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 "lab5" 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.

5.3 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 "lab5": `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)
```

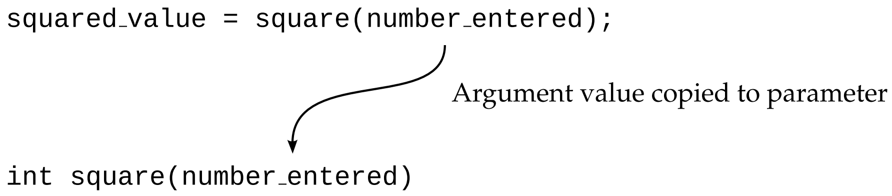
This function head declares the `square` function to return a value of type `int` and to require it to be called with one argument of type `int`. When the argument reaches the function it is given a name: `value`. It is then called a *parameter*. Parameters are very like variables, they have a type and 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.

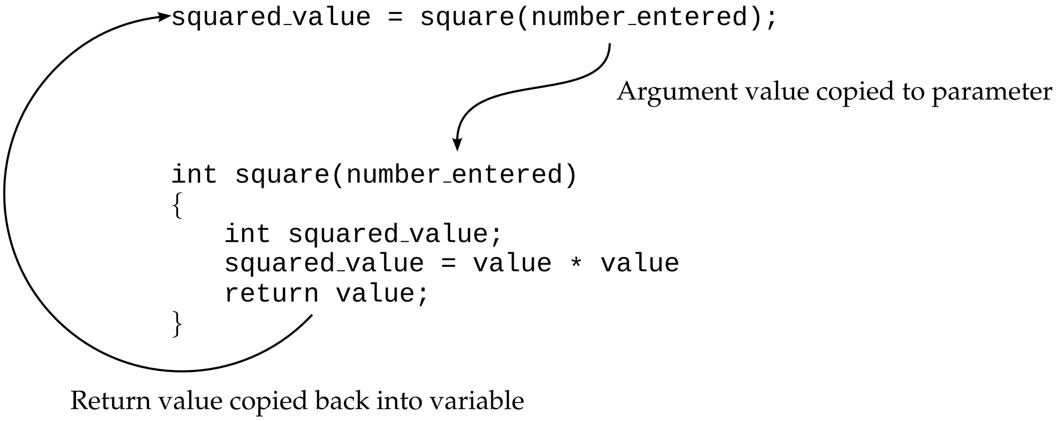


```
squared_value = square(number_entered);

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 ]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 5.3: Investigating Function Types

Alter the "lab5" 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 5.4: 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.

5.4 Functions: Why Bother?

Why did you move the input handling code into a separate function in exercise 5.4? 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.

5.5 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 5.5: 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 5.6: 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 one of the demonstrators 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();
```

5.6 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 B, 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 5.7: Calculating the Length of a Triangle's Hypotenuse

Alter the "lab5" 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.

5.7 Graphics Option

5.7.1 Creating More Manageable Code

Open the "graphics1" project 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 5.8: Splitting Your Program into Functions

Define suitable functions for:

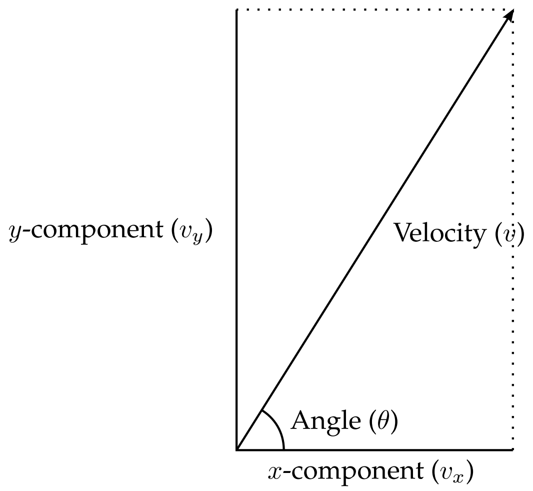
- drawing the stick person at any location on the screen. This function should have two parameters which specify the x- and y-coordinate at which the stick person will be drawn. It should also take a parameter which specifies the colour in which the stick person should be drawn.
- drawing the projectile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

5.7.2 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:

$$v_x = v\cos(\theta) \tag{5.1}$$

$$v_y = v\sin(\theta) \tag{5.2}$$

Exercise 5.9: 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.

5.7.3 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 29.

`setbkcolor` 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:

```
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 5.10: Using the New Graphics Functions

Now is your opportunity to use the new graphics functions (`cleardevice`, `setbkcolor` and `outtextxy`) 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.

5.8 Music Option

Open your "music2" project 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 5.11: 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 `midi_lib.h`.

Exercise 5.12: 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 5.13: 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*).

5.9 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!

Laboratory 6

Arrays and Strings

6.1 Overview

This laboratory introduces you to *arrays* which are a way to create a variable that has the capacity to store many values. Each of the values in the array can be accessed by using an index.

You will learn how to create and use arrays, and how to make sure that they are initialised with specific values when your program starts. You will also learn about *multidimensional arrays* which can be used to store data in a similar way to tables.

You have already been using text enclosed in quotation marks (`"..."`). In this laboratory you will be learn more about these *text* or *character strings*, which are actually a special form of array. You will learn how to create and manipulate them.

In the graphics option you will make use of arrays to make your graphics program more like a computer game. In the music option you will find that arrays are very useful for defining musical relationships, like the pitch relationships between notes in a scale.

6.2 Introducing Arrays

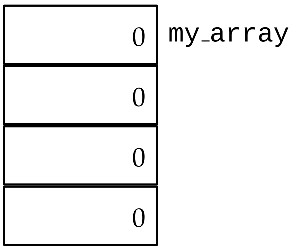
The variables you have been working with so far are a bit like scalar values in maths. This laboratory is about *arrays* which are more like mathematical vectors.

Each variable you declare defines a space in the computer's memory in which you can store one value of the correct type. We often show this as a diagram like this:

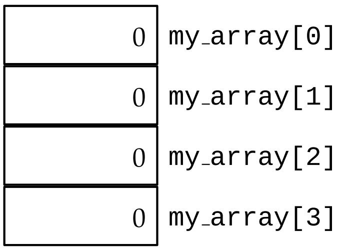
`int my_integer = 20;` $\equiv$ $\boxed{\quad 20}$ `my_integer`

The variable declaration on the left is equivalent to the diagram on the right. In the diagram, the box represents the space in the computer's memory in which the value of the variable is stored. In this case you can see that the box contains the value 20 to indicate that the variable is assigned the value 20. The text to the right of the box (`my_integer`) identifies the name of the variable: it provides a name for that space in the computer's memory.

Every variable you declare has a space in memory, and the name of that variable identifies the memory space. An array allows you to reserve *more than one* memory space for a single variable. All the memory spaces you reserve will have the same type (for example `int`). When you use the variable you have to specify which one of the spaces you wish to access. Diagrammatically, an array looks like this:



This shows that one variable name identifies multiple spaces in the computer's memory. These spaces are more properly called *elements*. To identify which element we are talking about we reference it by number. In C **elements are numbered from zero**. This 'element number' is called a *subscript* or an *index*. A subscript is placed after the variable name, in square brackets, like this: `my_array[3]`. We can show the way that subscripts work on a diagram:



Obviously the subscript must be an integer; it wouldn't make sense to ask for element number 2.3859.

You declare an array by specifying how many elements you want in that array. For example:

```
    int my_array[4];
```

declares a new array variable called `my_array`. Each element in the array contains an integer value. The array has four elements in it. Note that the last element is called `my_array[3]`. Try to remember that when you are declaring an array, the number in the square brackets is the number of elements, **not** the subscript of the last element. The subscript of the last element will be the size of the array minus one.

Syntax: Array Declarations

`type name``[`*`size`*`]` [, more variable delcarations]^n ;

e.g.

```
    int an_integer_array[10];
    double a_real_array[6];
    int an_int_variable, an_int_array[22];
```

Arrays are variables which have space for more than one value. The number of values that the array has space for is specified as a number in square brackets (*`size`*).

When the array variable is used, an integer must be included specifying the index of the element which is being referenced. This index is called a *subscript*. Subscripts are numbered **starting from zero**. For example:

```
    an_integer_array[6]
```

is the integer value which occupies the 7th element in the the array called `an_integer_array`.

6.3 Working with Arrays

You are now going to try using arrays in a program. The “lab6” project uses arrays to collect information. The program asks the user for the heights of up to ten people and then displays the mean height, calculated from the data that were entered. The heights that the user enters are stored in an array. It would be very difficult to produce a program like this without using arrays.

Copy the “lab6” project and open it in the IDE. Try building and executing the program. Test it out a few times to make sure you are familiar with what it does. The source code for “lab6” is shown below.

```
/*
 *  A program to demonstrate arrays
 *  C Programming laboratory 6
 */

#include <stdio.h>

/* Function prototypes */
void welcome_message(void);
double mean(double values[], int num_values);

/*
 * Program starts here
 */
int main(void)
{
    /* Declare an array which can store up to 10 doubles */
    double data_values[10];

    /* Keep track of the number of items we have here */
    int num_data_values;

    /* Some other useful variables */
    int curr_data_value, done;
    double mean_height;

    /* Set initial conditions */
    num_data_values = 0;
    done = 0;

    /* Display message to user */
    welcome_message();

    /* Loop to get all data values, stop when we have 10 values */
    /* Or when the user enters a zero height */
    for (curr_data_value = 0;
            (curr_data_value < 10) && (done == 0);
            curr_data_value++)
    {
        /* Get the data into the array */
        printf("Please enter the height for person %d (or 0 if done): ",
            curr_data_value + 1);
        scanf("%lf", &data_values[curr_data_value]);

        /* If the data was non-zero we have another height */
        /* If not, we are done */
        if (data_values[curr_data_value] != 0)
            num_data_values++;
        else
```

```
            done = 1;
    }

    /* Calculate the mean height */
    mean_height = mean(data_values, num_data_values);

    /* Display it */
    printf("\nThe mean height is %lf\n", mean_height);

    return 0;
}

/*
 * Displays welcome message
 */
void welcome_message(void)
{
    printf("Mean Height Calculator\n");
    printf("Enter the height of each person when prompted\n");
    printf("Enter 0 when you are done. You can enter up to 10 values\n\n");
}

/*
 * Calculates the mean of a set of values
 */
double mean(double values[], int num_values)
{
    double mean;
    int current_value;

    /* Initialise the mean to zero */
    mean = 0;

    /* Calculate the sum of all of the values */
    for (current_value = 0; current_value < num_values; current_value++)
    {
        mean += values[current_value];
    }

    /* Divide by the number of values, if there are any */
    if (num_values > 0)
        mean /= num_values;

    return mean;
}
```

The program uses the `main` function to collect information from the user. It defines two other functions: the `welcome_message` function displays useful text to the user when the program starts; the `mean` function calculates the mean of the values that the user has entered.

The program will accept up to 10 data values, so it declares an array of 10 `double` elements, like this:

```
    double data_values[10];
```

The program allows the user to enter less than ten values if they want, so we need to keep track of how many values they *have* entered. This is done using another variable:

```
    int num_data_values;
```

After the welcome message has been displayed the program enters a `for` loop. This loop counts through the elements in the `data_values` array. Each time the loop runs (each *iteration*) the user is asked for a height. The `scanf` function call places the result into the correct element in the array:

```
    scanf("%lf", &data_values[curr_data_value]);
```

This `scanf` call is just like the ones you have seen in previous labs, except that instead of a scalar variable this call puts the result into an element of the array. The element it puts the result in is decided by the value of the `curr_data_value` variable. The value of the `curr_data_value` variable is set by the `for` loop. The `for` loop will stop when it gets to the tenth element in the array (`data_values[9]`). It will also stop if the `done` variable is assigned a value other than zero. This is controlled by the condition part of the `for` loop.

To calculate the mean, the array is passed to another function, called `mean`. Note how the parameter which receives the array is declared. The prototype for the `mean` function looks like this:

```
double mean(double values[], int num_values);
```

The array is declared with empty square brackets because we don't need to limit this function to only working with arrays of a certain size. In this case, the function just needs to know how many values out of the array to use. Even if the program always used the whole array, we would still need to pass the size of the array as a parameter to the function. **There is no way to determine the size of an array in C**. You must store it in a variable somewhere, or you must fix it at the time when you write the program.

Exercise 6.1: Understanding the Array Program

Make sure you understand how the "lab6" program works. Try stepping through some or all of it if you are unsure. If you only want to step through some of it (perhaps you only want to step through the `mean` function) then use a breakpoint to interrupt the program when it reaches the part you are interested in.

If you do not understand anything about the program, ask one of the demonstrators for help.

When you are sure that you understand how the program works, change the source code to allow the program to accept up to 15 values, rather than 10. Rebuild the program and test it to make sure that your changes have worked.

Exercise 6.2: Add a Standard Deviation Function

The standard deviation of a sample of values measures how 'spread out' the values are and may be calculated using the following formula:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N} x_i^2 - (\bar{x})^2} \tag{6.1}$$

Where:

- σ is the standard deviation;
- x_i represents the ith data value;
- $\bar{x}$ is the mean of all of the data values;
- N is the number of data values.

The equation may also be written as the difference between the mean of the squared data values and square of the mean:

$$\sigma = \sqrt{(\bar{x^2}) - (\bar{x})^2} \tag{6.2}$$

Add a function to "lab6" called `standard_deviation` which takes the array of data values and the number of data values as arguments. It should return the standard deviation. You should add lines to the `main` function to call your new function and display the standard deviation on the screen.

Hints:

- Use equation 6.2 to calculate the standard deviation.
- Create a second array of the same maximum size as the `data_values` array (i.e. 15 elements).
- You should copy the original data items into this new array, but square them.
- Use the `mean` function to calculate the two means.
- You might want to look back at the last laboratory to remind yourself about the `pow` and `sqrt` functions.
- You will need to include `math.h`.

6.4 Initialising Arrays and Multi-Dimensional Arrays

Sometimes you will want to make sure there are specific values in elements of an array before you do anything with it. Imagine a situation where you want to know the number of days in a month. If you have the number of the month (from 0 to 11) in a variable called `month`, you could use an array to find out the number of days the month has:

```
int days_in_month[12];
...
printf("The number of days in Januray = %d", days_in_month[0]);
```

For this to work you would need to make sure the array `days_in_month` was set up with the right values. We could do that like this:

```
int days_in_month[12];
```

```
days_in_month[0] = 31;
days_in_month[1] = 28;
days_in_month[2] = 31;
days_in_month[3] = 30;
days_in_month[4] = 31;
days_in_month[5] = 30;
days_in_month[6] = 31;
days_in_month[7] = 31;
days_in_month[8] = 30;
days_in_month[9] = 31;
days_in_month[10] = 30;
days_in_month[11] = 31;
```

But this is a pain. Fortunately, C provides us with a way to initialise the value of an array when we declare it. Like this:

```
int days_in_month[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

The values on the right-hand side of the assignment operator (=), in the curly brackets, are known as *initialisers*. There must not be more initialisers that elements in the array.

The next program you will write will use initialisers to set numbers in an array. The difference from the arrays we have just been looking at, is that this next program will need a *multidimensional array*. The arrays you have seen so far have a single dimension; they work like a table with a single column. The subscript is like the row number, telling the program where to look in the table. A two dimensional array has two subscripts; it works like a table with multiple columns. The two subscripts act as the row and column numbers, selecting a single cell in the table. Three and more dimensions are also possible.

A two dimensional array is declared in C like this:

```
int data_table[5][3];
```

You could think of the `data_table` array as being a table with 5 rows and 3 columns. You would access the `data_table` array like this:

```
printf("The first element in the table is %d", data_table[0][0]);
```

Notice that you need to specify both subscripts.

Initialising a two dimensional array is like treating each row as a one-dimensional array, and then collecting all of the rows together. Look at this example:

```
int data_table[5][3] = { {1, 2, 3}, {2, 3, 4}, {3, 4, 5}, {4, 5, 6},
                         {5, 6, 7} };
```

If we treat the first subscript as the column index, and the second as the row index, `data_table` is equivalent to the following table of values:

1	2	3
2	3	4
3	4	5
4	5	6
5	6	7

Copy the "lab6a" project and open it in the IDE. "lab6a" is intended to be a program which informs the user of the distance between a small selection of major world cities. However, at the moment the program isn't finished. The best way for the program to work out the distance between cities is to look it up in a table (i.e. a two-dimensional array). You will need to create and initialise a suitable array to get the program to work. You should initialise it with data from the table below. The distances are in miles.

	Cape Town	Hong Kong	London	New York	Rio de Janeiro	Tokyo
Cape Town	0	7,375	6,012	7,764	3,773	9,156
Hong Kong	7,375	0	5,982	8,054	11,021	1,794
London	6,012	5,982	0	3,458	5,766	5,940
New York	7,764	8,054	3,458	0	4,817	6,740
Rio de Janeiro	3,773	11,021	5,766	4,817	0	11,533
Tokyo	9,156	1,794	5,940	6,740	11,533	0

Exercise 6.3: Using an Initialised Two-Dimensional Array

Add a two dimensional array to the "lab6a" program and initialise it with the data shown in the table. You can then use the indices that the user enters to look up the correct distance.

Some things to look out for:

- Remember that array subscripts are indexed from zero.
- Using a subscript that is beyond the range of the array can cause big problems. Validate the data that the user enters to make sure it is valid.

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

6.5 Strings: A Special Kind of Array

You have already used pieces of text many times in your C programs. They have usually been in `printf` statements and have been enclosed in double quotation marks `"like this"`. A piece of text like this is called a *string* because it is treated by the programming language as a *string of characters*. The name *string* is just an abbreviation.

In C, strings are treated as arrays of type `char`. Each element in the array holds a single character. For example, we could declare an array to hold strings like this:

```
char some_string[8];
```

Initialising a character array like this is quite easy, for example:

```
char some_string[8] = { "Hello" };
```

When a character array is initialised like this each letter is stored in an element of the character array `some_string`. Like this:

’H’	some_string[0]
’e’	some_string[1]
’l’	some_string[2]
’l’	some_string[3]
’o’	some_string[4]
’\0’	some_string[5]
-	some_string[6]
-	some_string[7]

You will notice that the element immediately after the last character in the string (some_string[5] in this case) contains a the character ’\0’ (that’s a backslash character and a zero character, not the letter ’o’). This is an *escape sequence* just like the special sequence for a new line (’\n’). ’\0’ is the escape sequence for the *null* character. It is used to signal the end of the string. A string that ends in a null character is said to be *null terminated*. All strings in C are null terminated. When you initialise a string the null termination is added to the array for you automatically. Because the null termination takes up an extra character, you **must** make sure that any character array you declare has one more space in it than the maximum number of characters that you want it to hold. For example, if you wanted to declare an array for holding strings of up to 20 characters, you should declare it as:

```
char another_string[21];
```

to make sure that there is room for the null termination character.

You can use the null termination to find the end of a string in an array. For example, you could check the first element in the array to detect whether a string is empty (i.e. it has no characters, like this: ""). You would check for the null termination like this:

```
    if (another_string[0] == ’\0’)
    {
        /* The string is empty */
    }
    else
    {
        /* The string is not empty */
    }
```

The example project “lab6b” contains an example of how to do some basic string handling. Copy the “lab6b” project and open it in the IDE. Try building and executing the project. What does it do?

The source code for “lab6b.c” is shown below.

```
/*
*  A program to demonstrate strings
*  C Programming laboratory 6
*/

#include <stdio.h>

/*
* Program starts here
*/
int main(void)
```

```
{
    /* Declare a character array to store the string in */
    char string_data[20];

    /* Ask the user for a string */
    printf("Enter a word: ");
    scanf("%19s", string_data);

    /* Show them the string */
    printf("The word you entered is: %s\n", string_data);

    /* Tell them what the first character was */
    printf("The first character of the word is %c\n", string_data[0]);

    return 0;
}
```

You should see that the program is quite simple; it uses a `scanf` statement to read in the string. It then displays the string using `printf` and then displays just the first character using another `printf` statement.

The `scanf` and `printf` statements use a placeholder that you haven't seen before for handling strings: `%s`. For example, the `printf` statement uses the `%s` placeholder to substitute the characters out of the string variable and into the string which is going to be displayed.

You will notice that the `scanf` statement also uses a `%s` placeholder, to read in a string. However, in the case of `scanf` a number appears between the percentage sign (%) and the `s` character. The number is the maximum number of characters to read into the string. It ensures that the program does not try and put too many characters into the array.

When using an array with `scanf` you **do not** need to use the ampersand character (&) before the variable name. The reason for this is a little complicated and you will study it in more detail in laboratory 13, next term.

Function Reference: `%s` — Using the `%s` placeholder with `scanf` and `printf`

```
scanf("%length s", string_variable);
printf("%s", string_variable);
```

The `%s` placeholder allows `scanf` and `printf` to handle strings. When using `scanf` it is good programming practice to make sure the length of the string that can be read is limited using a number (*`length`*) between the percentage sign and the `s` character. This number should be one less than the length of the array to allow room for the null termination character.

As the argument which matches the `%s` placeholder is an array. It does not need to be preceded by an ampersand.

Exercise 6.4: Understanding Character Arrays

Test the "lab6b" program with different input. You should find that scanf prevents you from being able to enter nothing at all. It also only ever uses the first word that you enter.

- Why is the scanf placeholder %19s? (Why is it not %20s?)
- What happens if you enter a word longer than 19 characters?
- What happens if you change the placeholder to have a number less than 19?

Exercise 6.5: Handling Null Terminated Strings

Add to "lab6b" to create a function with the prototype:

```
int string_length(char string[]);
```

The function should count the number of characters in the string. It should stop counting when it reaches the null character. It should be able to handle situations where there are no characters in the string (i.e. the first character is the null character). Call your new string_length function from within the mainfunction so that you can display the length of the word that the user enters.

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

Exercise 6.6: Character Array Handling

Add to "lab6b" to add some code into the main function which copies the characters from the string_data array into another array, but in reverse order. You will need to use the length of the string (which you determined in your string_length function) to do this. Display the reversed string using printf. You must make sure that the new string you create (the reversed copy) has a null termination character in the right place.

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

6.6 Forming Strings Using sprintf

Sometimes you want to form a more complicated string which will be stored in a character array, rather than displayed on the screen. The sprintf function works exactly like printf except that, rather than displaying the string it makes onto the screen, in places the result into a character array which you specify. For example, look at the following code snippet:

```
char number_string[10];
int an_integer;

an_integer = 20;
sprintf(number_string, "%d", an_integer);
```

You can see that the `sprintf` function call looks very like a `printf` function call, except that there is a new first argument. The first argument to `sprintf` is the string variable that results will be placed into. After the `sprintf` statement executes, the string `number_string` contains the characters '`2`', '`0`' and '`\0`'

Function Reference: `sprintf` — Forming strings using `printf`-like placeholders

```
int sprintf(char string_variable[], char format_string[], ...);
```

e.g.

```
    sprintf(number_string, "%d", int_variable);
    sprintf(answer_string, "The answer is %d", integer_answer);
    num_chars = sprintf(display_string, "%lf", double_variable);
```

The `sprintf` function works like the `printf` function except that, instead of outputting its results onto the screen, it places them into the character array `string_variable`. It is called in exactly the same way as `printf` but with an extra first argument (`string_variable`).

`sprintf` takes at least two arguments but, like `printf`, it can take more, depending on how many variables you want to include.

- `string_variable` is the character array which will receive the output of the `sprintf` function as a string;
- `format_string` is a string containing placeholders, just like with `printf`;
- after the first two arguments (where the `...` is) you should include variables whose values will be substituted for the placeholders in the `format_string`.

The placeholders you will use most often are:

- `%d` for integer (`int`) variables;
- `%lf` `double` variables;
- `%c` for single characters (`char`);
- `%s` for strings (`char[]`).

These are the same as for `printf`.

The return value of `sprintf` is an integer, which specifies how many characters were written into the `string_variable` array (not including the null termination).

The `sprintf` function is defined in `stdio.h`

Exercise 6.7: Using `sprintf`

Add to the `main` function in “lab6b” to prompt the user for an integer number. Use the `sprintf` function to create a string version of the integer in a character array (you could re-use one of the arrays you already have, if it is big enough).

Determine the length of the string it produces in two ways:

- use your `string_length` function;
- use the return value from the `sprintf` function call.

Display both answers and make sure that they are the same!

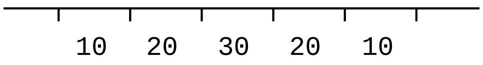
If you don’t understand how to do this, ask one of the demonstrators for help.

There is a library that is part of the C standard library which contains a number of functions especially designed for handling strings. All the functions are defined in `string.h`. If you want to find out more about these functions you should look in a book on C, or consult documentation on the internet (see Appendix B).

6.7 Graphics Option

Open your “graphics1” project. You will be adding even more functionality to it now. By the end of this session it should reach the stage where you can play it as a simple game.

Your graphics program is beginning to get quite sophisticated. The user now has control over the horizontal position of the stick person, and the launch angle of the projectile. You are now going to add a target for them to throw the object at. The target should be at ground level and have numbers on it identifying the score associated with a particular area. It could look something a bit like this:



How many areas there are, and what the scores are for each area is entirely up to you.

The best way to produce the target is using a loop. Each iteration of the loop should produce one of the small vertical lines which separate the scores. You could store the associated scores in array a bit like this:

```
int scores[5] = {10, 20, 30, 20, 10};
```

The code that you write to produce the target should have a structure a bit like this:

```
    int line;
    char label[3];

    for (line = 0; line < 5; line++)
    {
        /* Calculate position of line based on the value
        * of the line variable */

        /* Draw the line in the right place */

        /* Calculate position of label based on the value
        * of the line variable */
        /* Create a string with the label in it */
        sprintf(label, "%d", scores[line]);

        /* Use outtextxy to display the text */
        outtextxy(x, y, label);
    }

    /* Draw the last line here */
```

This is obviously just a skeleton, you will need to fill in the missing sections with code of your own.

Exercise 6.8: Drawing a Target

Create a function called `draw_target` which uses code based on the skeleton to draw a suitable target on the far right-hand side of the graphics window. Feel free to decide:

- how many areas your target will have;
- what scores the different areas will have;
- how big the different areas of the target will be.

Exercise 6.9: Giving the User a Score

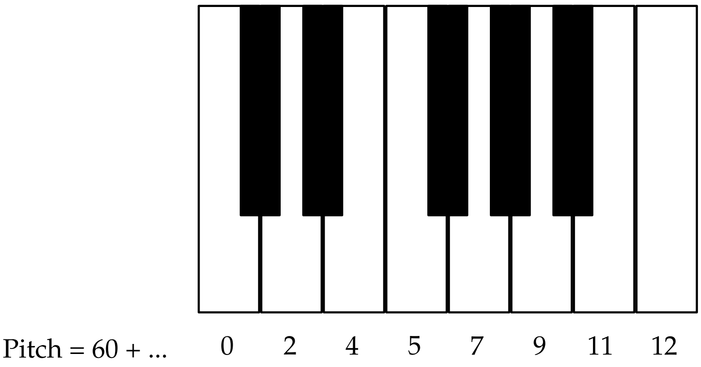
Now the user has a target to throw the object towards, the computer should reward them with a score that corresponds with the position in which the projectile path ends.

Your program calculates the vertical position of the projectile at each horizontal location. It should stop when the vertical location is not above ground level. The horizontal position at which the projectile is no longer above ground level will determine the score.

When the projectile reaches ground level (or below) your program should assign the corresponding horizontal position to a variable. You can then use that variable (which will be an x-coordinate) to calculate where on the target the object fell. You should then be able to use the values in the `scores` array to give the user a score for their throw.

6.8 Music Option

The improvisation program you have been working on has been limited to producing whole tone scales or random notes. It has been difficult to produce other scales as the differences in pitch between notes in say, a major scale, are not easily calculated mathematically. As an example, let’s look at a simple C major scale. If the scale starts on middle C the first note of the scale has a pitch value of 60. To get the other notes in the scale we could add small pitch values onto this starting value, like this:



If we chose a major scale in a different key then the pitch offsets would still be the same. But the pitch offsets are not related in any obvious mathematical way.

A way to approach this problem is to store these pitch offsets in an array. For example, you could define a major scale like this:

```
int major_scale[8] = {0, 2, 4, 5, 7, 9, 11, 12};
```

A loop like the following one would play a major scale:

```
int scale_note;
int key;

key = 60;

for (scale_note = 0; scale_note < 8; scale_note++)
{
    midi_note(key + major_scale[scale_note], 1, 64);
    pause(500);
    midi_note(key + major_scale[scale_note], 1, 0);
}
```

Arrays like these are very powerful for representing musical relationships such as scales and chords.

Exercise 6.10: Making Your Improvisation Use Non-Whole Tone Scales

Begin by opening your "music2" project. Add a function which uses as array to play a single ascending octave of a major scale. Try using it in place of the function which produced whole tone scales.

Now is your opportunity to innovate! Try and improve your improvisation program focussing on using scales other than whole tone scales.

- Try some different scales, other than a major scale.
- Try choosing which scale will be played at random.
- How about representing chords using an array? Can you get your program to improvise a chord line to accompany a melody built out of scales?
- Can you use the value of an array to predefine some `pause` values? This would allow you to predefine certain rhythms. Try playing these rhythms on MIDI channel 10 (this is the drum channel).

6.9 Summary

Now that you have finished this laboratory you should understand what arrays are and how they can be used. You should be able to declare both single and multidimensional arrays and initialise them with data.

You should know that strings are character arrays and that all strings in C are null terminated; so they end with the null character (`'\0'`). You should be able to manipulate strings and create them using the `scanf` and `sprintf` functions.

In the graphics option you added to your program to make it much more like a computer game. It should now have a target towards which the user throws the object. The user should then be given a score for their throw.

In the music option you should have found that arrays are very useful for defining sections of music by representing numerical relationships. You should understand how to use an array to represent a scale. You may also have explored how to use arrays for chords and rhythms.

Laboratory 7

Structuring a Software Project

7.1 Overview

This laboratory introduces good programming practice for structuring software projects. As your programs grow you will find that it is useful to split the functions you write across multiple source files. This laboratory teaches you the best way to do this. Although the concepts that are introduced in this lab will not have an effect on how your programs look once they are built, well structured source code is important. By using multiple files you can make your programs easier to understand and debug. You can also make it easier to reuse code you create for one program when you are writing another. Something which could save you a lot of effort!

Finally, a few tips on good programming practice are covered. Focussing on how to use indentation to make your programs more readable, and how to use the IDE to do this easily.

7.2 Splitting Your Program into Multiple Files

In laboratory 6 you learned how, and why, to split your programs into functions. One of the most important effects of splitting your code into functions is to wrap it up (or *encapsulate* it) into small chunks which can be *reused* easily. To be able to reuse your function in another program, you have had to copy it and paste it into another source file.

It is usual for more complicated C programs to be made up of more than one source file. Each source file typically contains multiple functions. The reasons for doing this are very similar to the reasons for using functions in the first place:

- using multiple files adds structure to your program and makes it easier to understand. For example, you could collect all the functions that produce output on the screen into one file, making it easier to find functions of a particular type.
- putting a set functions into a file allows you to use *the same file* in multiple projects. If there are bugs in the functions in that file, you only have to fix them once, rather than having to change the code in every project it is used in.

In this lab you will learn how to split your source code into multiple files, by placing different functions in each file.

Begin by opening the project that you have been using for your option work in the IDE. So, if you have been doing the graphics option you should open “graphics1”, or if you have been doing the music option you should open “music2”. These programs are getting quite complicated by now, and probably contain a number of functions. You are going to separate these functions into two files depending on their role in your program.

Exercise 7.1: Deciding on a File Structure

Before you actually create any new files, or change your program in any way, you should decide how, and why, you will structure the files.

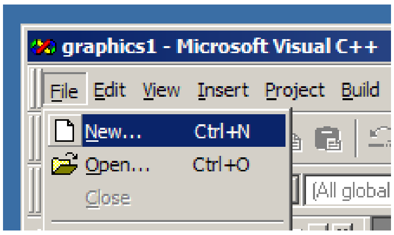
Decide how many files you want to use. This should be more than one and no more than about four at the moment. This decision should be based on what each of the functions you have written do. Some things to think about:

- You could collect together into one file the functions which interact with the user.
- If the `main` function is very complicated, you might want to leave it in a file on its own.
- You might want to collect the graphics/music functions together into one file.

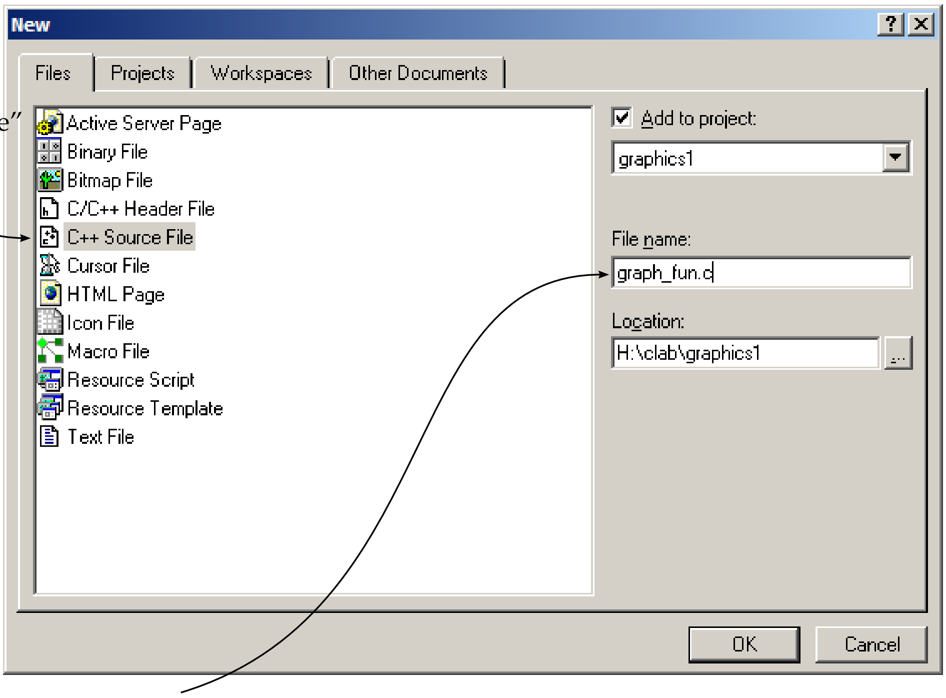
Don't make things more complicated than necessary; try to create a structure which is logical and would help someone else understand your program. Decide on names for your files which reflect what they do. To keep things simple, it is best to avoid file names that have spaces in.

Make some notes about your file structure to help you remember your decisions.

You can now create a new file in which to place some of your code. Select the “New” item from the “File” menu:



The “New” file dialog box will appear, it is shown below.



☑ Add to project: graphics1

File name: graph_fun.c

Location: H:\clab\graphics1

Select “C++ Source File”

Type the name of your new file

You wish to create a new source file. The IDE treats C source files as a special case of the more complicated language C++. To create a new C source file you should make sure that the "C++ Source File" option is selected. Type the name for your new file into the "File name" box as indicated. You **must** make sure that your file name ends in ".c".

A blank file will appear in the editor ready for you to put some source code in it. Copy (or cut) and paste at least one function into this file. **You will need to `#include` any header files which the function needs**. You can now save your new source file.

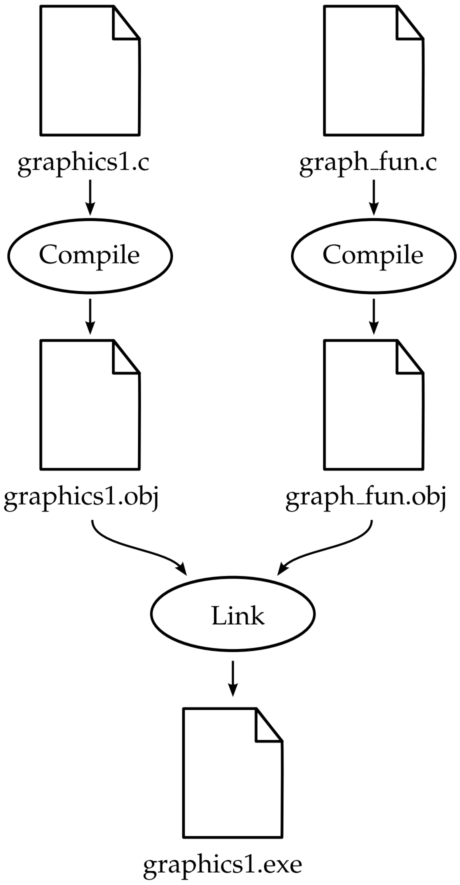
Repeat the process until you have the file structure that you decided on. If you want to remove a file from your project, select the file in the source file tree and press the "Delete" key. This **will not** actually remove the source file from the folder. But it will remove it from the project, so it is no longer part of your program.

You should now have a set of source files, which are all part of your program, and are all shown in the source file tree on the left-hand side of the IDE window. Each function in your program should only be in **one** of these source files.

Unfortunately, you are not yet ready to build your program. At the moment there is no way for a function in one file to know about a function in another file. To understand this, we must look at how projects are compiled and linked. You should remember from earlier labs that programs are built in two stages:

- **compilation** which produces an object file for each source file;
- **linking** which joins all of the object files together to make one executable.

The important part of this process is that the compiler only ever works on one source file at a time. This is shown in the diagram below:

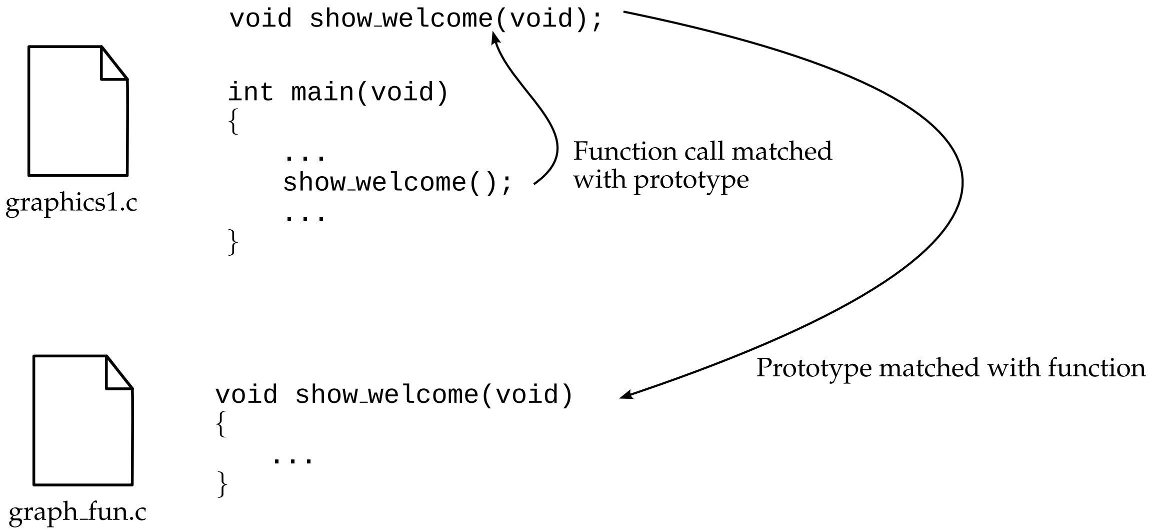


In this example, the "graphics1" project is being built. There two source files: "graphics1.c" and "graph_fun.c". The compiler first works on the "graphics1.c" file and produces the intermediate file "graphics1.obj". It then works on the file "graph_fun.c" and produces the intermediate file "graph_fun.obj". The compiler has now finished its work. The linker now runs. It takes the intermediate files "graphics1.obj" and "graph_fun.obj" and combines them together to make the executable file "graphics.exe". The files in your project will probably have different names, but the principle is the same.

To make sure that you don't get compiler errors or warnings, you must make sure that for each function that is used in one file, and declared in another, you must specify a prototype for that function. So, for example, in the example above:

- say "graph_fun.c" defines a function called `show_welcome`, which is called by the `main` function in "graphics1.c";
- for the compiler to know that the function exists in another file, there must be a prototype for `show_welcome` in "graphics1.c".

This situation is shown in the diagram below:



```
void show_welcome(void);

int main(void)
{
    ...
    show_welcome();
    ...
}
```

```
void show_welcome(void)
{
    ...
}
```

Exercise 7.2: Declaring the Correct Prototypes

You should now have created the files that you decided on, and moved the parts of your source code to the relevant files.

Make sure that each source file you have created contains the prototypes of each of the functions it needs that are in other files. It is standard practice to put a comment at the top of each file documenting what the purpose of the functions in the file are. Make sure you add a comment like this to the top of each file that you create.

Your program should now build and execute without warnings or errors.

7.3 Creating Your Own Header Files

To decide what prototypes are necessary you have to check your source file to find out what functions (if any) you use from other files. When your source files are large it is easy to forget which functions are called. Including prototypes which are not called will not generate an error or warning, but it is misleading for anyone reading your source code. The need to include prototypes for each function called in another file is inconvenient when you want to use one source file in more than one project.

The easiest way to solve these problems is to place the prototypes for all the functions in particular source file into a header file with the same name. For example, assume that a source file called "graph_fun.c" contains the following functions:

- `show_hello`
- `draw_stick_person`
- `draw_throw`

To accompany this source file a header file called “graph_fun.h” should be created. A header file is just like a source file, except that the code that it contains shouldn’t be code that actually executes. For example, the header file “graph_fun.h” will only contain function prototypes. The function prototypes will used by the compiler and linker to match up the functions in “graph_fun.c” to other files in which they are called. The function prototypes themselves are not converted into executable code.

In our example, the header file “graph_fun.h” should contain the prototypes for the three functions in “graph_fun.c”. For example, the contents of “graph_fun.h” might be as follows:

```
/*
 * Header file for graph_fun.c
 * Function prototypes only
 */

void show_hello(void);
void draw_stick_person(int x, int y);
void draw_throw(int x, int y, double velocity);
```

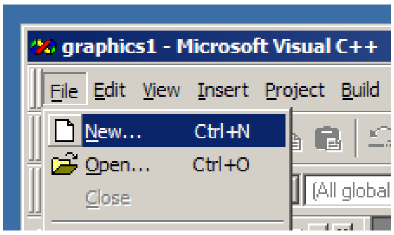
As you can see, header files can be very short!

In the file “graphics1.c”, which uses the functions `show_hello`, `draw_stick_person` and `draw_throw`, the header file must be `#include`d with the line

```
#include "graph_fun.h"
```

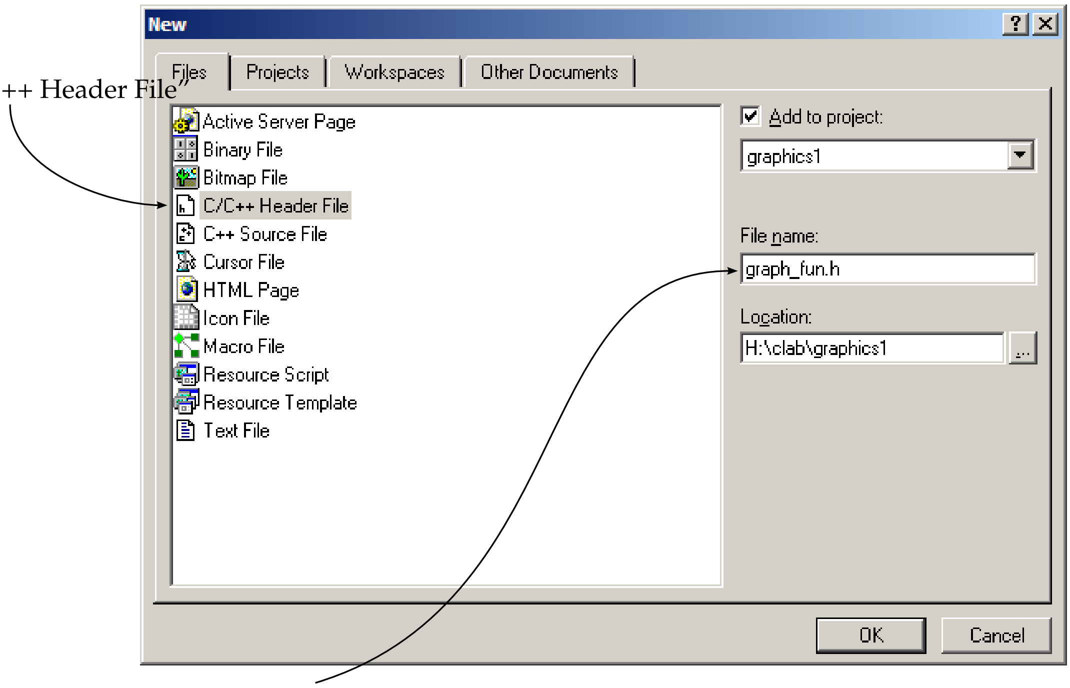
There is now **no need** to put any prototypes for the functions `show_hello`, `draw_stick_person` and `draw_throw` in “graphics1.c”. If we wished to use any of these functions in another file, we would simply put the `#include` directive at the top of the file. You will remember from laboratory 4 that the `#include` directive just includes the source from the file that is named into the current file. In this case it is used to make sure that prototypes for the functions in “graph_fun.c” are included wherever they might be used.

Creating a new header file is very similar to creating a new source file. Choose “New” from the “File” menu:


graphics1 - Microsoft Visual C++
File Edit View Insert Project Build
New... Ctrl+N
Open... Ctrl+O
Close
[All global

The “New” file dialog box will appear, just as when creating a new source file.

Select "C/C++ Header File"



☑ Add to project: graphics1

File name: graph_fun.h

Location: H:\clab\graphics1

Type the name of your new file

To create your new header file you should make sure that the "C/C++ Header File" option is selected. Type the name for your new file into the "File name" box as indicated. You should make sure that your file name ends in ".h".

Exercise 7.3: Creating Header Files

You should now create a header file for each one of your source files which contains functions that are used by another file. For example, if the file "music1.c" contains functions that are called **only** by other functions inside the "music1.c" file, there is **no need** to create a header file for it.

Use appropriate `#include` lines to allow you to remove function prototypes from your source files. Your program should still build and execute correctly.

7.4 Protecting Your Header Files

You have just created header files which allow you to encapsulate a set of functions in a source file in a way that makes them easy to use in other source files. Sometimes header files will have `#include` directives in them which include other header files. These header files may also have `#include` directives and so on. The process can become very complicated. In these situations two problems can occur:

- a *circular reference* can occur. This is where one header file includes another header file which includes the original header file. The compiler would then travel round this loop forever (actually it would run out of memory and crash, which is probably worse).
- a header file can be included into the same source file twice. If the header file only contains prototypes this is not a problem because you can declare the same prototype as many times as you want (as long as the definitions are the same). Later in the course you will put more complicated things in header files which can only be seen by the compiler once for each source file it processes.

Both of these problems can be solved if we make sure that the contents of a header file are seen **only once** by the compiler for each source file. We can do this by using the `#define`, `#ifndef` and `#endif` preprocessor directives you met in laboratory 4.

Let us take the example of a header file called “graph_fun.h”. At the top of the header file there should be a line which checks to see whether a label has been defined. The label **must be unique** to the header file. To try and ensure that the label is unique the name of the file is often used in some way. It is most common to use a label which is made of two underscores (“__”), the name of the file (except for the ”.h” extension) another underscore, the letter ‘H’ and finally another two underscores. As with all preprocessor labels, it is good programming practice to use only UPPERCASE letters for these labels. For example, for the header file “graph_fun.h” we would use the label `__GRAPH_FUN_H__`. We would check whether it is defined using the preprocessor directive `#ifndef` like this:

```
#ifndef __GRAPH_FUN_H__
```

If the label has not been defined then the preprocessor will continue reading the contents of the file. To make sure that the file does not get processed again, the first thing we do is define the label using the `#define` directive like this:

```
#define __GRAPH_FUN_H__
```

At the end of the source file we must include a `#endif` directive to match the `#ifndef` directive, like this:

```
#endif
```

Now that the compiler has seen the contents of the header file once the label `__GRAPH_FUN_H__` has been defined. If the header file is included again, the compiler will reach the line:

```
#ifndef __GRAPH_FUN_H__
```

where it will find that the label `__GRAPH_FUN_H__` has already been defined. It will therefore skip to the `#endif` directive at the end of the source file. So the contents of the header are only processed **once** no matter how many times the file is included.

The example header file that we looked at earlier, “graph_fun.h” should look like this:

```
/*
 * Header file for graph_fun.c
 * Function prototypes only
 */

#ifndef __GRAPH_FUN_H__
#define __GRAPH_FUN_H__

void show_hello(void);
void draw_stick_person(int x, int y);
void draw_throw(int x, int y, double velocity);

#endif
```

This system of protection, made up of a `#define` directive, a `#ifndef` directive and a `#endif` directive is called a *sentry*.

Exercise 7.4: Adding Sentries to Your Header Files

Make sure you understand how sentries work, and why they are useful. If you need any help, ask one of the demonstrators.

Include the correct preprocessor directives to make a sentry in each of your header files. Your program should still build and execute correctly.

7.5 Good Programming Practice: Some More Tips

This course has stressed, many times, the importance of creating source code that is easy to read and understand. The best way to achieve this is:

- structure your code logically into functions which allow for reusability;
- structure your functions logically into different source files;
- use header files so that your source files are easily reused and are not cluttered with prototypes;
- document your source code with plenty of comments that are aimed at someone who understands C but does not necessarily understand how your program works.

You can make your easy to read through careful use of *indentation*. Indentation is about changing where a line of code in your source file has its left-hand edge. It is very useful for making the structure of code inside functions clear. You should control where the left-hand edge of your lines of source code are using the "Tab" key. Whenever you start a new structure, like an `if` statement or a `while` loop, you should indicate which lines are part of this structure by moving the lines to the right by inserting a tab character at the start of each line.

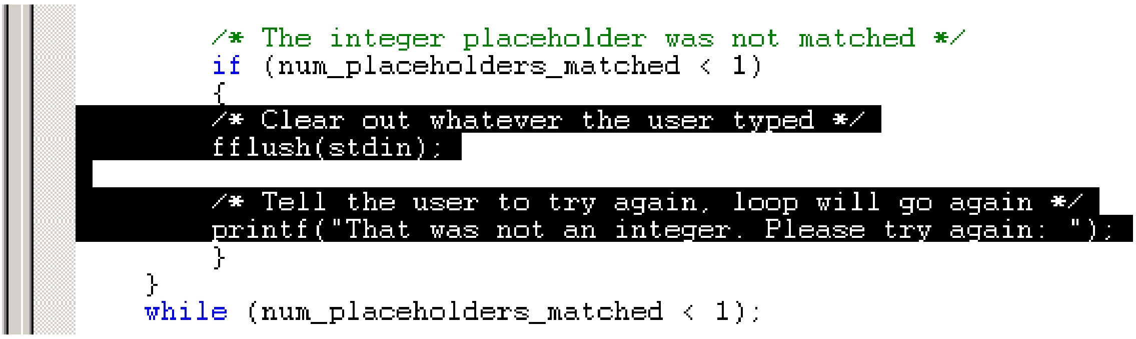
For example, an `if` statement should always be structured like this:

```
if (number_entered < 10)
    printf("That's less than 10\n");
```

Notice how the `printf` line is indented by one tab character relative to the `if` clause. The next line after this one is not part of the `if` statement and so should start in the same vertical position as the `if` clause.

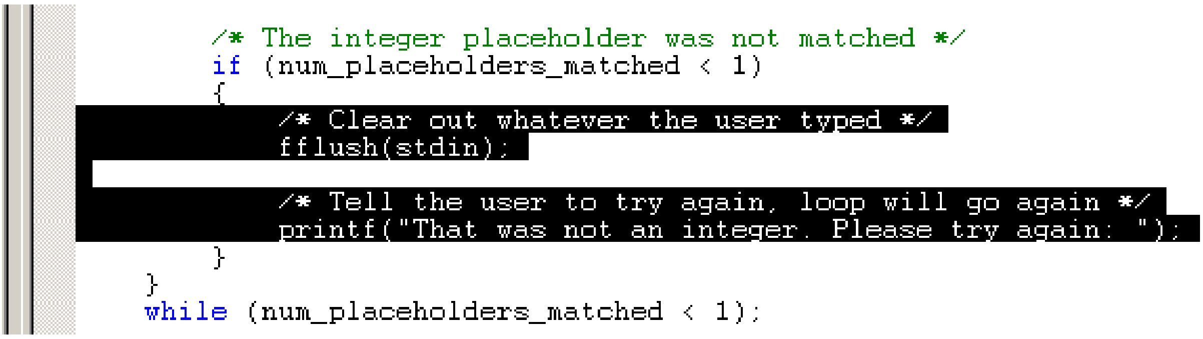
The editor in the IDE will try to carry out the correct indentation for you automatically. Sometimes, however, it may get it wrong. You can move a line to the right by placing the text cursor at the beginning of the line and pressing "Tab", which inserts a tab character. You can move the line left by holding down the "Shift" key and pressing "Tab".

In the IDE editor, this also works with blocks of code. If you highlight a block of code, like this:



```
        /* 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);
```

and press the "Tab" key, the editor inserts a tab character at the beginning of every line in the block. The block of code therefore moves to the right:



```
        /* 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);
```

You can move a block of code to the left by highlighting it and holding down the “Shift” key when you press “Tab”.

Exercise 7.5: Cleaning Up Your Code

This is the last time you will work on the program that you have been developing as part of your option. You will find the ideas that you have worked on and the code you have written very useful in later laboratories and in your assignments. To make sure your code is easy to understand you should:

- make sure you have commented your code thoroughly. If you are unsure, ask a friend to have a look at your code and see if they can understand what it does and how it works.
- make sure your indentation is correct and consistent. If you are unsure about this, check with one of the demonstrators.

There are no hard and fast rules for how to lay out and comment your source code in C, although there is some generally accepted good practice. With things like indentation it is most important that you are **consistent** across all of the source code in your project.

7.6 Summary

Now that you have finished this laboratory you should have either a music or graphics project that is split into multiple files in a logical way. It should use header files, protected by sentries, to allow easy use and reuse of the code your have written. You should understand the reasons for wanting to structure your code into multiple files, and how to apply these techniques to other projects.

You should also have spent some time making sure that you code is well-formatted, well-commented and easy to read.

You are now ready to tackle larger projects.

Laboratory 8

Practical Software Design

8.1 Overview

This laboratory looks at the process of designing software and documenting the development process in the form of a report. The lab is split into two parts:

- the first part of the lab (sections 8.3 and 8.4) gives you an existing design for a piece of software. This is exactly the kind of design that you will be expected to produce. You will implement the design in C, and then add to the design to give you practice at designing software.
- the second part of the lab (section 8.5) is a reference guide, giving you some simple tips to help you write your own software development reports.

Section 8.2 will introduce you to the lab, and the software development process.

8.2 The Structure of Software Projects

This laboratory is a bit different to the ones you have done so far. In this lab, rather than look at another part of the C programming language, we will look at how you should use the C that you already know about in a formal software development process.

The best way to produce good quality software is to follow a well structured software engineering approach. A formal approach to the development process helps to prevent errors. A small error in an early stage of the process can grow to become a large problem in the finished program. Such large problems tend to be very expensive and time-consuming to fix.

The software engineering process is usually broken down into the following steps:

- understanding the problem so that a set of **Requirements** may be created. This stage is often called *Requirements Capture* or *Requirements Elicitation*.
- analysing the requirements to understand how the problem may be solved. This stage is usually referred to as *Requirements Analysis* or just **Analysis**. The aim is to gather information ready to produce a formal specification for what the software should do.
- the creation of a formal **Specification**. A specification is typically split into things that the software is *required* to do, and things that are *desirable* for the software to do.
- using the specification as the starting point for a software **Design**. There are many approaches to designing software but the aim is always the same: to decide exactly how the software will work, and how it will be used *before* any actual programming takes place.
- the **Implementation** of the program. In your case this would be done in the C programming language. This is the only part of the process that actually involves programming. The way in which the design is implemented is documented in an **Implementation Report**.

- the software that has been produced must be carefully tested against the original specification and its functionality verified against the original requirements. There are a number of ways to approach the **Testing and Verification** of software. Usually more than one method is used to improve confidence in the correct operation of the program.
- before the software is passed on to a user, or the original customer or client, the operation of the program is documented in a **User Manual**.

An important part of the software engineering process is the documentation of each stage. In the list above, the words in **bold** indicate the names of each section of a typical software report. A high quality of documentation is essential when software is being developed by a team of engineers rather than a single programmer. Section 8.5 is a reference guide for software report writing, giving you some tips for producing your own programs and reporting on the development process.

Rather than spend a lot of time in this lab doing paper work you will be learning about program design by using a design which has already been written. This lab script will give you the specification and design for a piece of software, assuming that the requirements and analysis have already been done. They are not given here. You will have to implement the software design you have been given. Doing this will help you understand what is necessary in a good software design. This should help you when you come to write your own software reports.

There are two designs in this script: one is for the graphics option, the other is for the music option. You should attempt the one you are most familiar with.

8.3 Graphics Option

This design takes the example of a simple piece of software intended to behave like a simple drawing device (an "Etch-a-Sketch").

8.3.1 Specification

The program is required to:

- allow the user to draw a line using the arrow keys;
- allow the user to clear the drawing and start again;
- drawing should start from the centre of the screen;
- allow the user to exit at any time.

Additionally, it would be desirable if the program:

- ensures that the drawing does not leave the visible screen area;
- is easy and straightforward to use.

8.3.2 Design

The design will be split into three stages: the user interface design, a structural design and an algorithm design. This process roughly corresponds to a top-down design approach, which starts from what the program should appear to do and works towards the question of how that functionality should be achieved.

User Interface Design

The program will start by welcoming the user, informing them how to use the program and then displaying the graphics window in which they can draw.

At program start up, the following text should be displayed:

```
Welcome to SketchPad

Arrow keys let you draw
Press ’c’ to clear your drawing
Press ’q’ to quit

Press any key to start drawing...
```

The program should then wait for a key press.

Once the user has pressed any key the graphics window should be displayed. To ensure that the window fits comfortably onto most people’s screens, the graphics window should have a drawing area of 640 by 480 pixels. To allow these numbers to be changed easily they should be defined as symbols:

Symbol Name	Value
WINDOW_SIZE_X	640
WINDOW_SIZE_Y	480

Once the graphics screen is displayed, the drawing may begin. Drawing should begin at the centre of the screen, i.e. at the location (WINDOW_SIZE_X / 2, WINDOW_SIZE_Y / 2).

When the user presses a key it should be handled as follows:

- the ‘up’ arrow key should cause a line to be drawn from the current location upwards for a small amount;
- the ‘right’ arrow key should cause a line to be drawn from the current location right for a small amount;
- the ‘down’ arrow key should cause a line to be drawn from the current location downwards for a small amount;
- the ‘left’ arrow key should cause a line to be drawn from the current location left for a small amount;
- the ‘c’ key should cause the drawing to be cleared and the drawing location reset to the middle of the screen;
- the ‘q‘ key should cause the program to exit.

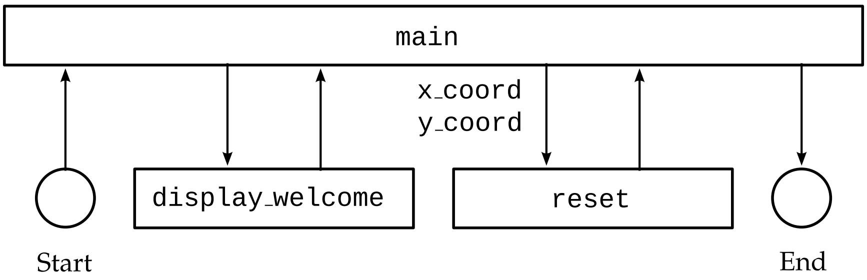
All key presses should be case insensitive. Once a line has been drawn, the “current location” will be the end of the new line. The amount that a line is drawn by should be small, but not too small, 10 pixels is reasonable in a 640 by 480 pixel window. To allow this amount to be changed easily, it should be defined as a symbol:

Symbol Name	Value
MOVE_DISTANCE	10

Program Structure Design

The program can be split up into separate functions corresponding to the different tasks the program must perform. Separate functions are used for programming clarity and to enable code reuse.

The structure of the program is shown in the diagram below:



Each of the structural units in the diagram is a function. These functions are documented in the table below:

Function Name	Input Parameters	Return Value	Description
`main`	(None)	Integer, error level passed to environment	Driving function, calls all others, always returns 0. Handles the actual drawing.
`display_welcome`	(None)	(None)	Displays the welcome message and instructions to the user
`reset`	x-coordinate (Integer) y-coordinate (Integer)	(None)	Clears the graphics window and sets the current location to the x- and y-coordinate specified

Program Algorithm Design

This section examines each of the functions identified by the structural design and outlines an algorithm for its implementation. All algorithms conform to the data conventions identified in the previous section.

The `main` function is responsible for carrying out most of the program's functionality. It calls the other functions, whenever necessary, to display the welcome message and reset the graphics window. The most important role of the `main` function is the processing of key input from the user and the drawing of lines in the graphics window.

The `main` function uses four variables:

Variable	Range	C Type	Description
`curr_x`	0 – 640	`int`	The x-coordinate of the current drawing location
`curr_y`	0 – 480	`int`	The y-coordinate of the current drawing location
`leave`	0 or 1	`int`	0 indicates that the program should continue running, 1 indicates the program should finish
`key`	←, ↑, →, ↓, 'c', 'C', 'q', 'Q'	`int`	The number of the key that was pressed by the user

The structure for the `main` function algorithm is as follows:

```
display welcome message
initialise graphics window
set initial position to be centre of screen
set leave to be zero
begin loop
    get a key press from the user
    if it was an arrow key, set current position correctly
    if it was 'c' or 'C' set the current position to be the centre of screen
    check that the position does not fall off the edge of the window
    if it does, adjust so that it is within the window
    if the key was an arrow key draw a line to the new current position
    if the key was ’c’ or ’C’ clear the screen and reset the position to centre
    if the key was ’q’ or ’Q’ set leave to be 1
loop while leave is zero
close the graphics window
```

The detailed algorithm for the `main` can be described in pseudo-code as follows:

```
function main
    call display_welcome function
    initialise the graphics window to WINDOW_SIZE_X, WINDOW_SIZE_Y
    set curr_x to be WINDOW_SIZE_X / 2
    set curr_y to be WINDOW_SIZE_Y / 2
    call reset specifying curr_x and curr_y
    set leave to be 0
    begin loop
        call getch to get a key from the user, put the result in key
        if key is zero then
            call getch again, putting the result in key
        end if
        switch using key
            case up arrow key
                subtract MOVE_DISTANCE from curr_y
            case left arrow key
                subtract MOVE_DISTANCE from curr_x
            case down arrow key
                add MOVE_DISTANCE to curr_y
            case right arrow key
                add MOVE_DISTANCE to curr_x
            case ’C’ or ’c’
                set curr_x to be WINDOW_SIZE_X / 2
                set curr_y to be WINDOW_SIZE_Y / 2
        end switch
        if curr_x is greater than 640
            set curr_x to be 640
        else if curr_x is less than zero
            set curr_x to be zero
        end if
        if curr_y is greater than 480
            set curr_y to be 480
        else if curr_y is less than zero
            set curr_y to be zero
        end if
        switch using key
            case any arrow key
                draw a line to the new curr_x and curr_y position
            case ’C’ or ’c’
                call reset specifying curr_x and curr_y
            case ’Q’ or ’q’
                set leave to be 1
        end switch
    loop while leave is zero
    close the graphics window
end function main
```

The `main` function always returns 0.

The two other functions are considerably simpler. The `display_welcome` function simply displays the welcome message and then waits for a key press. The pseudo-code is as follows:

```
begin function display_welcome
```

```
    display welcome message
    call getch to wait for a key press
end function display_welcome
```

The display_welcome function has no variables or parameters.

The reset function takes two parameters:

Parameter	Range	C Type	Description
x	0 – 640	int	The x-coordinate that the drawing location should be set to
y	0 – 480	int	The y-coordinate that the drawing location should be set to

The reset function clears the graphics window and sets the current location to the x- and y-coordinate specified by the parameters. The pseudo-code is as follows:

```
begin function reset
    call cleardevice to clear graphics window
    call moveto passing x and y parameters
end function reset
```

The reset function has no variables.

This completes the design.

8.3.3 Implementation

You should now take the time to make sure that you understand how the design is proposing to solve the problem. The pseudo-code for the main function is quite complicated. Have a think about how you are going to translate it into C.

Copy the "lab8" project. You should find that the "lab8.c" source file is completely empty. This is ready for your own code.

Exercise 8.1: Implementing the Design

Implement the 'SketchPad' program according to the design. You will need to choose appropriate C statements that match with the pseudo-code. You will also need to make sure that you #include the correct header files.

Exercise 8.2: Checking Your Implementation

Test your implementation to make sure that it works. Does it do what you expected it to?

Try out an implementation that a friend has done (you could let them try yours). How similar is it? If it is different, why is it different?

If a design is good then different implementations of it should be indistinguishable to a user.

Do you think the design could be improved? If so, how?

8.3.4 Extending the Design

You now have the chance to add to the design. Although you are not producing a software report for this program, you should make some notes as if you were writing a proper design. This gives you the chance to practice writing a small part of a design.

Exercise 8.3: Adding to the Design

Imagine that the specification contained an extra point: The program is required to:

- allow the user to change between red, green, blue and white drawing colours.

Make notes on how you would amend the design to meet this new part of the specification. Make sure that each decision you take is fully justified.

Exercise 8.4: Implementing Your Design Addition

Follow your design, and implement the additions you have just made. How easy is it to follow your design? Are you having to correct design mistakes whilst you program? Do you think that you missed anything in your design?

8.4 Music Option

This design takes the example of a simple piece of software intended to make the QWERTY (computer) keyboard behave like a basic musical keyboard with only one octave.

8.4.1 Specification

The program is required to:

- allow the user to play any note from an octave;
- use the octave that starts on middle-C;
- each note should be activated from a key press on the QWERTY keyboard;
- each note should be played for a short, but fixed amount of time;
- allow the user to exit at any time.

Additionally, it would be desirable if the program:

- uses keys on the keyboard that roughly correspond in layout to a musical keyboard;
- is easy and straightforward to use.

8.4.2 Design

The design will be split into three stages: the user interface design, a structural design and an algorithm design. This process roughly corresponds to a top-down design approach, which starts from what the program should appear to do and works towards the question of how that functionality should be achieved.

User Interface Design

The program will start by welcoming the user and informing them how to use the program. The program will then play notes in response to key presses, until the user requests that the program should exit by pressing the ′q′ key.

At program start up, the following text should be displayed:

```
Welcome to KeyNote
Press ′q′ to quit

The keyboard is the following keys:
[a] C
   [w] C#
[s] D
   [e] Eb
[d] E
[f] F
   [t] F#
[g] G
   [y] G#
[h] A
   [u] Bb
[j] B
[k] C
```

The program should then wait for a key press.

When the user presses a key that is part of the list displayed, the corresponding note should play for half a second (500ms). To allow this duration to be changed easily, it should be defined as a symbol:

Symbol Name	Value
NOTE_DURATION	500

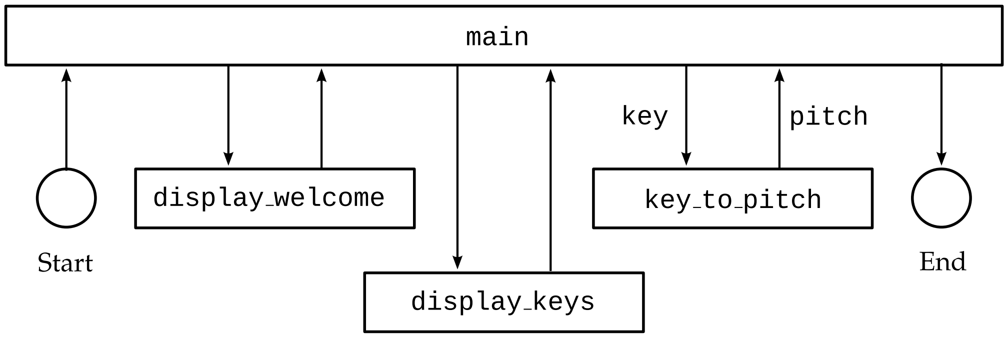
For example, when a user presses the ‘a′ key, a middle-C should be played for half a second. When the user presses the ‘t′ key, the F# above middle-C should be played for half a second. All key presses should be case insensitive, i.e. ‘a′ and ‘A′ should be treated identically.

If the user presses the ‘q′ key, the program should end.

Program Structure Design

The program can be split up into separate functions corresponding to the different tasks the program must perform. Separate functions are used for programming clarity and to enable code reuse.

The structure of the program is shown in the diagram below:



Each of the structural units in the diagram is a function. These functions are documented in the table below:

Function Name	Input Parameters	Return Value	Description
`main`	(None)	Integer, error level passed to environment	Driving function, calls all others, always returns 0
`display_welcome`	(None)	(None)	Displays the welcome message to the user
`display_keys`	(None)	(None)	Displays the instructions for how to use the keyboard to the user
`key_to_pitch`	key code (Integer)	Integer, the pitch of the note to play or zero if the key was not valid or -1 if the program should exit	Converts a key number to a pitch, or to zero if the key wasn't valid, or to -1 if the key was 'q' or 'Q' and the program should exit

The `key_to_pitch` function translates a key numbers (like that returned from `getch`) into a pitch. If the key was not a valid note key it returns zero. If the key was 'q' or 'Q' it returns -1. To allows these numbers to be easily read and recognised symbols should be defined for them as follows:

Symbol Name	Value
`PITCH_INVALID`	0
`PITCH_QUIT`	-1

The `main` contains the main loop which keeps the program running. Its most important role is to accept key presses from the user and then to call the `key_to_pitch` function to translate the key numbers into a pitch value. It then reacts to the pitch number that is returned and plays a note as necessary.

The `main` function uses three variables:

Variable	Range	C Type	Description
`leave`	0 or 1	`int`	0 indicates that the program should continue running, 1 indicates the program should finish
`key`	'a', 'A', 'w', 'W', 's', 'S', 'e', 'E', 'd', 'D', 'f', 'F', 't', 'T', 'g', 'G', 'y', 'Y', 'h', 'H', 'u', 'U', 'j', 'J', 'k', 'K', 'q', 'Q'	`int`	The number of the key that was pressed by the user
`pitch`	60 – 72, 0, -1	`int`	The pitch of the note to play or zero if the note should not be played, or -1 if the program should exit

The detailed algorithm for the `main` can be described in pseudo-code as follows:

```
function main
```

```
    call display_welcome function
    call display_keys function
    set leave to be 0
    begin loop
        call getch to get a key from the user, put the result in key
        if key is zero then
            call getch again, putting the result in key
        end if
        call key_to_pitch passing key, assigning the result to pitch
        if pitch is equal to PITCH_QUIT
            set leave to be 1
        else if pitch is not equal to PITCH_INVALID
            turn on a midi note of pitch, on channel 1 with velocity 64
            pause for NOTE_DURATION
            turn off a midi note of pitch, on channel 1
        end if
    loop while leave is equal to 0
end function main
```

The two display functions are very simple. The `display_welcome` function simply displays the first part of the welcome message. The pseudo-code is as follows:

```
begin function display_welcome
    display "Welcome to KeyNote"
    display "Press ’q’ to quit"
end function display_welcome
```

The `display_welcome` function has no variables or parameters.

The `display_keys` function displays the list of keys with their corresponding notes (see earlier in the design).

```
begin function display_keys
    display list of keys and notes
end function display_welcome
```

The `display_keys` function has no variables or parameters.

The `key_to_pitch` takes one parameter:

Parameter	Range	C Type	Description
`key`	‘a’, ‘A’, ‘w’, ‘W’, ‘s’, ‘S’, ‘e’, ‘E’, ‘d’, ‘D’, ‘f’, ‘F’, ‘t’, ‘T’, ‘g’, ‘G’, ‘y’, ‘Y’, ‘h’, ‘H’, ‘u’, ‘U’, ‘j’, ‘J’, ‘k’, ‘K’, ‘q’, ‘Q’ (Other values should be handled as invalid)	`int`	The number of the key that should be translated to a pitch

The `key_to_pitch` also has one variable:

Variable	Range	C Type	Description
`pitch`	60 – 72, 0, -1	`int`	The pitch of the note to play or zero if the note should not be played, or -1 if the program should exit

The way in which the `key_to_pitch` function works is very simple, but quite long. It simply recognises each of the valid key numbers and translates them to a pitch. The pseudo-code is shown below:

```
begin function key_to_pitch
    switch using key
        case ‘a’ or ‘A’
            set pitch to be 60
        case ‘w’ or ‘W’
            set pitch to be 61
        case ‘s’ or ‘S’
            set pitch to be 62
        case ‘e’ or ‘E’
            set pitch to be 63
        case ‘d’ or ‘D’
            set pitch to be 64
        case ‘f’ or ‘F’
            set pitch to be 65
        case ‘t’ or ‘T’
            set pitch to be 66
        case ‘g’ or ‘G’
            set pitch to be 67
        case ‘y’ or ‘Y’
            set pitch to be 68
        case ‘h’ or ‘H’
            set pitch to be 69
        case ‘u’ or ‘U’
            set pitch to be 70
        case ‘j’ or ‘J’
            set pitch to be 61
        case ‘k’ or ‘K’
            set pitch to be 72
        case ‘q’ or ‘Q’
            set pitch to be PITCH_QUIT
        any other case
            set pitch to be PITCH_INVALID
    end switch
    return from the function with pitch
end function display_welcome
```

This completes the design.

8.4.3 Implementation

You should now take the time to make sure that you understand how the design is proposing to solve the problem. The way that the `main` function uses the special pitch values `PITCH_INVALID` and `PITCH_QUIT` is quite complicated. Have a think about how this translates into C.

Copy the “lab8” project. You should find that the “lab8.c” source file is completely empty. This is ready for your own code.

Exercise 8.5: Implementing the Design

Implement the ‘KeyNote’ program according to the design. You will need to choose appropriate C statements that match with the pseudo-code. You will also need to make sure that you `#include` the correct header files.

Exercise 8.6: Checking Your Implementation

Test your implementation to make sure that it works. Does it do what you expected it to?

Try out an implementation that a friend has done (you could let them try yours). How similar is it? If it is different, why is it different?

If a design is good then different implementations of it should be indistinguishable to a user.

Do you think the design could be improved? If so, how?

8.4.4 Extending the Design

You now have the chance to add to the design. Although you are not producing a software report for this program, you should make some notes as if you were writing a proper design. This gives you the chance to practice writing a small part of a design.

Exercise 8.7: Adding to the Design

Imagine that the specification contained an extra point: The program is required to:

- allow the user to change the volume at which the notes are played using the up and down arrow keys.

Make notes on how you would amend the design to meet this new part of the specification. Make sure that each decision you take is fully justified.

Exercise 8.8: Implementing Your Design Addition

Follow your design, and implement the additions you have just made. How easy is it to follow your design? Are you having to correct design mistakes whilst you program? Do you think that you missed anything in your design?

8.5 Writing Your Own Report

This section contains some very brief tips to help you write your own software reports. These points should reinforce the material you have covered in the lectures. Each of the sections below is intended to reflect a section of your software report.

8.5.1 Requirements

The Requirements section should state the problem you have been given, in its entirety. You can imagine that the problem you have been given is a problem from a customer, or client. Quite often you will have requirements that are not explicitly stated in the problem. In your case these requirements arise because you are doing a course on C programming. Even though these requirements are implicit, you should still state them. They should include:

- the fact that you are expected to use the C programming language;

- the platform that you are expected to program for (a PC running Microsoft Windows);
- the graphics or MIDI library you are expected to use.

8.5.2 Analysis

Your Analysis section should take the Requirements section as its starting point. You should then attempt to analyse the requirements to extract more information from them. This is where you should flesh out the requirements with assumptions of your own, if you have to.

The most important rule is that when you are doing an analysis you must treat the problem as a *black box*. Treating something as a black box means that you should only be concerned with the the problem itself, and its effects, **not how you will solve the problem**. You are aiming to gather information so that you can put together a concrete specification for what the software should do.

The main things you should be concerned with in an analysis are:

- The inputs to the problem including units and expected range.
- The outputs the solution will **have** to produce including units and expected range.
- What kind of screen output is implied by the specification; will it be graphics or text?
- What size screen (especially for graphics) is reasonable on your platform?
- What is the available range of midi notes/channels?
- All formulae your program will use including calculation formulae and others such as physics, tempo conversion etc.
- State how the results of your program will appear/sound, in general terms.
- Who is your user? What are their needs? Are they technically or musically literate? Are they computer literate?

As a general rule of thumb, it shouldn't matter that you are going to implement your program in C. The design should be independent of programming language. This means that it should not contain C terms like `int`, `double` or functions like `scanf` or `midi_note`.

Be careful not to begin your design in your analysis. You cannot begin a design until you have a formal specification.

8.5.3 Specification

You should use your analysis to put together a list of **exactly** what your program needs to do. It is very useful if you split this list into things that your program **must** do, and what it would be **desirable** if your software did.

You may find it helpful if you number the points in your specification so that you can refer to them in later stages of your report.

8.5.4 Design

As you will have seen in this laboratory, your design is where you make all the major decisions about **how** your program will work. All these decisions should be made before you begin writing source code because you have a far better understanding of how the different pieces of your program fit together at the design stage.

A good design will be very detailed and should leave very few decisions to the programmer who implements the design. If you gave your design to someone else to code as well as yourself, a user should not be able to tell the difference between the two versions. No design decision should appear arbitrary — they should all be linked to the points in your specification.

Here are some guidelines for things that you should include in your design:

- Describe your user interface in detail including exact coordinates of all lines on the graphics screen (if appropriate), exact wording of text the user will see and how, when and where this will appear.
- Describe all acceptable user input and when it should be accepted by the program.
- Include a detailed structure diagram showing data flow between functions or parts of your program.
- Give a description of all algorithms used, including how you will validate user input.
- Specify a data table showing the name, type and value range of each variable you will use.

There should be a logical flow through your design, starting from the specifications and ending up at a full design for the program.

8.5.5 Source Code — Implementing the Design

You will be expected to include the full source code to your software as part of a software report. This is so that someone reading your report can understand how your software works and how they might add to it or change it.

Your code should be well commented. Assume the reader of your code understands C. Comments should explain why something is happening, not what is happening. They way you write code can also help a reader understand it. Make sure that you use variables and programming statements in a logical way. It helps if you choose logical names for variables and functions.

The way that you choose to lay out functions in your source files can heavily influence how logical your program seems to the reader. It is often best to put the `main` function first, in a C file with the same name as the program.

8.5.6 Implementation Report

Your source code should be supported by an Implementation Report. This section tells the reader how the different elements of the design have been turned into C.

You should include in your implementation report:

- decisions you have taken when implementing the design, such as what files you have used and what functions are in them;
- the way that your functions in different files interact, along with information about any header files that you have created;
- documentation and justification for any changes that you have made to the design whilst doing the implementation.

Someone should be able to understand your source code, and how it works, by reading your design, implementation report and the source code itself.

8.5.7 Verification and Testing

The Verification and Testing section of your report should aim to prove that your software functions correctly. This means that it should always react appropriately to user input, it should meet the specification that you set out after your analysis and it should satisfy the requirements that were originally given to you.

Testing software is a very complicated process. A perfect testing process, or *regime*, would test your software against all the possible inputs it could ever take, in all the conditions it all the conditions it could ever be run, making sure that the outputs are appropriate. In all but the most simple of cases, this is infeasible. It would just take too long. So most software testing is a compromise.

With your own testing, you must decide, and justify, what testing is necessary to prove that your software works. You should also comment on how you think your testing regime compares to the ideal in which every part of your program is tested. How much of your program has been tested is often called the test *coverage*.

Your testing and verification report should:

- describe your testing strategy to show that your program works correctly under all conditions;
- explain why you think you strategy is sufficiently comprehensive;
- include all test input data and test results and comment on your test coverage;
- detail any modifications you made following test failure and show the results of re-testing.

8.5.8 User Manual

You can imagine that most of your software report is written for someone who understands software development in C and wants to understand your program. The user manual, on the other hand, is written for the user that you identified in your analysis. It may be that this user is not technically literate. That way that your user manual is written should reflect the type of person you think will be using your program.

Your user manual should include:

- installation instructions — this may just be copying the executable file for your program;
- user and system requirements — you should say what type of computer is necessary to run the program and what kind of expertise you are expecting on behalf of the user;
- full usage instructions for all your programs features, ideally with examples;
- you might also like to include answers to frequently asked questions (FAQs).

8.6 Summary

Now that you have finished this laboratory you should have a good idea of what needs to be in a software design, and how you might go about writing one yourself.

You should also know that the reference section of this lab will be useful to you when you come to write your own software reports.

Laboratory 9

Assignment Work

9.1 Overview

This laboratory has been left open for you to work on any parts of previous labs that you have not finished yet, or to work on your assignment with the help of the demonstrators.

Part II

Spring Term Laboratories

Laboratory 10

Assignment Demonstration

10.1 Overview

In this laboratory you will be asked to demonstrate the software that you have been developing as part of your assignment to the demonstrators. You will not be able to ask for assistance in this session.

Laboratory 11

Structures and Defining New Types

11.1 Overview

In laboratory 6 you saw how arrays could be used to collect information together in a useful way. This lab introduces *structures*, which are another way of collecting information together. In an array, all the entries must be of the same type. The information stored in a structure may be of many different types. Structures allow you to write more compact programs and make them much easier to design and understand.

This laboratory will also introduce you to the `typedef` statement, which allows you to define your own types (like `int`, `double` etc.). This is most useful when used in combination with structures.

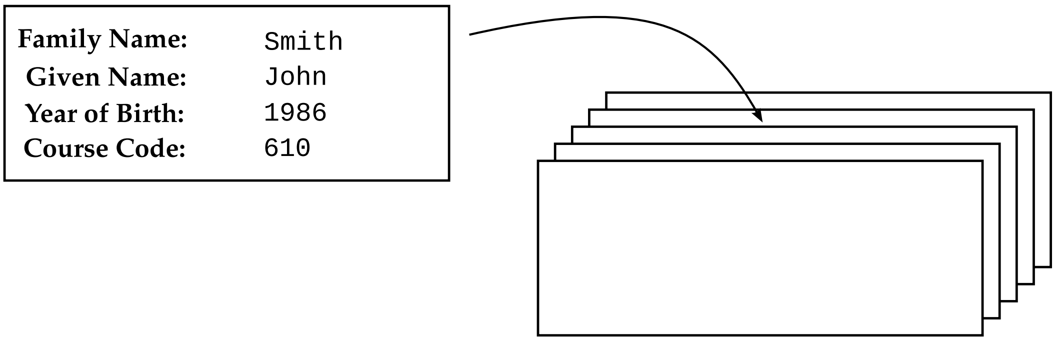
The graphics and music options each show you one way in which structures can be put to good use.

11.2 Introducing Structures

Imagine that we want to write a program to store details about students attending a course at a university. Let us say that we want to be able to store the following pieces of information about the student:

- their family name (or surname);
- their given name (or first name);
- the year in which they were born;
- the code of the course they are on.

Before computers were used to store this kind of information it would be common to put each of these details on a card. The card could then be filed with other cards which have the details of other students.



Family Name:	`Smith`
Given Name:	`John`
Year of Birth:	`1986`
Course Code:	`610`

A card is useful because it keeps the various pieces of information about the student all in one place.

In C we have used arrays to keep pieces of information together, like distances or positions or notes in a musical scale. In an array, all the items must have the same type. This can be very limiting. The details we want to store about each student are not naturally all the same type:

- the family and given names are character strings;
- the year in which they were born is an integer;
- the course code is a three digit integer.

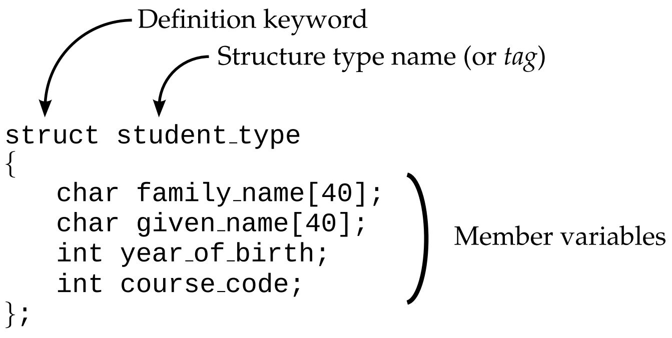
There are two strings and two integers. It would be possible to store the two integers as strings, but this would require conversions to store and retrieve the information. This would be inefficient and error prone.

C *structures* allow us to collect information together in a very similar way to the cards mentioned above. We can declare a structure using the `struct` keyword. This is normally done in two stages:

- Firstly you use the `struct` keyword to define a template for the structure. This is a bit like creating a card which has headings, but has not been filled in yet. The template is given a name so that you can refer to it.
- Once you have declared the template, you can now use it. To do this, you specify the name of the template when declaring a variable in place of a type name like `int` or `double`.

We will look at each of these stages in turn.

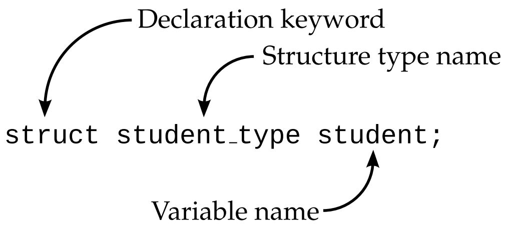
To define the template you first specify the `struct` keyword followed by a *tag* which is the name of the template you are defining. Following this, in a set of curly braces, you define one or more variables which are part of the structure. These are called *member variables*. A structure template definition for the details we want to store about a student is shown below.



```
struct student_type
{
   char family_name[40];
   char given_name[40];
   int year_of_birth;
   int course_code;
};
```

Remember that this **does not actually declare a variable yet**. All we have done is decided on a template, a layout for the card which we are going to fill in.

Now that we have a template, we can use it. We declare a variable called `student` which is a structure of the template `student_type`:



```
struct student_type student;
```

To use the `student` variable you must specify which one of the member variable you wish to access. For example, to set the year of birth to 1986, you would write:

```
student.year_of_birth = 1986;
```

The dot says that the variable before it is a structure, and the variable afterwards specifies which part of the structure we want to access.

Syntax: Structures

```
struct name
{
   member variable declarations
};
```

A structure is a way of grouping variables together where the variables may have different types. Structures also create more readable code because, unlike arrays, each item of data in the structure has a name, rather than just a number.

A structure definition specifies a new structure type called *`name`* which has member variables specified by the *`member variable declarations`*. For example, the following code defines a new structure type called `Point` with two member variables, `x` and `y`.

```
        struct Point
        {
            int x;
            int y;
        };
```

This structure definition can also be written:

```
        struct Point
        {
            int x, y;
        };
```

The *`member variable declarations`* follow exactly the same syntax as ordinary variable declarations.

To use the `point` structure, a variable should be declared using type `struct point`:

```
        struct Point some_point;
```

The `x` and `y` parts of this variable can now be accessed using the dot (`.`) operator. For example:

```
        some_point.x = 50;
```

Copy the “lab11” project and open it in the IDE. This project is a very simple demonstration of how to use structures.

Exercise 11.1: Using Simple Structures

Try building and executing the "lab11" project. Once you have run it, and tried entering some details, have a look at the source code. Make sure you understand what is happening and how it is being achieved.

Add a member variable to the structure for favourite colour. Add functionality to the program to allow the user to enter a favourite colour. The program should also display the student's favourite colour when the other details are displayed.

11.3 Defining New Types with `typedef`

The `typedef` keyword allows you to define new types, just like `int` and `double`. The new types you define must be based on existing ones. For example:

```
typedef int Distance;
```

declares a new type called `Distance` which is identical to `int`. `typedef` statements are usually placed in the *global scope* of the program (i.e. not inside any functions) or in a header file so that the type may be used by lots of different functions. The new type may be used in a variable declaration like this:

```
Distance distance_to_tokyo;
```

If at some point in the future you wanted to make all the variables of type `Distance` more precise, you could change the `typedef` statement to define `Distance` as a `double` rather than an `int`:

```
typedef double Distance;
```

In this way `typedef` statements can help make your program more readable by making your types more flexible, and your type names more descriptive. By creating a type called `Distance` you have quite a good idea about what variables of that type are going to store.

`typedef` statements are especially useful when dealing with structures, which is why they are being introduced in this lab. When we defined a structure for holding student details it was done likes this:

```
struct student_type
{
    char family_name[40];
    char given_name[40];
    int year_of_birth;
    int course_code;
};
```

This effectively defines a new type, but to use this type we have to use the `struct` keyword as well. You saw this in the "lab11" program:

```
struct student_type student;
```

We can make this shorter, and therefore easier to read (and type!), by using a `typedef`.

First, let's change the structure definition to rename the structure type name:

```
struct long_student_type
{
    char family_name[40];
    char given_name[40];
    int year_of_birth;
    int course_code;
};
```

The prefix `long_` has been added because we are going to create a type name which can be used as shorthand, and this is the long version. We can now define a new type as shorthand for `struct long_student_type`:

```
typedef struct long_student_type student_type;
```

This new type is easier to use than the old one, because we can omit the `struct` keyword when creating variables. So:

```
student_type student;
```

declares a new variable called `student` of type `student_type`.

When using type definitions with structures it is usual to combine the `struct` definition and the `typedef` all in one statement. Like this:

```
typedef struct long_student_type
{
    char family_name[40];
    char given_name[40];
    int year_of_birth;
    int course_code;
} student_type;
```

This defines a structure `struct long_student_type` and creates a shorthand for it called `student_type`, all in one line. As with other type definitions, **the best place to write this code is in the global scope**.

Syntax: Type Definitions

`typedef` *`old_type_name new_type_name`*`;`

A `typedef` statement defines a new name for an existing type. The new name is an exact replacement for the old name, it can be used interchangeably with the old name (i.e. it is a *synonym*). The old type name is still valid.

For example:

```
    typedef double Length;
```

Defines a new type called `Length` which is equivalent to the `double` type.

Exercise 11.2: Using `typedef` with Structures

Change the “lab11” program to use a `typedef` for the structure type so that the keyword `struct` does not need to be used when declaring variables.

Ensure that the type definition is in the global scope of your program (i.e. outside of any function).

Exercise 11.3: Using Your New Type Name

Create a function which displays all of the details about a student. Each piece of information should be shown on a separate line. Alter the program to use the function to display the contents of the structure in place of the final `printf` statement.

The function might have a prototype like this:

```
void display_student(student_type student);
```

11.4 Using Arrays of Structures

At the beginning of this lab the analogy was drawn between a structure and a card with details written on it. Usually there are many cards, all of the same type, kept together in a filing cabinet of some kind. So far we have only used one card. In this section we will use multiple cards by creating an array of structures.

An array of structures is easy to create by using the type that your have defined. An array of five student records would be declared like this:

```
student_type students[5];
```

You can now access individual structures in the array by specifying their index. For example:

```
    students[2].year_of_birth = 1986;
```

Exercise 11.4: Using an Array of Structures

Edit the "lab11" program so that the user is first asked how many students they wish to enter details for. This could be a number anywhere between 1 and 10. If the user enters 0 the program should end. Ask the user for the details of as many students as they specified. The student details should be stored in structures in an array.

When the user has entered all of the details the program should display all of the information stored in the array. This should be achieved by calling your `display_student` function repeatedly, in a loop.

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

11.5 Graphics Option

A structure can be used to store information about a graphical object such as a line, or a shape. *Vector graphics* editing programs manipulate information about graphical objects in this way. In this lab you will store the information for a set of lines in a structure.

A line can be fully defined by:

- its start coordinate (x, y);
- its end coordinate (x, y);
- its colour (an integer value).

This information could be stored in a structure like the following:

```
typedef struct long_line_type
{
    int start_x, start_y;
    int end_x, end_y;
    int colour;
} line_type;
```

The project "graphics3" contains the skeleton for the program you are about to write. Copy the project and open it in the IDE.

Exercise 11.5: Using Structures for Graphics

Write a program which prompts the user for the coordinates and colours of a number of lines. You might like the user to be able to state how many lines they want to use, although you will probably have to impose a maximum limit.

When the user has finished entering the information, draw the lines on the graphics window and wait for a key press.

You might like to add a feature to allow the user to edit the details of the lines that they have already entered so that they may correct mistakes.

A 2D coordinate is always made up of both x and y portions. It would make sense to define a structure to hold coordinates, or points, in a 2D space.

```
typedef struct long_point_type
{
    int x, y;
} point_type;
```

Exercise 11.6: Nesting Structures

Add the type definition for `point_type` to your program. Change the definition of `line_type` so that the start and end location of the lines are specified as points. You will end up with a structure inside a structure.

Exercise 11.7: Implementing Simple Vector Graphics Capability

How do you think you could use one type of structure to define a range of shapes (like lines, squares, circles, triangles etc.)? What things do they have in common? What special items of information might you need for some of them? How could you make a single drawing function draw all of the shapes? Can you include flood fill information?

Try implementing a few of your ideas. You will need to change your structure type definition, the user input code and the drawing routine.

11.6 Music Option

In this laboratory you will use structures to create a simple sequencer. The sequencer will allow the user to input the details of various notes and then play them back, in order.

Each note can be characterised by four pieces of information:

- pitch;
- channel;
- velocity;
- duration (in milliseconds).

This information could be stored in a structure like the following:

```
typedef struct long_note_type
{
    int pitch;
    int channel;
    int velocity;
    int duration;
} note_type;
```

The project "music3" contains the skeleton for the program you are about to write. Copy the project and open it in the IDE.

Exercise 11.8: Using Structures for Music

Write a program which prompts the user for the details required to play a number of notes (pitch, channel velocity, duration). You might like the user to be able to state how many notes they want to use, although you will probably have to impose a maximum limit.

When the user has finished entering the information, play the notes back to the user.

You might like to add a feature to allow the user to edit the details of the notes that they have already entered so that they may correct mistakes.

Exercise 11.9: Operating on the Sequence

Alter your program so that the user may choose to play back the note sequence they have just recorded in a number of different ways:

- transposed up or down;
- in reverse order;
- the instrument(s) that is/are being used.

Exercise 11.10: A More Advanced Sequencer

The sequencer you have created is limited to playing only one note at once. Have a think about how you might lift this restriction. How would this change the information that you store in a structure? How would the playback function have to be changed? Could you incorporate changes in instrument into the sequence?

Try implementing a few of your ideas.

11.7 Summary

This lab has introduced you to structures, a powerful way of collecting together information. An array collects information, but a structure is often useful because:

- the different parts of it (*members*) may have different types;
- the members are named, rather than simply referred to by index.

The middle part of the lab introduced type definitions, and showed a way in which they are commonly used when defining structures.

You should also have used a collection of structures in an array, both in a simple example for storing student details and in the graphics and music options. The graphics option used an array of structures to deal with graphical objects such as lines in a similar way to vector graphics editing programs. The music option used an array of structures as the basis for a simple sequencer.

Laboratory 12

Pointers and Passing Parameters by Reference

12.1 Overview

This laboratory begins by investigating how parameters are normally passed to functions. When a function is called, the value of an argument is copied into a parameter (which acts like a variable) inside the function. This is called *passing by value*.

You will then be introduced to a special kind of variable called *pointers* which, rather than store a value of their own, reference another variable. A pointer allows the program to find, and modify, another variable. In the next part of this lab you will use pointers when calling functions to allow them to modify the variables that have been passed as arguments. This is called *passing by reference*.

In the final part of the lab you will use pointers with structures to improve the way in which your programs from lab 11 are written.

12.2 More About Function Parameter Passing

This laboratory will introduce you to a useful and very powerful technique for working with variables. Before you can go on to this, we will take some time to look at the way in which variables work when functions are called. We will use very simple pieces of C code to understand important concepts.

The source code below is from the "lab12" project. Copy the project and open it in the IDE. Don't execute the program yet.

```
void change(int i)
{
    printf("The number is now %d\n", i);

    i = 20;

    printf("The number is now %d\n", i);
}

int main(void)
{
    int j;

    j = 5;

    printf("The number is %d\n", j);
    change(j);

    printf("The number is now %d\n", j);

    return 0;
}
```

You will see that there are two functions in this program. The `main` function and another function called `change`.

> ### Exercise 12.1: Investigating Function Parameters
>
> Before you try running the program look at it carefully. Decide *exactly* what you think it will do. What will it display on the screen?
>
> Try running the program. Does it do what you thought it would? Do you understand why it operates in the way it does?

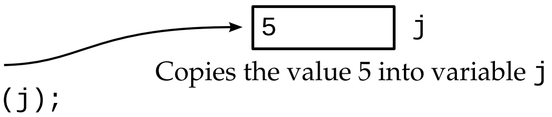
You should remember some important facts about functions from lab 5:

- parameters (the part in parentheses after the function name) are just like variables;
- all variables (and parameters) that are declared in a function are only accessible inside that function;
- when a function is called, it is common to specify some arguments in brackets after the function name;
- the values of these arguments are copied into the function parameter variables.

These points mean that if a parameter is changed inside a function the value of the variable that was used as an argument is **not** affected.

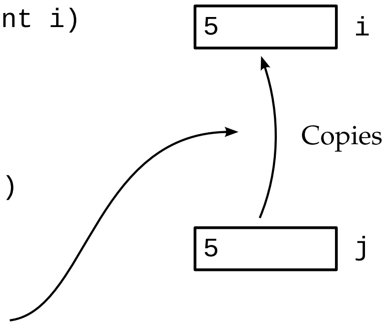
To emphasis this point, let's look at how the simple program from the "lab12" project works. To begin with, a value is assigned to the variable `j`. This is shown below:

```
void change(int i)
{
   i = 20;
}

int main(void)
{
   int j;
   j = 5;
   change(j);
   return 0;
}
```



When the function is called, the variable `j` is specified as an argument. Its value is copied into the `change` function's parameter `i` (see the diagram below).

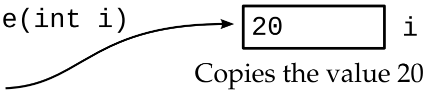
```
void change(int i)
{
   i = 20;
}

int main(void)
{
   int j;
   j = 5;
   change(j);
   return 0;
}
```



Copies the value from argument variable `j` into parameter `i`

In the `change` function, the value 20 is assigned to the parameter `i`. Notice that the value of the variable `j`, which only exists inside `main`, **is not affected**.

```
void change(int i)
{
   i = 20;
}
```



Copies the value 20 into parameter `i`

```
int main(void)
{
   int j;
   j = 5;
   change(j);
   return 0;
}
```

`main` variable `j` unaffected

When the `change` function returns the program flow to the `main` function, the value of `j` has not changed.

```
void change(int i)
{
   i = 20;
}

int main(void)
{
   int j;
   j = 5;
   change(j);
   return 0;
}
```

Value of variable `j` has not changed

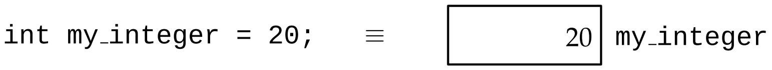
This system of copying the *value* of an argument into the parameter is called *passing by value*. It is the normal way in which C functions work. It has a very good side and a less good side:

- the good side is that functions are not able to affect the code that calls them. This stops them from damaging other parts of your program and ensures encapsulation (this was mentioned in laboratory 5).
- the less good side is that the only way to return a value is by using the `return` statement. This limits a function to only returning one value. Sometimes you might want to write code in which a function returns more than one value.

The next part of the lab introduces *pointers* which, amongst other things, allow us to get round this limitation.

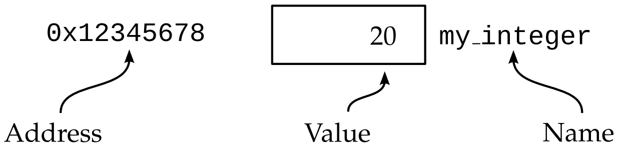
12.3 Introducing Pointers

When you store a number it in a variable, the program represents that number as binary and places in the computer's memory. In some diagrams in this lab script this process has been shown by depicting the memory location as a box containing the number. Like this:



(This diagram appeared in lab 6). You can think of the computer's memory being entirely made up of boxes like this one. Some of these boxes are used for your program's variables, others are used for your program's code and more are used by other programs.

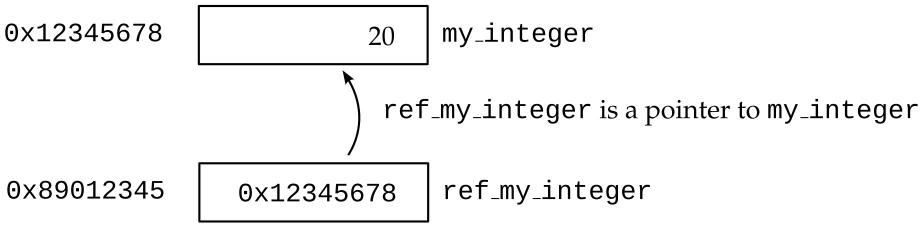
Every box or *location* in the computer's memory is numbered. The number for a location identifies it uniquely and is called an *address*. You can refer to any location in the computer's memory by specifying its address. Since addresses are just numbers it can be difficult to keep track of them so programming languages give us a way of representing location addresses with names. A variable name (like `my_integer`) is just an easier way of talking about a location in the computer's memory.



The address in this diagram is made up. It is difficult to know what address your variables will have before your program loads. Because C gives a name to each variable, you don't need to know the address.

C provides a way to use one variable to refer to the contents of another variable. This kind of variable is called a *reference*, because instead of holding a value of its own it references another memory location, which holds a value. In C, references are more commonly used *pointers*. The two names are interchangeable, we will most often use the name *pointer* in this lab script.

The diagram below shows a pointer variable called `ref_my_integer` which references the variable `my_integer`. The reference is shown with an arrow.



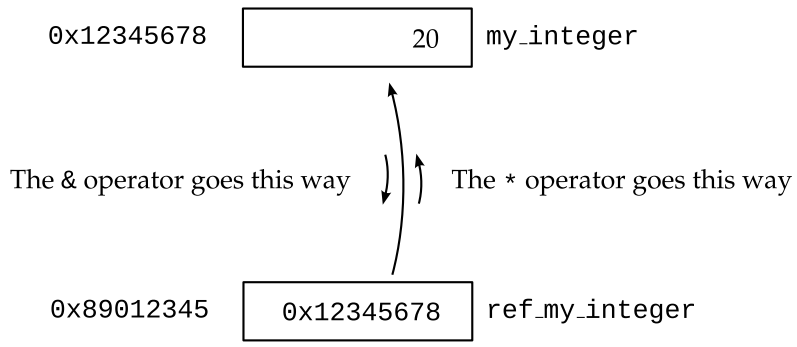
Notice that the contents of the memory location associated with the pointer variable `ref_my_integer` is `0x12345678`. This is the address of the memory location associated with the `my_integer` variable.

Pointers in C are declared like this:

```
int *ref_my_integer;
```

This line declares a new variable called `ref_my_integer` which is a pointer, or a reference, to an `int`. Notice that we have specified what this point will reference: an integer (type `int`). This pointer cannot be used to reference a `double` or a `char` or anything else. This helps to prevent some simple programming mistakes.

The star (`*`) is one of two C operators which are used when working with pointers, the other is the ampersand (&). You can think of these operators as travelling along the arrow which goes from a pointer to the memory location it is pointing to.



To make this clearer, let's look at an example piece of code:

```
int main(void)
{
   int my_integer;
   int *ref_my_integer;

   ref_my_integer = &my_integer;

   my_integer = 20;
   printf("my_integer is %d\n", my_integer);

   *ref_my_integer = 5;
   printf("my_integer is now %d\n", my_integer);

   return 0;
}
```

This piece of code is in the project "lab12a". Copy it and open it in the IDE.

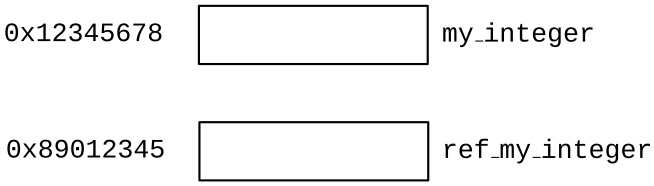
Exercise 12.2: Investigating Pointers

Build and execute the program "lab12a". Look at the source code and have a go at trying to understand how the program works. Don't worry if you don't understand, the next part of the lab will explain this program.

We will now investigate this program a line at a time.

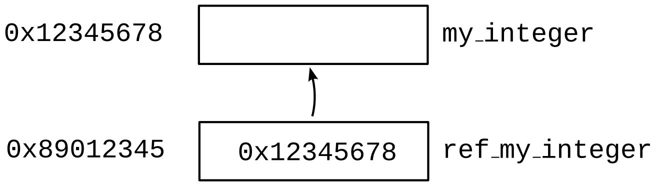
The first two lines of the program declare the integer variable `my_integer` and the pointer variable `ref_my_integer` which is specified as referencing integers. The program has not assigned values to these variables yet.

```
int main(void)
{
   int my_integer;
   int *ref_my_integer;

   ref_my_integer = &my_integer;

   my_integer = 20;
   printf("my_integer is %d\n", my_integer);

   *ref_my_integer = 5;
   printf("my_integer is now %d\n", my_integer);

   return 0;
}
```



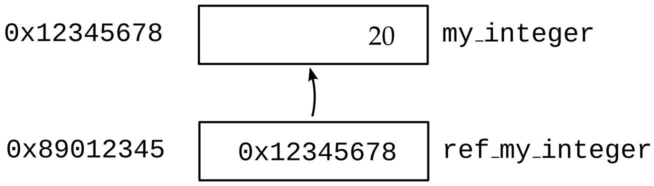
The next line uses the & operator to set up a pointer from the integer variable `my_integer`. The & operator is called the *reference operator* because it allows us to create a reference from a variable. A reference (pointer) to the variable `my_integer` is created and assigned to the pointer variable `ref_my_integer`.

```
int main(void)
{
   int my_integer;
   int *ref_my_integer;

   ref_my_integer = &my_integer;

   my_integer = 20;
   printf("my_integer is %d\n", my_integer);

   *ref_my_integer = 5;
   printf("my_integer is now %d\n", my_integer);

   return 0;
}
```



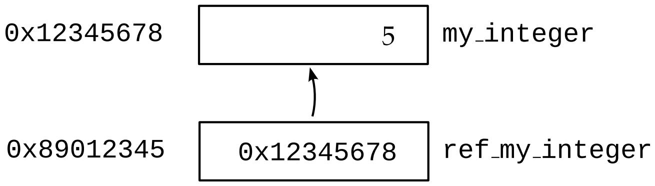
The next line assigns the value 20 to the variable `my_integer` and then uses a call to the `printf` function to display its value.

```
int main(void)
{
   int my_integer;
   int *ref_my_integer;

   ref_my_integer = &my_integer;

   my_integer = 20;
   printf("my_integer is %d\n", my_integer);

   *ref_my_integer = 5;
   printf("my_integer is now %d\n", my_integer);

   return 0;
}
```



The next line uses the `ref_my_integer` pointer variable with the * operator. The * operator is called the *dereference operator* and tells the program that we are not talking about the pointer variable, `ref_my_integer`, by what ever it references, in this case the `my_integer` variable.

```
int main(void)
{
   int my_integer;
   int *ref_my_integer;

   ref_my_integer = &my_integer;

   my_integer = 20;
   printf("my_integer is %d\n", my_integer);

   *ref_my_integer = 5;
   printf("my_integer is now %d\n", my_integer);

   return 0;
}
```



The assignment copies the value 5 into the location that is referenced by the pointer `ref_my_integer`, which changes the `my_integer` variable. The value that is displayed on the screen is therefore 5.

Exercise 12.3: More Pointer Investigation

Try stepping through the "lab12a" program. Watch the variables change in the watch window (you may need to click on the tab marked "locals" to display both the variables).

Make sure that you understand what is happening before you continue. If necessary, ask one of the demonstrators to explain it to you.

You can see that the reference (&) and dereference (`*`) operators do opposite things.

- The reference operator (&) creates a reference (a pointer) from a variable. It take simply takes the address of the memory location the variable is using to store its information in.
- The dereference operator (`*`) uses a reference to find the original variable. It uses the address stored in the pointer variable to find the memory location containing the information.

Syntax: Pointer Reference and Dereference Operators

```
& *
```

The reference operator (&) operates on a variable to create a reference. It does this by obtaining the address of the memory location which is being used to store the information for that variable. For example:

```
    &some_variable
```

is directly equivalent to the address of the variable `some_variable`.

The dereference operator `*` carries out the opposite operation to the reference operator: it takes an address in the form of a pointer and follows it to find the location that it refers to. So, if:

```
    some_pointer = &some_variable;
```

`*some_pointer` is directly equivalent to `some_variable` i.e. the lines

```
    *some_pointer = 20;
    some_variable = 20;
```

do the same thing.

The dereference operator is used when declaring pointer variables, as in:

```
    type *name;
```

For example, the line:

```
    int *some_pointer;
```

Declares a new variable called `some_pointer` which is a pointer to a variable of type `int`

You might have noticed that a pointer variable is declared using the dereference operator like this:

```
int *ref_my_integer;
```

This is saying "I would like to declare a variable with a type such that if you were to dereference it, you would get an `int`". If you dereference the `ref_my_integer` variable, you get an integer.

12.4 Using Pointers to Pass Parameters by Reference

The reason we started looking at pointers was to change the way parameters are passed into functions. The next thing you will do is to change the simple program "lab12" you met at the beginning of this lab so that the `change` function can alter the value of the argument variable in the `main` function.

This can be done by changing what we pass into the `change` function. Instead of passing the value of the variable `j` from the `main` function, we could pass a reference to `j` using the reference operator `&`. The `change` function should then be adjusted to accept and use a pointer to an integer instead of an integer.

This technique of passing a pointer rather than a value to a function is called *passing by reference*.

Exercise 12.4: Changing the “lab12” Program to Pass by Reference

Change the “lab12” program to pass the parameter into the change function by reference, rather than by value. You will have to change:

- The way in which the function is called. You will have to use the reference operator to create pointer.
- The parameter declaration part of the change function head. At the moment the change function accepts a single integer parameter. You should change this so that it accepts a pointer to an integer instead.
- The assignment statement inside the change function to use the dereference operator. By using the dereference operator you will affect the value of the j variable in the main function.

Try building and executing your program. You should find that the change function is now able to affect the value of j.

You may have noticed that you have been using the reference operator (&) since laboratory 2 in scanf lines like this:

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

This means that you are passing the scanf function a pointer to the my_integer variable, which allows it to change the value. This is another example of passing by reference.

Exercise 12.5: Using scanf with a Pointer

Alter the change function so that it asks the user what value they wish to set the variable to. It should then set the value of the location that i references using a scanf function call.

Be Careful! Remember that the variable your are using (i) is a pointer. Do you need to use the reference operator when calling scanf? If you don’t understand this, ask one of the demonstrators to explain it to you before continuing.

Exercise 12.6: Passing Multiple Values by Reference

Because a pointer is just another parameter, a function can accept a number of pointers as parameters, as well as other values. There are no limitations on this.

Change the “lab12” program so that the change function accepts two (or more!) pointers as parameters, and sets the values of them using scanf function calls.

12.5 Using Pointers with Structures

One of the most powerful ways to use pointers is in combination with structures, like those you met in lab 11. By passing a pointer to a function by reference it allows that function to change any of the members of that structure.

Suppose we have a student_type structure type just like the ones you looked at in lab 11:

```
typedef struct long_student_type
{
    char family_name[40];
    char given_name[40];
    int year_of_birth;
    int course_code;
} student_type;
```

We could declare a function to accept one parameter, which was a pointer to a structure of this type:

```
void change_student(student_type *student);
```

So that parameter `student` is a pointer to a structure of type `student_type`.

So how would the `change_student` function set the values in the structure? It **can't** do it like this:

```
student.year_of_birth = 1986;
```

because `student` is a *pointer* to structure **not** a structure. We need to use the dereference operator to follow the reference to find the actual `student_type` structure. It has to be done like this:

```
(*student).year_of_birth = 1986;
```

This dereferences the pointer `student` to find the structure, then uses the dot operator (`.`) to access the member variable `year_of_birth`. The brackets are necessary because otherwise the '`.`' takes precedence over the '`*`' and C thinks that the star refers to both the parts before and after the '`.`'.

Writing `(*student).`*`member`* is very cumbersome and difficult to read. To alleviate this problem, C includes a special operator for accessing the members of a structure to which you have a pointer. It looks like this: `->`. It is a combination of a minus sign, `-`, and a greater than sign, `>`, and is meant to look like an arrow. The line:

```
student->year_of_birth = 1986;
```

is exactly equivalent to:

```
(*student).year_of_birth = 1986;
```

The first notation is easier to read and is almost always used in place of the second.

Syntax: Pointer to Structure Member Access Operator

`->`

The `->` is used for accessing the member variables of a structure to which you have a pointer. As in:

`struct_pointer``->`*`member`*`;`

For example:

```
my_struct->some_value
```

Accesses the member variable `some_value` from the structure which is referenced by the pointer variable `my_struct`. This is directly equivalent to:

```
(*my_struct).some_value
```

Exercise 12.7: Using Pointers with Structures

Open the program that you were working on in lab 11 as part of your option. This will be “graphics3” if you are were working on the graphics option, or “music3” if you were working on the music option.

Both these programs have a loop in the `main` function which accepts values from the user to put into a structure in the array. Create a separate function which accepts a pointer to a structure. It should be this function which prompts the user for values and places them in the structure.

The function prototype for those doing the graphics option might look something like this:

```
    void get_line_values(line_type *new_line);
```

For those doing the music option it might look something like this:

```
    void get_note_values(note_type *new_note);
```

You will have to use the reference operator when you call the function, and the `->` operator.

Exercise 12.8: Improving Program Efficiency Using Pointers

When you pass a structure by value all of the member variables are copied into the parameter. This can make your programs inefficient, both in terms of their speed, and the amount of memory they use.

Change the `draw_line` (graphics option) or the `play_note` (music option) function so that the structure is passed by reference rather than by value.

12.6 Summary

Now that you have finished this laboratory you should know that the usual way in which functions are called involves a copy of the values of the arguments into the function parameters and is called *passing by value*.

You should understand what a pointer is and be able to use the reference (&) and dereference (`*`) operators to work with pointers. The technique of using pointers as parameters to functions is called *passing by reference*, you should know how to write and call functions which use this technique.

In the final part of the lab you used pointers with structures. You should have met the `->` operator which is designed for working with structures. You should also have altered one of the programs you created in lab 11 so that it passed structures by reference.

Laboratory 13

Arrays and Strings Revisited and Allocating Memory

13.1 Overview

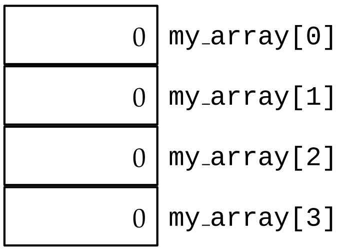
In lab 12 you were introduced to pointers. This lab begins by examining arrays, showing that there is a strong relationship between arrays and pointers. An array is essentially a pointer to a block of memory locations which the C compiler has set aside. Array and pointer notation can therefore be used interchangeably.

A pointer on its own is not very useful, as it has no memory to refer to. This lab introduces the `malloc` function which you can use to allocate blocks of memory whilst your program is running. Any memory you allocate with `malloc` must be de-allocated before your program ends using the `free` function. C provides an operator called `sizeof` which helps you determine how much memory is required by different variable types.

Finally the lab looks at allocating memory for structures, and arrays of structures. You will alter the graphics or music program you have been working on to use the memory allocation functions `malloc` and `free`.

13.2 More About Arrays

Lab 6 introduced you to arrays. You may remember diagrams of arrays with boxes representing the elements of the array. Like this:



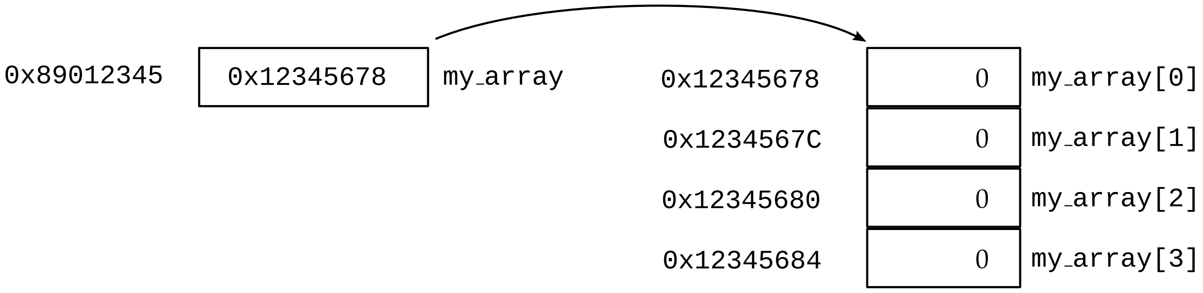
The boxes in array diagrams like this one represent memory locations in just the same as the boxes in the diagrams representing pointer/variable relationships in lab 12. These diagrams simply the way in which arrays work. An array actually has two parts to it:

- A block of memory locations in which the information for each element is stored;
- The array variable is actually a pointer which references the first memory location in the block.

An array that is declared as:

```
int my_array[4];
```

Can be drawn diagrammatically as:



The variable name my_array is actually a pointer, this is shown on the left-hand side of the diagram. This pointer references the first location of a block of four integer-sized memory locations. The block of memory locations is shown on the right-hand side of the diagram.

When you use the square brackets ([ ]) after an array name it is very similar to using the dereference operator (*). In fact, writing:

```
my_array[0] = 5;
```

is exactly the same as writing:

```
*my_array = 5;
```

Accessing any element other than the first element in an array requires placing a non-zero number in the square brackets. The pointer equivalent to this is to add a number to the pointer. Like this:

```
*(my_array+3) = 20;
```

Which is exactly the same as writing:

```
my_array[3] = 20;
```

Using mathematical operations, such as addition, on pointers is called *pointer arithmetic* and must be used carefully. Pointer arithmetic changes what the pointer is referencing. There is nothing in C to stop you changing a pointer so that it does not reference a valid memory location. This mistake, often called a *floating pointer*, is very difficult to spot and generally causes your program to crash completely.

In general the array notation, using the square bracketed subscript, is much clearer and easier to read than pointer arithmetic. **Use pointer arithmetic very carefully**, try and avoid it if possible, there are usually clearer ways to write the same thing.

The fact that arrays are just 'pointers with some memory attached' is often reflected in functions which accept arrays as parameters. In lab 6 you used a mean function to calculate the mean of an array of values. The prototype for this function was:

```
double mean(double values[], int num_values);
```

This prototype could equally have been written using pointer notation, as:

```
double mean(double *values, int num_values);
```

The two prototypes are effectively identical. The body of the mean function used the square bracket notation for accessing the values parameter as an array. Like this:

```
mean += values[current_value];
```

Even with the function prototype that declares the `values` parameter as a pointer, this line **does not** need to be changed. Just as an array variable is actually a pointer, it is perfectly legitimate to treat a pointer as an array.

If you look at other people's code, including standard C functions, you will find that the pointer notation for array parameters to functions is very common.

Exercise 13.1: Investigating Arrays as Pointers

When you pass an array to a function, you are passing a pointer. You can think of arrays as always being passed by reference. The project "lab13" contains a skeleton for a program which might use an array. Copy this project nd open it in the IDE.

Based on the very simple programs you used in laboratory 12, create a program which demonstrates that arrays are effectively always passed by reference. You can make the program as simple or complicated as you like.

13.3 More About Strings

In lab 6 you saw that character strings are also arrays: arrays of type `char`. Each element is a character and the last element in the string is always the null character, `'\0'`. Since string variables are arrays, they are also pointers. You will often find that functions that operate on strings expect a parameter of type `char *` i.e. a character pointer. For example, the prototype for the standard C function `strlen` is as follows:

```
int strlen(char* str);
```

The `strlen` function calculates the length the string it was passed by finding the null-termination character. You wrote a function to do this in lab 6.

You can assign strings to pointers that are specified in your program, like this:

```
char *message;
message = "It's a long way to Tokyo\n";
```

You could then display the message using a `printf` statement, like this:

```
printf(message);
```

You can access individual characters of the message using the array subscript square brackets, for example:

```
message[5]
```

would be equivalent to the character `'a'`. However, you **can not** change the string by writing to it. The string is pre-specified so it may be in memory which you are not allowed to write to. For example, you **should not** do the following:

```
message[5] = '1';
```

If you want to be able to change the message, you must put its contents into an array, like this:

```
char message[] = {"It's a long way to Tokyo\n"};
```

This will allocate writeable memory especially for the message and copy the message into it. You can now change its contents.

Exercise 13.2: Investigating Strings as Pointers

Open the "lab6b" project. This project does some operations on a word that you enter. It should count the length of the word and then reverse the characters in the word and display it. Currently, the reversal operation takes place in the `main` function.

- create a function called `reverse_string`, which reverses the string that you give it;
- for both the `reverse_string` function and the `string_length` function change the prototypes so that the string parameters are declared as pointers, rather than arrays.

So, your prototypes should be:

```
int string_length(char *string);
```

and:

```
void reverse_string(char *string);
```

You should find that declaring these parameters as pointers rather than arrays makes no difference to the operation of your program. However, when handling strings it is far more common to use the pointer notation than the array notation.

13.4 Allocating Memory

In the last two sections you found that an array is a pointer to a block of memory locations. The block of memory is allocated by the C compiler when your program is running. The C compiler then sets up the array pointer so that you can use it. There are a number of circumstances under which you might not want the C compiler to allocate the memory for you. You might want to do it yourself. The most common of these reasons are:

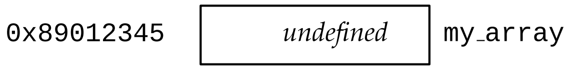
- You don't know how big an array you need until the program is running. When you specify an array size, it must be fixed when you write the program. If you allocate the memory yourself you can control how big an array is whilst the program is running, perhaps in response to user input.
- The amount of memory that can be set aside by the compiler for an array is limited. Sometimes you may wish to allocate a large block of memory to use as an array, for example to manipulate a bitmap image or a sound sample. Allocating memory whilst the program is running is more flexible, and the blocks of memory which can be allocated are bigger.

When the compiler allocates array memory for you, you must have specified the size when you wrote the program. This size is fixed, so this is called *static memory allocation*. Allocating memory whilst the program is running is more flexible. This is called *dynamic memory allocation* because the amount of memory that is allocated can be changed every time the program is run.

The standard C library provides a number of functions for working with memory. The normal way to allocate a block of memory is to use the `malloc` function (which is an abbreviation for memory allocation). When you are finished with the block of memory, your program **must** hand it back, using the `free` function.

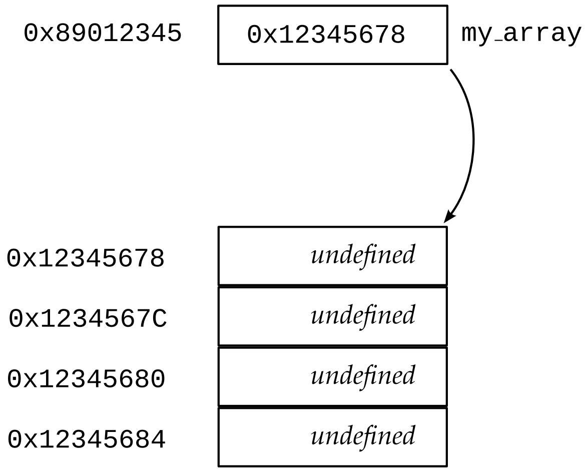
To demonstrate the use of `malloc` and `free` we will look at a very simple program snippet. The bit of code we will look at simply allocates some memory, we imagine that it uses it, and then the memory is freed. When the memory is allocated we will use the pointer my_array to reference it. When the program starts the pointer does not yet have a meaningful value.

```
int main(void)
{
   int *my_array;

   my_array = malloc(size);

   ...

   free(my_array);

   return 0;
}
```



The call to malloc allocates a block of memory of a specified size. The code snippet has the word *size* where you would specify the size of the block of memory you require. We will look at how to specify memory sizes in a moment. Imagine that we specified that we wanted room for four integers.

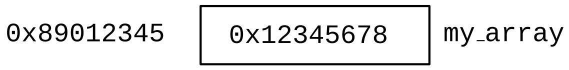
```
int main(void)
{
   int *my_array;

   my_array = malloc(size);

   ...

   free(my_array);

   return 0;
}
```



The diagram shows that we have allocated a block of memory, and that the pointer my_array references it. Notice that the block of memory has no variable name. It is not a variable; the only way we have to access this memory is using the my_array pointer.

When we have finished using the memory we must tell the system that we no longer need it. This allows the memory to be reused. If we did not do this the system would use up memory every time we ran the program. Allocating memory and forgetting to free it is called a *memory leak*.

```
int main(void)
{
   int *my_array;

   my_array = malloc(size);

   ...

   free(my_array);

   return 0;
}
```



Notice that when you free a block of memory, the pointer that you used to access the memory (in this case my_array) does not change. However, the memory that this pointer references no longer belongs to the program. **You must not attempt to access memory you have freed**.

A call to the malloc function may not always succeed. You are asking the system for a block of memory. If it does not have enough memory to give you the malloc function will fail. If it does, the malloc function returns the special pointer value NULL. If a pointer is NULL it indicates that the pointer is invalid. If you try to use a NULL pointer your program will crash. It is good practice to always check to see if malloc returned a NULL or a valid pointer.

You have just seen how the malloc and free functions are used to allocate and free up blocks of memory. A call to the malloc function takes one argument: the size of the block you wish to allocate. C has a special operator called sizeof to help you calculate how much memory different things take up. The sizeof operator looks like a function call but is in fact built in to the C language. To use sizeof you specify the type name for a variable in brackets after the keyword sizeof. For example, if you want to know the size of an integer, you would write:

```
sizeof(int)
```

The size of four integers would be:

```
sizeof(int) * 4
```

Therefore, to allocate space for four integers using malloc you would write:

```
my_array = malloc(sizeof(int) * 4);
```

The code snippet we have been looking at is in the “lab13a” project. Copy this project and open it in the IDE.

Exercise 13.3: Investigating Memory Allocation

Have a look through the source code for the “lab13a” project. Do you understand what it is doing? Once you have tried executing it, try stepping through it and watching the variables. If you cannot understand what is happening, ask one of the demonstrators to explain it to you.

Function Reference: malloc — Memory Allocation

```
pointer = malloc(size);
```

e.g.

```
    some_pointer = malloc(sizeof(int));
    real_array = malloc(sizeof(double) * 20);
    string = malloc(sizeof(char) * 256);
```

The malloc function allocates a block of memory of the requested size and returns a pointer to the memory. Once the memory is not longer required, it must be deallocated using the free function.

Should the malloc function be unable to grant the request for memory, for example if there is insufficient remaining free memory on the system, the function will return the NULL pointer.

The malloc function is defined in stdlib.h

Syntax: The `sizeof` Operator

`sizeof(`*`type`*`)`

The `sizeof` operator calculates the amount of memory a type requires for storage. For example, the following:

```
    sizeof(int)
```

calculates how much size is required by one integer variable. `sizeof` evaluates to a number, so it may be combined with mathematic operations such as multiplication. For example:

```
    1024 * sizeof(char)
```

The most common use for the `sizeof` operator is when using the `malloc` memory allocation function.

Function Reference: `free` — Memory Deallocation

`free(`*`pointer`*`);`

The `free` function deallocates memory that was previously allocated using the `malloc` function. The `free` function takes one argument: a pointer to the block of memory to be deallocated.

- The `free` function should always be used to deallocate memory as soon as it is no longer required.
- You should **never** attempt to free a block of memory twice, or to free statically allocated memory such as arrays.
- Once a block of memory referenced by a pointer is deallocated, the pointer should **not** be used again, unless it is assigned a new value.

The `free` function is defined in `stdlib.h`

Exercise 13.4: Arrays and Memory Allocation

The "lab6" program allowed the user to enter data which the program placed in an array. Two statistical operations are then carried out: mean and standard deviation. At the moment, the memory in the array is statically allocated. The number of data items the user can entered is therefore fixed when the program is compiled.

Change the "lab6" program so that the user is asked how many data items they wish to enter at the beginning. You should then use `malloc` to dynamically allocate the data array. You can put a maximum limit of the number that the user enters if you want.

Remember to free the memory before the program ends.

Exercise 13.5: Strings and Memory Allocation

In a similar way to Exercise 13.4, change the "lab6b" program so that the user can decide how long a word they wish to be able to enter. You will need to be careful in how the `scanf` function call is constructed.

13.5 Dynamic Memory Allocation for Structures

Dynamically allocating memory for structures is very similar to allocating memory for arrays. The structure name is used with the `sizeof` operator to calculate the size of the block of memory required. For example, to allocate a block of memory for one `student_type` structure you would call `malloc` in the following way:

```
student = malloc(sizeof(student_type));
```

Allocating memory for an array of structures is similar. For example:

```
student_array = malloc(sizeof(student_type) * 20);
```

As you can see, this follows the same pattern as allocating memory for standard C types such as `int` and `double`.

Exercise 13.6: Structures and Memory Allocation

If you have been following the graphics option, open the "graphics3" project; if you have been following the music option, open the "music3" project. These projects currently use a statically allocated array of structures. Alter the program so that the array memory is dynamically allocated and freed.

Exercise 13.7: Reallocating Memory

If the user of your "graphics2"/"music3" program wanted to a a line/note whilst the program was running, how would you accommodate this (it would require more memory to be allocated). If the user removed a line/note could you free memory up? Have a go at implementing this into your program.

If the user is adding/removing lines/notes continuously allocating and freeing memory can make your program slow. How could you make this process more efficient?

13.6 Summary

This laboratory re-examined arrays and strings to show their close relationship with pointers. You should now know that an array is simple a block of memory, allocated by the compiler, which is referenced by a pointer. Memory that is allocated by the compiler is called *statically allocated* memory.

Now that you have finished this lab you should know how to *dynamically* allocate memory using the `malloc` function call. You should know that all dynamically allocated memory should be deallocated using the `free` function call.

In the last part of the lab you should have used dynamic memory allocation with an array of structures.

Laboratory 14

Creating Buffers

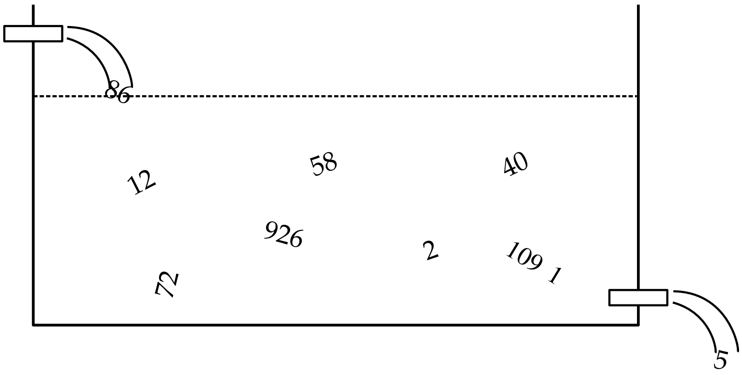
14.1 Overview

This lab uses the C you have learned so far to examine an important real-life application: the use of buffers. Buffers are simply a way of storing information in your program. This lab looks at two different type of buffer and shows you the basic principles of using them.

To begin with this script looks at what buffers are, and why they are useful. Two sorts of buffer are then introduced which differ in the way in which information is extracted from the buffer. At the end of the lab you will see a more sophisticated arrangement for handling buffers which is used a great deal in practice.

14.2 Buffers: Flexible Storage

A buffer is a part of a software system which stores data. In this way it acts a bit like a tank. There is a path for information to flow in, and a path for information to flow out.



The tank is useful in situations where the rate of information coming in is different to rate at which information travels out. A buffer acts as a flexible way to store a small amount of data.

Unlike in the picture of the tank, above, buffers usually store information is some kind of order. This lab is going to examine two types of buffer which differ in the way information is taken *out* of the buffer.

- *Stacks* are buffers where the item of data you get out, is the *last* item of data you put in. The next item of data you get out is the second to last item of data you put in and so on. Just piling objects up into a stack. A stack is also called a *last in first out* buffer, or LIFO.

- *Queues* are buffers where the item of data you get out is the first item of data you put in. The next item of data you get out was the second data item you put in and so on. Just like a queue of people waiting to be served at, say, a bar. A queue is also called a *first in first out* buffer, or FIFO.

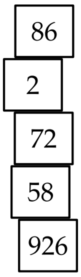
The lab will look at LIFOs first, and then FIFOs.

Buffers are widely used but the most common uses are in situations where you will use them without knowing. LIFO buffers are used by almost every microprocessor system, whether complex like a PC or simple like a washing machine or a microwave. You will learn more about this use of LIFO buffers in some of your other courses. You have already seen the use of a FIFO buffer in lab 5. This keyboard input buffer, which stores key presses before they are collected by function calls such as `scanf` and `getch` is a FIFO buffer. FIFO buffers are used in many other situations and are perhaps the most widely used type of buffer.

All the buffer examples in this lab use integers as the data type. There is no reason why this has to be the case. The data items could be `double`s, `char`s, arrays, structures or pointers.

14.3 Stacks — LIFO Buffers

Imagine you have a collection of blocks, like a child's building blocks. Each block has a number on it, and represents an item of data that you might want to store. A LIFO buffer would be represented by stacking these blocks, one on top of another.

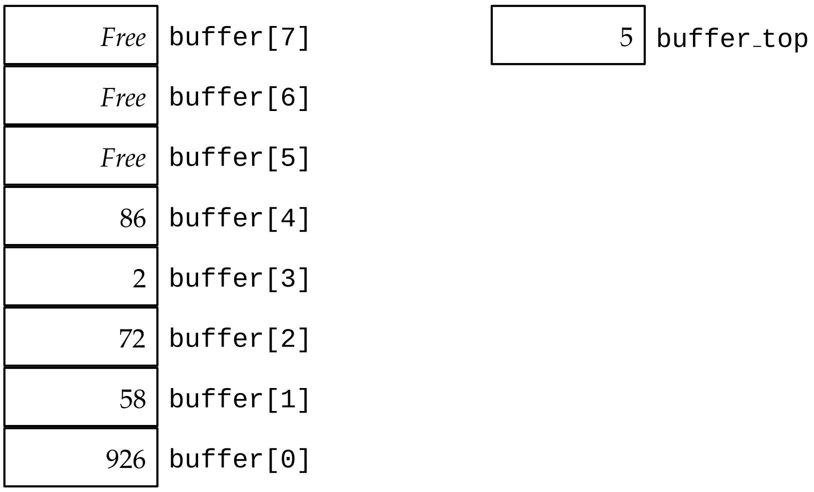


In this diagram, the last block to be added to the stack was number 86. This is next block we may take away. The last block we added to the stack is the first we can take away — this gives the stack its name *last in first out*.

We can create a LIFO buffer using an array. We can use a variable to tell us which element in the array is the next free one. For example, if we create an array of integers called `buffer` to use as a stack we can represent its contents using the box diagrams used in previous labs:

Free	`buffer[7]`
Free	`buffer[6]`
Free	`buffer[5]`
86	`buffer[4]`
2	`buffer[3]`
72	`buffer[2]`
58	`buffer[1]`
926	`buffer[0]`

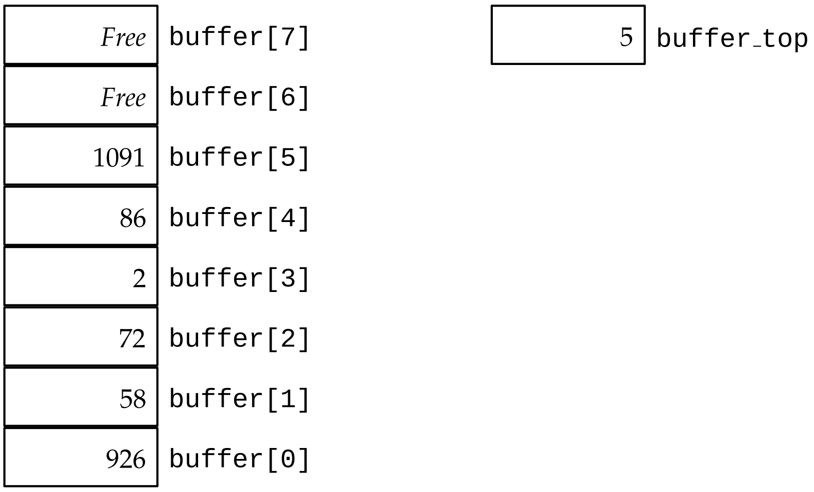
To make the array work as a stack we must keep track of the location of the next free slot. In the situation shown in the diagram above, the next free element in the array is `buffer[5]` so we would store the number 5 in a variable. The next free location is usually called the *top* of the stack. In this case we could have `buffer_top` variable which is set to 5 (see the next diagram).



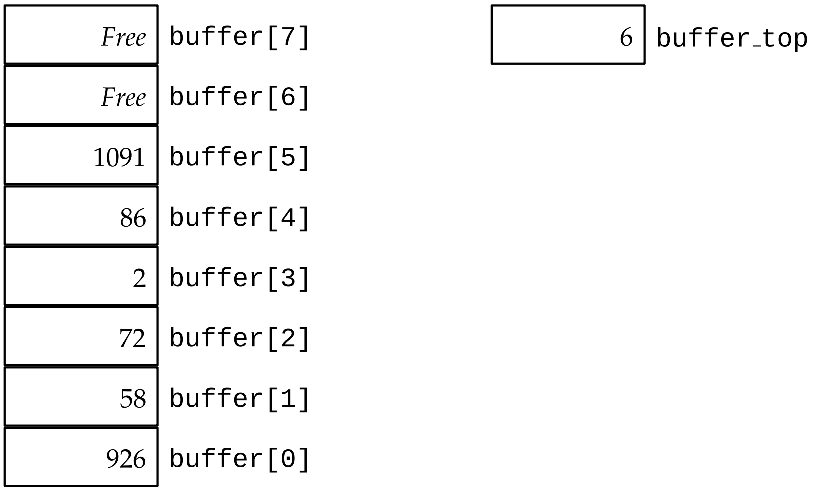
To add a new item of data we must place the data in the next free location in the array. The next slot is buffer[buffer_top]. For example, if we wish to add the number 1091 to the stack, we would execute a line of code like the following:

```
buffer[buffer_top] = 1091;
```

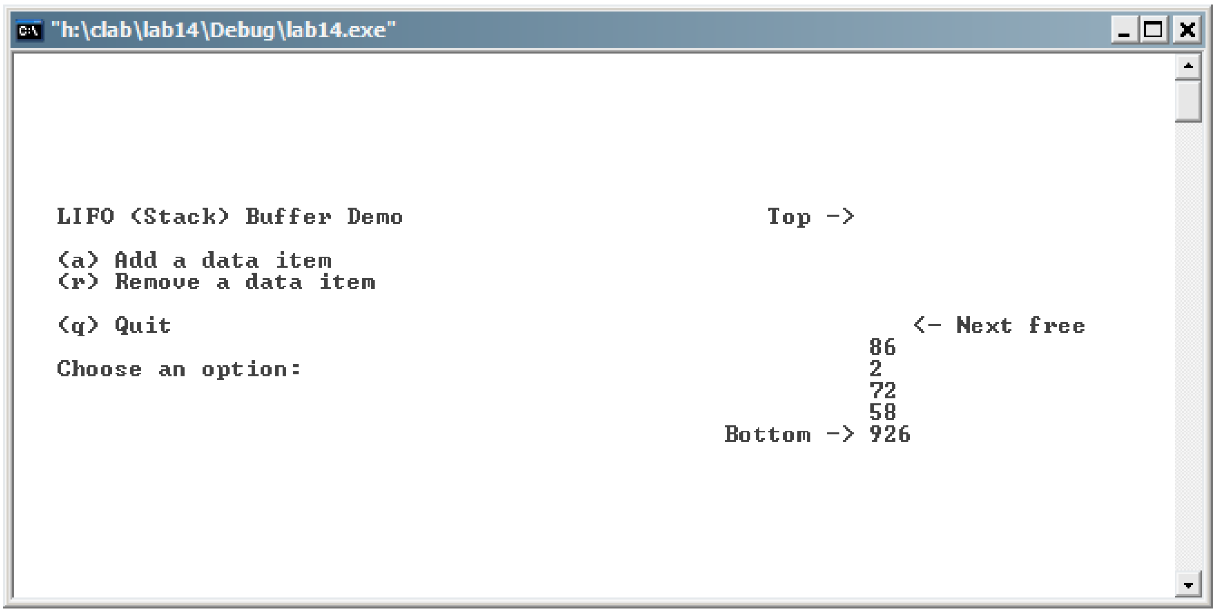
The result of this is shown below:



As we have now used another slot we must update the buffer_top variable:



The “lab14” project implements a stack as an array. When the program is running you have the option of adding or removing data items. The current stack is shown on the left-hand side. A screen shot of the “lab14” program running is shown below:



```
LIFO (Stack) Buffer Demo                         Top ->

(a) Add a data item
(r) Remove a data item

(q) Quit                                                   <- Next free
                                                        86
Choose an option:                                       2
                                                        72
                                                        58
                                              Bottom -> 926
```

Copy the “lab14” project and open it in the IDE. You will find that it is made up of three source files and two header files:

- “lab14.c” contains the `main` function and the `add_item` and `remove_item` functions which respond to user requests to add and remove data items to/from the buffer.
- “display.c” contains the `display_buffer` function which displays the current contents of the stack and the `display_menu` function which displays the short user menu.
- “display.h” contains prototypes for the functions in “display.c”.
- “buffer.c” contains functions for handling the actual buffer. The `buffer_init` initialises the buffer, `buffer_add` does the work of adding an item to the buffer, `buffer_remove` is intended to do the work of removing an item from the buffer. This function is not finished.
- “buffer.h” contains prototypes for the functions in “buffer.c”.

The “lab14” program uses two functions that you have not seen before. These are `clrscr` and `gotoxy`. The `clrscr` function takes no arguments and clears all text from console text window. The cursor is set to the top left-hand side. The `gotoxy` function takes two integer arguments which specify an x and y location in the console text window. When the function is called, the cursor is positioned at the specified location. For example:

```
gotoxy(4, 20);
```

Sets the x location of the cursor to be 4 characters from the left of the screen, and the y location of the cursor to be the 20th line from the top of the screen.

Function Reference: `clrscr` — Clears the Console Text Window

```
void clrscr(void);
```

e.g.

```
    clrscr();
```

The `clrscr` function clears the console text window of all text and sets the cursor location so that the next text to be displayed will appear in the top left-hand corner of the window.

The `clrscr` function is defined in `console_lib.h`.

Function Reference: `gotoxy` — Sets the Console Text Window Cursor Position

```
void gotoxy(int x, int y);
```

e.g.

```
    gotoxy(5, 10);
```

The `gotoxy` function sets the cursor position in the console text window. The cursor is set to the position specified by the function arguments x and y such that the next text to be displayed in the window appears at that location.

The x coordinate specifies the cursor position in character spaces from the left-hand edge of the window. The y coordinate specifies the cursor position in lines from the top of the window. The top left cursor position is (1, 1).

The `gotoxy` function is defined in `console_lib.h`.

Exercise 14.1: Working with LIFO Buffers

Build and execute the “lab14” project. You should find that you can add data to the stack, but not remove it. This is because the buffer handling function `buffer_remove` is not finished.

Read through the source code and make sure that you understand how the program works. If you have problems understanding parts of the program try stepping though them, by setting a breakpoint if necessary. When you feel that you understand the way that the program works, complete the `buffer_remove` function so that data items can be removed from the stack.

14.4 Buffer Problems: Overflow and Underflow

When you first looked at arrays in lab 6 it was mentioned that the array subscript (the bit in square brackets after the array name) could be any value. When using arrays you must be careful not to try and access a value which is not really in the array.

In the context of stacks, there are two situations which could potentially break this rule:

- if you attempt to add more items to the stack than the buffer array can hold, this is called *overflow*;

- if you attempt to remove an item from the stack when the stack is empty, this is called *underflow*.

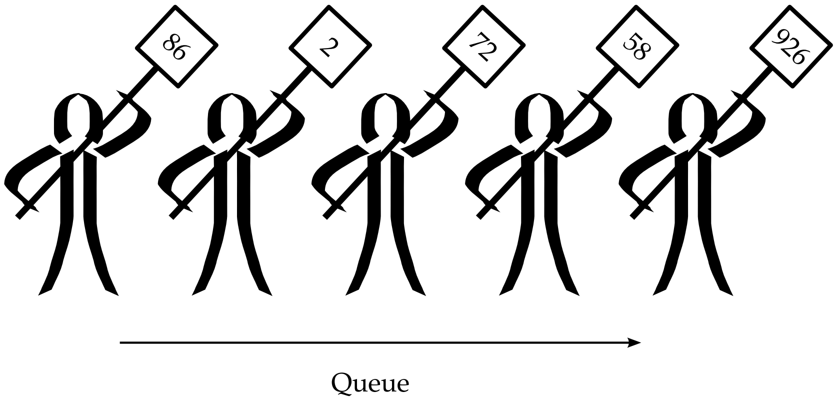
Both of these situations could cause a program to malfunction or crash seriously.

Exercise 14.2: Handling Overflow and Underflow Conditions

The "lab14" program currently has no protection against overflow or underflow. Make sure you trap these conditions so that they can never happen. Warn the user if they attempt to add more data items than the stack will hold, or if they try to remove data items when the stack is empty.

14.5 Queues — FIFO Buffers

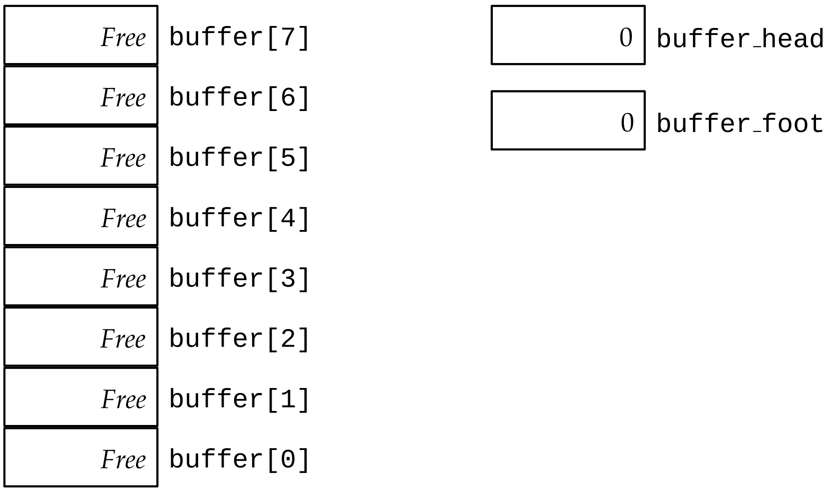
In section 14.3 of this lab the analogy of a set of building blocks was used to represent a LIFO buffer. Each building block represented a data item. You can imagine a FIFO buffer as a queue of people waiting to be served at a bar. Each carries a sign with a number on it. In this case, the people in the queue with their signs represent the data items.



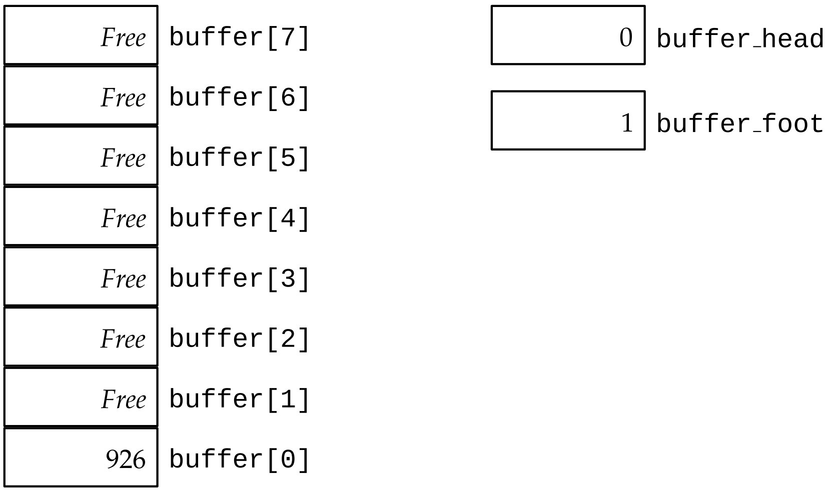
The first person to join the queue at the bar will be the first person to be served (this is a perfect world!). All of the people in the queue will be served in order as they reach the bar. The first into the the queue will be the first out of it — this gives the queue its name a *first in first out* buffer.

A FIFO buffer can be based on an array in much the same way as a LIFO buffer. Except now we must keep track of two points in the array, the head of the queue and the foot (end) of the queue. To understand how a FIFO is managed in an array we will use a box diagram again. The array name `buffer` is used again. The `buffer_head` variable indicates the element containing the next data element to take out of the queue (i.e. the next person to be served). The `buffer_foot` variable indicates the next free slot in the array.

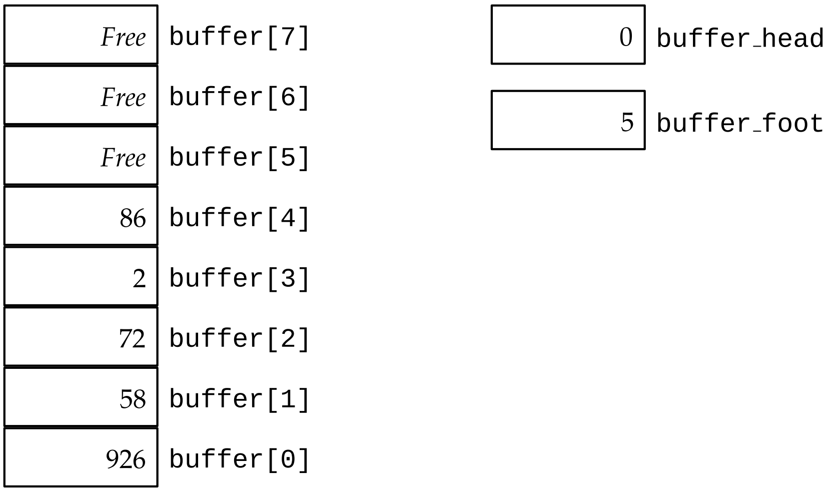
To begin with, the buffer is empty:



Notice that both buffer_head and buffer_foot are zero. This indicates that there is no data between the head and foot of the queue, i.e. the queue is empty. The next diagram shows that the data item 926 has been added to the queue:



The buffer_foot variable has changed to indicate the next free element in the array. Notice that the buffer_head has not changed because we have not removed any data items from the queue. The next diagram shows that we have added the data items 58, 72, 2 and 86 to the queue.



These diagrams have shown what happens when data items are added to the queue, what about removing items from the queue? Providing the queue is not empty, the buffer_head variable tells us where the next data item is that we may remove. In this case, `buffer_head` is 0, so we remove the data item 926:

Free	`buffer[7]`
Free	`buffer[6]`
Free	`buffer[5]`
86	`buffer[4]`
2	`buffer[3]`
72	`buffer[2]`
58	`buffer[1]`
Free	`buffer[0]`

1	`buffer_head`
5	`buffer_foot`

Notice that the `buffer_head` variable has changed to indicate where the new head of the queue is.

Exercise 14.3: Implementing a FIFO Buffer

Change the "lab14" example so that it uses a queue to store data instead of a stack. You will need to add a variable to indicate the foot or bottom of the queue. You will also need to change the `buffer_init`, `buffer_add` and `buffer_remove` functions.

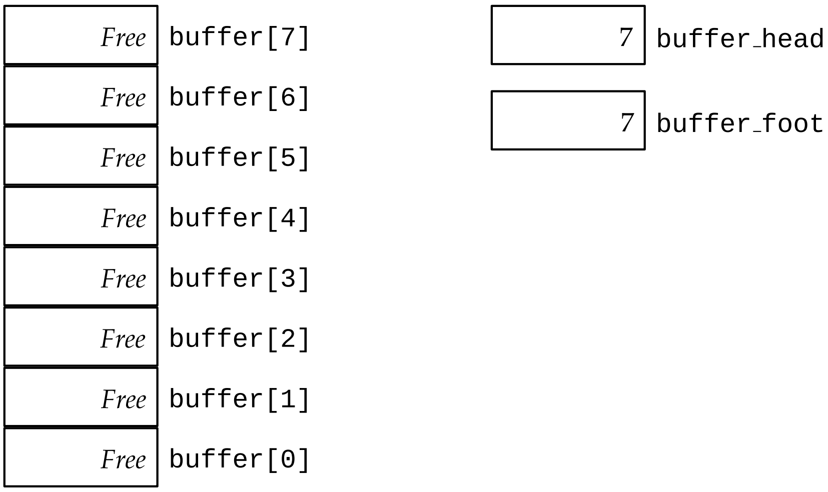
14.6 Managing FIFOs as Circular Buffers

There is a major problem with the FIFO buffer implemented in an array as discussed in the previous section. The problem occurs when the buffer has been filled and then emptied. For example, say that we add seven items to the buffer:

Free	`buffer[7]`
12	`buffer[6]`
1091	`buffer[5]`
86	`buffer[4]`
2	`buffer[3]`
72	`buffer[2]`
58	`buffer[1]`
926	`buffer[0]`

0	`buffer_head`
7	`buffer_foot`

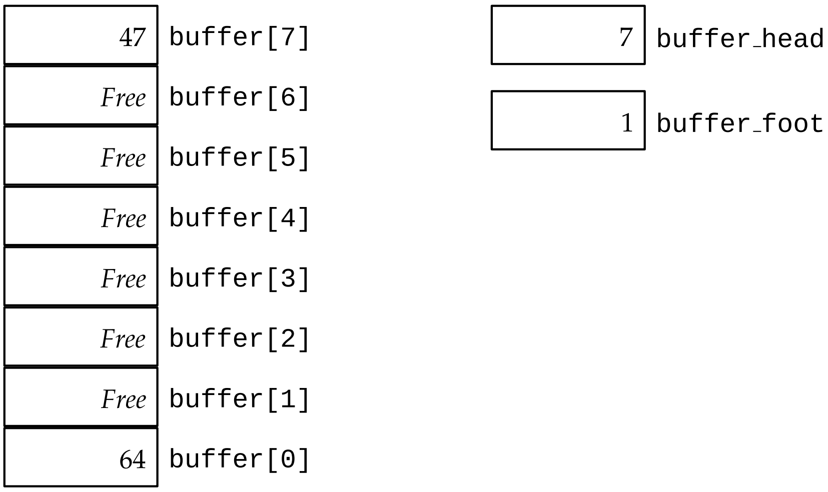
Now we will remove all of those seven items:



Free	buffer[7]	7	buffer_head
Free	buffer[6]	7	buffer_foot
Free	buffer[5]		
Free	buffer[4]		
Free	buffer[3]		
Free	buffer[2]		
Free	buffer[1]		
Free	buffer[0]		

Notice that the array is completely empty and both the buffer_head and buffer_foot variables are 7. Despite the buffer being totally empty, how many more data items can we add to the queue? It is likely that we cannot add more than one data item without the buffer overflowing.

The way to solve this problem is to imagine the array as if the top element (buffer[7]) were joined to the bottom element (buffer[0]) to form a circle. We may then continue adding data items to the queue. This is called a *circular buffer*. To continue the example from above, once we have added seven items to the buffer, and removed them, suppose we add another two data items: 47 and 64. This is shown in the diagram below:



47	buffer[7]	7	buffer_head
Free	buffer[6]	1	buffer_foot
Free	buffer[5]		
Free	buffer[4]		
Free	buffer[3]		
Free	buffer[2]		
Free	buffer[1]		
64	buffer[0]		

Notice that the buffer_foot variable has a lower value than the buffer_head variable. This is a sign that we are treating the buffer as circular, and that the queue data has *wrapped round* from one end of the array to the other.

Exercise 14.4: Making the FIFO Example Circular

Change the "lab14" program so that queue is handled as a circular buffer. You will need to be careful how you detect and trap the buffer overflow and underflow conditions. If you do not understand how to do this, ask one of the demonstrators for help.

14.7 Summary

Now that you have finished this lab, you should know what a buffer is and how one might be constructed using an array. You should know about two important types of buffer stacks (LIFOs) and queues (FIFOs).

You should understand two problems that can arise when using buffers: overflow, when the buffer is full; and underflow, when the buffer is empty. In the case of queues, you should also understand how to manage these as circular buffers, making best use of the storage space available.

Laboratory 15

Lists and Queues (1)

15.1 Overview

This laboratory starts by looking at arrays of pointers as a way storing information. This approach gives a lot of flexibility without taking up too much memory.

The next section of the lab begins by examining some of the problems with storing large amounts of data in an array. The solution to this is something called a *linked list*. This is a useful way to store large amounts of data very efficiently.

This course will introduce you to two kinds of linked list, the first is presented in this lab, you will meet the second kind in lab 16.

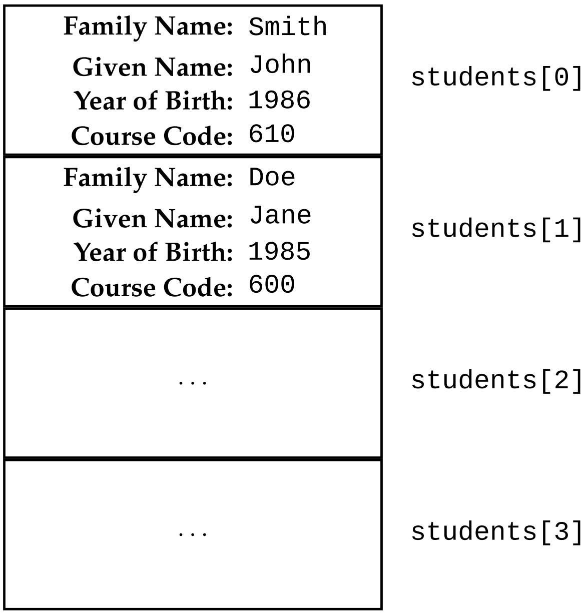
15.2 Storing Arrays of Pointers

Lab 14 introduced buffers which stored integer data items in an array. If you wanted to store more complicated information, student details for example, you could create an array of structures.

For example, say that we create an array of four `student_type` structures of the kind that we used in lab 11.

```
student_type students[4];
```

In memory, this array looks something like this:

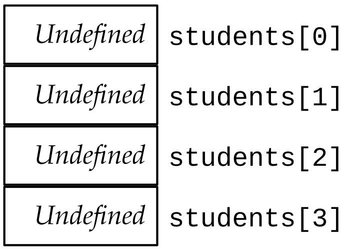


An array of structures uses enough memory to be able to store all the data items, for each of the structures that form the array. So an array of 100 structures is 100 times the size of a single structure in memory. If the buffer is empty this arrangement is exceedingly wasteful of memory.

A way to solve this problem is to create an array of pointers and manage the array just like a LIFO buffer — use a variable to tell us how much of the array is being used and how much is free. The pointers in the used section would all reference structures. We would declare an array of pointers like this:

```
student_type *students[4];
```

All this does is create an array of pointers which, to begin with, are undefined:

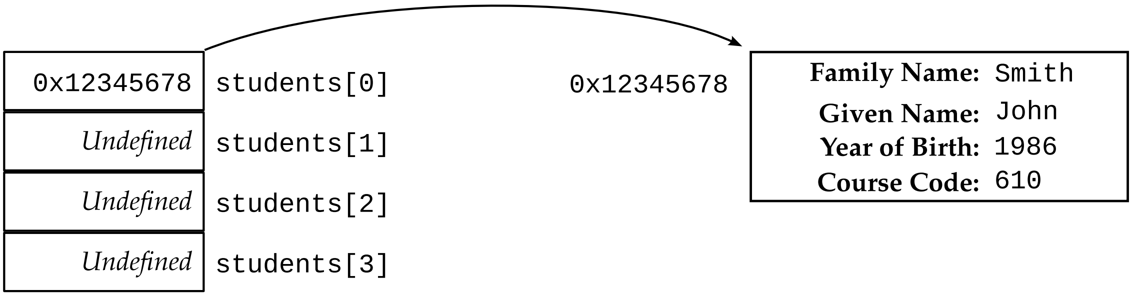


We would record the number of students that the arrays holds in a variable. Initially this variable would be set to zero. A pointer does not use very much memory, so this array is considerably smaller than the array of structures.

When we want to add a student, we allocate a new structure using `malloc` and save a pointer to it in the array. For example:

```
student_type *new_student;

new_student = malloc(sizeof(student_type));
students[0] = new_student;
```

This sets the first pointer in the array to reference the new structure:



The variable that keeps track of how many students there are in the array must also be incremented.

As each element of the array is now a pointer to a structure, we must use the `->` operator to access the structure members. Like this:

```
students[0]->year_of_birth = 1986;
```

Removing a student from the end of the list is just like removing an item from a LIFO buffer. The only difference is that we must use the `free` function to deallocate the memory that was being used by the structure we have just removed.

Exercise 15.1: Using Arrays of Pointers

If you have been following the graphics option, open the “graphics3” project; if you have been following the music option, open the “music3” project. In lab 13 you changed the program so that the array of structures was allocated dynamically and deallocated at the end of the program.

Replace the dynamically allocated array of structures with a statically allocated array of pointers to structures. This will make it easier for you to allow the user to add and remove lines/notes. You will need to keep track of how many of the elements of the array are in use. You might like to use the LIFO buffer code from lab 14 as inspiration for this.

Use dynamic memory allocation for the structures themselves, remember to free any memory that you allocate.

15.3 The Limitations of Arrays

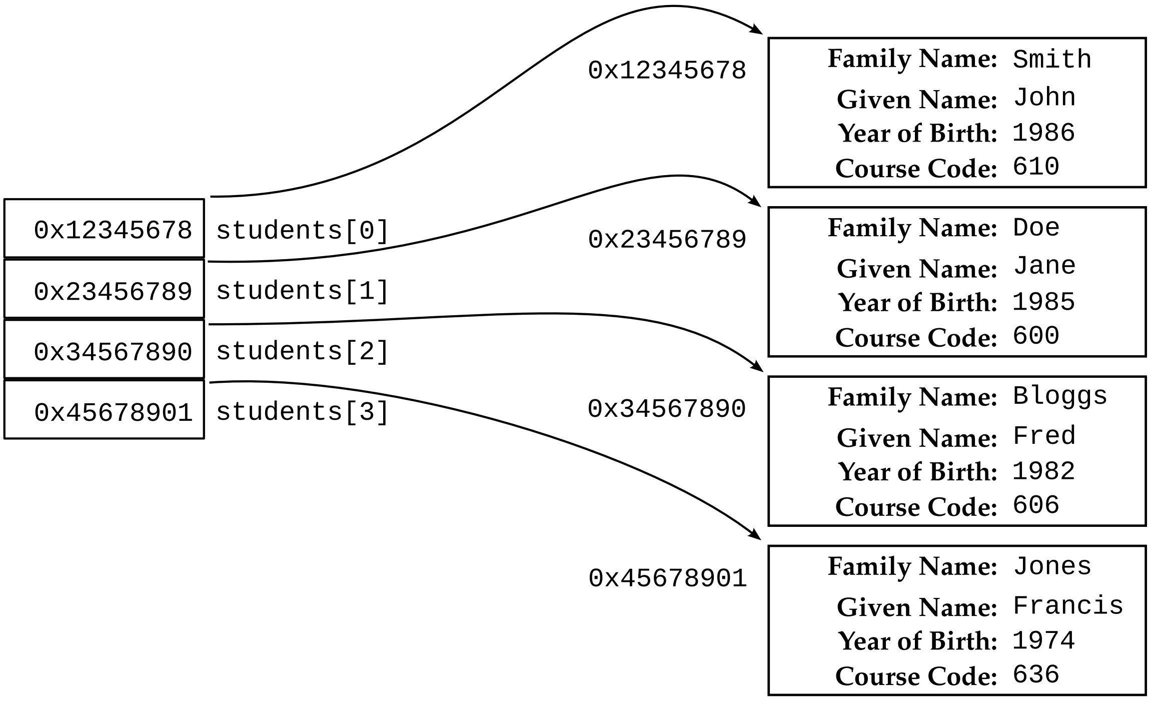
Only storing pointers in an array, rather than the structures themselves, is a vast improvement. However, this arrangement still has a number of limitations. The largest limitation is that adding or removing a structure to/from anywhere in the list other than the top is very difficult. It is possible, but the process is inefficient.

Imagine a situation where a list using this technique is needed to store anywhere between a small and large number of structures. As the array of pointers is statically allocated, if only a small number of its entries are being used this is very wasteful. If we make the array of pointers small we must dynamically reallocate it when it is full to prevent overflow. This is very inefficient.

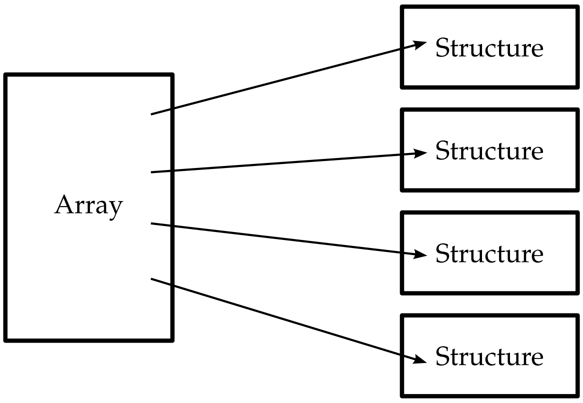
The rest of this lab introduces linked lists, which get round all of these problems.

15.4 Introducing Linked Lists

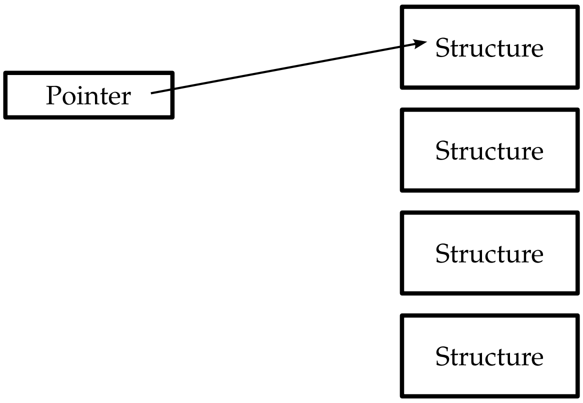
To begin with, consider the example of an array of four student structure pointers that was introduced above. Imagine that each of the four pointers in the array each referenced a structure:



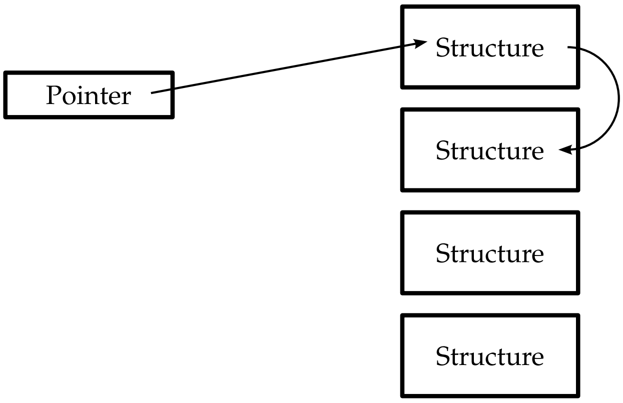
In this situation we have a single list (the array) which tells us where to find each of the structures. We could simplify the diagram and show it like this:



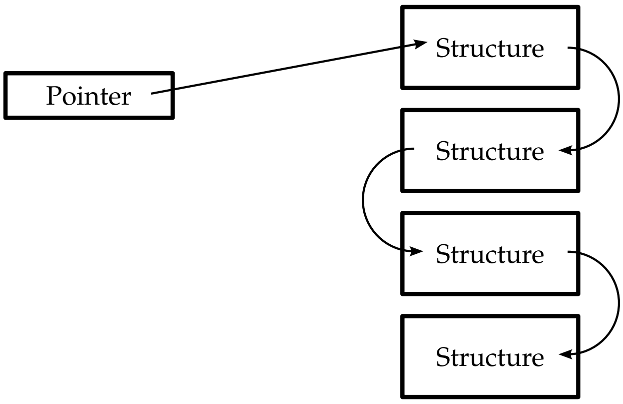
Rather than have an array, which has limited size, we could simply keep one pointer, which only ever references the first structure. Like this:



We could arrange for the first structure to contain a pointer which references the second structure:

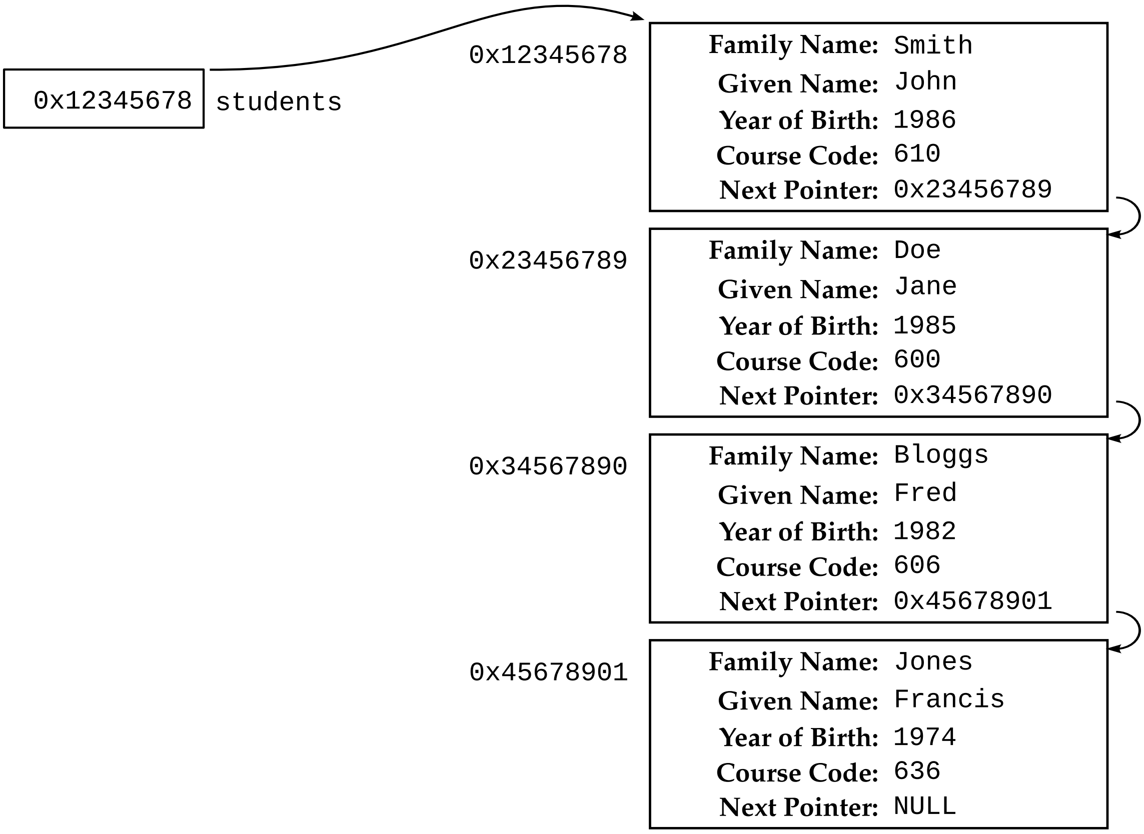


This process could then continue, so that each structure contains a pointer which references the next structure in the list:



It is then possible to find any item in the list simply by starting at the beginning and following each of the references in turn. The list can now grow to any size. An empty list consumes as much space as one pointer. This efficient way of storing information is known as a *linked list* because the list of structures is linked together by references.

A linked list is usually created by adding an extra member variable to the structure. The member variable is a pointer which references the next structure in the list:



0x12345678
Family Name: Smith
Given Name: John
Year of Birth: 1986
Course Code: 610
Next Pointer: 0x23456789

0x23456789
Family Name: Doe
Given Name: Jane
Year of Birth: 1985
Course Code: 600
Next Pointer: 0x34567890

0x34567890
Family Name: Bloggs
Given Name: Fred
Year of Birth: 1982
Course Code: 606
Next Pointer: 0x45678901

0x45678901
Family Name: Jones
Given Name: Francis
Year of Birth: 1974
Course Code: 636
Next Pointer: NULL

The last structure in the list does not need to reference anything, but it does need to indicate that it is the last structure in the list. To do this, we may sure that the 'Next Pointer' of the last structure in the list has the special value `NULL`. As you may remember from lab 13, the value `NULL` indicates that a pointer is not valid.

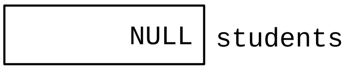
The pointer that references the first structure in the list is usually known as the *head pointer* or the *first pointer*. The member variable in each structure which references the next structure in the list is usually known as the *next pointer*.

15.5 Working with Linked Lists

When a program using a linked list starts, it will make sure that the 'first pointer' is set to `NULL`:

```
students = NULL;
```

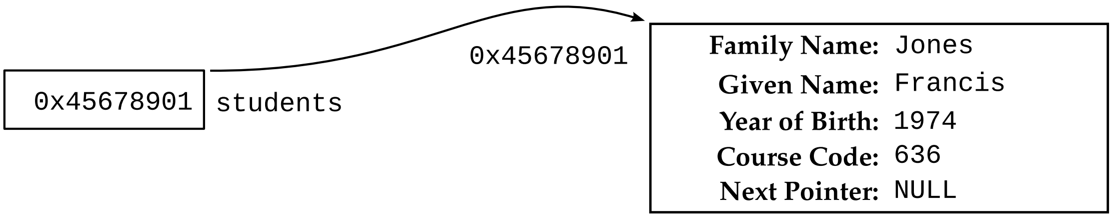
This indicates that the list is empty:



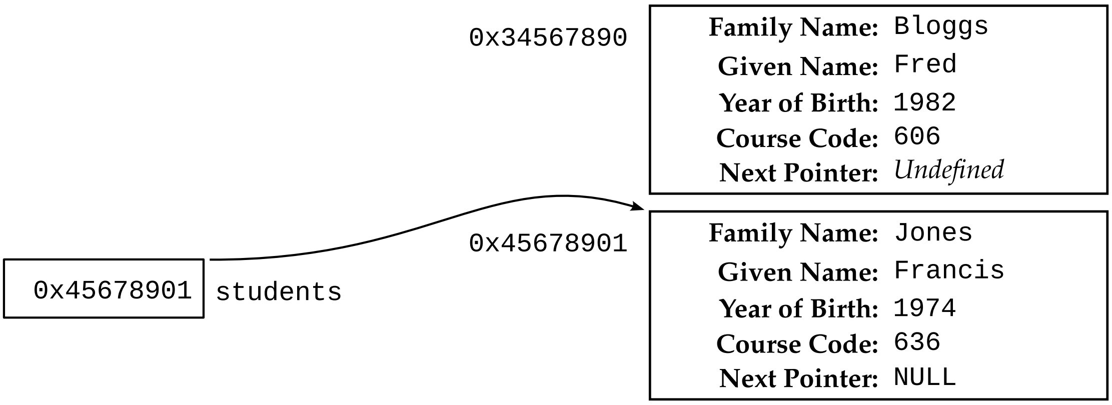
To add the first item to the list we must go through the following steps:

- allocate the memory for the new item;
- set appropriate member variables in the new item;
- set the 'next pointer' of the new item to `NULL`;
- set the 'first pointer' to reference the new item.

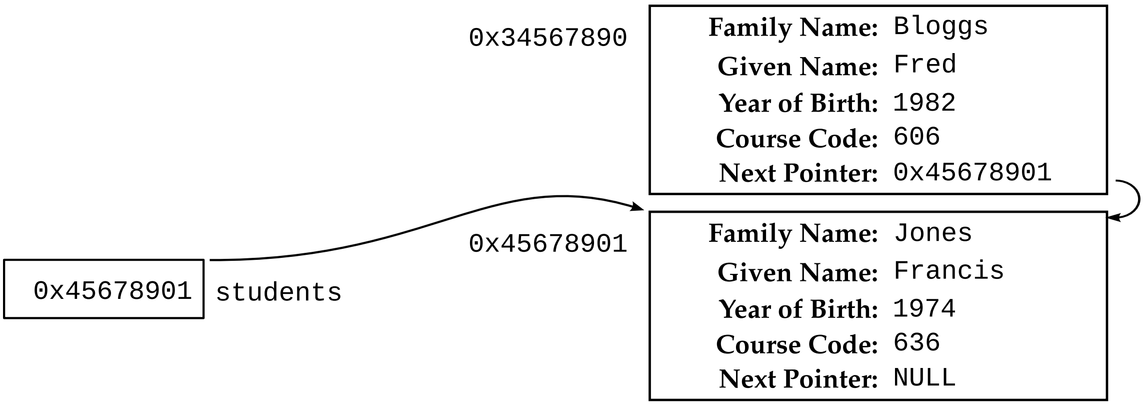
The result of this is shown in the diagram below:



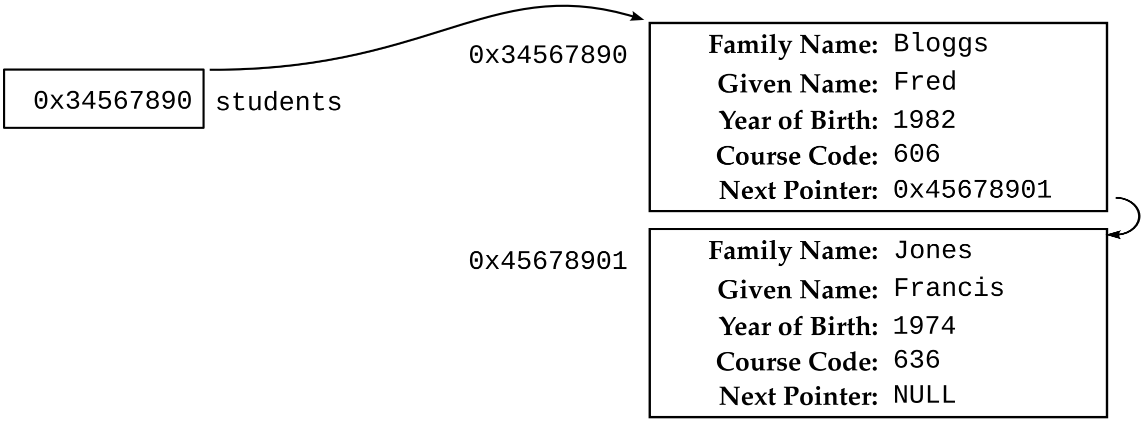
The steps for adding the next item to the list are slightly different. After allocating the new structure, and setting up the member variables to store data we must set the 'next pointer'. The situation at this point is shown in the diagram below:



We wish to insert the new structure **before** the structure that is already in the list. To do this we copy the value of the 'first pointer' into the 'next pointer' of the new structure. This makes sure that the new structure references the same thing that the 'first pointer' does:



Finally, we make sure that the 'first pointer' references the new structure and the list is complete again:



To summarise, the steps that we followed this time were:

- allocate the memory for the new item;
- set appropriate member variables in the new item;
- set the 'next pointer' of the new item to be whatever the current value of the 'first pointer' is;
- set the 'first pointer' to reference the new item.

You may notice that if you try to apply these steps to the first situation, where the very first item is being added to the list, they will still work. In fact, these steps will always work for adding new items to the list, however full or empty it already is.

Removing items from the list is the exact reverse of the steps for adding, except some are unnecessary:

- use the 'first pointer' to find the first structure on the list, this is the one we will remove;
- set the 'first pointer' to be whatever the 'next pointer' of the structure references, this causes the list to 'bypass' the structure we want to remove;
- free the memory for the item, it has been removed from the list.

These steps may be translated directly into code. The rest of this lab looks at how you can implement a linked list in C.

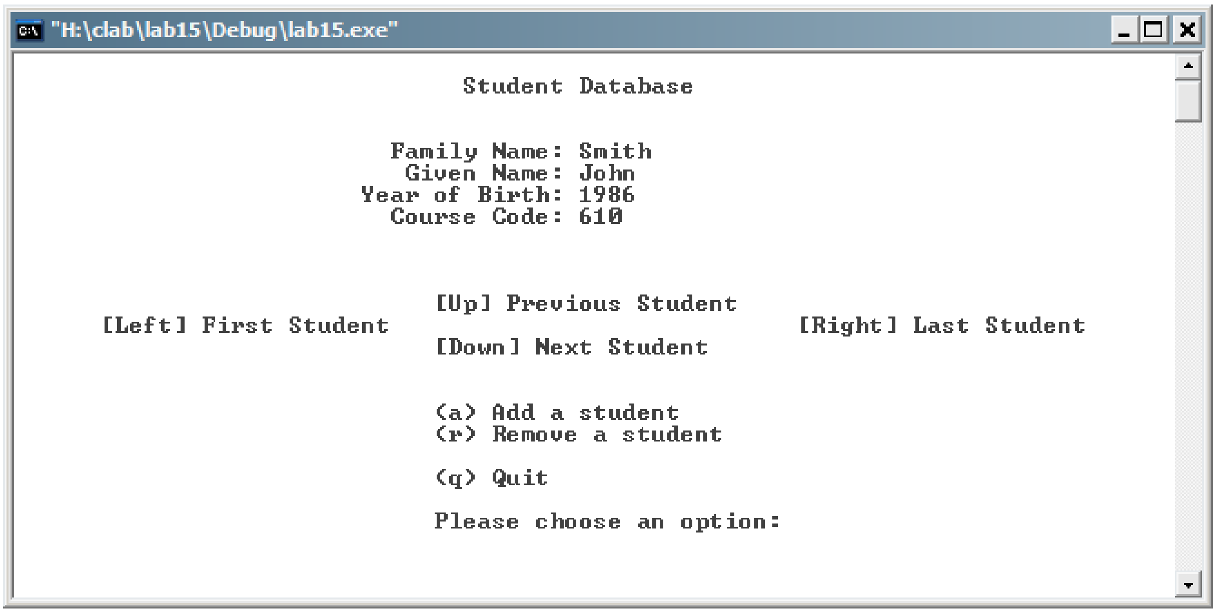
15.6 A Linked List Example

The "lab15" project is an example of how to construct a program which uses linked lists. The program stores student details in a very similar way to the programs you worked with in lab 11.

Copy the project and open it in the IDE. You will find that it is made up of the following files:

- “lab15.c” contains the `main` function as well as the `create_student` function which creates a new student structure and fills in the details and the `query_remove` function which asks the user if they really want to remove a student from the list.
- “display.c” contains the `display_student` function which displays the currently selected student and the `display_menu` function which displays the simple user menu.
- “display.h” contains prototypes for the functions in “display.c”.
- “list.c” contains functions for handling the actual linked list.
- “list.h” contains prototypes for the functions in “list.c”.
- “types.h” contains type definitions for the two structures used in the program.

A screen shot of the “lab15” program running is shown below:



```
                                Student Database


                           Family Name: Smith
                            Given Name: John
                         Year of Birth: 1986
                           Course Code: 610


                              [Up] Previous Student
     [Left] First Student                               [Right] Last Student
                              [Down] Next Student


                              (a) Add a student
                              (r) Remove a student

                              (q) Quit

                              Please choose an option:
```

Exercise 15.2: Investigating Linked Lists

Familiarise yourself with the “lab15” program, try adding students and browsing through the list. You should find that two important functions do not work:

- the ability to remove a student;
- the ability to view the previous student in the list;
- the ability to view the last student in the list.

When you add a student, does the student appear at the beginning or the end of the list?

The “lab15” program uses a structure to store student information in. It is defined like this (from `types.h`):

```
typedef struct long_student_type
{
    char family_name[40];
    char given_name[40];
    int year_of_birth;
    int course_code;
    struct long_student_type *next;
}
student_type;
```

This definition is identical to the structure you used in lab 11, with the addition of a pointer called `next`. Notice that this has to be defined using the long type name `struct long_student_type`, rather than the abbreviated `student_type`. This is because the structure needs to define the type of the `next` pointer **before** the `typedef` is complete. At the point at which the `next` pointer is defined, the `student_type` type name is not yet valid.

The program uses another structure to store information about the list as a whole. This structure is also defined in `types.h`. It is defined like this:

```
typedef struct long_list_type
{
    student_type *first;
    student_type *current;
}
list_type;
```

The `main` function has one of these structures to keep track of the first student in the list (the 'first pointer'). This is done with the member variable `first`. The member variable `current` is used to reference the student structure which is currently being displayed. A pointer to this structure is passed to all of the list handling functions. This allows them to update the `first`, `last` and `current` pointers if they need to.

Exercise 15.3: Understanding the Linked List Example

Read through the source code and make sure that you understand how the program works. If you have problems understanding parts of the program try stepping though them, by setting a breakpoint if necessary.

Exercise 15.4: Keeping Track of the Last Student in the List

Change the definition of `list_type` in `types.h` to add a 'last pointer', you should call it `last`. This will allow the program to keep track of the last student in the list, as well as the first.

To make this pointer work, you must add the following things:

- you should add to the `list_init` function to make sure the `last` pointer is initialised to `NULL`;
- the `list_add` function must be changed to set the `last` pointer if the list was previously empty.

Fill in the `list_last` function so that it sets the `current` pointer to point to the last student in the list.

Exercise 15.5: Adding Remove Functionality

Implement the `list_remove` function so that it removes the first item from the list. Remember to use the `free` function to deallocate the structure once it is removed.

Be very careful of the following situations:

- What if the list is already empty? Make sure your code detects this condition.
- What if the student you are removing is the only remaining student in the list? What extra step(s) do you need to do in this situation?
- What if the student you are removing (the first one in the list) is the one currently being displayed? If you do not change the `current` pointer it will be invalid.

15.7 Summary

Now that you have finished this lab you should understand the basic principles of linked lists, and some of the reasons why they are used. You should have investigated how linked lists can be handled in C and written some code to remove items from simple linked list.

This laboratory has only covered the basics of linked lists, lab 16 will go into more depth, making linked lists more useful.

Laboratory 16

Lists and Queues (2)

16.1 Overview

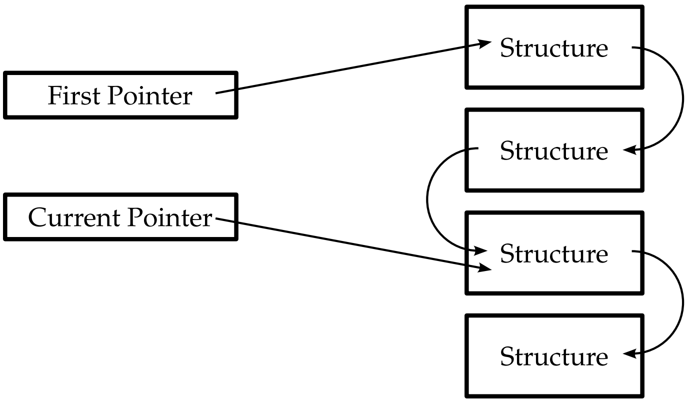
The last lab introduced the idea of storing structures by linking them together with references. This is called a linked list. This lab looks further at linked lists and examines some of the problems with the way in which linked lists were approached in the last lab. You will be introduced to a more sophisticated form of linked list, which alleviates many of these problems.

16.2 The Limitations of a Single-Linked List

The kind of linked list you have been working with so far has a single pointer in each structure, which references the next structure in the list. Because there is only one pointer in each structure, this is called a *single-linked list*. A single-linked list is very simple and very efficient. In this section you are going to explore some of the difficulties with single-linked lists.

The largest problem with single-linked lists is caused by the fact that the references that link each of the structures only go in one direction. This means that, if you are at a particular point in a linked list, it is difficult to find the item that comes *before* the one you are looking at.

For example, consider the situation in the diagram below:



In this case, the 'current pointer' is referencing the third structure in the list. How do we find our way to the second structure in the list?

The only way to do this is to go back to the beginning of the list (using the 'first pointer') and to move along the list keeping track of two pointers:

- a pointer to the item we are considering, call this pointer A, this pointer starts at the beginning of the list;
- a pointer to the item before pointer A, call this pointer B.

We will set Pointer A to start at the beginning of the list and then move it, link by link, through the list. We will stop when pointer A is equal to the 'current pointer'. Every time pointer A is moved on one link, pointer B is altered to be one behind it. When pointer A reaches the 'current pointer', B will reference the structure that comes before it.

We will now look at an example of how to do this in pseudo-code:

```
set pointer_a to equal first_pointer set pointer_b to be NULL

loop while (pointer_a does not equal current_pointer and pointer_a
is not NULL)
    set pointer_b to equal pointer_a
    set pointer_a to equal the next structure after pointer_a
end loop
```

When the loop exits, `pointer_b` will reference the structure before the one referenced by the `current_pointer`. In two situations `pointer_b` will be equal to `NULL`:

- the structure referenced by the `current_pointer` is the first one in the list, in this case there is no structure before it;
- for some reason (perhaps there is a bug elsewhere in the program) the structure referenced by the `current_pointer` is not actually part of the list.

This method is very similar to the one used in the `list_clear` function in the "lab15" project. The `list_clear` function goes through the whole list and deallocates the memory for all of the structures on the list. It is called when the program ends to make sure that all dynamically allocated memory is freed.

Exercise 16.1: Finding the Previous Item in the List

In the "list.c" source file in the "lab15" project, add a function to find the previous structure in the linked list. It should have the following prototype:

```
student_type *find_previous(student_type *student);
```

The function takes a single argument (`student`) which should reference a structure in the linked list. The function should return a reference to the structure that comes before `student` in the list.

Use the `find_previous` function to implement the `list_previous` function. This should allow the user to move both forwards and backwards in the list using the arrow keys.

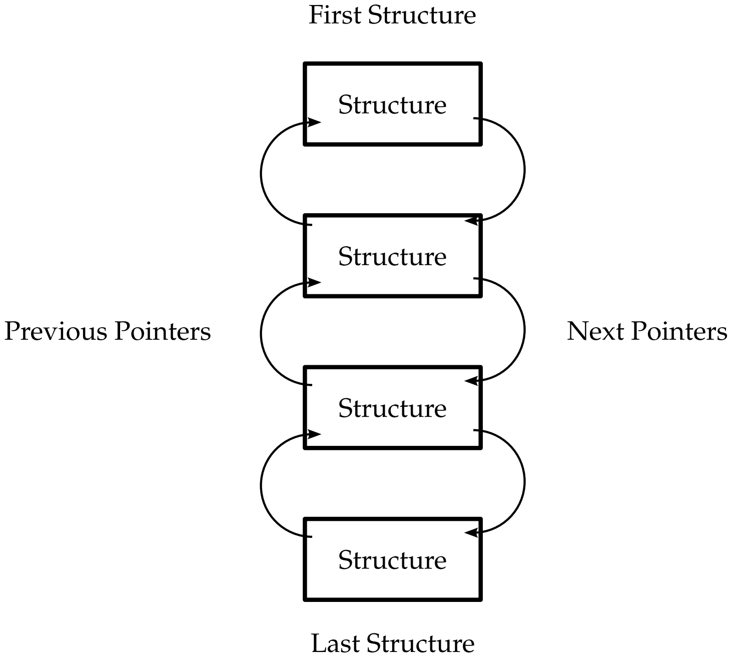
Exercise 16.2: Removing the Current Item

At the moment the `list_remove` function always removes the first item from the list. Using the `find_previous` function that you wrote in Exercise 16.1 change the `list_remove` function so that it always removes the **current** item from the list. There are three situations to think about, depending on where the structure is in the list. What happens if the structure is:

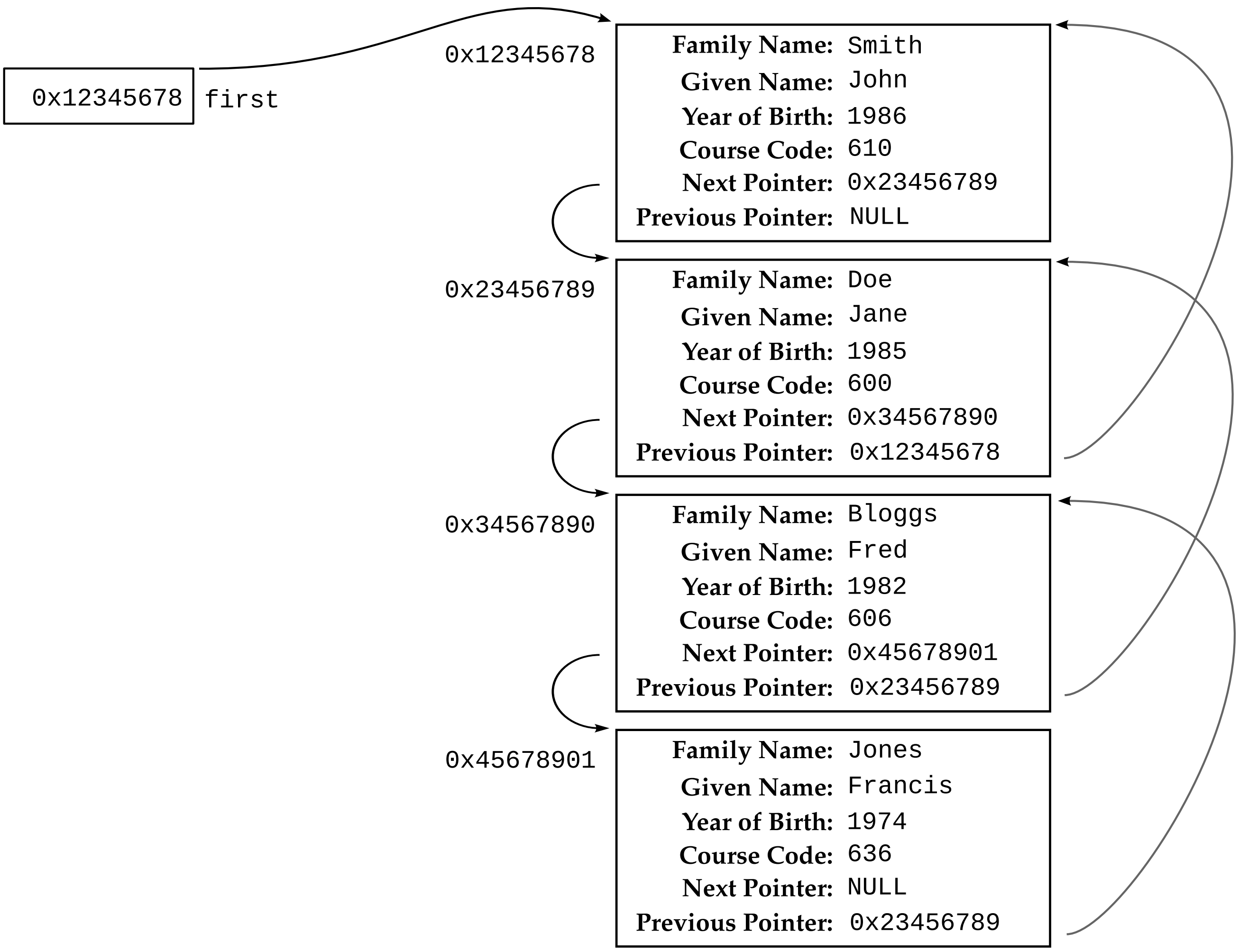
- in the middle of the list?
- at the beginning of the list?
- at the end of the list?

16.3 Introducing Double-Linked Lists

In many situations finding the previous structure in a linked list is essential. The method that you used to do this is very inefficient, but unavoidable if the list is only single-linked. A much more efficient way to solve this problem is to introduce another pointer into every structure which references the structure that comes *before* it in the list. Each structure then has both a 'next pointer' and a 'previous pointer'. This is depicted in the diagram below:



The previous pointer can be used to 'travel' in the reverse direction along the list. For a single-linked list, the 'next pointer' of the last structure in the list had the special value `NULL` to indicate that it was the last structure in the list. In a double-linked list the first structure has its 'previous pointer' set to `NULL` in a similar way. This is shown in the diagram below:



A list like this is a lot more flexible, but is more difficult to set up and manage. Great care must be taken to ensure that all pointers reference valid memory.

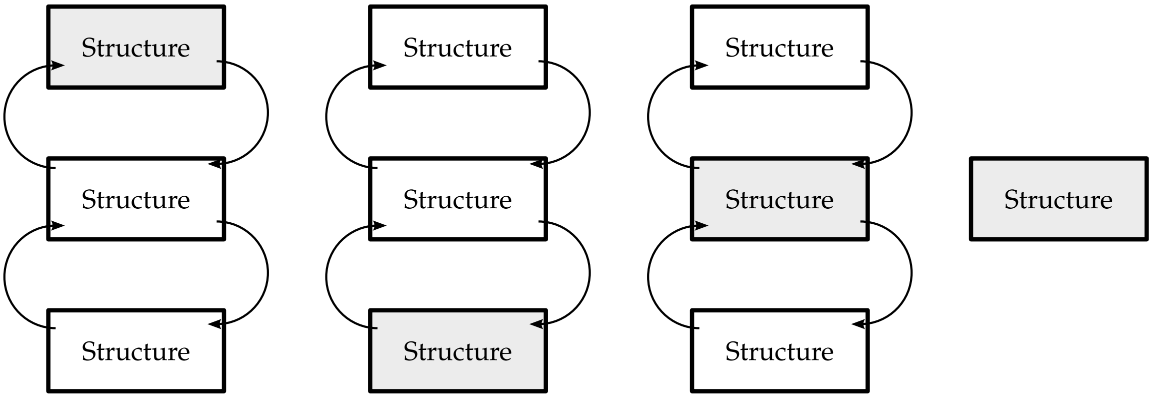
16.4 Manipulating Double-Linked Lists

In a moment you will have to write some code to manage a double-linked list. Before this we will look at the key steps of the processes for adding and removing structures into a double-linked list.

For both adding and removing operations there are four different situations to consider. These are to do with the position in the list that structure will have (in the case of adding) or has (in the case of removing). The structure could be:

- first in the list;
- last in the list;
- somewhere in the middle;
- the only structure in the list (this is a bit like being *both* first and last).

These situations are shown in the diagram below:



First in the List Last in the List Middle of the List Only Item in the List

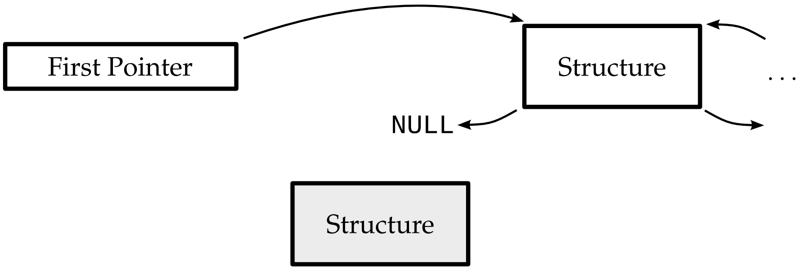
This section will examine first adding and then removing. Each operation will consider these four cases.

16.4.1 Adding into a Double-Linked List

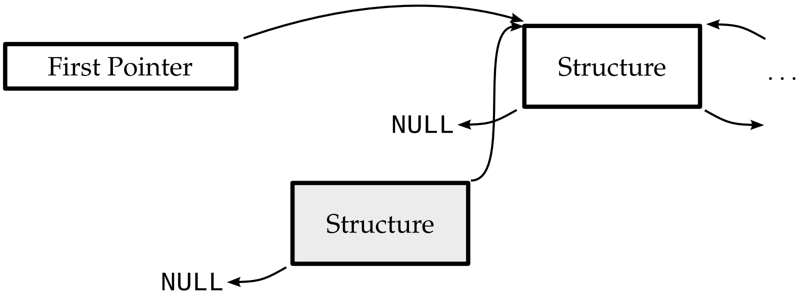
This section will look at the steps involved in adding a structure to a double-linked list. We will examine each of the four different situations under which this can occur in turn.

Adding to the Head of the List

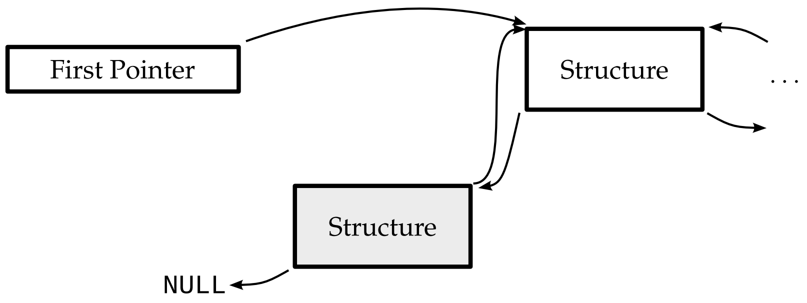
To begin with, the new structure (shown below shaded) is not linked into the list at all:



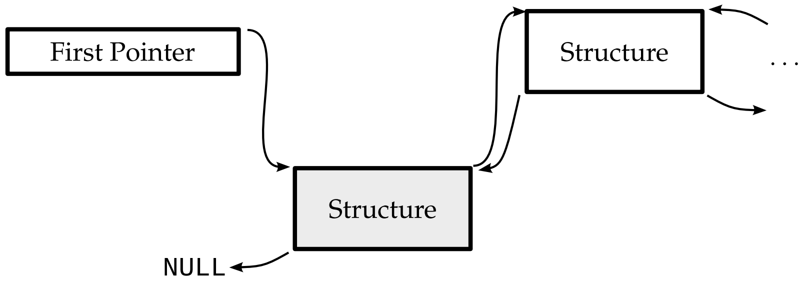
The new structure must be set so that its 'next pointer' references the same thing as the 'first pointer', and its 'previous pointer' is set to `NULL`:



The 'previous pointer' of the structure which is currently first on the list must be set to reference the new structure:



Finally, the 'first pointer' must be set to reference the new structure:



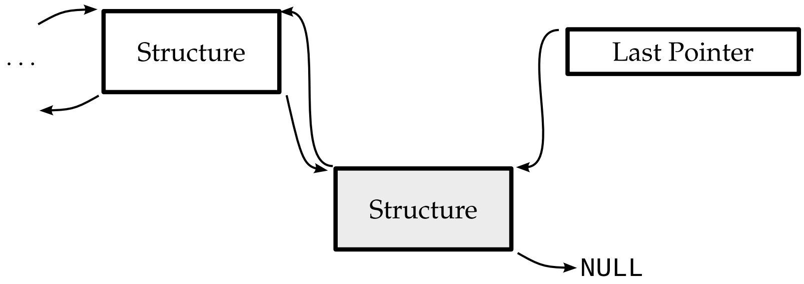
The structure has been successfully added to the head of the list.

Adding to the Foot of the List

Adding to the foot of a list is very similar to adding to the head of the list. Rather than using the 'first pointer', the 'last pointer' is used. The steps are (you can compare these with the steps for adding to the head of the list):

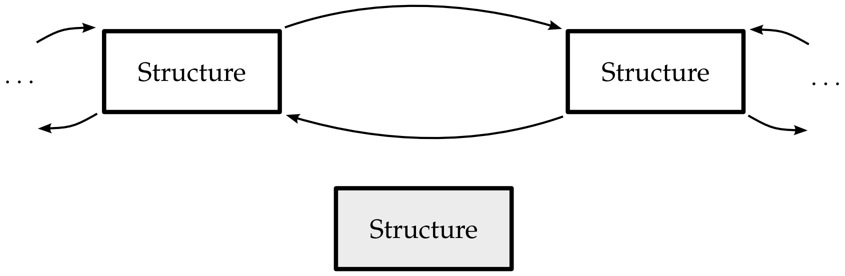
- The new structure must be set so that its 'previous pointer' references the same things as the 'last pointer' and its 'next pointer' is set to `NULL`.
- The 'next pointer' of the structure which is currently last on the list must be set to reference the new structure.
- Finally, the 'last pointer' must be set to reference the new structure.

The completed operation results in the following situation:



Adding into the Middle of the List

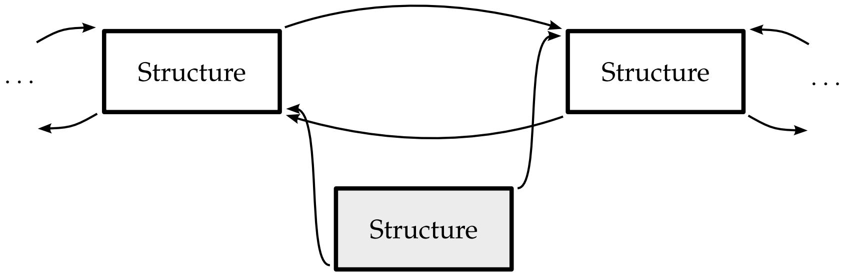
When adding to the middle of the list, there is a structure before the point of insertion, and there is a structure after it. This is shown in the diagram below; the new structure is shown shaded, and is not yet linked into the list:



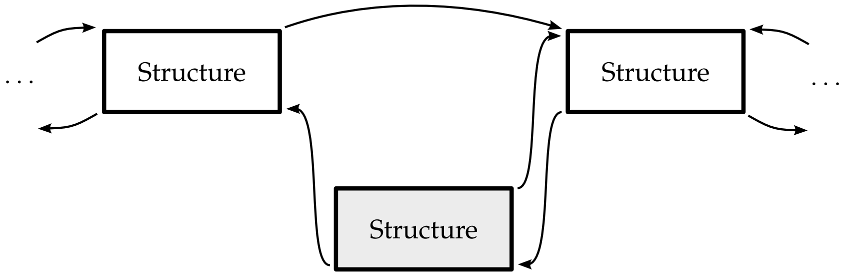
The new structure must be set so that:

- its 'next pointer' is the same as the 'next pointer' of the structure before it in the list;
- its 'previous pointer' is the same as the 'previous pointer' of the structure after it in the list.

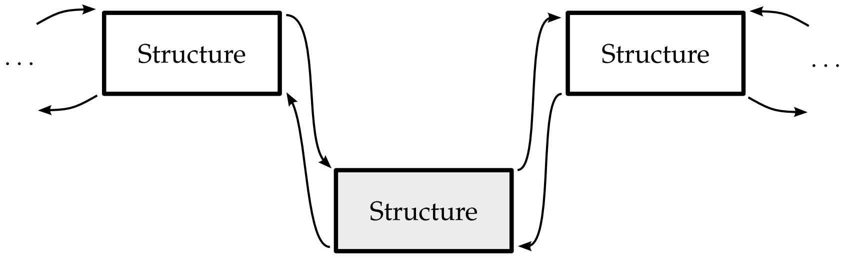
This is shown below:



The 'previous pointer' of the structure which is next on the list must be set to reference the new structure:



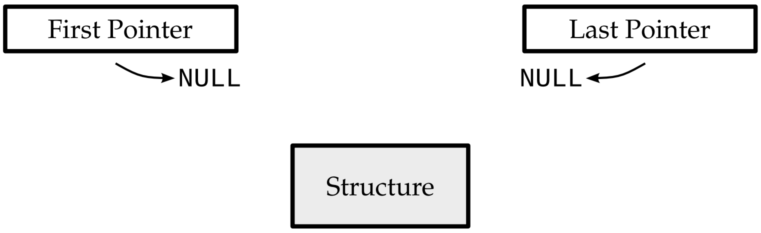
Finally, the 'next pointer' of the structure after the new structure on the list must be set to reference the new structure:



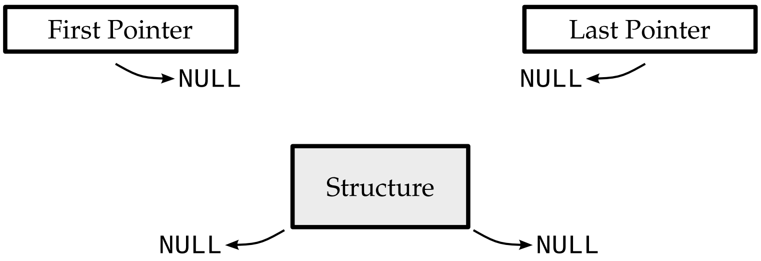
The structure has been successfully added to the middle of the list.

Adding to an Empty List

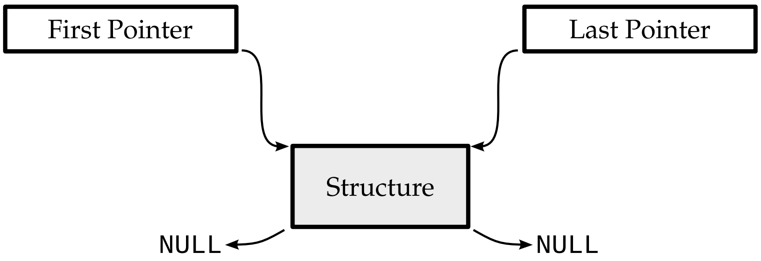
Adding to an empty list is a simple operation. To begin with, both the 'first pointer' and the 'last pointer' will be set to `NULL`:



The 'next pointer' and 'previous pointer' of the new structure should both be set to be `NULL`:



Finally, both the 'first pointer' and the 'last pointer' should be set to reference the new structure:



The structure has been successfully added to the list.

16.4.2 Removing from a Double-Linked List

This section will look at the steps involved in removing a structure from a double-linked list. We will examine each of the four different situation under which this can occur in turn, each of them has strong similarities to the steps involved in adding structures to the list.

Removing from the Head of the List

The steps involved in removing a structure from the head of the list are:

- Set the 'first pointer' to reference the next structure in the list.
- Set the 'previous pointer' of the next structure in the list to be `NULL`.
- The structure is no longer part of the list, its memory can be deallocated.

Removing from the Foot of the List

The steps involved in removing a structure from the foot of the list are:

- Set the 'last pointer' to reference the previous structure in the list.
- Set the 'next pointer' of the previous structure in the list to be `NULL`.
- The structure is no longer part of the list, its memory can be deallocated.

Removing from the Middle of the List

The steps involved in removing a structure from the middle of the list are very similar to removing from either the head or the foot. The object to be removed will have a structure both before it and after it in the list. The steps are:

- Set the 'next pointer' of the previous structure in the list to reference the next structure in the list.
- Set the 'previous pointer' of the next structure in the list to reference the previous structure in the list.
- The structure is no longer part of the list, its memory can be deallocated.

Removing the Only Remaining Structure in the List

When there is only one remaining structure in the list, the removal operation is simple. Both the 'first pointer' and the 'last pointer' should be set to `NULL`. The structure has then be removed from the list and its memory can be deallocated.

Exercise 16.3: Creating a Double-Linked List

Change the "lab15" program so that it uses a double-linked list. You may need to draw some diagrams of your own to make sure that you fully understand the processes involved in both adding and removing items from the list. You should find that using double-linked list simplifies some aspects of list handling.

16.5 So When Are Linked Lists Useful?

Linked lists are a very powerful technique but, as you have seen from this and the previous laboratory, are more difficult to work with than a simpler storage system such as arrays. Linked lists are appropriate in many situations, here are some pointers to help you identify when a linked list might be suitable for your program:

- when each data item is represented by a structure, if a data item is just a number an array is likely to be more efficient;
- when you need to be able to store anything from just a few data items to a large number;
- when you need to be able to add data items into the middle of a list and remove them, or if you need to be able to change the order of data items easily;
- if the list will be very big the amount of memory an array would need would be very big and the computer would have to allocate it all in one chunk, with a linked list the memory can be allocated in smaller pieces which is easier to accommodate.

The largest drawback with linked lists, even double-linked lists, is that you cannot simply ask for the nth data item. You must work through the list n times from the beginning (or the end). Arrays do not have this problem. The property of being able to get to the nth data item immediately is called *random access*.

You may have noticed that the easiest way to handle single-linked lists is to add and remove to/from the head of the list. This means that single-linked lists as most appropriate in situations where the storage will be used in a LIFO fashion. If FIFO-type storage is required, a double-linked list is more appropriate.

16.6 Summary

Now that you have finished this lab you should be able to work with both single-and double linked lists. You should know how each type of linked list is constructed and how they can be used. You should also understand how to write programs which use linked lists.

Laboratory 17

File Handling

17.1 Overview

Up to this point in the course you have used functions from the standard C library to display information on the screen and receive information from the user. This laboratory will introduce you to a set of very similar functions which allow you to both write and read information to/from files stored on the computer or a network drive.

In the final part of the lab you will use this to store information from the program you have been working as part of you graphics or music option.

17.2 Getting Access to Files

Since very nearly the beginning of this course you have been using the `printf` function for output to the screen and the `scanf` function for input from the keyboard. This lab will use very similar functions, `fprintf` and `fscanf`, to output and input to/from a file.

Before you can either read or write to/from a file you have to request access to the file. When you do this, the file you have requested is usually *locked* meaning that only your program can access the file at that time. This is called *opening* the file. When you are finished with the file you must *close* it again.

When you open a file you are given a pointer which acts a shorthand for the file you are talking about. This pointer is called a *file handle*. You get a different file handle for each file that you open. Once you have closed the file, the file handle is no longer valid. A file handle variable is declared like this:

```
FILE *file_handle;
```

To open a file you should use the `fopen` function. This function requires two arguments, both of which are strings: the name of the file you want to open and the *access mode* you wish to have. The access mode is to do with what kinds of operations you wish to carry out on the file. Specifying `"w"` indicates that you wish to write to the file; `"r"` specifies you wish to read and `"w+"` specifies that you want to be able to both read and write. For example, the following line of code:

```
file_handle = fopen("number.txt", "w");
```

opens the file `numbers.txt` for writing. If the `fopen` function could not open your file, for whatever reason, the file handle (in this case the `file_handle` variable) is set to `NULL`.

When you are finished with the file, you must close it again using the `fclose` function. This function takes only one argument: the file handle of the open file you wish to close:

```
fclose(file_handle);
```

Function Reference: `fopen` — Opens a File

```
FILE* fopen(char* filename, char* access_mode);
```

e.g.

```
    file_handle = fopen("data.txt", "w");
    secret_file = fopen("important.dat", "r");
    temp_file = fopen("temporary.tmp", "rw");
```

The `fopen` function attempts to open the file specified by the `filename` argument with the access mode specified by the `access_mode` argument. Both of these arguments should be strings. Valid access modes are:

- `"r"` open file for reading, the file must already exist;
- `"w"` open file for writing, if the file already exists it will be overwritten;
- `"a"` open file for writing, if the file already exists it will be not overwritten, writing takes place at the end of the file (this is called *append* mode);
- `"r+"` open file for both reading and writing, the file must already exist;
- `"w+"` open file for both reading and writing, if the file already exists it will be overwritten;
- `"a+"` open file for both reading and writing, if the file already exists it will be not overwritten, reading and writing take place at the end of the file.

If the file cannot be opened the `fopen` function returns `NULL`.

The `fopen` function is defined in `stdio.h`.

Function Reference: `fclose` — Closes an Open File

```
int fclose(FILE* file_handle);
```

The `fclose` function closes an open file identified by the argument `file_handle`. If there is a problem during the close operation the `fclose` function returns the predefined constant `EOF`. If there were no problems the function returns zero.

The `fclose` function is defined in `stdio.h`.

17.3 Simple File Read and Write Operations

Copy the “lab17” project and open it in the IDE. The source code for the “lab17.c” file is shown below:

```
/*
 * A Program to Demonstrate File Handling
 * Lab 17
 */

#include <stdio.h>
```

```
int main(void)
{
    FILE *file_handle;
    int number;

    /* Ask the user for a number */
    printf("Please enter an integer number: ");
    scanf("%d", &number);

    /* Get access to the file by opening it */
    file_handle = fopen("number.txt", "w");

    /* Write a single number to the file */
    fprintf(file_handle, "%d", number);

    /* We’re done with the file, close it */
    fclose(file_handle);

    return 0;
}
```

This simple program first obtains an integer number from the user. It then opens the file `number.txt` for writing. The next line of code uses the `fprintf` function to write to the file. The `fprintf` function works exactly like the `printf` function, or the `sprintf` function. The difference between the `fprintf` function and the `printf` function is that the `fprintf` function takes an extra first argument: the handle of the file to write to. For example, the code:

```
fprintf(file_handle, "Hello");
```

Outputs the text “Hello” to the file identified by the file handle `file_handle`. The line in the “lab17” program:

```
fprintf(file_handle, "%d", number);
```

Outputs the integer number held in the `number` variable to the file as text.

Function Reference: fprintf — Outputting to a file using printf-like placeholders

```
int fprintf(FILE* file_handle, char* format_string, ...);
```

e.g.

```
    fprintf(file_handle, "%d", int_variable);
    fprintf(result_file, "The answer is %d", integer_answer);
    num_chars = fprintf(temp_file, "%lf", double_variable);
```

The `fprintf` function works like the `printf` function except that, instead of outputting its results onto the screen, it places them into the file specified by the `file_handle` argument. It is called in exactly the same way as `printf` but with an extra first argument (`file_handle`).

`fprintf` takes at least two arguments but, like `printf`, it can take more, depending on how many variables you want to include.

- `file_handle` is a handle to a file, previously opened with `fopen`, which will be used as the destination for all `fprintf` write operations;
- `format_string` is a string containing placeholders, just like with `printf`;
- after the first two arguments (where the `...` is) you should include variables whose values will be substituted for the placeholders in the `format_string`.

The placeholders you will use most often are:

- `%d` for integer (`int`) variables;
- `%lf` `double` variables;
- `%c` for single characters (`char`);
- `%s` for strings (`char[]`).

These are the same as for `printf`.

The return value of `fprintf` is an integer, which specifies how many characters were written into the file (not including the null termination). This number will be negative if an error occurs.

The `fprintf` function is defined in `stdio.h`

Exercise 17.1: Investigating File Writing

Try building and executing the “lab17” program. When it has terminated, open “My Documents” from your desktop, double click on your “clab” folder and the “lab17” folder. You should see a file called `number` or `number.txt`. If you double click on this file it will open. You should see that the number you entered into the “lab17” program is stored in this file as text.

Just as writing to a file can be done with `fprintf`, reading from a file can be done with `fscanf`. The `fscanf` function works just like the `scanf` function except that it takes an extra first argument: the handle of the file that it should read from. For example, the following line of code reads an integer from a file:

```
fscanf(file_handle, "%d", &int_variable);
```

Function Reference: `fscanf` — Performs a `scanf`-like read from a file

```
int fscanf(FILE* file_handle, char * format_string, ...);
```

e.g.

```
    fscanf(file_handle, "%d", &an_int_variable);
    fscanf(text_file, "%c", &a_char_variable);
    number_of_matches = fscanf(result_file, "%lf", &a_double_variable);
```

The `fscanf` function obtains textual input from the file identified by the `file_handle` argument. This handle must identify a file previously opened by the `fopen` fucntion for reading. The input is converted according to the `format_string` argument. The `format_string` argument specifies placeholders in an identical manner to the `scanf` function.

`fscanf` takes at least two arguments but, like `scanf`, it can take more, depending on how many variables you want to include.

- `file_handle` is a handle to a file, previously opened with `fopen`, which will be used as the source for all `fscanf` read operations;
- `format_string` is a string containing the place holders that `fscanf` will try to match input to;

`fscanf` uses the same place holder system as `scanf`. 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 return value of `fscanf` is the number of place holders that were successfully matched or the predefined constant `EOF` (End Of File) if there was a problem.

The `fscanf` function is defined in `stdio.h`.

Exercise 17.2: Reading from a File

Alter the "lab17" program so that it reads a number back from the `number.txt` file using `fscanf` and displays it to the user. Remember to change the access mode in the `fopen` function call.

17.4 Reading and Writing Many Lines to/from a File

The `fprintf` and `fscanf` operations are very powerful, they are not just limited to writing a single number into a file and reading it back again. There is no reason why you cannot place a `fprintf` function call like:

```
fprintf(file_handle, "%d\n", numbers[index]);
```

into a loop to write many consecutive number to a file from an array. Notice the use of the new-line escape sequence (`\n`) to make sure that each number is on a new line in the file.

The next exercise will use the `random_number` function from the defined in the music library header file `midi_lib.h`. Those doing the music option will already be familiar with this function. The function generates an integer random number within the bounds that are specified by the two arguments to the function. For example, the function call:

```
number = random_number(1, 10);
```

Will assign a random number to the `number` variable. The random number will be anywhere between (and including) 1 and 10.

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 `midi_lib.h`.

Exercise 17.3: Writing Multiple Numbers to a File

Alter the "lab17" so that it fills an array of 40 elements with random numbers and writes all of the numbers to a single file. Each number should be on a new line in the file. Open the file you have written into to check the write operation worked correctly.

Adjust the program to write 500 random numbers between 1 and 100 to a file called `unsorted.txt`. **Keep this file, you will need it in lab 18.**

Exercise 17.4: Reading Multiple Numbers from a File

The "lab17a" project is a nearly-empty template for this exercise, copy the project and open it in the IDE. Write a program that opens the file that was to written by the program in Exercise 17.3. It should read 20 numbers from this file into an array. At the end of the program it should display the numbers.

Exercise 17.5: Making the Read Operation More Flexible

The “lab17a” program you have just written depends on there being 20 numbers in the file it is reading from. If there are more than that, the numbers do not get read. If there are fewer, there will be a problem. (In this case the `fscanf` function will return the constant `EOF`, which stands for End Of File).

Alter the “lab17a” program so that it will read any amount of numbers in from a file. The numbers should be read into an array, the memory for which is dynamically allocated using `malloc`. To do this you will need to read the file twice:

- once to determine how many numbers there are in the file so that you can allocate the array;
- a second time to read the numbers into the array.

You will need this program when you come to attempt some of the exercises in laboratory 18.

17.5 File Operations with Structures

The examples we have looked at so far have only written or read a single number with each `fprintf` and `fscanf` function call. There is no reason why this has to be the case. A `fprintf` function call could insert multiple numbers into a file, like this:

```
fprintf(file_handle, "%d, %d\n", first_number, second_number);
```

This would write two numbers on a single line in the file, separated by a comma and a space. To read these numbers back in from the file we would use the following `fscanf` function call:

```
fscanf(file_handle, "%d, %d\n", &first_number, &second_number);
```

There is also no reason that we must only work with numbers. We may also work with strings.

In labs 11, 15 and 16 you worked with a structure which stored information about a student. You created a type definition for this structure called `student_type`. To write information from this type of structure to disk you would use a piece of code like the following:

```
fprintf(file_handle, "%s, ", student.family_name);
fprintf(file_handle, "%s, ", student.given_name);
fprintf(file_handle, "%d, ", student.year_of_birth);
fprintf(file_handle, "%d\n", student.course_code);
```

Notice that each of the pieces of information about the student will all appear on the same line in the file, each separated by a space and a comma. A similar system could be used for reading information in.

Exercise 17.6: Reading and Writing Structure Information to/from a File

Alter the "lab17" program so that it prompts the user for details about a student and stores them in a structure. Write a separate function which will write the details of that student to a file. The student details should all appear on the same line in the file, separated by commas.

The "lab17b" project is another near-empty template. Basing your "lab17b" program on "lab17a", write a program which uses a separate function to read the details of a single student in from the file that was written to by the "lab17" program. The program should display the details it reads in. **Do not alter the "lab17a" program, you need this in lab18**.

You might like to copy some of your code from the "lab11' project to make this task easier.

Exercise 17.7: Adding Save and Load Functionality to a Program

If you have been following the graphics option, open the "graphics3" project; if you have been following the music option, open the "music3" project. Each of these programs use an array of structures to store either a drawing (in the case of the graphics option) or a musical sequence (in the case of the music option). Based on the exercises you have done in this lab, add functionality to your program to allow the user to save the contents of the array to a file, or to load all the information from a file into the array. This will allow a user to save and then reload their drawing or sequence.

17.6 Summary

Now that you have finished this laboratory you should understand how to carry out basic file operations. You should know that:

- to begin operating with a file, you must *open* it with the `fopen` function;
- when you have finished using the file you should *close* it with the `fclose` function;
- when operating on the file you use an identifier that you obtained from the `fopen` function called a *file handle*;
- you can write to the file using the `fprintf` function, which is very similar to `printf`;
- you can read from the file using the `fscanf` function, which is very similar to `scanf`.

You should have used these concepts to add load and save functionality to the program that you have been working on as part of your option. This gives you the ability to save drawings as a file (in the case of the graphics option) or musical sequences as a file (in the case of the music option).

Laboratory 18

Sorting, Searching and Recursion

18.1 Overview

This laboratory takes a very brief look at two activities that are often carried out by software programs:

- *sorting*, where a collection of data is sorted according to some criteria;
- *searching*, where the presence and position of some item in a collection of data must be determined.

Both of are very complicated topics in their own right. You will only write one program for each of these topics.

Searching data often uses a technique called *recursion*. A function is recursive if it calls itself. This can cause the computer to execute many times, just like a loop. This lab introduces recursion with a few simple examples, and then asks you to write a recursive program to search and array.

18.2 Sorting Data Using a Bubble Sort

You have been taught about *bubble sorts* in your lectures. A bubble sort is a simple sorting technique which ensures a list of data items is sorted, simply by comparing items, a pair at a time. The program determines whether the items are in order, if there are not, the items must be swapped over in the list.

Exercise 18.1: Sorting a Collection of Numbers

Open the “lab17a” project that you worked on in lab 17. This program should read in a list of numbers from a file, into a dynamically allocated array. You should also have a file called `unsorted.txt` which you created in lab 17.

Alter the “lab17a” program so that it reads the numbers from `unsorted.txt`. You should write a function called `sort` which the program should call to sort the contents of the array. The `sort` function should use a bubble sort technique. Once the `sort` function has sorted the array of numbers, your program should write the numbers into a file called `sorted.txt`.

When you have executed your program, open the `sorted.txt` file and check that the numbers are in order.

Determining the order of numbers is fairly easy, you can use the mathematical greater than (>) and less than (<) operations. Comparing strings is more difficult. Fortunately, there are some standard C functions to do this for you. The most useful string comparison function is called `strcmp`. The `strcmp` function takes two arguments, both of which are strings. For example:

```
char* first_string = "hello";
char* second_string = "world";
int result;

result = strcmp(first_string, second_string);
```

The `strcmp` function compares the two arguments and determines the order they should be in, alphabetically, and returns an integer number. If the first argument comes before the second, `strcmp` will return a negative number. If the second argument should come before the first, `strcmp` will return a positive number, greater than one. If the two arguments are identical, `strcmp` will return zero.

The `strcmpi` function is identical to the `strcmp` function, except that `strcmpi` ignores the case (upper or lower) of the characters when it carries out the comparison.

Function Reference: `strcmp, strcmpi` — String Comparison

```
int strcmp(char* string1, char* string2);
int strcmpi(char* string1, char* string2);
```

e.g.

```
    result = strcmp(first_string, second_string);
    security_check = strcmp("Open Sesame", password);
    are_not_equal = strcmpi(magic, "abracadabra");
```

The `strcmp` and `strcmpi` functions compare their two string arguments `string1` and `string2`. In each case the function returns an integer indicating the relative order of the two strings:

- if `string1` comes before `string2` the function returns a number less than zero;
- if `string1` comes after `string2` the function returns a number greater than zero;
- if `string1` matches `string2` the function returns zero.

`strcmp` performs a case sensitive comparison; `strcmpi` performs a case *in*sensitive comparison.

Both the `strcmp` and `strcmpi` functions are defined in `string.h`

Exercise 18.2: Sorting a Linked List Based on Strings

Alter your "lab15" program to add a 'Sort' option to the menu. When the user selects this option, the program should sort all the students in the linked list according to their family name. You should use a bubble sort for this.

18.3 Introducing Recursion

A *recursive* function is a function which calls itself. A function which calls itself usually causes some of the code in that function to be run a second time. By calling itself, a recursive function is forming a loop, just like the `while` or `for` loops you have used on this course. To differentiate the two types of looping, using a `while`- or `for`-type loop, which will occur *within* a function, is called *iteration*.

Recursion is an alternative way to produce a loop in a program, and any iterative program can be re-written as recursive and vice-versa. In fact, there are some programming languages that do not have any way of doing iteration. In these languages, the only way to produce a loop is using recursion.

Copy the "lab18" project and open it in the IDE. The source code for the "lab18.c" file is shown below:

```
/*
* A Program to Demonstrate Recursion
* Lab 18
*/

#include <stdio.h>

/*
* Sums all the numbers between start and end
* start must be smaller than end
*/
int sum_series(int start, int end)
{
    /* If there is no difference between the start
    * and end numbers, there is nothing to sum */
    if (start == end)
        return end;

    /* Add the current start number to all the other
    * numbers in the series */
    return start + sum_series(start + 1, end);
}

/*
* Requests two numbers from the user and displays the
* sum of all the integer numbers in between
*/
int main(void)
{
    int start, end, sum;

    /* Ask the user for numbers */
    printf("Please enter two integer numbers\n");
    printf("The first number must be smaller than the second\n");
    printf("First number: ");
    scanf("%d", &start);
    printf("Second number: ");
    scanf("%d", &end);

    /* Calculate the sum of the series */
    sum = sum_series(start, end);

    /* Display the result */
    printf("The sum of all the integers from %d to %d is %d\n", start, end, sum);

    return 0;
}
```

This program uses recursion to calculate the sum of all the integers from one number to another (including the numbers themselves). For example, the sum of all the numbers from 1 to 5 is $1+2+3+4+5 = 15$. This sum is calculated by the `sum_series` function. You will notice from the source code above that the `sum_series` function calls itself. It uses recursion to loop through the numbers in the series.

Exercise 18.3: Investigating Recursion

Try building and executing the "lab18" program. Try putting different combinations of numbers in and check the results(the program will break if you put a large number in first!). Try stepping through the program and watching the contents of the variables change. Make sure you understand how the program works.

Exercise 18.4: Calculating a Factorial Recursively

In the "lab3a" program you wrote a program which calculated the factorial of a number using a `for` loop. Change this program (or write a new one) so that it calculates the factorial of the number using a recursive function.

18.4 Searching Using Recursion

You have been taught about *binary searches* in your lectures. A binary search looks for a particular item in a sorted list by repeatedly breaking the list down into two parts.

Exercise 18.5: A Recursive Binary Search

Alter the "lab17a" program so that, after it has sorted the list of numbers, it asks the user for a number to search for. It should then perform a binary search using a recursive search function to find the number. The program should tell the user the line number of the number they requested in the `sorted.txt` file. (This is the same as its index in the array).

18.5 Summary

Now that you have finished this laboratory you should understand how to implement a bubble sort algorithm, both for simple arrays of numbers and for more complex situations like strings and linked lists. Whilst doing this you will have used the `strcmp` and `strcmpi` string comparison functions.

In the second part of the lab you learnt about recursion, which is a way of looping parts of your program by creating a function which calls itself. You should have used this technique to calculate a factorial and to perform a binary search on a sorted array of integers.

Appendix A

Summer Term Laboratory Outline

A.1 Overview

You have four laboratories in the summer term: labs 19, 20, 21 and 22. Laboratory 22 is set aside for you to demonstrate the software you have developed as part of your second assignment. Labs 19 to 21 are free for you to use to work on your assignment. You may choose how you want to use these labs, however, it is suggested that you spend:

- lab 19 working on your analysis, specification and design;
- lab 20 working on you implementation;
- lab21 testing your implementation and making any changes as necessary.

During each of these labs the demonstrators will be available to help and advise you.

Appendix B

Suggested Further Reading

B.1 Tutorial Material

[1] Peter Aitken and Bradley L. Jones. *Teach Yourself C in 21 Days*. SAMS, 2002.

[2] Paul Davies. *The Indispensable Guide to C with Engineering Applications*. Addison Wesley, 1995.

[3] Andy M. Tyrrel, Stephen L. Smith, and Jonathan A. Dell. *Essence of C for Electronic Engineers*. Essence of Engineering. Prentice Hall, 1998.

[4] Tony Zhang. *Teach Yourself C in 24 Hours*. Sams, 2000.

B.2 Reference Material

[5] Brian W. Kernigan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988.

[6] Grant Macklem, Gregory Schmelter, Alan Schmidt, and Ivan Stashak. Graphics library function reference. http://www.cs.colorado.edu/˜main/cs1300/doc/bgi/, (Accessed November 2005).

[7] Microsoft Developer Network. Msdn reference library (includes C standard library reference). http://msdn.microsoft.com/library/default.asp, (Accessed November 2005).

[8] Peter Prinz and Tony Crawford. *C in a Nutshell*. O'Reilly, 2005.