Introduction to the C Programming Language

Laboratory Scripts

Dr. Adar Pelah
Dr. Julian Miller (original author)
with
Dr. Andy Hunt, Dr. Peter Mendham,
Dr. Andy Pomfret, Dr. Steve Smith,
Dimitrios Zantalis, Amir Dehsarvi and Tautvydas Mickus

September, 2016

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 the Autumn term. There is a laboratory script for each lab in the Spring term. The course is assessed via a programming assignment and a written report (submitted electronically). The assignment period begins at the end of the taught component of the course.

There are some labs that are in the appendix. These will not be covered in the allocated lab sessions, but nevertheless they are important and should be studied and used when appropriate, especially in assignments.

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, it is strongly suggested that you attempt parts of the other option. The C programming course is assessed through one assignment. The assignment will be handed out towards the end of term. 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]`$^{\mathbf{n}}$, 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.

Part I

Laboratories

Laboratory 1

Simple C Programs in Code::Blocks

1.1 Developing C programs

In order to write programs (in C or Java etc.) we need some tools, system tools that is, that help us accomplish this. In particular we need a compiler, a linker, a debugger and an *Integrated Development Environment* or IDE to aid with the development process.

An IDE 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 a source files and translates it 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;
- a **linker** which takes one or more compiled files (object files) generated by a compiler and combines them into a single executable program;
- a **debugger** which allows you to test out your programs in a very controlled way and diagnose problems with them.

To read more about these topics in more detail you could follow the links below (reading the introduction of each article should suffice):

- `Compiler: http://en.wikipedia.org/wiki/Compiler`
- `Linker: http://en.wikipedia.org/wiki/Linker_(computing)`
- `Debugger: http://en.wikipedia.org/wiki/Debugger`
- `IDE: http://en.wikipedia.org/wiki/Integrated_development_environment`

1.2 Code::Blocks and MinGW

In this first laboratory you will be introduced to the programming tools we will use on the C programming course. All the tools that we use (for Windows OS) are free and open source. This means that you will be able to install them on your own computers with no cost.

To write your C programs you will use two pieces of software (in addition to libraries for doing graphics and music (MIDI) - more on this later). The first is called MinGW which stands for"Minimalist GNU for Window". This is a C programming development environment and includes the GNU C compiler. The second is an integrated development environment (IDE) called Code::Blocks. Further information about these is available through the following URLs:

- `MinGW (http://www.mingw.org/)`
- `Code::Blocks (http://www.codeblocks.org/home)`

There are more system tools that are used when developing applications (such as assemblers, archivers etc.) but the aforementioned ones are all we need for completing this course. The idea is to write cross-platform code i.e. code that is system independent. You only have to write your code once and you can compile it on different operating systems without any modifications. The C language is ideal for such kind of programming; compilers exists for all major operating systems (Windows, Linux, OSX).

1.3 Starting to write C programs

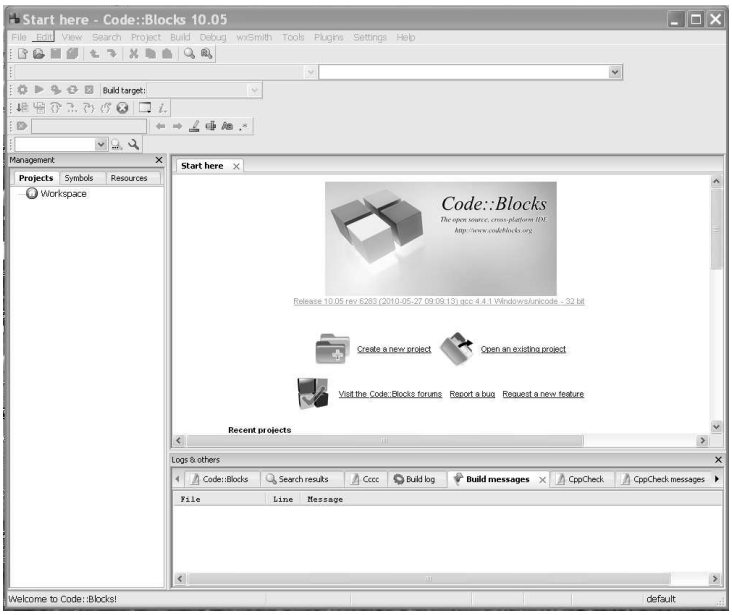
In the first C lab you will be introduced to the following concepts:

- 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 the graphics and music libraries.

1.4 Introducing the IDE

The Code::Blocks GUI comprises all the main features of an IDE. It looks something like this:



Locate "Code::Blocks" on your computer. This can be found by going to the "Start" icon on your Windows taskbar, then select "All Programs" and look for "Programming Tools". Select "Code::Blocks" and you should see something like the image shown in figure above.

- On the top of the window we see the familiar menu bar which has all available options for the software.

- Below we see various toolbars. Make sure that the compiler toolbar is visible. (If not sure go to "View" → "Toolbars" and make sure that the "Compiler" option is checked). The compiler toolbar has important functions that we need to select, compile and run our programs. These functions can also be selected through the "Build" menu.
- On the left of the window we see the "Management" panel. From here we manage our project and all related files.
- The middle part of the screen is split into two panels. The top panel is currently displaying the start panel for Code::Blocks, however this panel will display the text editor when a file is open/selected. The bottom panel, "Logs & others" comprises various tabs each containing important information about the output of various operations. An important tab is "Build messages" which outputs the state of the compilation process (all errors will be displayed here).

1.5 Creating a Project

In Code::Blocks the highest level of organization is a "Workspace", which can contain a number of "Projects". Projects are collections of software components. There may be dependencies between these projects. For instance projects may contain a collection of useful software components that are used by a number of other projects. The information about which projects depend on which others is stored in the workspace.

1. Now we are going to create a project. There are two ways of doing this. If Code::Blocks has just been started you can just click on the link "Create new project" or alternatively you can go to the menu bar and select "File" → "New" → "Project".
2. Select "Empty Project" and press "Go".
3. Enter a title for your project (Do not use spaces when naming a project). A suggestion for this is "helloworld".
4. Choose the folder to create the project in. Please note that Code:Blocks will automatically create a folder with the same name as your project name, in which all related files will be placed.
5. Make sure that the resulting filename has the desired name and location. Press "Next".
6. Make sure your selected complier is "GNU GCC Compiler".
7. Make sure the Create Debug and Release configuration boxes are ticked.
8. Leave all other options set to their default values. Press "Finish". In a few moments you will see an icon appear in the "Management" panel with the name you gave your project.
9. Click on "File → "New" → "Empty File".
10. Add this new file to the active project (Choose "Yes" to the message box).
11. Enter an appropriate file name (e.g. main.c, helloworld.c for example). Use the .c extension to inform the compiler that this is a C program and (not C++ which has extension .cpp).
12. This file should belong to both the Debug and Release versions of the software. So make sure that the Debug and Release boxes are ticked. Press "OK".
13. Now in the "Manager" panel, in the "Projects" tab if you expand your project tree you should see the new file that you just created. The file should also be visible in the editor panel (top half of the middle area of the GUI). If it is not then double click on the source file (i.e. main.c or helloworld.c).

1.6 Targets

In projects it is often necessary to have different variants of the project available. Variants are called "Build Targets"[1]. They differ with respect to their compiler options, debug information and/or choice of files.

When creating a project in the previous section of this guide we checked some boxes labeled Debug and Release. Usually when software is developed, two versions of the program are created: a Debug and a Release version (the Debug and Release targets). The Debug version is the one that is used for the main development of the program and contains important debug information (often called debug symbols) which is helpful when testing the program. For further information you might like to read:

```
http://en.wikipedia.org/wiki/Debug_symbol
```

The Release version of the software is the one that is given to the end user and has all debug symbols stripped out. This means that it will run much quicker.

So projects in Code::Blocks can comprise many different targets. Each target can have its own settings and its own files. This gives us the ability to have many different programs (targets) inside one project and keep all related files (source files, header files etc.) conveniently organised. This idea is fundamental to the creation of the "clab project" that we will be using for the much of this course. For now let us focus on our newly created project and the two different versions of it that we can create (Debug and Release).

1.7 Writing Code, Compiling and Running the Program

1. Make sure your main program file is open in the text editor and type in the following lines:

```
#include <stdio.h>

 int main (void)
{
   printf("Hello World\n");
   return 0;
}
```

2. Now in the compiler toolbar make sure the selected target is "Debug" and hit the "Build and run" button. Look in the "Logs & others" panel in the "Build log" tab as the compilation process executes.

3. Everything should compile fine and a command window will pop up displaying the output of our program: "Hello World". Below that there is an empty line and then some information about the process (our program running on the OS) and the execution time.

4. Press any key to close the command window and return to Code::Blocks.

5. Note that the information that we get in the command window is also displayed in the "Build log" tab.

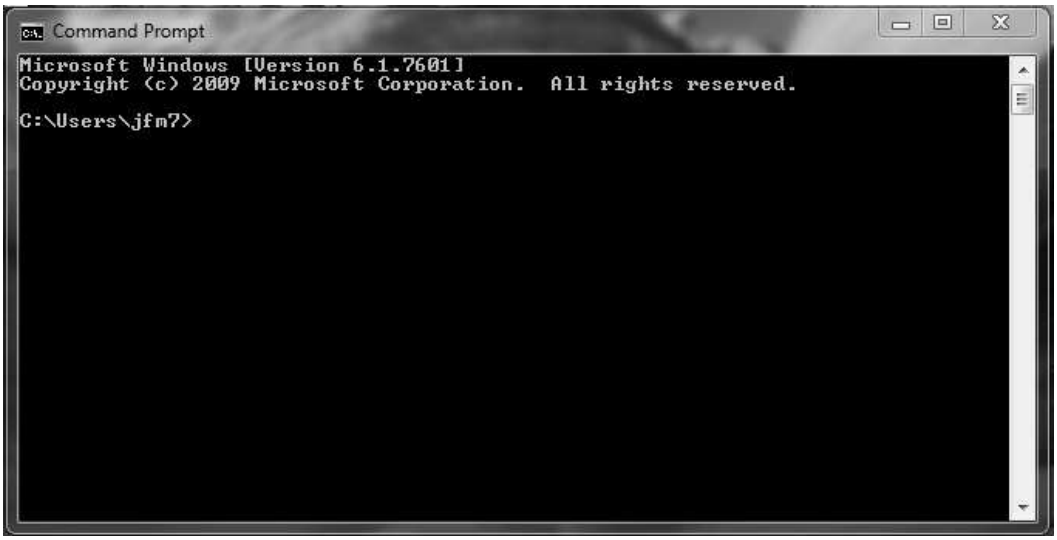
6. Now repeat the process but this time choose Release as your target.

Now close the Code::Blocks IDE.

At this stage we have compiled two different versions of our program; a debug and a release version. Open windows explorer and navigate to the location of your project. Inside your project folder there is a folder called "bin" (short for binaries) and inside that folder there are two folders named "Debug" and "Release"; each contains the corresponding target. Both targets are files with a name that is the name of your project (e.g. hello-world) and the extension ".exe". These files are executable (they are machine code) and you can run them. There are a couple of ways of doing this.

1. You can double click the file icon. If you try this you will, very briefly, see a small back window appear (like the one that opened when you chose "Build and run" in the Code::Blocks IDE.
2. The other way to run your executable is to use a MS OS tool called "Command Prompt". To locate this tool you need to go to the "Start" tab on the Windows taskbar (at the bottom of your screen). Select "All Programs" and then choose "Accessories". You should see "Command Prompt" on the list (a small black icon). Select this and a black window will open.

The "Command Prompt" should look something like this:



In this example the "Command prompt" window shows the path:

```
C:\Users\jfm7
```

When you do this it will show a path corresponding to the computer you are using now. You need to navigate to the folder where the executable program is located. The simplest way to do this is to go back to windows explorer window that shows your executable program and copy the path to it. This can be done by clicking in the top menu box (this shows the path) and selecting "CTRL C" which will copy the path. Now in the "Command prompt" window, type "cd" followed by a space and right click and select "Paste". This will fill in the path after the "cd". Now press "Return" and now the "Command Prompt" will be ready to run the program. Type the name of your program (e.g. "hello-world" you don't need to type the extension ".exe") and then press return. Now you will see the message "Hello world" and the window does not disappear!

This procedure may seem tedious but actually this is often how programmers run their C program executable files. If you want to give a friend a program you have written (assuming they are going to run it on a computer using the same operating system type (i.e. Windows), you can give them the .exe file and they should be able to run it. Note that sometimes when you do this the computer will complain it cannot find a .dll file, in which case just find it on the internet and copy it to the folder where the executable is. So, once you have the .exe file you do not need the IDE, you have a standalone file that can be run, so your friend would not need a C compiler or the Code::Blocks IDE. This is important to remember.

[1]For further information go to "Help" → "CodeBlocks" and read section "1.4 Create projects from Build Targets"

1.8 Using the "clab" Code::Blocks Project

In the previous sections we went through the process of creating a new project containing only one source file and no other dependencies. The exercises in this course and the program you are going to develop as your final assignment, comprise more than one file and depend on third party libraries (allegro and portmidi in particular) with which we have to link. Because the process of setting up a project using these libraries is lengthy (and essentially the same for each project that contains these libraries) we have created a ready made project for you, called "clab", that contains all necessary files (source files, header files, libraries, resource files, documentation) for completing the exercises of this course. It is recommended to use this project for creating your own programs too. Doing this will keep all files organised and will eliminate the need to create and set up a new Code::Blocks project every time you want to write a new program. Another advantage of having a single project with multiple targets is that you can just copy the main project folder anywhere and you are sure that all your work is copied and nothing is left behind. So for example you could run this project from a USB drive that you always carry around and use it on different machines (assuming that Code::Blocks and MinGW are installed and properly set up on the machine that you run the project).

If you later want to create a separate project for any of the programs you can always do that by creating a project from a target; this is discussed in an appendix on the Code::Blocks IDE.

1.9 Getting the "clab" Project

- Go to the Electronics Department internal web pages. Locate "DSE" below Information for Taught postgraduates" and click it '. Then click on "Modules". Then click on the link to "C Programming for MSc ". If you scroll down the page to the section "Code::Blocks project required for Lab exercises" just below that you will see "CLab code for Windows (available)". Click on this link and choose "Save As". Save the file "clab.zip" to the Department of Electronics network drive called "M:" It should be visible on the left-hand side of the "Save As" window. Now right click on the file "clab.zip" and choose "Extract All...". Once the unzipping process has finished you should have a folder:

  `M:\clab`

1.10 Examining the contents of the "clab" folder

- Using windows explorer, navigate to the "clab" folder and have a look inside. You should see the following a list of folders which includes the following:
- bin (folder) : This folder contains all compiled executables. This is where the executable programs are created.
- include (folder): This folder contains the allegro, portmidi library header files and the wrapper files `graphics_lib.h`, `amio_lib.h`.
- lib (folder): This folder contains the allegro and audio/midi libraries (.a and .dll files).
- obj (folder): This folder contains the object files that are produced after the compilation process. This is of no interest to us at the moment.
- .objs (folder): This folder contains more object files
- src (folder): This folder contains all the source code files. For the exercises each file corresponds to a different program (target), so each file contains a main function. Your program might contain more than one source file. We will see how to add multiple files to a target later on.
- data (folder): This folder contains files that the allegro graphics uses to define the fonts that can be used when writing text output to the graphics window.
- clab.cbp (file): This is the Code::Blocks project file.

When writing your program you will be adding files to the "include" and "src" folders. This is a typical folder structure in open source software and is a convenient way to organise your files.

Often there are a few other files that are related to the project (e.g. clab.depend and clab.layout) which need not concern us at this point.

1.11 Settings of the "clab" project

The clab project has been set up in such a way so that the allegro, portaudio and portmidi libraries are included and are available for all targets (programs) that need them. So you don't have to download anything extra or perform any more set up steps to use these two libraries. Everything is there and ready for you to use. Each exercise of this course is a different target, with its own settings and corresponding source files. All you have to do is pretty much select your target, compile and run. Of course the source code must be complete in order for the compilation process to succeed; but completing the exercises is your job. Sometimes a complete working source file will be given and you have to expand on it by writing more code. Just follow the individual lab scripts. Before doing any programming we should take a look inside the clab project and see how things are set up. Understanding and getting comfortable with the project settings is important as you have to create your own targets later on in the course.

Start the Code::Blocks IDE from the Start menu on the far left hand side of the Windows taskbar. You will see a window in the centre of the Code::Blocks IDE entitled "File associations". Just click "OK". This will automatically associate C programs (and other files) with the Code::Blocks IDE. Now click on "Open an existing project". A window entitle "Open file" will appear. You will see a file called "clab.cbp" (with a Code::Bocks icon next to it). Left click on the file and then left click on "Open". You will see three window panels. The left panel is called the "Management" panel and it shows you a "clab" icon and two sub-folders called "Sources" and "Headers". The Sources folder contains all necessary source files for completing this course.

1.12 Building your target

Having covered the basics of our development tools, we can continue with the process of building our programs. The building process consists of several stages. To understand this process better you could consult:

```
http://en.wikipedia.org/wiki/Software_build
```

Now focus on the "Management" tab and click on the "+" symbol by the folder "Sources". You will see a list of C programs in the 'src" folder. Double click on the file "graphics1.c". The IDE will open an editing window and show you the program, the first line of which is

```
/* please delete this line */
```

Go ahead and delete this line. Click on the save file icon.

Now look for the green triangle like an arrow on "Compiler" toolbar in Code::Blocks. Near it you will see a drop-down menu. Click on this and select "graphics1". This drop-down menu lists all the targets. This is how you select what program you want to compile and run.

Now you can try to build the program associated with the C source code file "graphics1.c". Building means invoking the compiler which translates your C program (in this case "graphics1.c") into the machine code program that runs on the processor chip inside the computer. To start the build process first locate the yellow gear button on the compiler toolbar and then click on it. Watch the output of the build log during the build process. In the build log we see that the build process first goes through compilation stage, then linking stage and finally reports the output size of the executable. The two blue lines report the status of the terminated process and time of execution followed by the number of errors and warnings (which should be zero in this case).

Now using **Windows Explorer** navigate to "clab\bin\Debug" and you should see the file "graphics1.exe". The file is the product of the build process and its name is taken from the corresponding

option in the build targets tab in the project/targets option window (output filename setting). You could run this program by double clicking the file. But don't do it just yet; we will run it through Code::Blocks.

1.13 Running the program

Focus again on the compiler toolbar and now execute the run command (green triangular "play" button). The application should run and you are presented with a couple of windows. One is the black command run window (like "Command prompt"), the other is a graphics window that the program opened. It will show a small red circle.

Make the command run window active (by clicking on the top menu bar) and then type return. The graphics window will disappear.

Press any key to close the command window when you finish [2]

The third icon in the compiler toolbar runs the two steps above together in succession. All the commands found in the compiler toolbar can also be run through the build menu item of Code::Blocks.

1.14 Looking in more detail at a C Program

If you haven't already done so close any program that is currently open (i.e. close the editing window).

Double click on the icon for the source file "lab1.c". The C language *source code* will appear in the editor window.

Select "lab1" as the target in the "Build target" drop-down menu on the Code::Block menu bar.

The source code for this program is shown 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 */
```

[2] If you don't want to close the command window by pressing a key every time you exit your application then you will need to change the settings.

```
    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 Code::Blocks when you type a comment the editor will understand and colour it grey. 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;
```

A statement of this type 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`. You might like to think of the '=' sign in C as meaning "becomes equal to", so x = 3; would be read as "x becomes equal to 3" or even "set x to 3".

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, when 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 for help.

1.15 Compiler errors, warnings and debugging the lab1.c program

We are now going to use investigate how the compiler tells us about various things wrong with a program we are working on.

- Press the "Build and run" button. Look in the "Logs & others" panel in the "Build log" tab as the compilation process executes.
- Everything should compile and a command window will pop up displaying the output of our program.

When there are problems with the C source code, the compiler will generate errors and warnings. These will appear in the "Build log" tab.

- **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!

Now if you looked at the "Build log" when you built the program it will have given a warning. This is shown below. The text about the path to the folders has been removed. The first line appears in black and the warning appears in dark blue.

```
In function 'main':

15   warning: variable 'average_speed' set but not used
[-Wunused-but-set-variable]
```

This warning is nothing to worry about. The compiler is just pointing out that although the integer variable `average_speed` has been declared and a calculation using it has been carried out (see towards the end of the program), nothing is done with this (i.e. we have not printed out the value or used it in a further caculation). Effectively, the compiler is reminding you about that, as you probably want to do something with the value. It also points out that it thinks the problem is at line 15 of the program, which is where `average_speed` is declared.

Now, let's deliberately generate an error, just to see what happens when we try to build the program.

Remove the "`;`" symbol at the end of the line in the program which looks like:

```
distance_to_tokyo = 9720;
```

So now it should look like:

```
distance_to_tokyo = 9720
```

Now build the program.

Now it will say in red:

```
21  error: expected ';' before distance_to_airport
```

It will also show a red square on the line 21 on the program listing.

Now correct the error, save the file and Build and run the program again.

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!

A 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.

position the cursor over line 29 in "lab1.c", which should be the line

```
time_to_fly = (distance_to_tokyo - distance_to_airport) / speed_of_plane;
```

Now right click the mouse and select "Toggle breakpoint" (first option on menu). After you have done this a red circle will appear by the line number. A breakpoint is a point in Debug mode where the program will stop and wait for further instructions from the user. Now go to the Debug menu (at the top of the IDE) and select "Start" (this can be done through a function key shortcut "F8"). The debugger will now run through the program and stop on the breakpoint. Once again return to the Debug menu and select "Debugging windows" → "Watches". A window will open in the IDE with two items in the window: "Local variables" and "Function arguments". Each has a small plus sign on the left. Click on the plus sign next to "Local variables". When you do this you will see a list of variables that are used in the program. The first four look like:

```
distance_to_tokyo = 9720
distance_to_airport = 120
speed_of_plane = 1200
speed_of_car = 100
```

These variables have all been given these values by the C program as it has executed all statements before the breakpoint. After, the above variables it lists the other variables in your program:

```
time_to_fly = 4200702
time_to_drive = 2293592
time_to_tokyo = 4200608
average_speed = 2009288258
```

These have not received values from the program as the statements that define the values these variables take has not been reached yet. Consequently these variables have more or less random values. Don't worry if these uninitialized variables have different values to those above as they are processor dependent.

To begin stepping through your program starting at line 29, press "F7", and you will see a small yellow triangle appear next to the line number. This indicates which line the debugger will execute next. Repeating this will advance the program execution another line. The debugger will skip lines that it cannot execute, such as comments and variable declarations.

As you step through the program you should be able to see the values change in the watch window. When you reach the last line of the program "return 0;" stop the debugger via the debug menu.

Exercise 1.2: Using the Debugger

Remove the breakpoint on line 29 and create a breakpoint on line 15. Now 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 for help.

1.16 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`". The header name 'stdio' is short for 'Standard Input and Output' and it contains ready-made functions for getting information from the keyboard and to the screen, as well as to and from files. In this case 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 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`*]$^{\mathbf{n}}$ `);`

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` long floating point used for 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`.Note that's a lower-case letter 'L' and not a digit '1' in 'lf'.

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 for help if you do not understand how to do this.

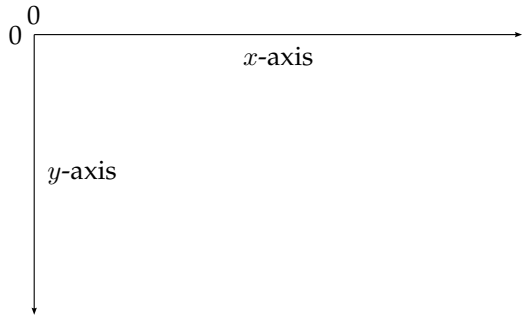
1.17 Graphics Option

There are two kinds of optional sections in the C Labscripts. One involves graphics and the other audio or midi related exercises. You are strongly encouraged to attempt both. However, if you are on a Music Technology related degree you might like to try the graphics exrecises after you have already completed the music options. On the other hand if you are not on a Music Technology degree you should try the graphics exercises first and then the music ones. The section for the music option begins after this one (Section 1.18).

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.

1.17.1 A Simple Graphics Program

In "Build target" select "graphics1" and also open the code listing `graphics1.c`.

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;

    /* choose red pen colour */
  setcolor(RED);

    /* draw a circle at x_position, y_position
    with radius 10 and line thickness 2 */
    circle(x_position, y_position, 10, 2);

    /* move the contents of the screen buffer to the display */
    update_display();

    /* 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:

```
circle(x_position, y_position, 10, 2);
```

The circle has its centre at (`x_position`, `y_position`) and has a radius of 10 pixels. The fourth parameter (after 10) is 2. This determines the thickness of the pen used to draw the circle.

Function Reference: `circle` — Draws an unfilled circle in the graphics window

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

The `circle` function draws an outlined 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. The thickness of the circle perimeter is specified by *`thickness`*.

For example

```
circle(200, 300, 20, 5);
```

will draw a circle with its centre at (200, 300) with a radius of 20 pixels and line thickness of 5.

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

The next line is:

```
update_display();
```

Before we explain the next line, it is important to realise that all graphics functions actually write to an internal graphics display, called a screen buffer. We need to call the function `update_display` to move the contents of the screen buffer to the "live" display. This may seem a little tedious at first, but it allows us to do something that is very useful. We can create the appearance and possible movement of multiple graphics entities.

Function Reference: `update_display` — Moves the contents of the screen buffer to the screen

```
update_display();
```

The `update_display` makes the contents of the screen buffer "live" by moving it to the screen memory. It should be called after using graphics functions when you want the drawn elements to appear.

`update_display` 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 1.5: A First Look at Graphics

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

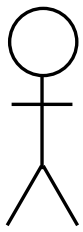
Some things to investigate for yourself:

- 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.

1.17.2 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, thickness);
```

e.g.

```
    line(100, 150, 200, 250, 2);
```

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. The last argument defines the line thickness.

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

Exercise 1.6: 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, 2);
```

This defines the line with respect to the position of the centre of the stick man's head which is given by `x_position` and `y_position`. If you define all the drawing elements of the stick man in this way, you will be able to change the position of the stick man by just changing the values held by the variables `x_position` and `y_position`.

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

Remember: Once you have created your executable program you can run it outside of the Code::Blocks IDE. To do this, locate the "clab" folder and within it, the "bin" folder. Inside that you will see there is a folder called "Debug". Inside this you will see a file called graphics1.exe. If you double click this icon it will run graphics1 outside of the IDE.

1.18 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. Later on we will use another software librray called "portaudio" which will allow music files

(such as .wav) to be read into a program and manipulated.

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*.

1.18.1 A Simple Music Program

In "Build target" select "music1" and also open the code listing `music1.c`.

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". Before you execute any midi program you **must** place headphones or speakers into the headphone socket of your computer. If you don't you will get a strange looking error message which looks like this:

Portmidi found host error. There is no driver installed on your system

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 <amio_lib.h>

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

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

/* 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);

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

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 <amio_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 is:

```
    midi_start();
```

This is essential and initializes the midi interface so that sounds can be played.

Function Reference: `midi_start` — Initializes the midi interface

```
midi_start();
```

`midi_start` is defined in `amio_lib.h`.

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. MIDI Channel 10 is often reserved for a drum-map, and so it reacts differently to all other channels, by allowing individual MIDI pitches to activate individual drums. But more on this later.

The musical pitch is specified by a number. Each semitone (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 `amio_lib.h`.

If we turned the note off again straight away it would play for such a short time that you might not even 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 `amio_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.

After all the midi notes have stopped we need to reset the midi interface using the command:

```
midi_close();
```

Function Reference: `midi_close` — Closes the midi interface

`midi_close();`

`midi_close` is defined in `amio_lib.h`.

Exercise 1.7: 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!

N.B. If you notice that your last note is being 'cut short' it is probably because the midi_close()function is being called before your note has had a chance to fully play. If this is the case, add an extra pause (of, say, 1 second) just before the midi_close()function call, and see if this cures the problem.

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

1.18.2 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 1.8: 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!

If you are not sure of what musical intervals are required for the above chords discuss these amongst yourselves, or type "forming chords" into a search engine.

1.18.3 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 `amio_lib.h`.

Exercise 1.9: Changing the Program

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

Exercise 1.10: 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 for help.

1.19 Summary

Now you have completed this lab you should know how to use the Code::Blocks IDE to build and run C programs. You will also 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.