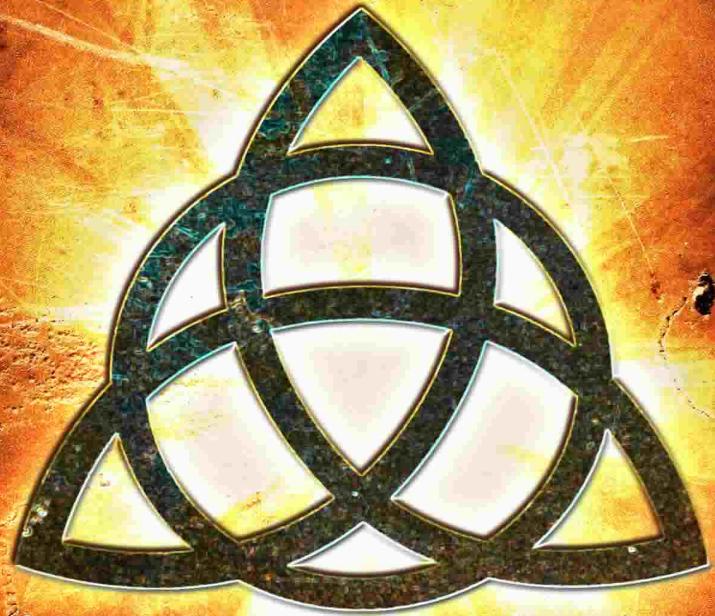


PROGRAMMING ARCANA



2011

ANDREW CAÍN

Welcome neophyte! So you seek to master the arts of Magic. Well, you have come to the right place. Here we will give you the opportunity to explore magic, to understand its working and use. Before we start I have one piece of advice for you, practice, for it is only through practice that you can hope to learn these secret arts.

Welcome to the Programming Arcana¹, a book about learning to program. This book contains a number of chapters that take you from knowing nothing, or little, about programming to a position where the mysteries are revealed. By the end of the material you will be able to create your own programs and you will be ready to start learning other programming languages and approaches to software development.

This book is divided into a number of chapters, each of which introduces you to a programming task and the arcane knowledge that must be attained to understand how the task is accomplished. As with any arcane knowledge there are special terms that are used by those who know its secrets. In each chapter you will be introduced to the terms you need to understand in order to perform the current task. This will provide you with the tools you need to describe programs to other software developers, and will help you understand how the structures within your programs work to achieve their goals.

Before we start getting into details let us have a look at how you can use this book to help you become a master of the arcane arts of programming.

Book Overview

This book is designed for people who want to gain an in depth knowledge of software development. It assumes that you are familiar with computers and how to use them, but does not require you to have any prior programming experience. You should feel comfortable using a computer, and be prepared to start looking at how it works in more detail.

There are a number of different ways in which texts can present their material. In this book, the content is presented using a concept first approach. This approach places the programming concepts as the central focus, with details of syntax being secondary. The logic behind this is that you can learn new syntax if you understand the concepts, whereas it will be difficult to be productive if you understand the syntax but do not understand the concepts behind this syntax. To this end, this book gives you the choice to study either the C++ or Pascal programming language. Both languages are very capable, and offer different advantages and challenges.

The chapters of this book build upon each other. Each chapter covers a new group of concepts, that will expand your programming capabilities and enable you to create larger and more capable programs. The layout of each chapter is the same, with the concepts having the main focus. Each chapter is laid out in the following order:

1. **Concepts:** The first part of the chapter introduces the concepts that will be covered. This is done in a language neutral manner, with the focus being on how to think about the tools being presented. This will introduce each concept with an illustration, and accompany this with explanatory text.

¹Arcana is defined as secrets or mysteries, Wiktionary defines it as “specialized knowledge that is mysterious to the uninitiated.”. This fits well with the idea of programming, and we think it is a cool word to describe the *magic* of programming!

-
- 2. **Using the Concepts:** The next section shows how these concepts can be used to achieve a task. This task will try to cover all the concepts presented in a practical manner. This is done in a language neutral way, and talks about how to use the concepts to achieve a goal.
 - 3. **Languages:** The next two sections present the syntax you need to use these concepts in **C++** and **Pascal**. You should use this as a reference, and can read this alongside reading about how to use the concepts.
 - 4. **Understanding:** Following the language specific details, the next section explains in detail how the concepts work within the computer. Use this to get an understanding of how the concepts work in more detail. This section will show you illustrations of what is happening within the computer when your code is running.
 - 5. **Examples:** Each chapter will have at least one example showing you how these concepts can be used. This will include the code, and some explanatory text to discuss what is being presented.
 - 6. **Exercises:** The exercises allow you to put into practice what you have read about. You cannot learn to program without practice. These exercises are a good start, but you should try to come up with your own project so that you can test out these new concepts on something you are interested in working on.r

Which Language?

A programming language defines a set of rules that determine how you write the code for your programs. Each language defines its own rules, and so there is always the temptation to focus heavily on these details and place the overriding concepts in second place. We believe that when you are starting to learn to program, a good understanding of the programming concepts is far more important than the details of the programming language you are using. This book is not an in depth study of either the C++ or Pascal language, it is a book about learning to program.

To really learn these concepts well you will need to practice putting them to use. This will require you to use a programming language. Each chapter will provide you with enough information to put the concepts to use in either the C++ or Pascal language. So the main question you need to answer now is which language will you use?

Both C++ and Pascal are very capable languages. Pascal was designed as a teaching language, which means that it does make it a little easier to see how the concepts being covered apply to your code. C++, on the other hand, is a commercial language designed for professionals to build programs. This is both an advantage and disadvantage for C++. The advantage is that the language is widely used in industry, but the disadvantage is that it lacks the clarity that is offered by Pascal. Remember that this is only your first programming language. A professional software developer will know many different languages, and by the end of this material you will be equipped to learn many new languages.

C++

C++ is a very flexible language that is widely used in industry, though its syntax is more of a challenge to learn.



Pascal

Pascal is a powerful languages that was designed to teach programming. It is used in industry, but not to the same extent as C++.



Formatting

This book has a number of visual formatting guides. These are designed to help you navigate through the material easily.

Pseudocode

Text formatted in this way relates to an algorithm description. This will describe the steps that need to be performed in a way that is language neutral and can be applied to C++, Pascal, and possibly other languages. 

C++

Text formatted in this way relates to the C++ programming language. If you are going to use C++ you need to pay attention to the text in these boxes, otherwise you can skip over them. 

Pascal

Text formatted in this way relates to the Pascal programming language. If you are going to use Pascal you need to pay attention to the text in these boxes, otherwise you can skip over them. 

Note

Text formatted in this way covers notes related to the current concept or illustration. This book makes extensive use of notes to capture important points, so do not skip over these. 

The language sections of each chapter also add markers to each page to clearly mark where they start, and where they end. If this is your first programming experience you should stick with one of these languages, so you can skip the pages that are marked as being for the other language.

Programming Jargon and Concept Taxonomy

Programming has a lot of its own jargon. As you learn to develop software it is also important that you start to learn this *special language* that software developers use to discuss their programs. You will find that this terminology is used in many places. It is used in programming texts, in discussions between developers, in discussion boards, blogs, anywhere that developers are discussing software development. Having a clear understanding of this terminology will help you make the most of these resources.

The concepts in this book are closely linked to this programming terminology. To help you understand each concept, we have classified them using one of the following categories:

- **Artefact:** An artefact is something that you can create in your code.
- **Action:** Actions are things that you can *command* the computer to do.
- **Term:** These are general terms, used to describe some aspect.

When you are reading about the different concepts in this book you can use these classifications to help you think about how you may use the knowledge you are gaining.

Artefacts: Artefacts are things that you create in your code. Programming is a very *abstract* activity, you spend most of your time working with concepts and ideas. You write text, code, that will create things within the computer when your code is run.

When you are learning about a new kind of artefact come up with ways of visualising it. It is a **thing** that you are creating with your code. Try to picture the artefact within your code.

These artefacts are the basic building blocks that you have to work with. You need to be very familiar with them, how they work, and what you can do with them.

Actions: Actions get the computer to perform a task. Your actions will be coded within the **artefacts** that you create, and will define how artefacts behave when they are used. The actions themselves are commands that you issue to the computer. They are executed one at a time, and each kind of action gets the computer to carry out certain tasks.

When you are learning a new kind of action you need to see what this action does. To start with you should play with it, test it out, and see if you can understand what it is getting the computer to do. As you progress you need to start thinking about how you can sequence these actions so that the computer performs the tasks you want it to. There are only a very few kinds of actions, so it is by combining them that you can get the computer to do what you want.

Terms: The remaining terms are words that developers use to explain concepts. These are not things that you create, or actions that you request. These are just words that you need to *know*.

When you are learning a new term you need to try to commit it to memory. Memorise the terms, try to use them in sentences, explain them to others. All of these tasks will help you understand, and remember these terms.

Advice

If you want, or need, to learn to program then you can not do this just by reading a book, even one as magical as this. Learning to program requires practice. This book is designed to give you the concepts you need in order to understand how to go about creating your first programs. To really understand these concepts you need to apply them to the creation of your own programs.

When you are getting started, programming can appear quite daunting and the tools you use can be unforgiving. Work through these initial challenges, and with practice you will be able to overcome them. Once you have some success there is nothing better than seeing a program you created running on a computer. You have brought the machine to life, getting it to perform a task the way you want it performed. Once you get a program working it can become easy to get hooked and working on new features and functions becomes a real joy. The greater the challenge the program offers, the greater your sense of achievement when you see the working product in operation.

Other people are the best resources to help you get over these initial challenges. Fellow students studying this material can provide you with support, and a chance to discuss the challenges you are facing. Teaching staff are also a good resource when you are really stuck. If you do not have access to anyone who can help, use discussion boards and websites. Getting the right help will make a large difference to your learning experience.

Remember that you will need to study this material. That is not just reading it, but thinking and reflecting on what you have read. Try to think about each of the concepts, and how they relate to the other material that has been presented to you. Try to design and build your own programs with the material you are learning. If you do think deeply and apply the concepts to programs you create, you will eventually get the light bulb moment when things become clear and programming can become truly joyful.

0

Getting Started

 Magic requires both knowledge and tools. Our first lesson will uncover the tools of the Magi. Tools that you will need to use to practice magic. Here, take this ancient wand this orb and cauldron. Each of these tools is essential to the working of even the most basic spells. Now lets see if you can wield that wand. Take it in your hand like this, and ...

Software development requires both knowledge and tools. This first chapter introduces key details of your computer, how programs are created and run, and the tools you will use to build your own software.

When you have understood the material in this chapter you will be able to create programs from source code, run your programs, and identify cause of errors when creating or running your program.

Contents

0.1 Learning Focus	6
0.2 Concepts Related to Getting Started	6
0.2.1 What are programs?	6
0.2.2 How can you create a program?	6
0.3 Understanding Getting Started	6
0.4 Secrets of the Magi – Advanced Shell Usage	6

0.1 Learning Focus

Programming is about providing instructions to get a computer to do the things that you want. In order to learn how to do this we need to start by understanding a little about the computer and how it works. This first chapter focuses on developing your understanding of how the computer works, how programs are created, and how they run. When you have successfully understood the material in this chapter you will be able to:

- Describe key components of your computer, and how they relate to program execution.
- Describe the environment within which programs execute, and how this can impact on programs working.
- Setup a computer to build and run programs.
- Run programs from the command line.
- Setup and use a code editor to create small programs.
- Compile programs from the command line.

Once you have achieved these learning outcomes the next chapter will help you start to explore the key concepts for the components within a program.

0.2 Concepts Related to Getting Started

There are two big questions we need to address here:

1. What are programs?
2. How can you create a program?

0.2.1 What are programs?

0.2.2 How can you create a program?

0.3 Understanding Getting Started

0.4 Secrets of the Magi – Advanced Shell Usage

1

Program Creation

 asting spells crafted by others is alright, but the true power of magic can only be realised by crafting your own spells. You have done well mastering the tools, so now let us turn our attention to the study of the arcane knowledge of spell craft. To create your own spells you need to know...

Compiling programs crafted by others is alright, but the true power of programming can only be realised by learning to craft your own programs. Chapter ?? introduced you to the tools you need to compile programs from source code, so now we can turn our attention to the study program creation.

This chapter introduces the artefacts used to create programs, and how you can code these using a programming language. You will start by learning to create simple programs that output information to the Terminal, and then we will look at making use of SplashKit to create some more interesting programs.

When you have understood the material in this chapter you will be able to write the code needed to declare a program, and within this program create your own custom procedures to perform simple tasks. You will be able to convert this code into an executable file using a compiler, and then run the program you created. You will have made those first important steps in your journey to master this arcane knowledge.

Contents

1.1 Program Creation Concepts	9
1.1.1 Program	10
1.1.2 Statement	11
1.1.3 Procedure Call	12
1.1.4 Procedure	13
1.1.5 Expression	14
1.1.6 Literal	15
1.1.7 Type	16
1.1.8 Identifier	17
1.1.9 Library	18
1.1.10 Comments	19
1.1.11 Procedure Declarations	20
1.1.12 Summary	21
1.2 Using these Concepts	22
1.2.1 Designing Output Test	22
1.2.2 Understanding Output Test	23

1.2.3	Choosing Artefacts for Output Test	23
1.2.4	Writing the Code for Output Test	24
1.2.5	Compiling and Running Output Test	26
1.3	Program Creation in C++	33
1.3.1	C++ Program (with Procedures)	34
1.3.2	C++ Statement	36
1.3.3	C++ Procedure Call	37
1.3.4	C++ Procedure Declaration	38
1.3.5	C++ Identifier	40
1.3.6	C++ Expression	41
1.3.7	C++ Literal	42
1.3.8	C++ Types	43
1.3.9	C++ SplashKit Terminal Output	45
1.3.10	C++ Comments	46
1.3.11	Some Common C Errors with Program Creation	47
1.4	Program Creation in Pascal	49
1.4.1	Pascal Program	50
1.4.2	Pascal Statement	51
1.4.3	Pascal Procedure Call	52
1.4.4	Pascal Identifier	53
1.4.5	Pascal Expression	54
1.4.6	Pascal Literal	55
1.4.7	Pascal Types	56
1.4.8	Pascal Terminal Output	57
1.4.9	Pascal Comments	58
1.5	Understanding Program Execution	59
1.5.1	Starting a Program	59
1.5.2	Calling the Write Procedure	64
1.5.3	Summary	70
1.6	Program Creation Examples	71
1.6.1	Seven Times Table	71
1.6.2	Circle Area	73
1.6.3	Shape Drawing	75
1.7	Program Creation Exercises	78
1.7.1	Concept Questions	78
1.7.2	Code Writing Questions: Applying what you have learnt	79
1.7.3	Extension Questions	80

1.1 Program Creation Concepts

Our first program is going to display some text to the Terminal. In this section you will be introduced to the programming artefacts and terminology you will need to use to create this program. This first step is important and will require you to have installed a C++ or Pascal compiler, see Chapter ?? ?? for instructions.

A programming **artefact** is something that can be created and used within your code. In this chapter we will look at creating programs, and using a number of other artefacts. The following artefacts will be covered in this chapter:

- **Program**: A program is a sequence of instructions that when compiled creates an executable file that a user can run.
- **Procedure**: A procedure is a named sequence of instructions that will get the computer to perform a task. When you want the task performed you can call the procedure.
- **Library**: The program can use code from other Libraries. These libraries contain reusable Procedures and Types.
- **Type**: A type defines how data is interpreted by the program. The programming language will support a number of basic types by default, and libraries can add other types.

In addition to these artefacts, you will need to understand some programming **terminology**. The following terms are discussed in this section:

- **Statement**: An **instruction** within the program.
- **Expression**: A **value** used in a statement.
- **Identifier**: The **name** of an artefact.

This section also introduces the following kinds of instructions. You can use these to get the computer to perform certain **actions** within your program.

- **Procedure Call**: The instruction to run a procedure.

We can then use these concepts, artefacts, and instructions to create a program that will write some text to the Terminal as shown in Figure 1.1.

```
dhcp7-206:program-creation acain$ ./OutputTest
Extended Hello World
1 + 1 = 2
Area of a circle with radius 3 = 28.27
dhcp7-206:program-creation acain$
```

Figure 1.1: Hello World run from the Terminal

1.1.1 Program

In most software projects the top level *artefact* you are aiming to create is a **program**. Within your software a program is a list of instructions the computer will perform when that program is run on the computer.

When you create a program in your code you should be thinking about the tasks you want the program to achieve, and the steps you must get the computer to perform when the program is run. These then become the instructions within the program, with each instruction being a **Statement** of what you want performed.

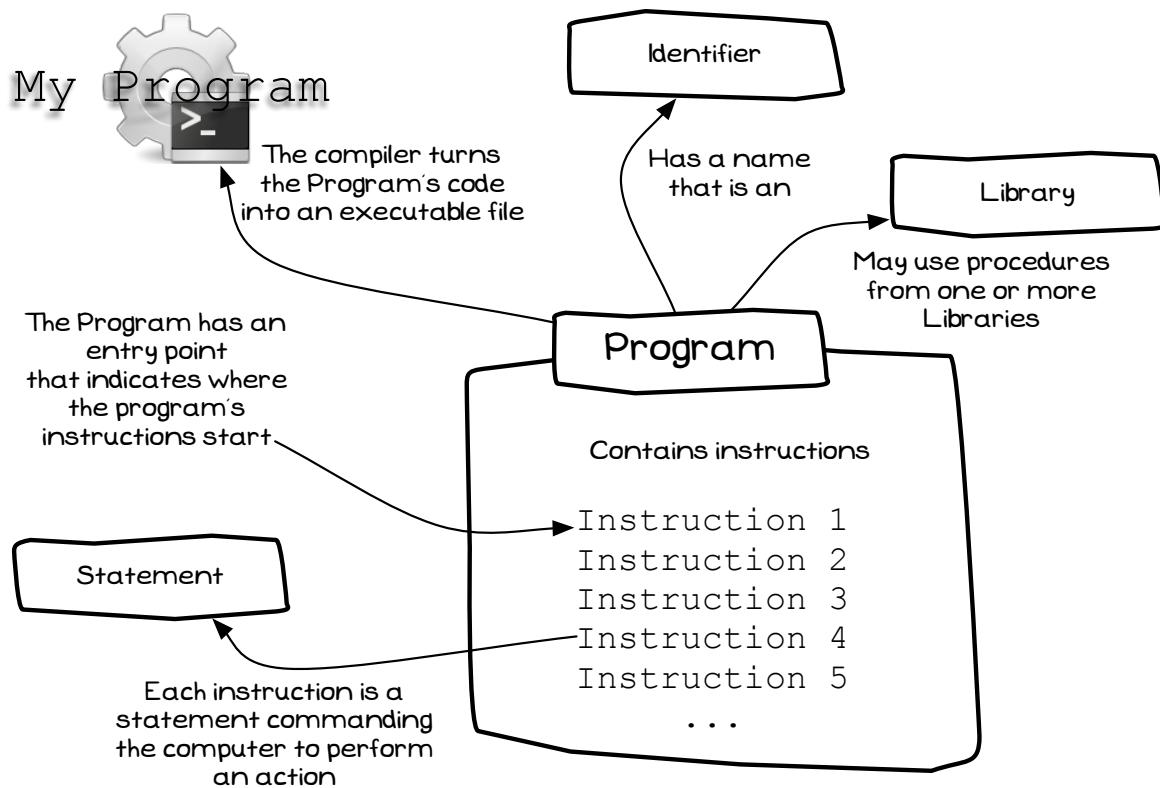


Figure 1.2: A program contains instructions that command the computer to perform actions

Note

- A program is an **artefact**, something you can create in your code.
- Figure 1.2 shows the concepts related to the program's code.
- A program is a programming artefact used to define the steps to perform when the program is run.
- You use a compiler to convert the program's source code into an executable file.
- By declaring a program in your code you are telling the compiler to create a file the user can run (execute).
- The program has an **entry point** that indicates where the program's instructions start.
- The name of the program determines the name of the executable file.
- Your program can use code from a **Library** or number of libraries.
- In programming terminology, an instruction is called a **Statement**.

1.1.2 Statement

When you are created a program you define the actions the computer will perform when the program is run. Each of these *actions* is coded as a **statement** within the program. This style of programming is known as **imperative** programming. Imperative means to give authoritative commands, and that is what we do in our programs. Our programs are lists of *authoritative commands*, statements, that *tell* the computer the actions it is to perform.

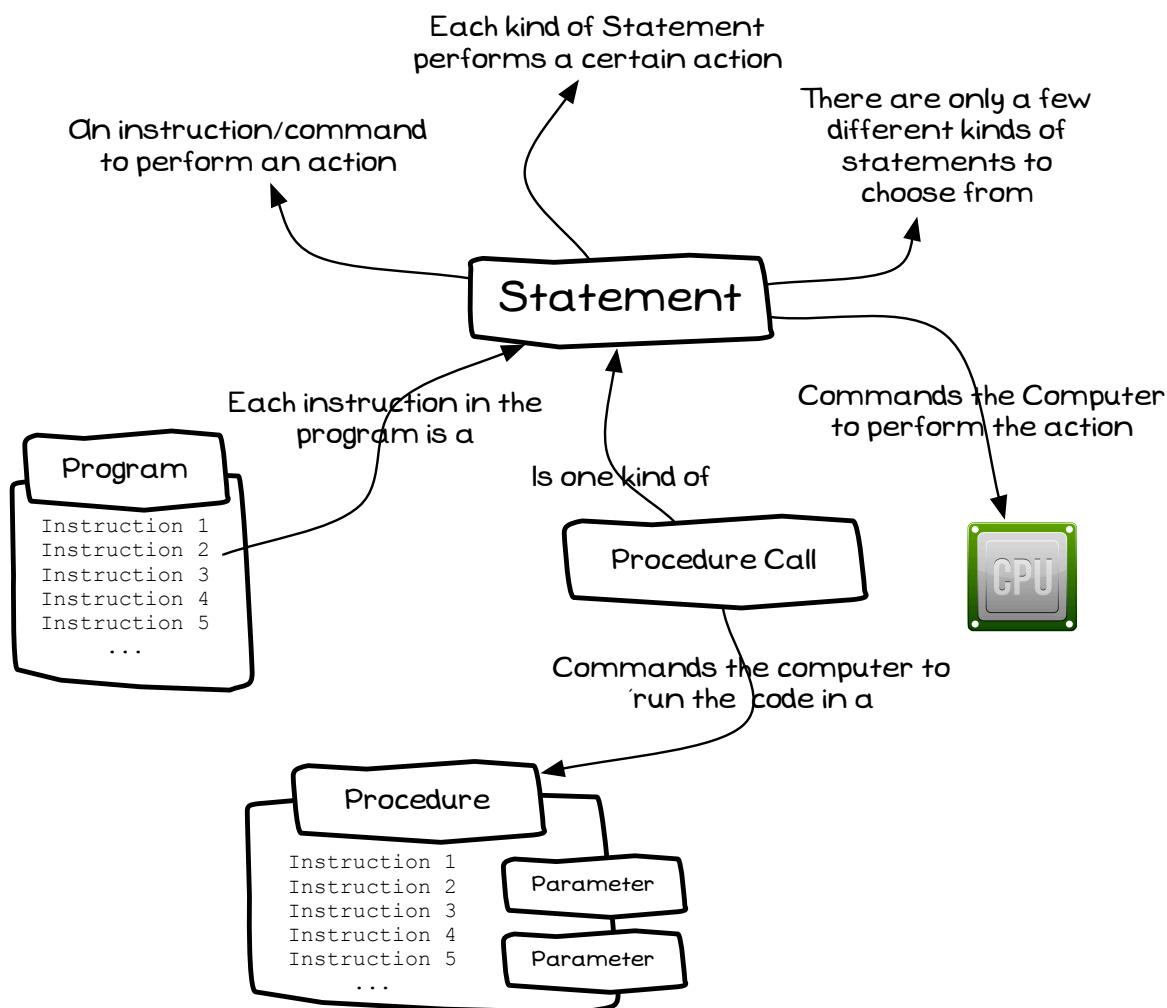


Figure 1.3: A statement is a command for the computer to perform an action

Note

- A statement is a **term** used to describe the instructions in your code.
- Figure 1.3 shows the concepts related to statements.
- A statement is a **command**, an instruction to perform an action.
- A **Program** has a list of statements that are followed when it is executed.
- There are only a few kinds of statements. Each statement has a defined set of actions the computer performs to carry out the *command*.
- A **Procedure Call** is a kind of statement that tells the computer to run the code in a **Procedure**.

1.1.3 Procedure Call

A procedure call is a kind of [Statement](#) that instructs the computer to run the code in a [Procedure](#). This statement uses the procedure's name to identify the procedure that must be run. If the procedure called requires some data, this data is *passed* to the procedure as part of the procedure call.

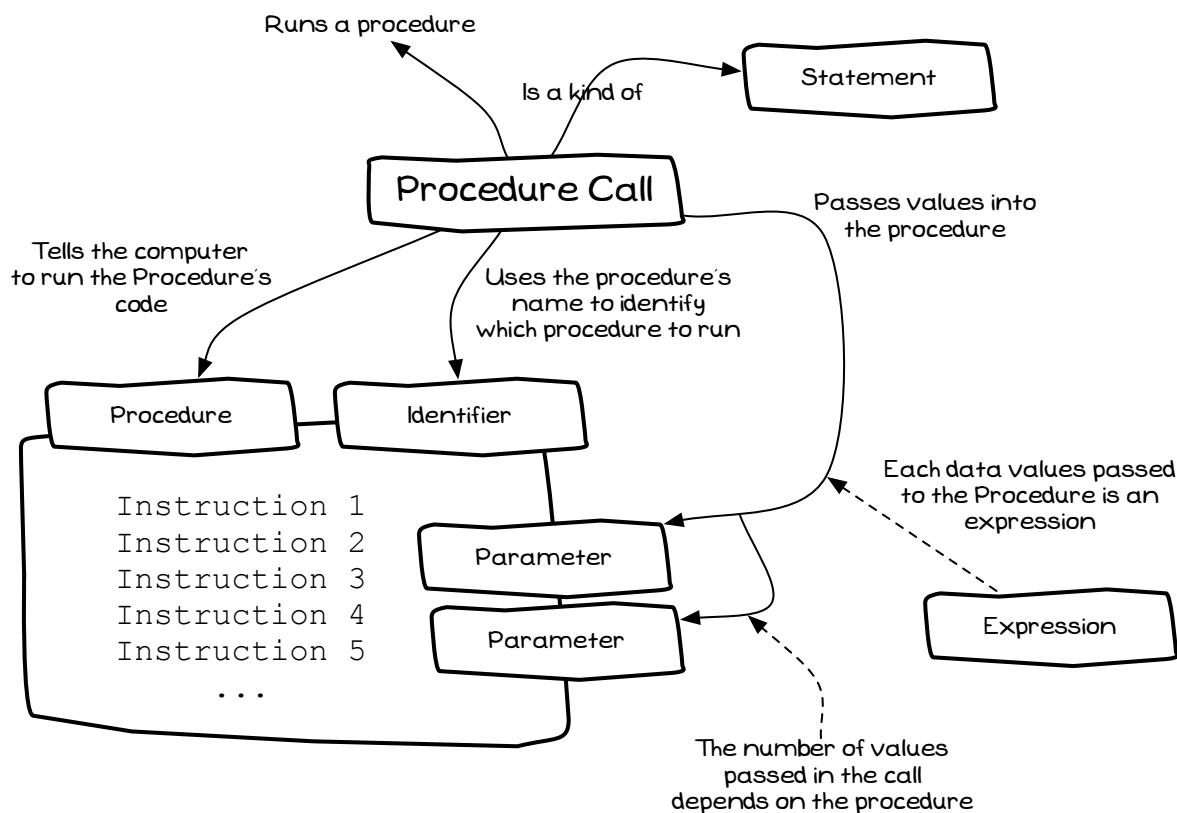


Figure 1.4: A procedure calls runs a procedure, passing in values for the procedure to use

Note

- A procedure call is an **action**, you can call procedures in your code.
- Figure 1.4 shows the concepts related to the procedure call.
- A procedure call is an instruction to execute a procedure.
- You can code a procedure anywhere you can code a statement.
- The [Identifier](#) indicates the [Procedure](#) to run.
- Data values passed to the procedure are coded using [Expressions](#).
- When the procedure's task is complete the program continues with the next [Statement](#).



1.1.4 Procedure

The computer is unintelligent, so performing anything meaningful requires a large number of instructions. Coding all of these directly in the program would be slow and time consuming. To avoid this programming languages offer the capability to group the instructions to perform a task into a **procedure**.

A procedure is a list of instructions that gets the computer to perform a specific task. When a procedure is called it gets control of the computer and instructs it to perform the steps needed. Often these steps require data, so the procedure may need to be passed data when it is called. When the procedure finishes its task, control returns back to the code that called the procedure.

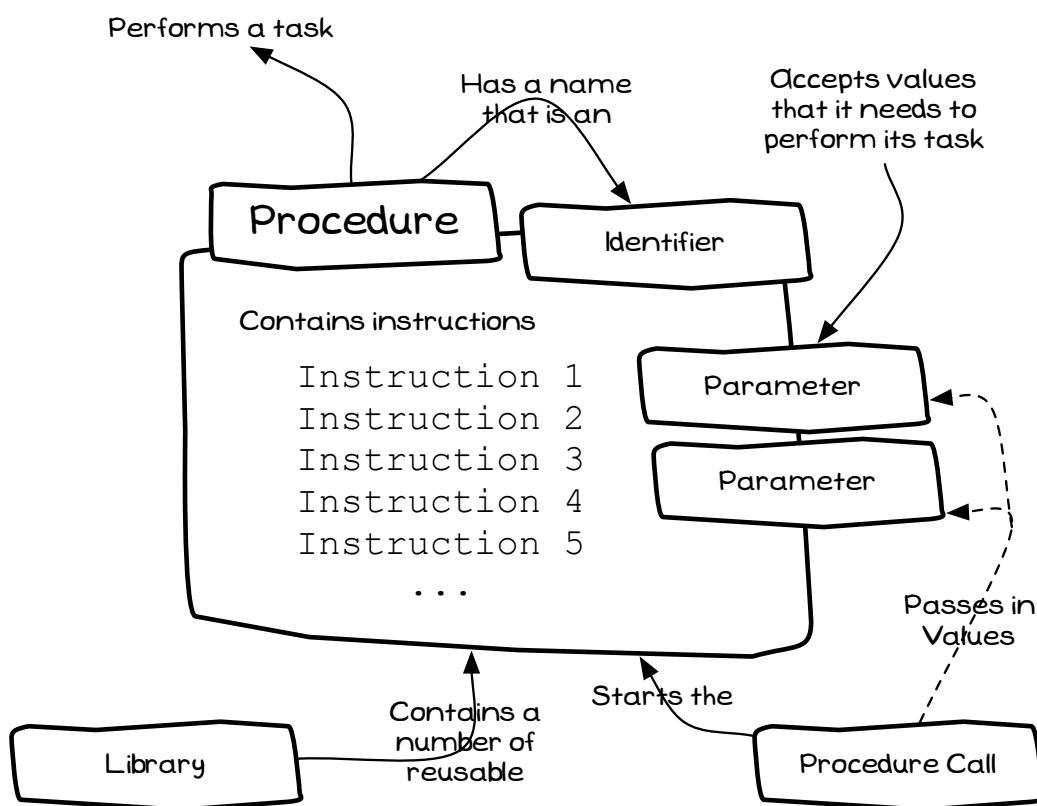


Figure 1.5: A procedure contains instructions to perform a task, and may need to be passed data in order to do this

Note

- A procedure is an **artefact**, something that can be created in code.
- Figure 1.5 shows the concepts related to procedures.
- A procedure is a programming artefact that can be called to perform a certain task.
- The name of a procedure is an **Identifier**.
- Each **Library** will contain a number of procedures to perform common tasks.
- The standard library will include procedures to write values to the Terminal.
- The SwinGame libraries contain procedures that can draw images on the screen, play sounds, and perform other tasks needed to create small games.
- Procedures are also known as **subroutines**, **sub-programs**, **methods** or **sub-procedures**.

1.1.5 Expression

Some statements need data, this data can be calculated or provided as a literal value in the code. The term **expression** is used in programming to describe the places in a statement where data must be supplied. At run time each expression becomes a value that is used by the statement.

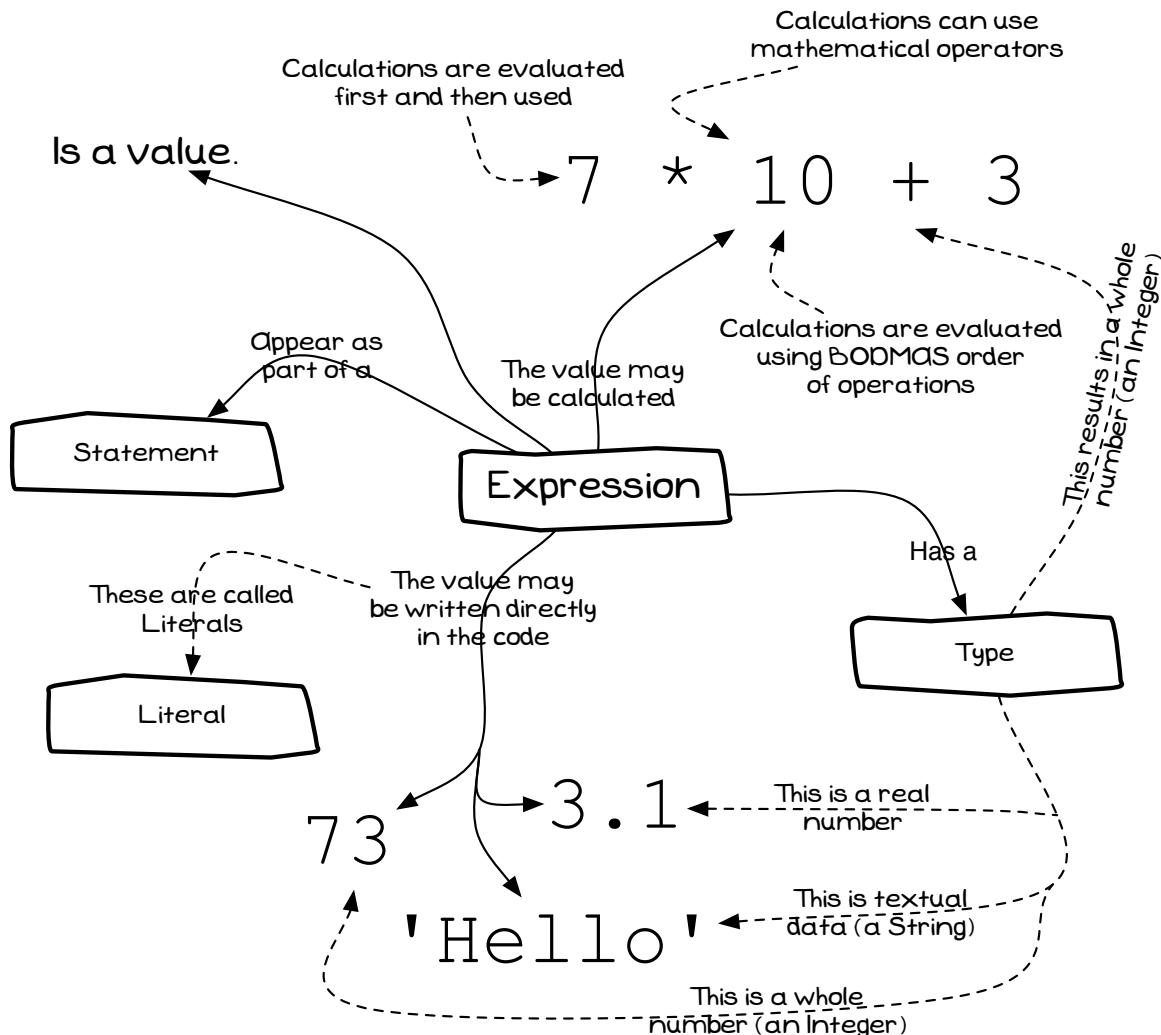


Figure 1.6: An expression provides a **value** to be used in a Statement.

Note

- An expression is a **term** given to code that calculates a value.
- The concepts related to expressions are shown in Figure 1.6.
- An expression provides a **value** that is used in a Statement.
- The expression's value may be calculated or entered directly into the code.
- Calculations can use mathematical operators: + for addition, - for subtraction, * for multiplication, / for division, and parenthesis () for grouping.
- Expressions are evaluated using the BODMAS^a order of operations.
- Values entered directly within an expression are **Literal** values.

^aBODMAS indicates that expressions are evaluated **B** brackets first, **O** orders (which includes powers and square roots), **DM** for division and multiplication (which are of equal precedence, and are evaluated left-to-right), then **AS** addition and subtraction (of equal precedence, evaluated left-to-right).

1.1.6 Literal

A Literal is a whole, or part of, an [Expression](#) where the value is entered directly into the code.

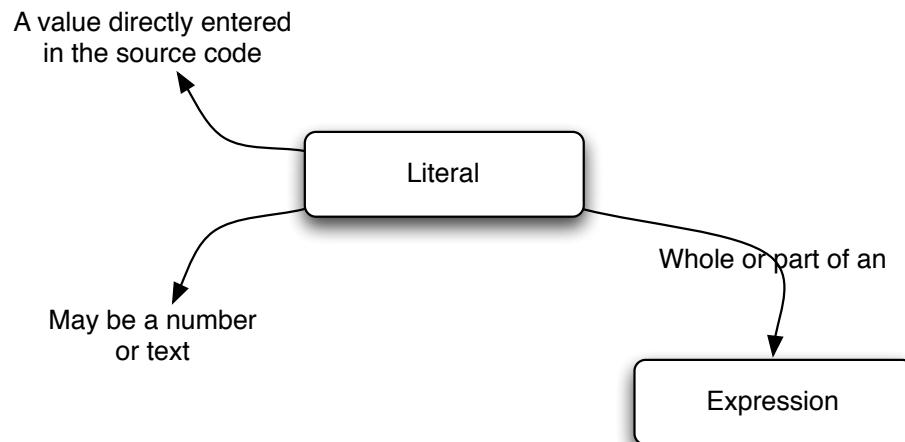


Figure 1.7: Concepts related to Literals.

Note

- Figure 1.7 shows the concepts relate to Literals.
- A Literal is a value entered directly into the program's source code.
- The value of a Literal can be a number or text.
- A Literal can be part or all of an [Expression](#).
- These values are *hard coded* into the program.



1.1.7 Type

All values within a program will have a **type**. The type indicates how the data stored in the computers memory is interpreted by the program. There are three basic data types available in a programming language, as shown in Figure 1.8.

- **Textual** data such as ‘Fred’, ‘Hello World’, ‘23’, and ‘This is text’.
- **Whole numbers** such as 1, 0, -5, and 37.
- **Real numbers** such as 0.5, -126.0, 3.141516, and 23.981.

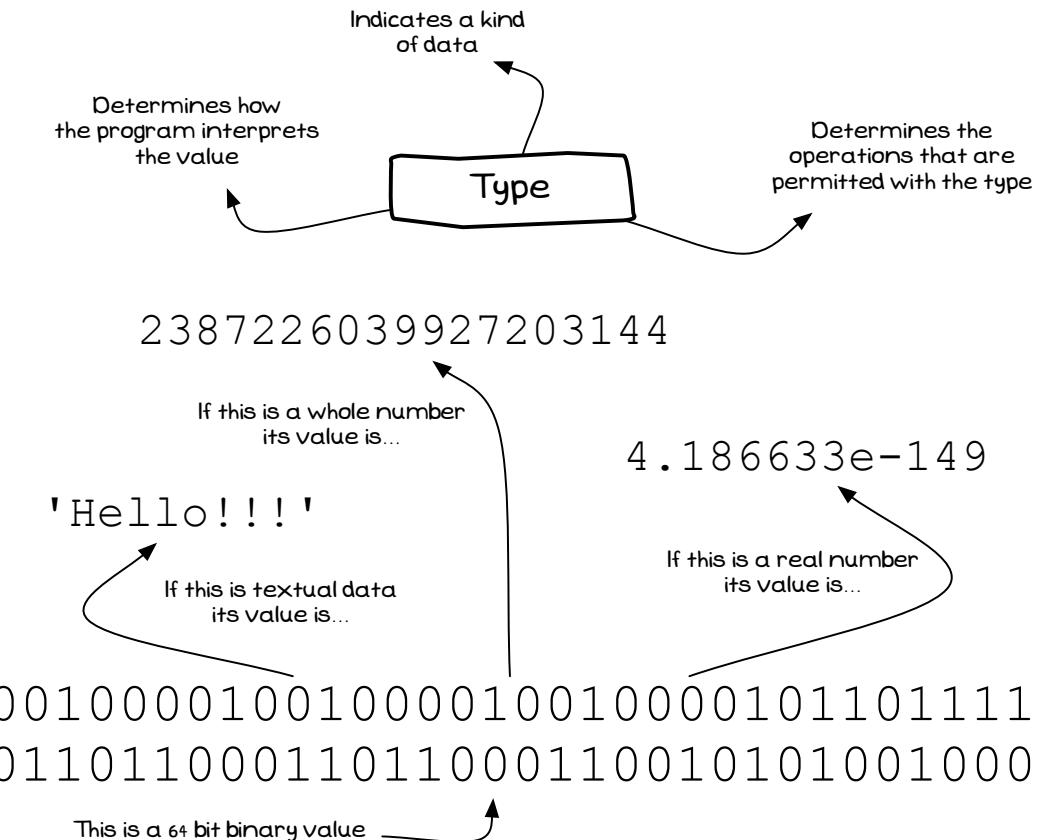


Figure 1.8: A types define how values are interpreted and the operations that can be performed on the data.

Note

- A type is an **artefact**, there will be a number of existing types that you can use, and later you will see how to create your own types.
- The concepts related to expressions are shown in Figure 1.8.
- A type is a programming artefact that indicates a kind of data.
- The type determines the basic actions that can be performed on the value.
- The type determines the amount of memory needed to store a value of that kind.
- Whole numbers are usually called **Integers**.
- Real numbers are usually represented as **Floating Point** values. These values have a limited precision, supporting only a certain number of digits of precision.
- Textual values can contain numbers as text characters. For example, the text '23' is the character '2' followed by the character '3' - it is not the number 23.
- You can perform mathematic operations on numeric data, but not on textual data.



1.1.8 Identifier

An identifier is the technical term for the name/word that *identifies* something for the compiler. These can be the **name** of a programming artefact (such as a Program, Library, or Procedure) or words that have special meaning for the compiler. You will use identifiers to name the artefact you create, and to select the artefact you want to use.

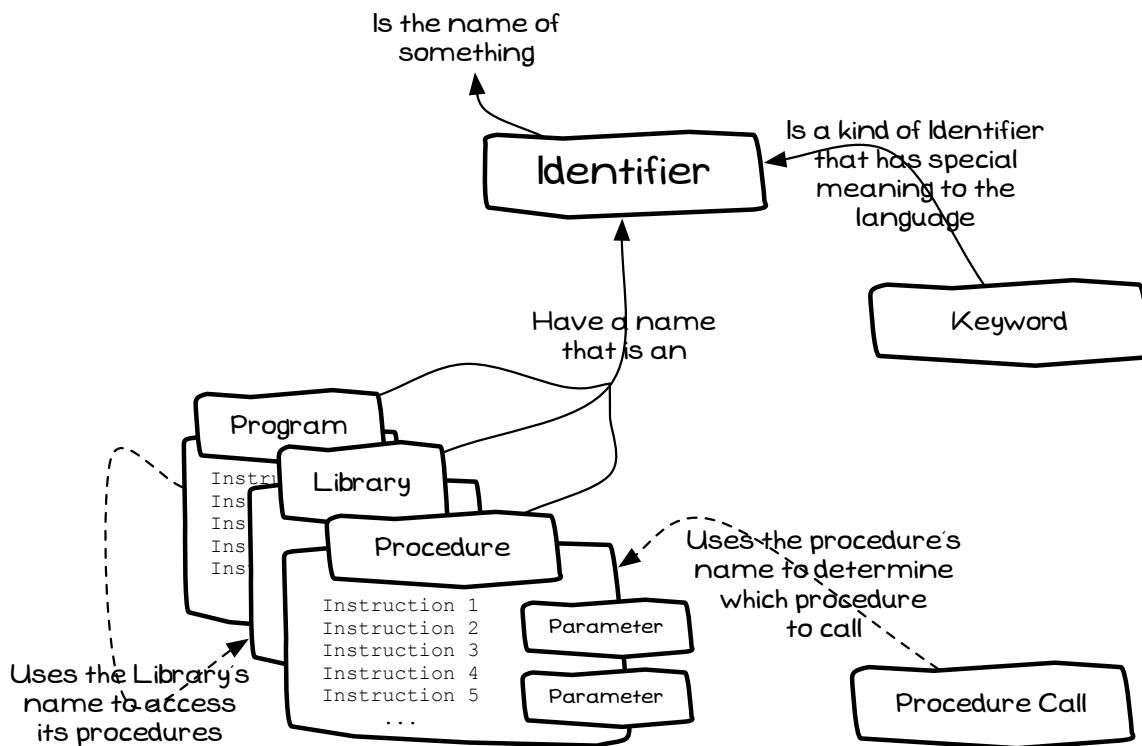


Figure 1.9: An Identifier is the name of a programming artefact such as a Program, Library, or Procedure.

Note

- Figure 1.9 shows the concepts related to an Identifier.
- The **name** used to identify a programming artefact (such as a [Program](#), [Library](#) or [Procedure](#)) is an identifier.
- You use identifiers to indicate which libraries you want to access in your program.
- Each [Procedure Call](#) uses the procedure's identifier to determine which procedure is run.

1.1.9 Library

A library is a collection of reusable code artefacts. Each programming language has its own library, and your programs can make use of the code available in this library.

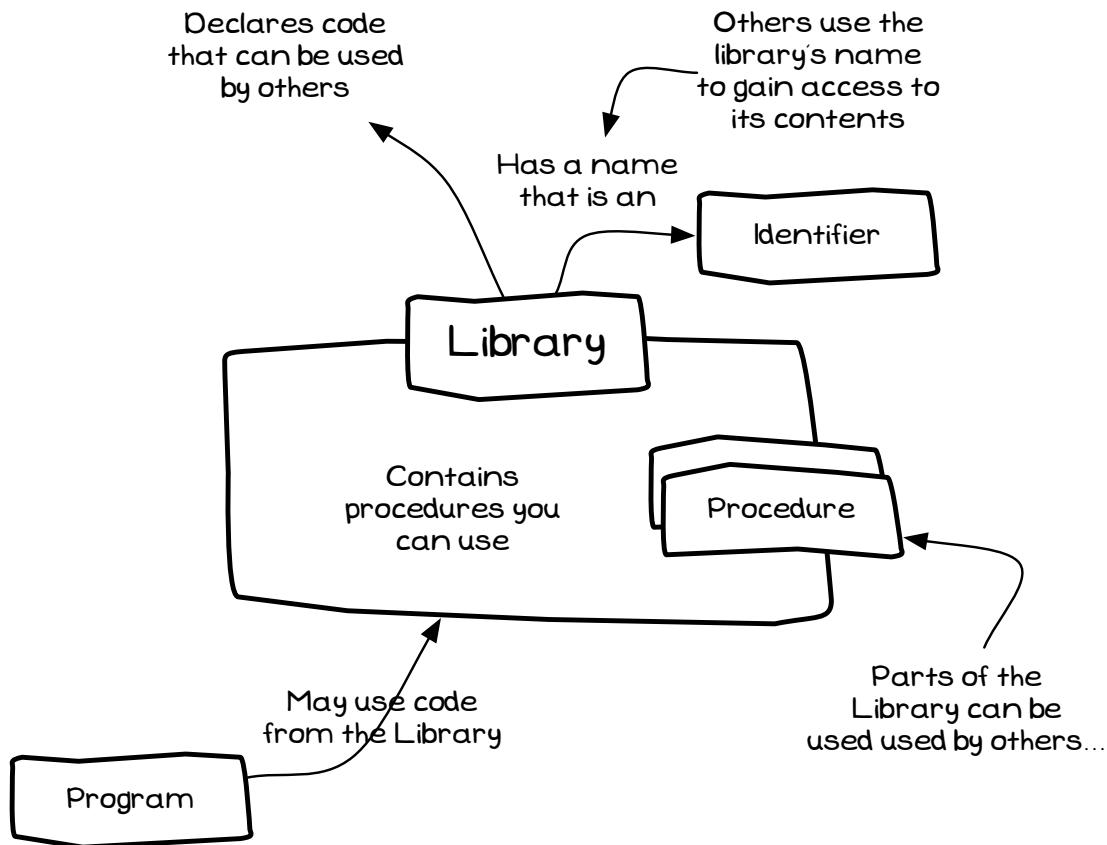


Figure 1.10: A library contains code that can be used by your Program

Note

- A library is an **artefact**, it contains reusable artefacts.
- Figure 1.10 shows the concepts related to a library.
- A library is a collection of reusable code artefacts that you can use to perform certain tasks.
- The library will contain **Procedures** that perform a number of tasks.
- Each language has a standard library with code to perform many commonly performed tasks.
- Other libraries extend the capability of the languages further.
- SwinGame is a external library containing code to help you build games.

1.1.10 Comments

A program's source code contains instructions for the actions the computer must perform. However, this code is written and maintained by people. It is often useful to be able to place comments in the code to help someone reading that code understand how the code works or what it is trying to achieve. This text is not something that should be translated into machine code.

Programming languages support the ability for programmers to embed *comments* into the source code that are ignored by the compiler.

Note

- It is good practice to place a comment at the top of your code explaining what the program does.
- Comments should be included to help other people read your code. You will also find these comments useful when you return to your code after a long break.
- Make your comments meaningful, try to capture your intentions and ideas.
- Comments have no impact on the output produced by the compiler.



1.1.11 Procedure Declarations

Procedures contain code that define the steps the computer performs when the procedure is called. In your Program you can define your own Procedures, allowing you to divide a program's tasks into separate Procedures.

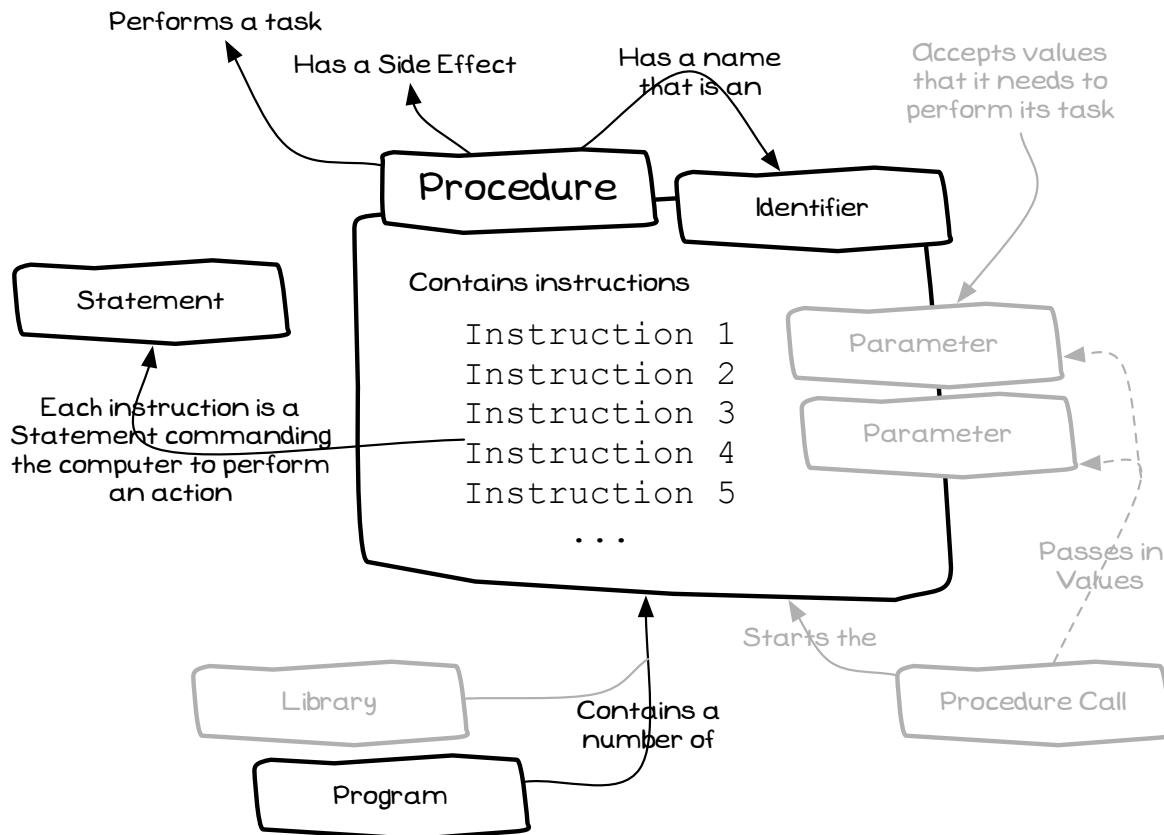


Figure 1.11: Procedure Declaration

Note

- A Procedure is an **artifact** that you can *create* and *use* in your code.
- Each Procedure contains code to perform a certain task. When you want the task performed you call the Procedure.
- Procedures should have a **side effect^a**, meaning that it changes something when it is executed.
- The Procedure's declaration defines its **name**, and the **steps** it performs.
- Each instruction in the Procedure is a **Statement**.
- The Procedure's **Identifier**:
 - Is the name used to call the Procedure.
 - Should be a **verb** that **reflects the task** the Procedure performs.
- When the Procedure is called its instructions are executed.
- Each Procedure's instructions are isolated from the other code in your Program. When you are working on a Procedure you do not need to know about the internal workings of the other procedures.



^aOutput to the Terminal is an example of a Side Effect. After calling these procedures the text you wanted to appear was written to the Terminal. These Procedures changed the Terminal.

1.1.12 Summary

This section has introduced a number of programming artefacts, some programming terminology, and one kind of instruction. An overview of these concepts is shown in Figure 1.12. The next section will look at how you can use these concepts to design some small programs.

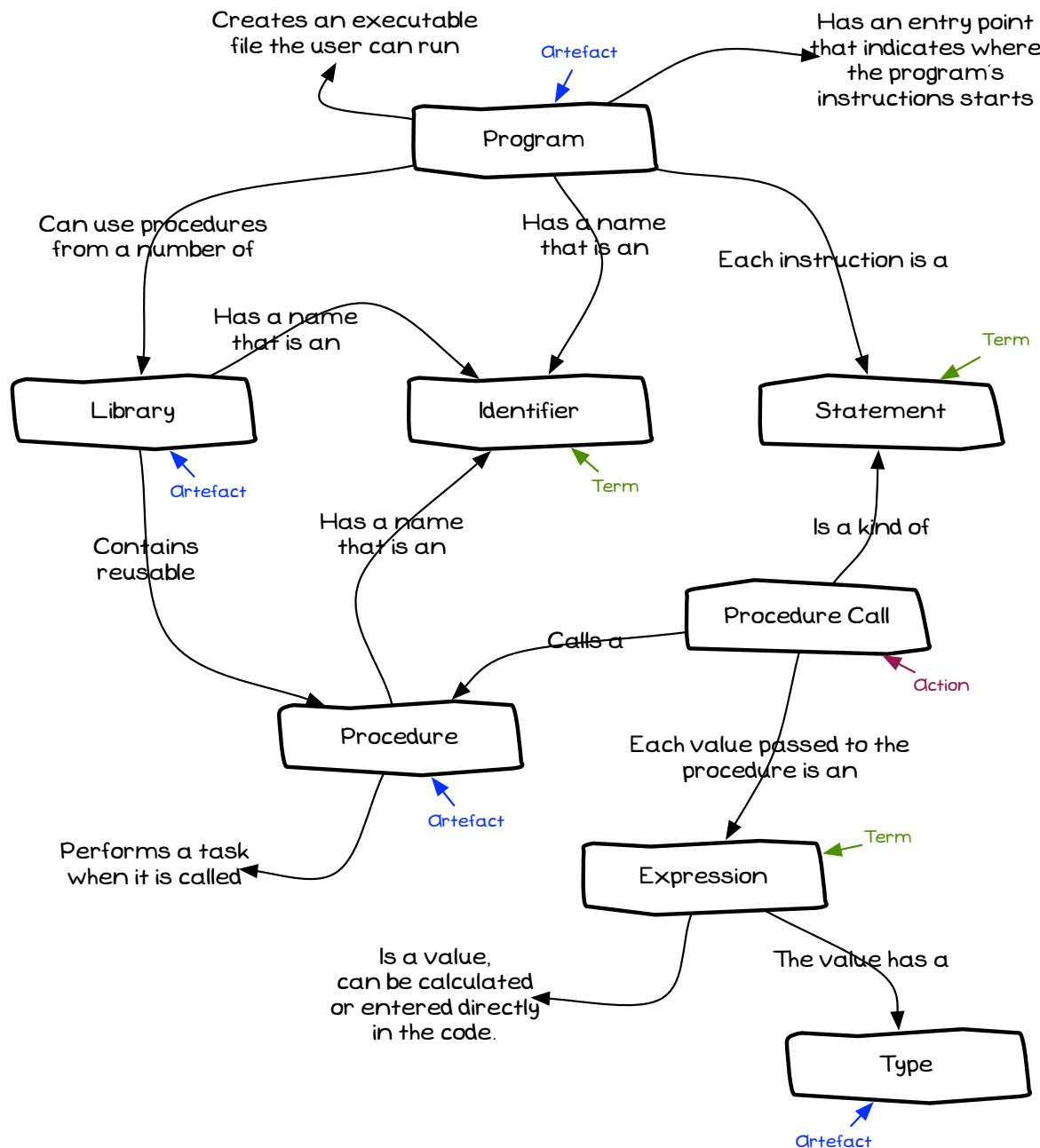


Figure 1.12: Key Concepts introduced in this Chapter

Note

- **Artefacts** are things you can *create* and *use*.
- **Terms** are things you need to *understand*.
- **Actions** are things you can *command* the computer to perform.

1.2 Using these Concepts

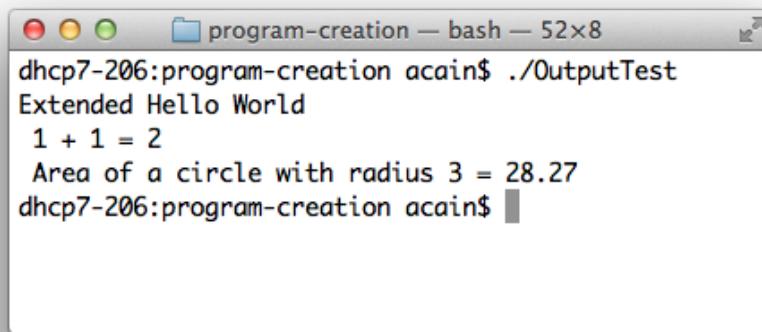
Armed with the knowledge you have gained in the Section 1.1 you can now start to make your own small program.

1.2.1 Designing Output Test

Our first programming task is to extend the classic ‘Hello World’ program to also output some data values. We will call this program **Output Test**. This program will allow you to see all of the different concepts from this chapter in action. A description of the program is shown in Table 1.1, and a sample execution is shown in Figure 1.13.

Program Description	
Name	<i>Output Test</i>
Description	Displays the text ‘ Output Test Program! ’ on the Terminal, followed by ‘ 1 + 1 = ’ and the result of the calculation $1 + 1$, and then ‘ Area of a circle with radius 3 = ’ and the result of calculating this value.

Table 1.1: Description of the *Output Test* program.



A screenshot of a terminal window titled "program-creation — bash — 52x8". The window contains the following text:
dhcp7-206:program-creation acain\$./OutputTest
Extended Hello World
1 + 1 = 2
Area of a circle with radius 3 = 28.27
dhcp7-206:program-creation acain\$

Figure 1.13: Output Test executed from the Terminal

To design and implement this program we need to follow a number of steps:

1. Understand the problem, and get some ideas on the tasks that need to be performed.
2. Choose the artefacts we will create and use.
3. Map these artefacts to code.
4. Compile and run the program.

1.2.2 Understanding Output Test

The first step in creating a program is to analyse the problem, and find related material that we can use to make sure we know what the computer needs to do. You need to understand what the program needs to do before you can start to design the solution.

For this program you need to understand how to calculate the answer of the equation $1 + 1$ (which is a trivial task), and also how to calculate the area of a circle a , given its radius r . A quick search of the internet and you can find the equation is $a = \pi r^2$. Using these equations you have the knowhow needed to calculate the values needed to be output.

1.2.3 Choosing Artefacts for Output Test

Designing a program is all about making decisions around how to organise the program's instructions. This means **deciding** on which artefacts you will **create**, and which you will **use**. This chapter has already introduced the concept of **creating** a [Program](#), and **using** [Procedures](#). With these tools it is possible to design simple programs that will get the computer to perform actions by running procedures in sequence.

Let us return to the design of the *Output Test* program. The program will require:

- The **creation** of a [Program](#). This program will contain the instructions to write the three messages to the terminal.
- The **use** of a procedure to write to the terminal. Programming languages provide a standard [Library](#) that, amongst other things, contains procedures to write data to the terminal. This means you can call one of these procedures, passing it the data you want written to the Terminal, and let it take care of the details of how this is done.

The design for this program can be documented using a programming language neutral **pseudocode**¹. This is a structured textual description of how the program works that is independent of the programming language used to implement it.

[Listing 1.1](#) shows the pseudocode for the *Output Test* program. This represents the **program** that needs to be created, and shows the three **procedure calls** that need to be made.

Pseudocode

```
-----
Program: Output Test
-----
Steps:
1: Output 'Output Test Program' to the Terminal
2: Output ' 1 + 1 = ', and the result of 1 + 1
3: Output ' Area of a circle with radius 3 = ',
   and the result of 3.1415 * 3 * 3
```

Listing 1.1: Pseudocode for the Output Test program.

C++

In C the procedure to write data to the Terminal is called `printf`, see [C++ SplashKit Terminal Output](#).

Pascal

Pascal has two procedures to write data to the Terminal, `Write` and `WriteLn`.

¹This is literally translated as 'false code'; it looks like code, but it is not real code!

1.2.4 Writing the Code for Output Test

The pseudocode in Listing 1.1 contains the instructions that will get the computer to perform the actions needed by the *Output Test* program. These pseudocode instructions are not in a form that can be used by the computer, which can only use machine code. To make this into a program the pseudocode must be translated into source code, and then to compile this into machine code.

Section 1.3 Program Creation in C++ and Section 1.4 Program Creation in Pascal, contain a description of the syntax needed to create programs using the C and Pascal programming languages. Each section outlines how to write the code so that it can be understood by a compiler. The programming languages are expressed as a number of related syntax rules. You will find the rules that you need to create a program, the rules for calling a procedure, and other related rules.

In this book the syntax rules are expressed using **syntax diagrams**. An example is shown in Figure 1.14. This diagram shows the syntax related to two rules, *first rule*, and *second rule*, and shows the four main parts of all the syntax diagrams.

1. Text found at the start of a line (not contained in a box) is the name of a rule. There are two rules in Figure 1.14: *first rule*, and *second rule*.
2. Arrows show the order in which the parts of the rule are applied. They start at the rule name, and point in the direction you need to follow. Each box pointed to by an arrow represents either another rule to apply, or the text that must be written.
3. Rectangular boxes (nodes) on a line indicate points where other rules need to be applied. For example, the node **second rule** within the *first rule* indicates that you **must** apply the *second rule* at this point.
4. Boxes with rounded corners represent text that must be entered into the code. For example, the node **write in the code** within the *first rule* indicates that you **must** write the text ‘*write in the code*’ at this point in your code.

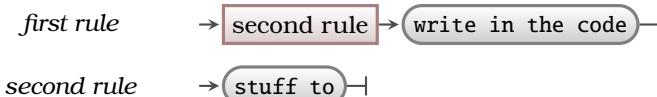


Figure 1.14: An example rule

In order to make use of these syntax diagrams, you must know what it is that you *want* to write in code. The rules in Figure 1.14 indicate that you can write either a *first rule* or a *second rule*. If you want to write a *first rule* you find that rule in the diagram, and then follow its arrows. Reading the *first rule* indicates that the *second rule* must be applied first.

The *second rule* tells you that you must write the text ‘*stuff to*’. The vertical bar at the end of the line indicates the end of the *second rule*. This means at this stage the code is:

stuff to

Having finished the *second rule*, you can return back to finish the *first rule*. This indicates that you need to write ‘*write in the code*’ in the code. This is the last part of the *first rule*, and so the code needed to write a *first rule* from the syntax diagram in Figure 1.14 is shown below.

stuff to write in the code

For a more realistic example have a look at the syntax diagram in Figure 1.15.² This shows

²The is used as shorthand to avoid having to list all of the characters between 'A' to 'Z'.

the rules you need to follow to code an Identifier³ in either C or Pascal.

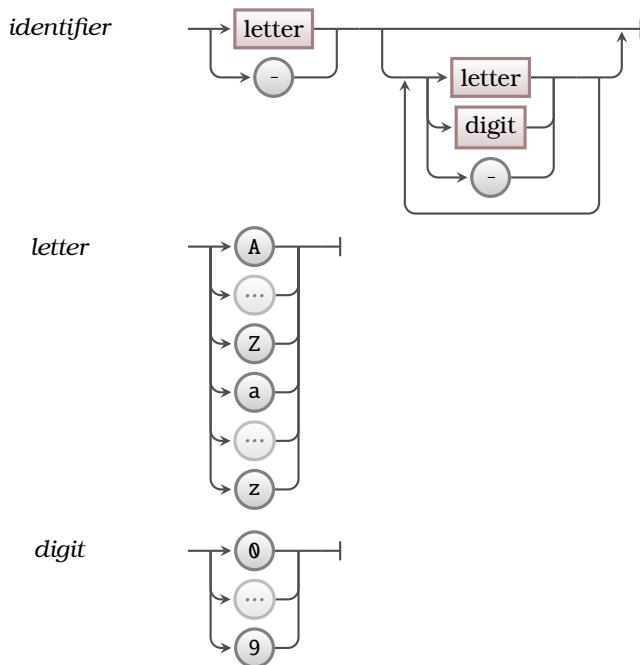


Figure 1.15: The Syntax Rules for an Identifier

There are three rules in Figure 1.15, including the **identifier** rule itself and rules for **letter** and **digit**. These rules show arrows that give you *options*, and the ability to *repeat* parts of the rules.

- A **letter** is one alphabetic character: i.e. one of 'A' to 'Z' or 'a' to 'z'. This is an example of options in the syntax, where you follow **one** of the available arrows.
 - A **digit** is a single number: i.e. a number between '0' and '9'.
 - The **identifier** has a more complicated rule, with the following parts:
 1. The first thing in an **Identifier** must be either a *letter* or an underscore (_).
 2. Next you have the option of following the top line and ending the identifier, or following the downward arrow and including other letters, numbers, and underscores.
 3. Following the downward arrow you have a new option where you can choose to have either a *letter*, a *digit*, or an underscore as the second character in your identifier.
 4. Continuing after this option you have another option where you can return back to repeat the previous step, allowing you to have identifiers with more than one or two characters.

Using the Syntax Diagrams

Syntax diagrams can help you to map a concept to actual code that needs to be written in your source code. To use these diagrams you must first know *what* it is that you want to create or use, and then you can *look up* the related syntax. This is where pseudocode code comes into play. It contains a description of the things that need to be created.

[Listing 1.2](#) shows the pseudocode for the Output Test program. This tells you what needs to be created; you need to create a [Program](#). This means you need to find the syntax that tells

³Most programming languages have the same rules for identifiers.

you the rules of how a *Program* is written in a programming language.

Pseudocode

```
-----  
Program: Output Test  
-----  
Steps:  
1: Output 'Output Test Program' to the Terminal  
2: Output ' 1 + 1 = ', and the result of 1 + 1  
3: Output ' Area of a circle with radius 3 = ',  
    and the result of 3.1415 * 3 * 3
```



Listing 1.2: Pseudocode for the *Output Test* program (repeated from Listing 1.1)

When you write the code for the program you will need to know the actions you want the computer to perform. The pseudocode in Listing 1.2 indicates that you need to output text and calculated values to the terminal; this can be achieved using [Procedure Calls](#). You can lookup the language syntax for a procedure call, and use this to write the required code.

How the Syntax is Presented

The C and Pascal Syntax needed to create a program are shown in Section [1.3 Program Creation in C++](#) and Section [1.4 Program Creation in Pascal](#). Each part of the Syntax is presented on its own page that shows a syntax diagram followed by an example and some notes. The best way to approach this is to do the following:

1. Find the page with the syntax rule you are interested in knowing about.
2. Have a quick look at the **syntax diagram** and the rules it contains. Read each rule, and get a basic feel for how it is going to come together for your program.
3. Read the **example** to see one way of using the rule. A syntax diagram can be used to create any number of variations of the rule. However examples show you at least one way these rules can be coded.
4. Return to the diagram and make sure you can match each part of the example back to the rule that created it.
5. Now look up any related rules that are not explained on this rule's page. For example, a **Program** will use the **Statement** rule to code its instructions. The actual rule for a Statement will have its own page. When you read the rules for a Program you will need to also find the page with the rules for a Statement so that you know how to code these within the program.

As you follow this process it is also a good idea to take notes and to try to use these rules in your own programs. Have your code editor open, and see if you can follow the rules or mimic the examples to start building your own program's code. You can also try typing in some of the examples to see how they work.

Learning to program, and learning a language's syntax, takes time and practice (like juggling). Reading about these concepts is one thing, but being able to successfully apply this ideas is something different. Make sure that you practice using these concepts and syntax.

1.2.5 Compiling and Running Output Test

Previous sections have shown pseudocode for the *Output Test* program. These steps can be coded, using either C or Pascal, into a source code file. In order to actually use these instructions you will need to **compile** the source code file. This will produce an executable file that

you can run. To do this you can use the terminal and the command line compiler as shown in Chapter ??.

- On Ubuntu **Linux** you can find the Terminal in the **Accessories** folder within **Applications**. See Figure 1.16.
- On **MacOS** you can find the Terminal in the **Utilities** folder within **Applications**. See Figure 1.17.
- On **Windows** you will need to download and install *MSys2*, follow the steps in the SplashKit installer. The *Msys2 Shell* is then the equivalent of Terminal on the other operating systems. You will find this in **Program Files**, **MSys2**. See Figure 1.18.

Once you are in the Terminal you have the ability to run a number of text based commands. These commands instruct the computer to perform actions for you.

- **pwd** stands for *Present Working Directory*, and shows you where you are in the file system.
- **ls** stands for *List* and it prints out a list of the files that are in the current directory.
- **cd** stands for *Change Directory* and it moves you to another directory.

To compile your program you need to do the following:

1. Change into the directory where your code is located using the **cd** command. For example, if your code is in a *Code* folder in your *Documents* folder you would use:
 - **Linux**: `cd /home/username/Documents/Code`
 - **MacOS**: `cd /Users/username/Documents/Code`
 - **Windows**: (in MinGW Shell) `cd /c/Users/username/Documents/Code`⁴
2. Run the compiler, passing in the name of the file you want to compile. See the language specific notes below
3. Execute the program using `./OutputTest`

C++

The C compiler is called **gcc**. To compile your *Output Test* program you will need to run the following:

```
skm clang++ output_test.cpp -o OutputTest
```



Pascal

The Pascal compiler is called **fpc**. To compile your *Output Test* program you will need to run the following:

```
skm fpc -S2 OutputTest.pas
```



⁴This example moves you into the `c:\Users\username\Documents\Code` folder. You need to use the `/c/` to refer to the C drive.

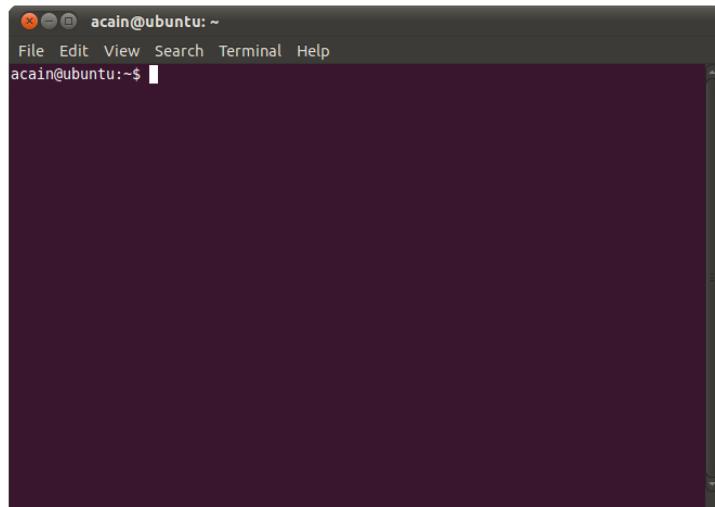


Figure 1.16: Terminal on Ubuntu Linux

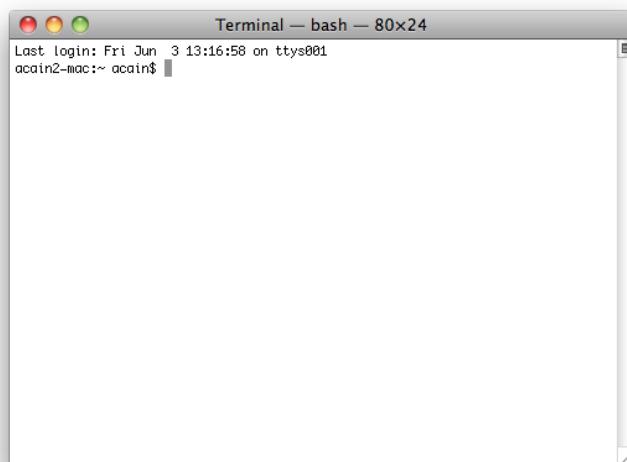


Figure 1.17: Terminal on MacOS

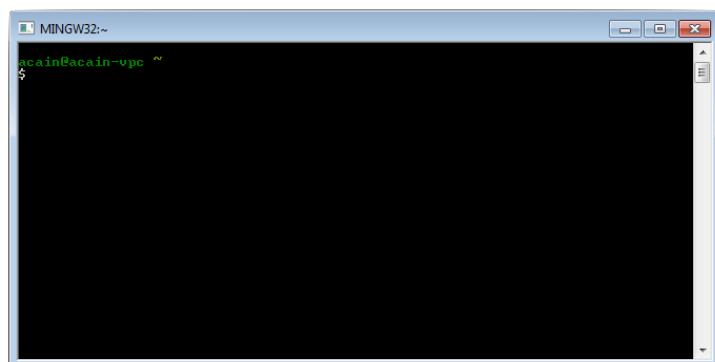


Figure 1.18: MinGW Shell (Terminal) for Windows

Figure 1.19 shows an example of the instructions needed to compile and run the C version of the *Output Test* program on Linux. Figure 1.20 shows an example of the instructions needed to compile and run the Pascal version of the *Output Test* program on Linux.

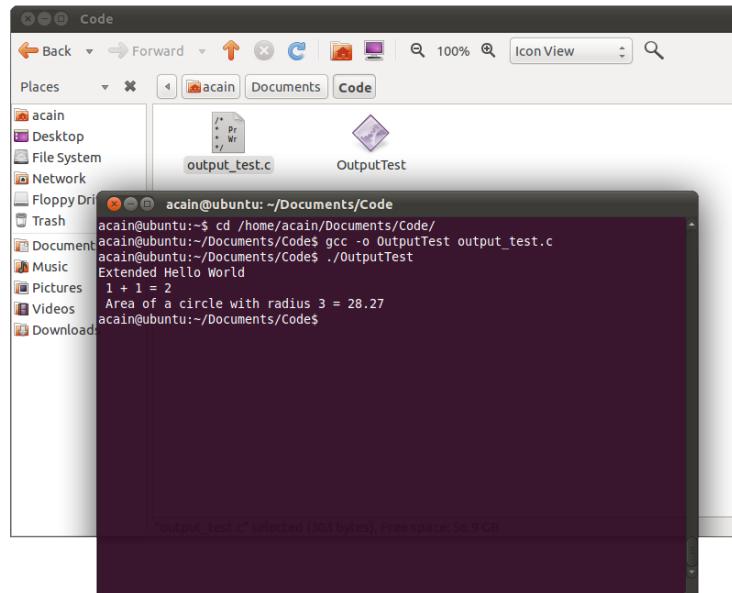


Figure 1.19: Example of compiling and running a C version of Output Test on Linux

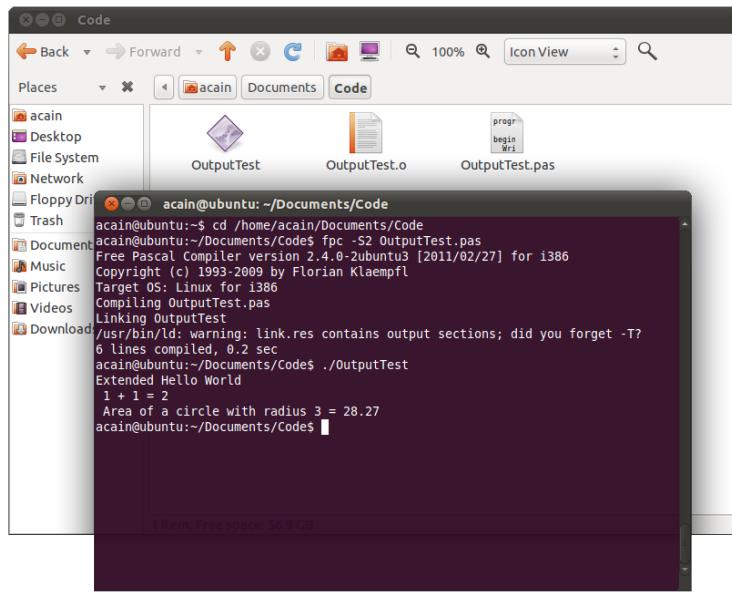


Figure 1.20: Example of compiling and running a Pascal version of Output Test on Linux

Figure 1.21 shows an example of the instructions needed to compile and run the C version of the *Output Test* program on MacOS. Figure 1.22 shows an example of the instructions needed to compile and run the Pascal version of the *Output Test* program on MacOS.

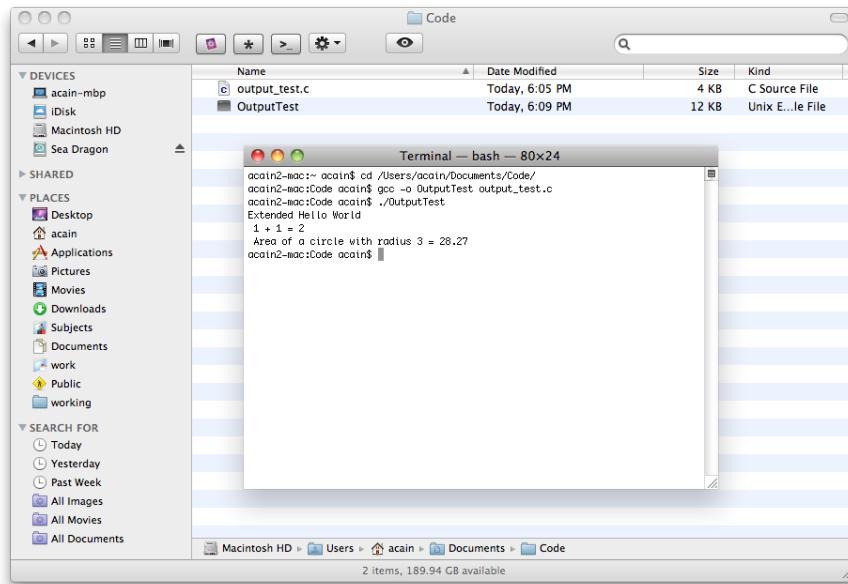


Figure 1.21: Example of compiling and running a C version of Output Test on MacOS

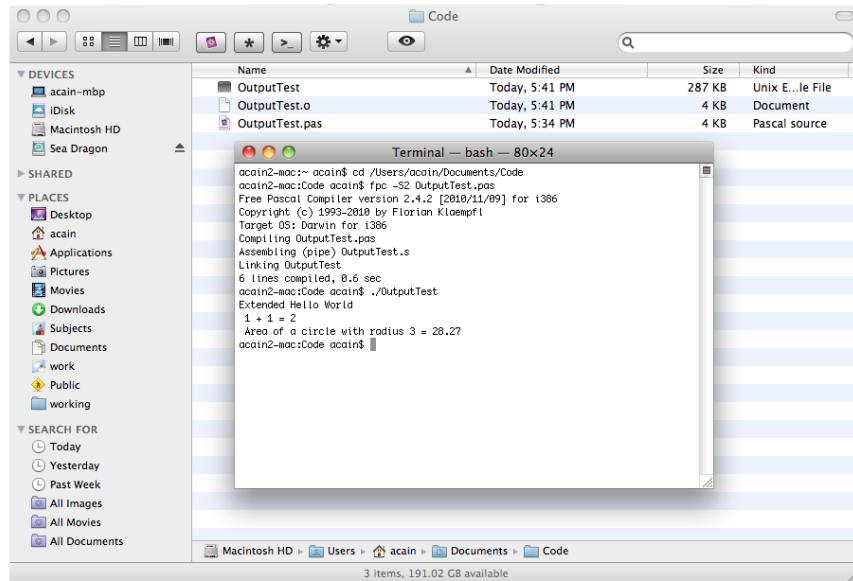


Figure 1.22: Example of compiling and running a Pascal version of Output Test on MacOS

Figure 1.23 shows an example of the instructions needed to compile and run the C version of the *Output Test* program on MacOS. Figure 1.24 shows an example of the instructions needed to compile and run the Pascal version of the *Output Test* program on MacOS.

Figure 1.23: Example of compiling and running a C version of Output Test on Windows

Figure 1.24: Example of compiling and running a C version of Output Test on Windows

Compiler Errors

A compiler is a very sensitive piece of software; they require that you follow the language's syntax *precisely*. One small mistake, and the compiler will fail to compile your program and end with an error message. This is a *good* thing! (Although it may not feel like it at first.) Why? Because when programs become complicated we need the exact details to help us, and to assure the program works the way we need it to.

Here are some handy hints related to dealing with compiler errors.

1. **Know that errors are inevitable.** You will get compiler errors! As you gain more experience you will get fewer errors, but it will always be a rare event that everything is exactly as it should be the first time.
2. **Error messages may appear cryptic at first.** In many cases the compiler's error messages can appear cryptic, you need to learn to decode the messages the compiler gives you.
3. **Start with the first error message, and do not move on until it is fixed.** The compiler will read your code 'top down'. So the first error it outputs will be the first in the file. The problem is that the compiler will try to continue on, despite the error. This can mean that other errors further on in the output may actually be caused by the compiler being *confused* due to the first error. By fixing the first error you may also fix subsequent errors.
4. **Deal with one error at a time.** It is easy to feel overwhelmed when you see a huge list of errors, but do not be scared off! Start at the first error, and solve them one by one. Compile after fixing each problem to see if the others still exist.
5. **Do not add more code until the errors are fixed.** Compile your program frequently. Add small pieces of functionality, and then fix any syntax errors before adding the next piece of functionality. Coding in this way makes sure that you do not get too many errors, and reduces the code you have to search in order to fix the problem.
6. **Read the error message carefully.** The message will try to tell you what has gone wrong, and once you understand what the messages are trying to say you will be able find and fix the issues quickly.
7. **Work out what the error messages mean.** Understanding the error messages will mean that you will know what to look for, and will help you avoid these errors in the future. If you not sure what an error message means ask others developers or search for the message on the internet.
8. **Find the line and character number of the error.** One important detail in the error message will be the line number of the error. This gives you a starting point to help you locate the problem. It is important to note that this is where the compiler got to when it noticed the error - this *does not* mean this is where the error actually is (but it is a good place to start looking). You may need to look back one or more lines to find the actual source of the problem.
9. **Watch for typos.** It is easy to mistype an identifier. When this happens the compiler will not know what to do (and it will complain about you trying to use 'unknown' things). Make sure you check for these tiny typos when you get errors related to the compiler not being able to find an identifier.
10. **When you get stuck ask for help.** Compiler errors are likely to be something small, but sometimes the causes are hard to find. If you get stuck, *ask for help!* Having access to other more experienced developers will be a valuable resource as you learn to program. Learning to program can be tough at times, and having someone who can help you when you get stuck will make all the difference.

1.3 Program Creation in C++

Section 1.2 on page 22 of this chapter introduced an ‘Output Test’ program, and its design. The pseudocode from this section is shown in Listing 1.3. In this Section you will see the rules for translating this program’s design into the C code shown in Listing 1.4.

Pseudocode

```
-----
Program: Output Test
-----
Steps:
1: Output 'Output Test Program' to the Terminal
2: Output ' 1 + 1 = ', and the result of 1 + 1
3: Output ' Area of a circle with radius 3 = ',
               and the result of 3.1415 * 3 * 3
```

Listing 1.3: Pseudocode for Hello World program (from Listing 1.1).



C++

```
/*
*  Program: output_test.c
*  Writes some messages to the Terminal.
*/
#include "splashkit.h"

int main()
{
    // Output the messages...
    write_line("Output Test Program");
    write_line(" 1 + 1 = " + to_string(1 + 1));
    write_line(" Area of a circle with radius 3 = " + to_string(3.1415 * 3 * 3) );

    // Finish
    return 0;
}
```

Listing 1.4: Output Test in C



Note

- Save the C code in a file named `output_test.c`.
- Compile this using `gcc -o:OutputTest output_test.c`.
- Run using `./OutputTest`.
- The code at the start is a Comment describing what is in the file, see [C++ Comments](#).
- The code `#include <stdio.h>`, is part of the [C++ Program \(with Procedures\)](#). It is a *header include*, and gives access to the code in the *Standard IO Library*.
- `int main() ...` is part of the [C++ Program \(with Procedures\)](#), it marks the entry point and contains the instructions that are executed when the program runs.
- The `main` function contains three [C++ Procedure Calls](#). Each is a call to the `printf` [Procedure](#), which is used to output text to the Terminal. See [C++ SplashKit Terminal Output](#).
- Each of the procedure calls contains one or two [Expressions](#) that pass values to the `printf` procedure, which will output these to the Terminal.
- This code uses `c-strings`, `int`, and `float` types, see [C++ Types](#).



1.3.1 C++ Program (with Procedures)

C++ does not have an explicit Program artefact. Rather, you create a program by having a function called ‘`main`’ in your code. Figure 1.25 shows the structure of the syntax used to create a program using the C++ language.

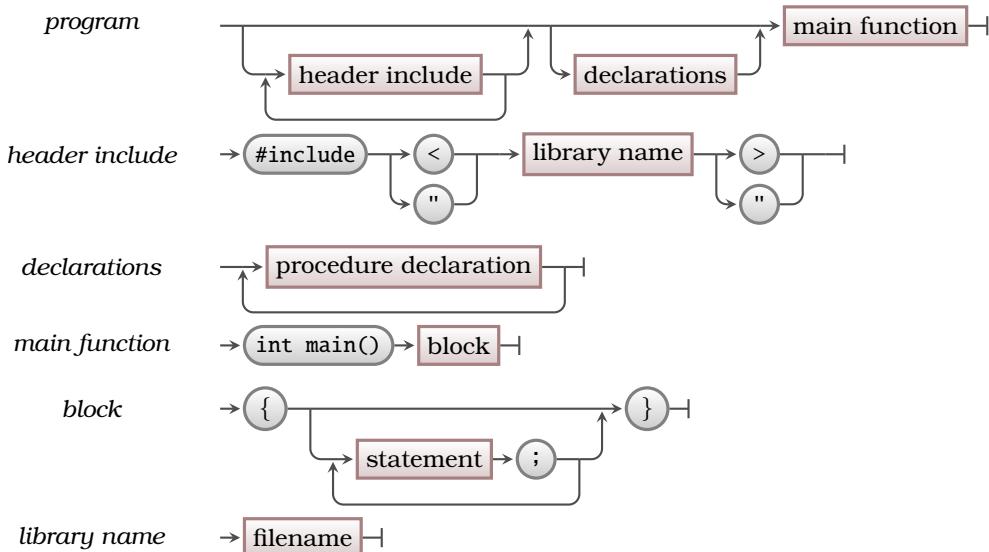


Figure 1.25: C++ Syntax for a Program (with procedures)

Listing 1.5 shows a small C++ Program. You should be able to match this up with the syntax defined in Figure 1.25. This program does not include any custom procedures, but does use a **header include** to include the `splashkit.h` header file. Following this is the `main` function that includes the instructions that are run when the program is executed.

C++

```
#include "splashkit.h"

int main()
{
    write_line("Hello World!");
    return 0;
}
```

Listing 1.5: C++ Hello World

Note

- With the *header include* syntax you use `#include <...>` to include standard libraries, and `#include "..."` to include other external libraries.
- Header files contain a summary of the features available within a library. By including the header file you gain access to these features.
- When a C++ Program runs, it starts running the instructions from the first **Statement** within the `main` function (line 5).
- A **Function** is a kind of **Procedure**, and their details will be covered later (see Section 2.1.9).
- The ‘`return 0`’ code is a **Statement** that ends the `main` function (and the program). The **Return Statement** is covered later in Section 2.3.14.

Listing 1.5 shows another example C++ Program. This code includes two custom procedures: `say_hello` and `say_is_anyone_there`. These procedures are called within `main`.

C++

```
/*
* Program: say-hello-proc.c
* Outputs 'Hello' messages to the Terminal.
*/
#include "splashkit.h"

void say_hello()
{
    write_line("Hello... ");
}

void say_is_anyone_there()
{
    write_line("Is anyone there?");
}

int main()
{
    say_hello();
    say_hello();
    say_is_anyone_there();

    return 0;
}
```

Listing 1.6: Is Anyone There?

Note

- You place **declarations** after the *header includes* and before the *main function*.
- The *declarations* can contain any number of *procedure declarations*. See [C++ Procedure Declaration](#) for details on this code.
- The code in Listing 1.6 shows a Program with two procedures: `say_hello()` and `say_is_anyone_there()`.
- Notice that these procedures are declared after the *header include* `#include "splashkit.h"` and before the *main function*.

1.3.2 C++ Statement

In a [Statement](#) you are commanding the computer to perform an action. There are only a small number of statements you can choose from. At this stage the only statement is the [Procedure Call](#), technically the *procedure statement* in C++. This is shown in Figure 1.26, where we can see that at this stage all Statements are calls to [Procedures](#).

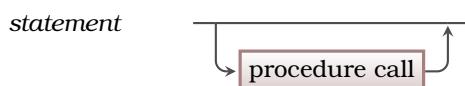


Figure 1.26: C++ Syntax for Statement Syntax

C++

```
#include "splashkit.h"

int main()
{
    //Introduce knights
    write_line("We are the Knights who say 'Ni'!");
    write("We are the keepers of the sacred words:");
    write_line(" 'Ni', 'Peng', and 'Neee-wom'!");
    write_line("The Knights Who Say 'Ni' demand a sacrifice.");
    write_line("We want... a shrubbery!");

    return 0;
}
```

Listing 1.7: C++ Knights

Note

- The code in Listing 1.7 contains a [C++ Program \(with Procedures\)](#).
- This Program contains five procedure calls, see [C++ Procedure Call](#).
- Each procedure call runs the `write_line` or `write` procedure to output text to the Terminal. See the section on [C++ SplashKit Terminal Output](#).

1.3.3 C++ Procedure Call

A procedure call allows you to run the code in a Procedure, getting its instructions to run before control returns back to the point where the procedure was called.

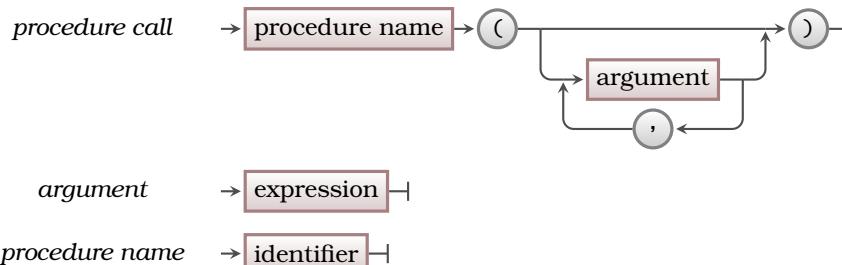


Figure 1.27: C++ Syntax for Procedure Call Syntax

C++

```
#include "splashkit.h"

int main()
{
    write_line("Count back from 2 . . .");
    write_line(2);
    write_line(1);
    write_line(0);
}
```

Listing 1.8: C++ Count Back

Note

- A procedure call is an **action** which commands the computer to run the code in a procedure.
- The procedure call starts with the procedure's name (its **Identifier**) that indicates the procedure to be called.
- Following the identifier is a list of values within parenthesis, these are the values (coded as **Expressions**) that are passed to the procedure for it to use.
- Remember that C++ is case sensitive so using `Write_Line` instead of `write_line` will not work.
- The code in Listing 1.8 contains a **C++ Program (with Procedures)**.
- This Program contains four procedure calls.
- Each procedure call runs the `write_line` procedure to output text to the Terminal. See the section on **C++ SplashKit Terminal Output**.

1.3.4 C++ Procedure Declaration

The Syntax for a C++ Procedure Declaration is shown in Figure 1.28.

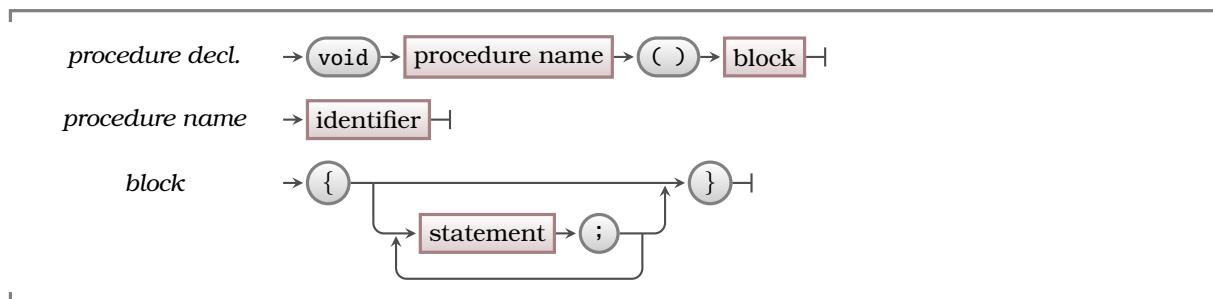


Figure 1.28: C++ Syntax for a Procedure

C++

```
/*
* Program: print_steps.c
* Prints out the steps to cook a meal... (partial)
*/
void find_what_to_cook()
{
    printf("Step 1 - Find what to cook -\n");
    printf("1 - Find a recipe book\n");
    printf("2 - Pick recipe \n");
}

void purchase_missing_ingredients()
{
    printf("Step 2(a) - Purchase Missing Ingredients -\n");
    printf("1 - Goto Shop\n");
    printf("2 - Find ingredients and put in basket\n");
    printf("3 - Go to register and pay for ingredients in basket\n");
    printf("4 - Return home\n");
}

void get_ingredients()
{
    printf("Step 2 - Purchase Ingredients -\n");
    printf("1 - Read recipe\n");
    printf("2 - Write a list of ingredients \n");
    printf("3 - Check food stocks, and tick off ingredients you already have\n");
    purchase_missing_ingredients();
}

int main()
{
    find_what_to_cook();
    get_ingredients();
    // etc...
    return 0;
}
```

Listing 1.9: Cooking a Meal

Note

- There are three Procedures declared in the code in Listing 1.9.
- A **Procedure Declaration** starts with the word **void**. This indicates that the following code is a procedure declaration to the compiler.
- The **Procedure Name** is an identifier. It is the name of the Procedure. This can be any valid **C++ Identifier** that has not been used before.
- The empty parenthesis must appear after the procedure's name, and before the *block*.
- The **block** should look familiar. This is the same as was used in the *main function* of the program to define its instructions, and is used for the same purpose within the *Procedure Declaration*.
- There are a number of conventions, called coding standards, that describe how your code should appear for a given language. In this text we will use a common C convention of having all *Procedure Names* in **lower case**, with underscores (_) used to separate words. So the *Get Ingredients* procedure becomes `get_ingredients`.

1.3.5 C++ Identifier

The C++ **Identifier** syntax is shown in Figure 1.29. In C++, as in most programming languages, the identifier must start with an underscore (_) or a letter; in other words your identifiers cannot start with a number or contain other symbols. This is because the compiler needs a way of distinguishing identifiers from numbers entered directly into the code.

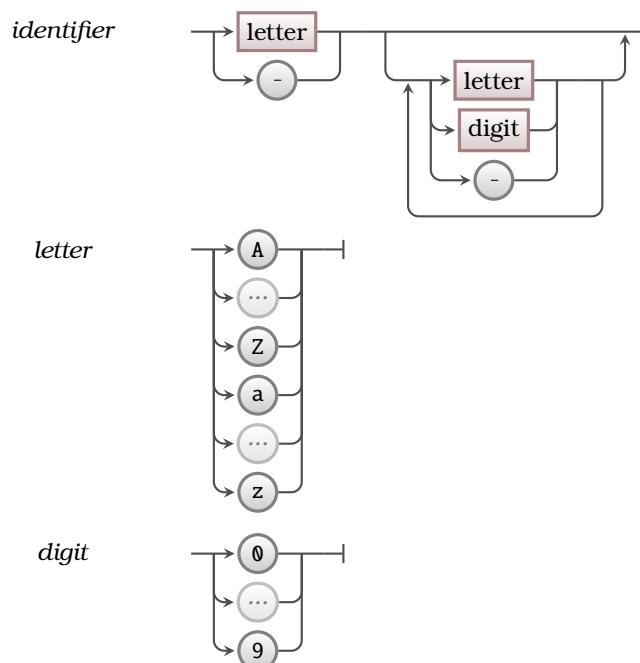


Figure 1.29: C++ Syntax for an Identifier

Reserved Identifiers (Keywords)					Example Identifiers
auto	break	case	char	const	printf
continue	default	do	double	else	scanf
enum	extern	float	for	goto	bitmap
if	int	long	register	return	sound_effect
short	signed	sizeof	static	struct	name
switch	typedef	union	unsigned	void	draw.bitmap
volatile					age
					my.alien
					height
					test
					alien
					name3
					.23
					i

Table 1.2: C++ Keywords and other example identifiers

Note

- In the syntax definition an identifier cannot contain spaces, or special characters other than underscores (_).
- A letter is any alphabetic character (a to z and A to Z).
- A digit is a single number (0 to 9).
- Each item in Table 1.2 is a valid identifier.
- The **keywords** are identifier that has special meaning to the language.
- The **example identifiers** give you examples of the kinds of names that could be given to artefacts we create.

1.3.6 C++ Expression

An **Expression** in C++ is a mathematical calculation or a **C++ Literal** value. Each expression will have a **Type**, and can contain a number of mathematic operators. Table 1.3 lists the operators that you can include in your expressions, listed in order of precedence.⁵ The operators you can use depend on the kind of data that you are using within the expression.

Operator	Description	Example
()	Parenthesis	$(1 + 1) * 2$
% * /	Modulo ^a , Multiplication and Division	$1 / 2 * 5 \% 3$
+ -	Addition and subtraction	$10 + 3 - 4$

^aThe remainder after division. For example 9 modulo 3 is 0, 10 modulo 3 is 1, 11 modulo 3 is 2 etc.

Table 1.3: C++ Operators and Example Expressions

Example Expression	Value	Type
73	73	int
2.1	2.1	float
"Hello World"	"Hello World"	string ^a
"Fred"	"Fred"	string
3 * 2	6	int
1 + 3 * 2	7	int
(1 + 3) * 2	8	int
7 - 3 + 1	5	int
3 / 2	1 ^b	int
3.0 / 2.0	1.5	float
3 % 2	1	int
11 % 3	2	int
3 / 2.0	1.5 ^c	float
1 + (3 / 2.0) + 6 * 2 - 8	6.5	float

^aThis is technically a `char*` which denotes a reference to a `string` of characters.

^bC does integer division for int values, rounding the value down.

^cIf either, or both, values are real (floating point) numbers the result is also a real number.

Table 1.4: Example C++ Expressions and their values

Note

- Table 1.4 shows some example expressions, their values, and types.
- Expressions can be literal values, entered in the code.
- Expression can contain mathematical calculations using standard addition, subtraction, multiplication, division, and groupings.

⁵Expressions follow the standard mathematic order of precedence (BODMAS).

1.3.7 C++ Literal

A literal is either a number or text value written directly in the code. In other words, it is not calculated when the program runs - the value entered is **literally** the value to be used. Figure 1.30 shows the syntax for the different literal values you can enter into your C++ code.

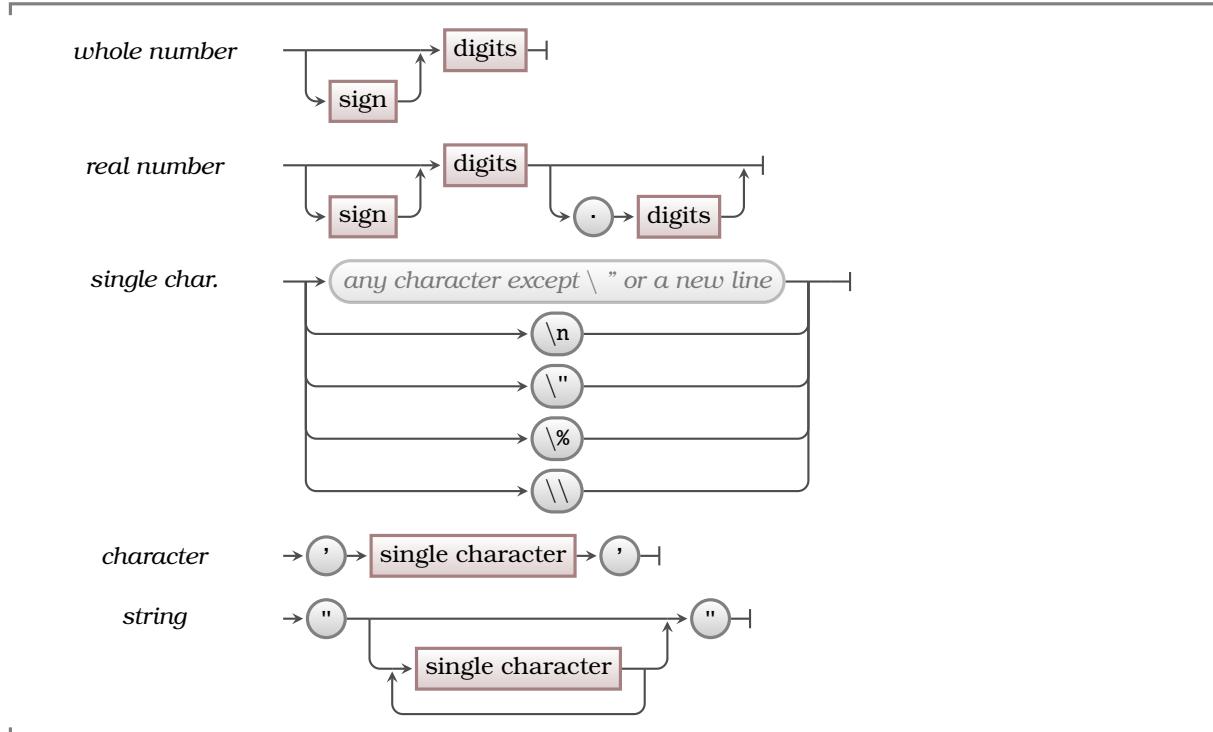


Figure 1.30: C++ Syntax for Literals

Note

- Most of these are self evident. For example, a whole number can be 127, -8711, +10 for example. Real numbers are written as 3.1415 for example.
- Within a string the \ character is used to indicate that the next character has a special meaning. The following list includes the most useful special characters:
 - \n creates a new line
 - \" creates a double quote
 - \% creates a % character
 - \\ creates a \

1.3.8 C++ Types

Types are used to define how data is interpreted and the operations that can be performed on the data. Table 1.5 shows the three basic types of data, the associated C++ type, size in memory, and other related information. Table 1.6 shows the operators that are permitted for each Type.

Whole Number Types			
Name	Size	Range (lowest .. highest)	
short	2 bytes/16 bits	-32,767 .. 32,767	
int	4 bytes/32 bits	-2147483648 .. 2147483647	
int64_t	8 bytes/64 bits	-9,223,372,036,854,775,807 .. 9,223,372,036,854,775,807	

Real Number Types			
Name	Size	Range (lowest .. highest)	Significant Digits
float	4 bytes/32 bits	1.0e-38 .. 1.0e38	6
double	8 bytes/64 bits	2.0e-308 .. 2.0e308	10

Text Types		
Name	Size	Known As
char	1 byte/8 bits	
string	various ^a	c-string

Table 1.5: C++ Data Types

^aThe size in memory is determined by the number of characters within the string, and some overhead

Type	Operations Permitted	Notes
Whole Numbers	() + - / * %	Division rounds down if all values are whole numbers.
Real Numbers	() + - / *	
Text	() +	You can use + for concatenation. ^a

Table 1.6: C++ Permitted Operators by Type

^aTo concatenate literals you **must** tell the compiler to make them strings. This can be done using `string("...")`. See Figure ??.

Note

- The int type is the typical whole number type.
- The double type is the typical real number type.
- C++ builds on the C programming language, and inherits a the ‘c-string’ standard literal. This can be converted to a C++ string, but needs you to indicate this as shown. You cannot perform concatenation on the c-string type, but you can on C++ strings.
- For example values see Table 1.4 on page 41.

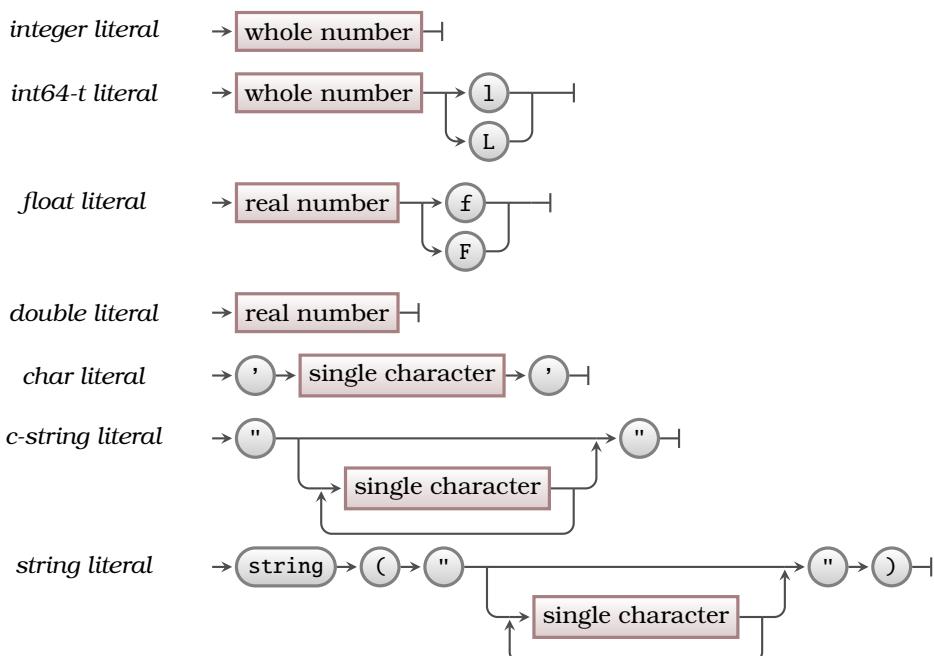


Figure 1.31: C++ Syntax for Typed Literals

1.3.9 C++ SplashKit Terminal Output

The C and C++ programming languages include standard libraries that allow you to write text to the terminal. However, the way the C procedures work is prone to errors (none of which will help you learn about programming) and the C++ way is unique to C++ and not something other languages have adopted. As the focus of this book is on learning to program, not to learn to use any one particular language, we will use the SplashKit procedures to output text. These avoid the issues inherent in the C library, and use a more standard approach than that taken in C++. You can revisit the standard libraries once you are confident in programming in general.

SplashKit includes two procedures to output text to the terminal: `write_line` and `write`

Procedure Prototype	
	<pre>void write_line(string text) void write_line(int text) void write_line(double text) void write(string text) void write(int text) void write(double text)</pre>
Parameter	Description
<code>text</code>	The text that is to be written to the Terminal.

Table 1.7: Parameters that must be passed to write line.

C++

```
#include "splashkit.h"

int main()
{
    write_line("Multi\nLine\nOutput\n");
    // outputs then goes to new line
    write_line("1 + 1 = " + to_string(1 + 1));

    write("3 x 2 = "); // outputs, but does not go to a new line due to write
    write_line(3 * 2); // so this will appear on same line as "3 x 2 ="

    return 0;
}
```

Listing 1.10: C++ write line examples

Note

- To convert a number into a string you can use the [to_string Function](#).
- There are multiple versions of `write_line` so that you can easily output numbers and text.
- The `write_line` procedure writes out the text and then adds a new line, whereas `write` just writes out the text (remaining on the same line).

1.3.10 C++ Comments

Comments allow you to embed documentation and explanatory text within your program's code. The comments are skipped by the compiler, so they have no affect on the program's machine code. You write comments to help yourself and other people understand what you intend the program to do, and any thoughts you want to record along with the code.

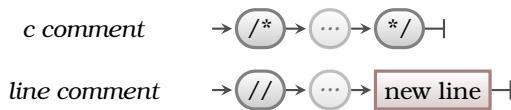


Figure 1.32: C++ Syntax for comments

Note

- Figure 1.32 shows the syntax for comments in C++.
- In standard C++ the first style of comments must be used, `/* Comment */`.
- Most modern C++ compilers also allow single line comments using `// Comment`.
- Standard C++ comments can span multiple lines, these are also known as '*block comments*'.
- A compiler ignores comments when compiling your code.
- You can type almost anything in the comment, represented by the ... in the diagram.

1.3.11 Some Common C Errors with Program Creation

The following list shows some of the more common errors you are likely to encounter at this stage. The C compiler is particularly cryptic with its error messages, but these few should help you with the more common problems.

error: expected ';' before ...

Each statement in C must be ended with a semicolon (;). This error is indicating that the compiler has reached a point where it believes a statement should have ended. Look back at the previous line, it is likely that you forgot to put the ending semicolon.

warning: implicit declaration of function '...'

The compiler doesn't know about the procedure/function that you are calling at this point in the code. This could be caused by one of two common problems. Firstly you may have a typo in the name of the procedure you are calling, check the name carefully and remember that C is case sensitive. Secondly, you may have forgotten to include a library that you are using. If this is the case check the includes at the top of the file.

Undefined symbols: "main", referenced from:...

The entry point for a C program must be called `main`. C is case sensitive, so `Main` and `main` are not the same. Check that you have a `main` function. This error is indicating that the compiler cannot find the program's entry point, it cannot find the `main` function.

warning: control reaches end of non-void function

You are missing the return at the end of the `main` function. Add the code `return 0;`.

error: filename: No such file or directory

This error is likely to occur if you mistype the name of the library's header (.h) file that you are including. Check the `#include<...>` at the start of the code. The error here is indicating that the compiler cannot find the file. If the filename is spelt correctly then there may be an issue with your compiler's installation.

error: expected '=', ',', ';', 'asm' or '__attribute__' before 'main'

You must declare `main` has `int main()`. This error will occur if you have a typo in the `int` part.

error: expected '=', ',', ';', 'asm' or '__attribute__' before '{' token

This is the error message you get if you forget to put the parenthesis after `main` but before the open brace ({).

1.4 Program Creation in Pascal

Section 1.2 on page 22 of this chapter introduced an ‘Output Test’ program, and its design. The pseudocode from this section is shown in Listing 1.11. In this Section you will see the rules for translating this program’s design into the C code shown in Listing 1.4.

Pseudocode

```
-----
Program: Output Test
-----
Steps:
1: Output 'Output Test Program' to the Terminal
2: Output ' 1 + 1 = ', and the result of 1 + 1
3: Output ' Area of a circle with radius 3 = ',
               and the result of 3.1415 * 3 * 3
```

Listing 1.11: Pseudocode for Hello World program (from Listing 1.1).



Pascal

```
//
// OutputTest: Writes some messages to the Terminal.
//
program OutputTest;
uses SplashKit;

begin
  WriteLine('Output Test Program');
  Write(' 1 + 1 = ');
  WriteLine(1 + 1);
  Write(' Area of a circle with radius 3 = ');
  WriteLine(3.1415 * 3 * 3);
end.
```

Listing 1.12: Output Test in Pascal



Note

- Save the Pascal code in a file named `OutputTest.pas`.
- Compile this using `fpc -S2 OutputTest.pas`.
- Run using `./OutputTest`.
- The code at the start is a Comment describing what is in the file, see [Pascal Comments](#).
- Each of the procedure calls contains one or two [Expressions](#) that pass values to the `WriteLn` procedure, which will output these to the Terminal.
- This code uses [strings](#), [int](#), and [float](#) types, see [Pascal Types](#).



1.4.1 Pascal Program

Figure 1.33 shows the structure of the syntax you can use to create a program using the Pascal language.

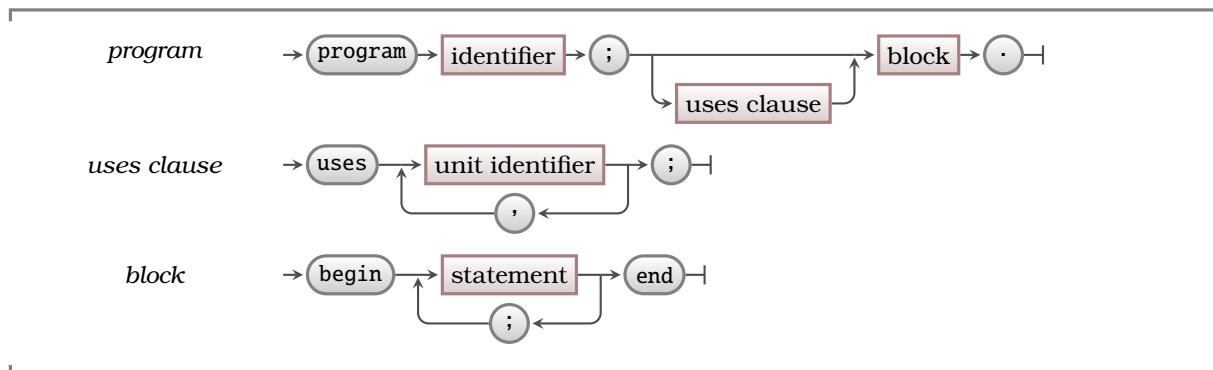


Figure 1.33: Pascal Syntax for a Program

The code in Listing 1.13 shows an example Pascal Program. You should be able to match this up with the syntax defined in Figure 1.33. Notice that the uses clause is skipped, this clause allows you to access code in external libraries (called units in Pascal). By default all Pascal programs have access to the `System` unit which contains the `WriteLn` procedure and many other reusable artefacts.

Pascal

```

program HelloWorld;
uses SplashKit;

begin
  WriteLine('Hello World!');
end.
  
```

Listing 1.13: Pascal Hello World

Note

- A Pascal `Program` starts at the first `Statement` after the program's `begin` keyword (line 2).
- The program ends when the instructions get to the program's `end` keyword (line 4).
- The `WriteLn` procedure comes from a `System` unit (Pascal's name for an external library) that is automatically include in each program.

1.4.2 Pascal Statement

In a [Statement](#) you are commanding the computer to perform an *action*. There are only a small number of statements you can choose from. At this stage the only statement we have discussed is the [Procedure Call](#), formally known as the *procedure statement* in Pascal. This is shown in Figure 1.26, where we can see that at this stage all Statements are calls to [Procedures](#).

```
statement → procedure call
```

Figure 1.34: Pascal Syntax for Statement Syntax

Pascal

```
program Knights;
uses SplashKit;

begin
  //Introduce knights
  WriteLine('We are the Knights who say ''Ni'''');
  WriteLine('We are the keepers of the sacred words:');
  WriteLine('  ''Ni'', ''Peng'', and ''Neee-wom'''');
  WriteLine('The Knights Who Say ''Ni'' demand a sacrifice.');
  WriteLine('We want... a shrubbery!');
end.
```

Listing 1.14: Pascal Knights

Note

- The code in Listing 1.14 contains a [Pascal Program](#).
- This Program contains five procedure calls. See [Pascal Procedure Call](#).
- Each procedure call runs the printf procedure to output text to the Terminal. See the section on [Pascal Terminal Output](#).



1.4.3 Pascal Procedure Call

A procedure call allows you to run the code in a procedure, getting its instructions to run before control returns back to this point in the program.

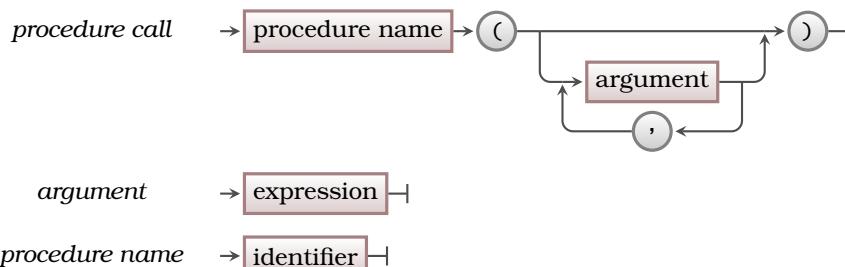


Figure 1.35: Pascal Syntax for Procedure Call Syntax

Pascal

```

program CountBack;
using SplashKit;

begin
  WriteLine('Count back from 2... ');
  WriteLine(2);
  WriteLine(1);
  WriteLine(0);
end.
  
```

Listing 1.15: Pascal Count Back

Note

- A procedure call is an **action** which commands the computer to run the code in a procedure.
- The procedure call starts with the procedure's **Identifier**, this indicates the name of the procedure to be called.
- Following the identifier is a list of values within parenthesis, these are the values (coded as **Expressions**) that are passed to the procedure for it to use.
- The code in Listing 1.15 contains a **Pascal Program**.
- This program contains four procedure calls.
- Each procedure call runs the `WriteLn` procedure to output text to the terminal. See the section on **Pascal Terminal Output**.

1.4.4 Pascal Identifier

The Pascal Identifier syntax is shown in Figure 1.36. In Pascal, as in most programming languages, the identifier must start with an underscore (_) or a letter; in other words your identifiers cannot start with a number or contain other symbols. This is because the compiler needs a way of distinguishing identifiers from numbers entered directly into the code.

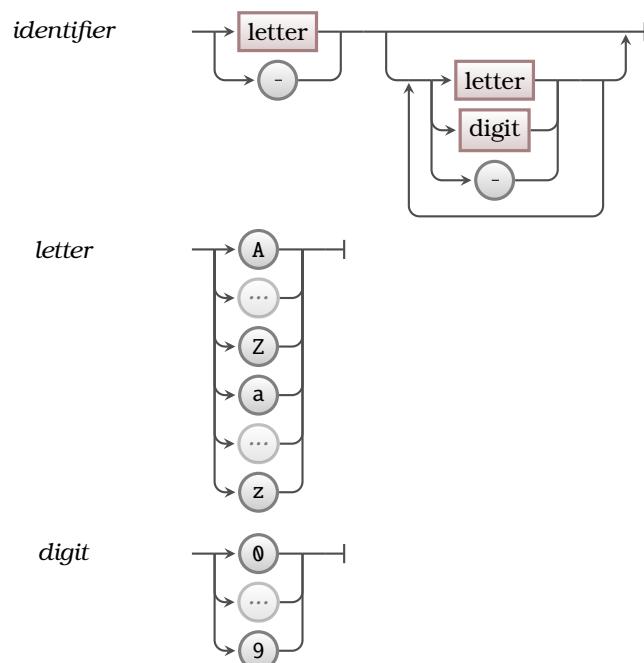


Figure 1.36: Pascal Syntax for an Identifier

Reserved Identifiers (Keywords)					Examples
absolute	and	array	as	asm	WriteLn
begin	case	class	const	constructor	Write
destructor	dispose	dispinterface	div	do	ReadLn
downto	else	end	except	exports	Bitmap
exit	false	file	finalization	finally	myAlien
for	function	goto	if	implementation	age
in	inherited	initialization	inline	interface	height
is	label	library	mod	new	.23
nil	not	object	of	on	i
operator	or	out	packed	procedure	name
program	property	raise	record	reintroduce	test
repeat	resourcestring	self	set	shl	height
shr	string	then	threadvar	to	DrawRectangle
true	try	type	unit	until	FillCircle
uses	var	while	with	xor	CheckRange

Table 1.8: Pascal identifiers

- In the syntax definition an identifier cannot contain spaces, or special characters other than underscores (_).
 - A letter is any alphabetic character (*a* to *z* and *A* to *Z*).
 - A digit is a single number (0 to 9).
 - Each item in Table 1.8 is a valid identifier.
 - The **keywords** are identifier that has special meaning to the language.

1.4.5 Pascal Expression

An **Expression** in Pascal is a mathematical calculation or a literal value. Each expression will have a **Type**, and can contain a number of mathematic operators. Table 1.9 lists the operators that you can include in your expressions, listed in order of precedence.⁶ The operators you can use depend on the kind of data that you are using within the expression.

Operator	Description	Example
()	Parenthesis	$(1 + 1) * 2$
mod * / div ^a	Modulo ^b , Multiplication and Division	$1 / 2 * 5$ $1 \text{ div } 2 * 5$
+ -	Addition and subtraction	$10 + 3 - 4$

^adiv performs an integer division, ignoring any remainder.

^bThe remainder after division. For example 9 modulo 3 is 0, 10 modulo 3 is 1, 11 modulo 3 is 2 etc.

Table 1.9: Pascal Operators and Example Expressions

Example Expression	Value	Type
73	73	Integer
2.1	2.1	Single
'Hello World'	'Hello World'	String
'Hello ' + 'World'	'Hello World'	String
'Fred'	'Fred'	String
3 * 2	6	Integer
1 + 3 * 2	7	Integer
(1 + 3) * 2	8	Integer
7 - 3 + 1	5	Integer
3 / 2	1.5	Single
3 div 2	1	Integer
3 mod 2	1	Integer
11 mod 3	2	Integer
3.0 / 2.0	1.5	Single
3 / 2.0	1.5	Single
1 + (3 / 2.0) + 6 * 2 - 8	6.5	Single

Table 1.10: Pascal Example Expressions

Note

- Table 1.10 shows some example expressions, their values, and types.
- Expressions can be literal values, entered in the code.
- Expression can contain mathematical calculations using standard addition, subtraction, multiplication, division, and grouping.

⁶Expressions follow the standard mathematic order of precedence (BODMAS).

1.4.6 Pascal Literal

A literal is either a number or text value stated directly in the code. In other words, it is not calculated when the program runs - it is already in the code. Figure 1.37 shows the syntax for the different literal values you can enter into your Pascal code.

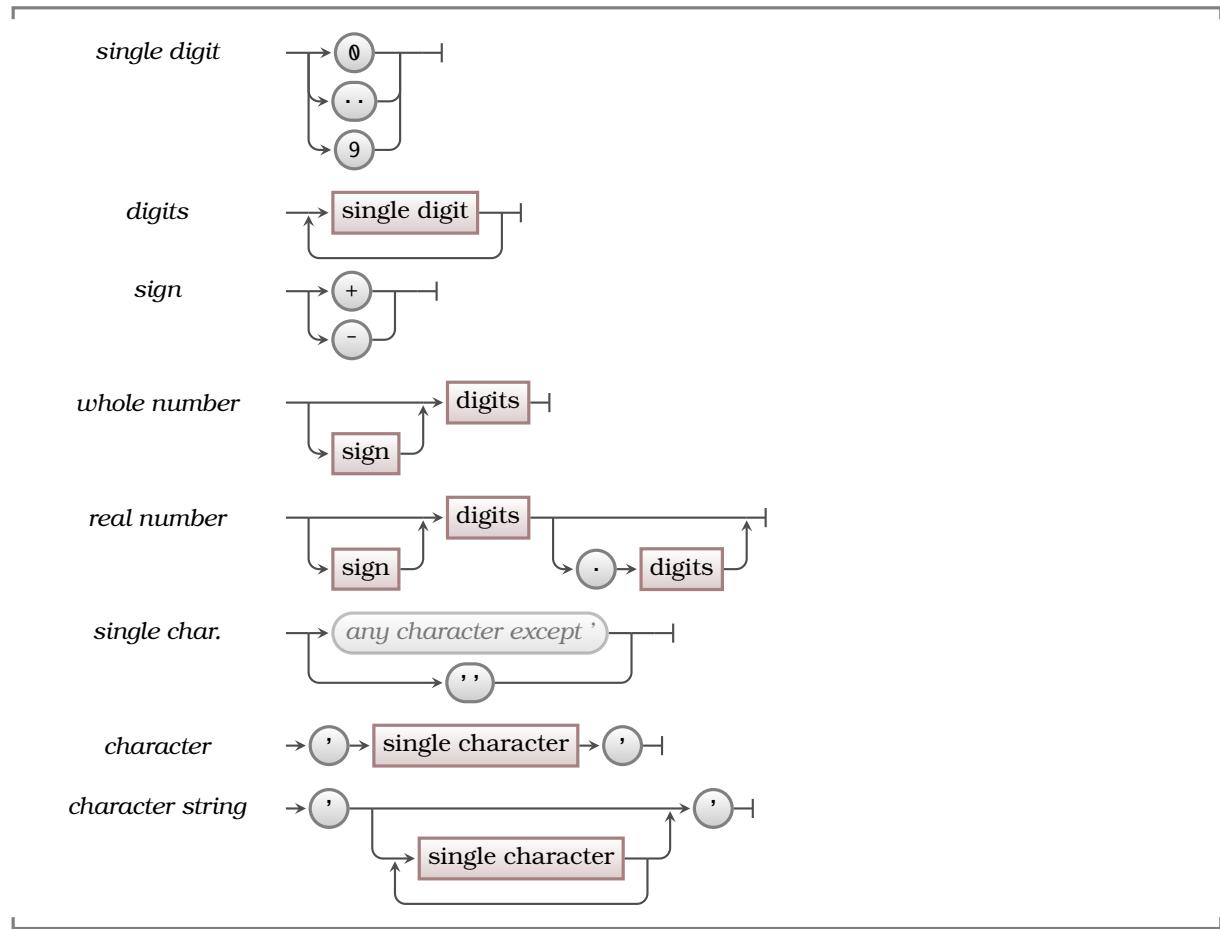


Figure 1.37: Pascal Syntax for Literals

Note

- '0..9' means the digits 0, 1, 2, etc. up to 9.
- To embed a single quote (') in a string you need to use two single quotes, for example `WriteLn('It''s a lovely day!');`

1.4.7 Pascal Types

Types are used to define how data is interpreted and the operations that can be performed on the data. Table 1.11 shows the three basic types of data, the associated Pascal type, size in memory, and other related information. Table 1.12 shows the operators that are permitted for each type.

Whole Number Types			
Name	Size	Range (lowest .. highest)	
Byte	1 bytes/8 bits	0 .. 255	
SmallInt	2 bytes/16 bits	-32,767 .. 32,767	
Integer	4 bytes/32 bits	-2147483648 .. 2147483647	
Int64	8 bytes/64 bits	-9,223,372,036,854,775,807 .. 9,223,372,036,854,775,807	

Real Number Types			
Name	Size	Range (lowest .. highest)	Significant Digits
Single	4 bytes/32 bits	1.0e-38 .. 1.0e38	6
Double	8 bytes/64 bits	2.0e-308 .. 2.0e308	10
Extended	10 bytes/80 bits	1.9e-4932 .. 1.1e4932	20

Text Types		
Name	Size	Notes
Char	1 byte/8 bits	
String	256 bytes/2048 bits	Supports up to 255 characters by default, compiler options allow longer strings.

Table 1.11: Pascal Data Types

Type	Operations Permitted	Notes
Whole Numbers	() + - div / * mod	
Real Numbers	() + - / *	
Text	() +	The + performs concatenation.

Table 1.12: Pascal Permitted Operators by Type

Note

- The Integer type is the typical whole number type.
- The Double type is the typical real number type.
- For example values see Table 1.10 on page 54.

1.4.8 Pascal Terminal Output

Pascal comes with a range of libraries (called units in Pascal) that provide reusable programming artefacts, including reusable [Procedures](#). The `System` unit is automatically included in all Pascal programs. This unit includes artefacts you can use to perform input and output tasks, including procedures to write output to the Terminal.

Procedure Prototype	
procedure Write(...)	
Parameter	Description
...	The <code>Write</code> procedure takes a variable number of parameters. Each of the parameters are written to the Terminal in sequence. See notes for details on formatting numeric values.

Table 1.13: Parameters that must be passed to `Write`

Procedure Prototype	
WriteLn(...)	
Parameter	Description
...	Works in the same way as <code>WriteLn</code> , but also advances to a new line after writing each parameter to the Terminal.

Table 1.14: Parameters that must be passed to `WriteLn`

Pascal

```
program SampleWriteLn;
uses SplashKit;

begin
  Write('Hello ');
  WriteLine('World!');
  WriteLine('Single Line Output... ');
  WriteLine('Multi' + LineEnding + 'Line' + LineEnding + 'Output');
  Write('1 + 1 = ');
  WriteLine(1 + 1);
  WriteLine('It''s a lovely day!');
end.
```

Listing 1.16: Pascal `writeln` examples

Note

- Numeric values can be formatted using two format modifiers. You can specify the minimum number of characters and the number of decimal places.
 - To output with a minimum number of characters use `- value:min` characters.
 - * `WriteLn(2.4:10);` This will write the text ‘ 2.4E+0000’ to the Terminal.
 - * `WriteLn(2.4:-10);` This will write the text ‘2.4E+0000 ’ to the Terminal.
 - To output with decimal places use `- value:min characters:decimal places`.
 - * `WriteLn(2.4:8:3);` This will write the text ‘ 2.400’ to the Terminal.
 - * `WriteLn(2.4:-6.2);` This will write the text ‘2.40 ’ to the Terminal.

1.4.9 Pascal Comments

Comments allow you to embed documentation and explanatory text within your program's code. The comments are skipped by the compiler, so they have no affect on the program's machine code. You write comments to help other people understand what you intend the program to do.

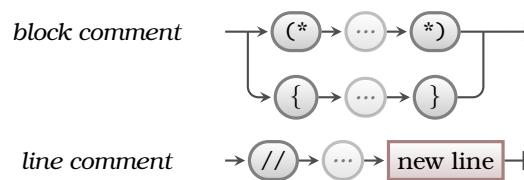


Figure 1.38: Pascal Syntax for comments

Note

- Figure 1.38 shows the syntax for comments in Pascal.
- Block comments can span multiple lines.
- You can type almost anything in the comment, represented by the . . . in the diagram.
- A compiler ignores comments when compiling your code.



1.5 Understanding Program Execution

The earlier Sections of this Chapter have covered the concepts and code related to program creation, but have not looked at how these concepts actually affect the computer when the program is run. This Section illustrates the actions that occur inside the computer when your program is executed. A good understanding of these concepts work will enable you to use them effectively.

This Section will help you answer the following questions:

- What happens when the program is started?
- What happens when the code executes a procedure call?

1.5.1 Starting a Program

Double clicking a program's icon, or launching it from the command line, causes the program to run. This is as much as most normal users need to know about using programs. However, as a Software Developer you need to know more about what is actually happening as you will be the one who defines what the computer does when your program runs.

Starting a program is the responsibility of the *Operating System*. When the program is launched the following steps are performed. A discussion of each of these steps follows.

1. Space is allocated in memory for the Program, and partitioned into areas for the program's **code**, and the call **Stack**.
2. The program's code is loaded into memory, into the **code** Section.
3. A **frame** is added to the **Stack** with the location of the first instruction in the program.
4. The computer starts running the instructions based on the current frame in the stack.

Allocating Memory for the Program

To start the program the Computer first needs to get the program's instructions into memory. This task is performed by the Operating System when the program is launched. The Operating System allocates memory for the program to use, and partitions this memory into different areas. Each area will be used to store different kinds of information needed by the program. An illustration of this is shown in Figure 1.39.

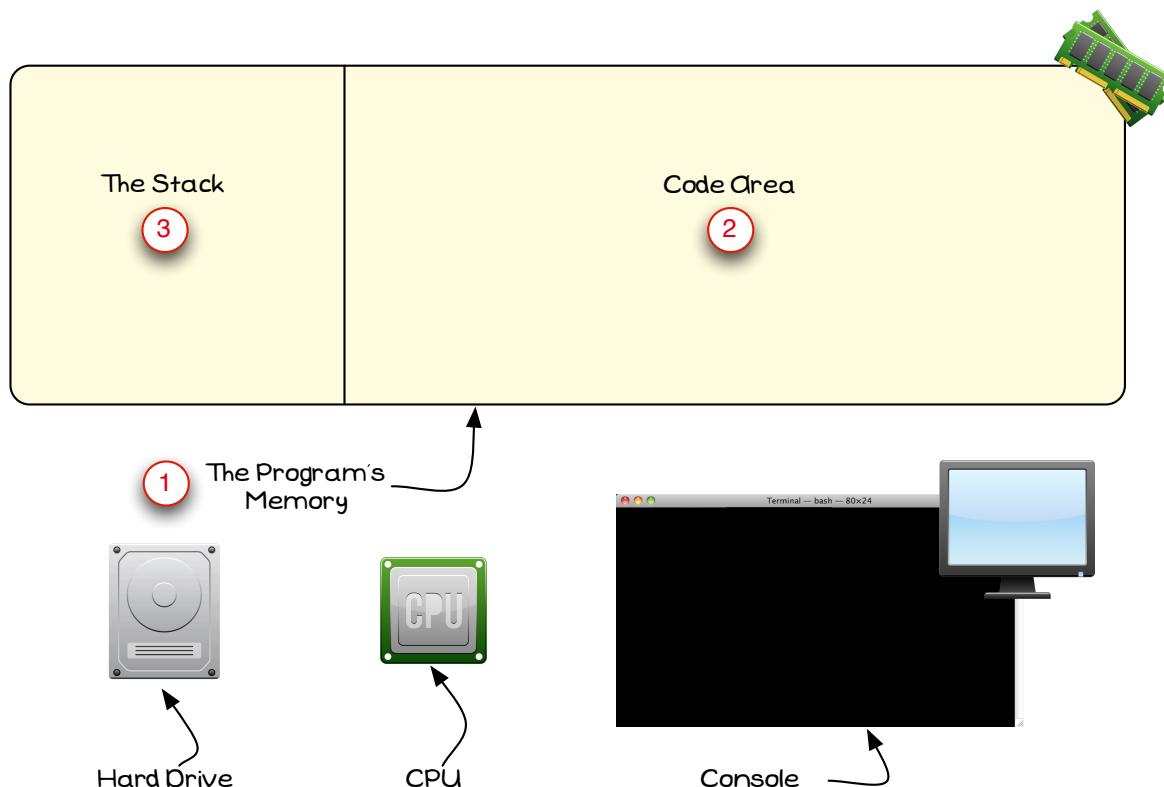


Figure 1.39: Operating System prepares memory for the Program

Note

- In Figure 1.39 the indicated areas show the following:
 1. The Operating System allocates memory for the program.
 2. Part of the allocated memory will be designated to store the program's instructions. This can be thought of as the *Code Area*.
 3. Another part of the allocated memory will be set aside to keep track of the current instruction. This area is called the *Stack*.

Loading the Code

Having allocated the program some memory, and partitioned this space into the **Stack** and **Code** area, the Operating System then reads the program's instructions from the executable file and loads these into the Code area. This is illustrated in Figure 1.40.

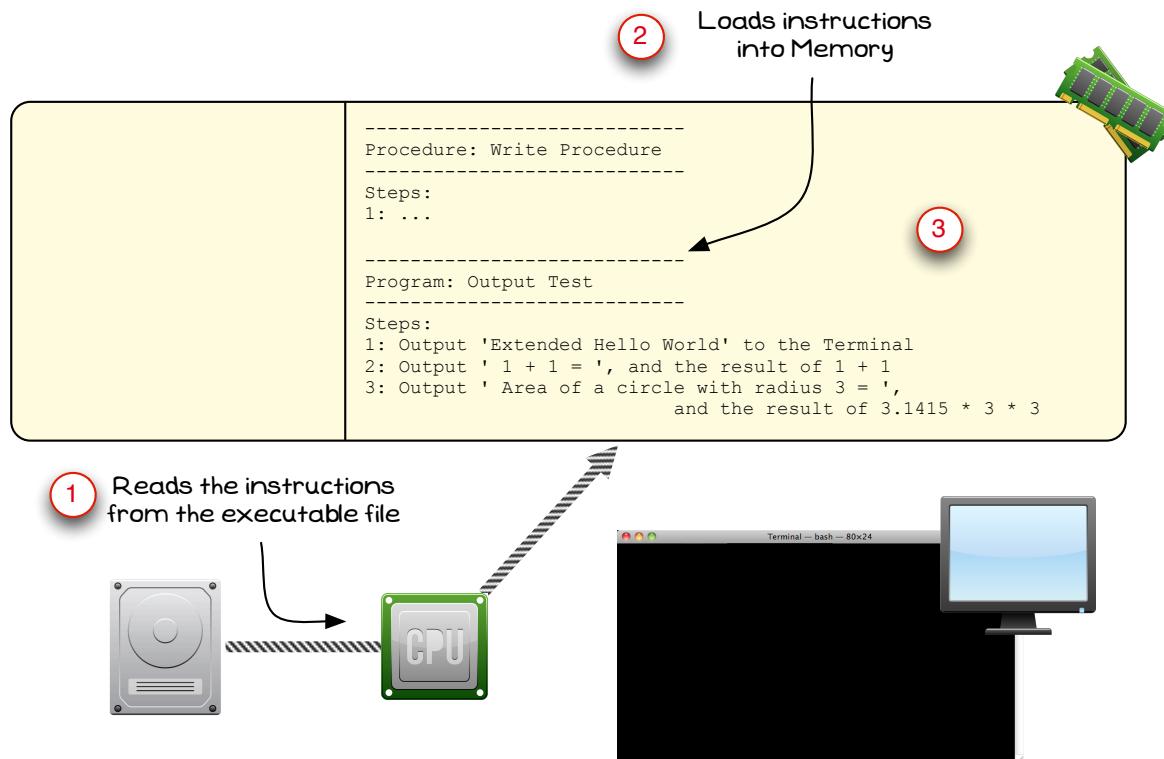


Figure 1.40: The Operating System loads the program's code into memory

Note

- In Figure 1.40 the indicated areas show the following:
 - The program's instructions are read from the executable file the user launched.
 - The instructions are stored into the program's memory: into the code area.
 - When this finishes, all of the program's instructions are loaded into memory.
 In Figure 1.40 the instructions are shown as the pseudocode from Listing 1.1. In reality these will be the *machine code* instructions that were saved into the executable file by the compiler.
- The *Operating System* is a software component that is used to control access to the hardware. In this case the Operating System takes the responsibility for setting up the machine so that it can run the program the user launched.

Setting Up The First Instruction

Now that the code is loaded into memory, the Operating System uses the details saved in the executable file to setup the program's first instruction. This will be loaded onto the Stack, which is responsible for keeping track of the current instruction. The compiler will have used the program's **entry point** to store these details when the program was compiled. This is shown in Figure 1.41.

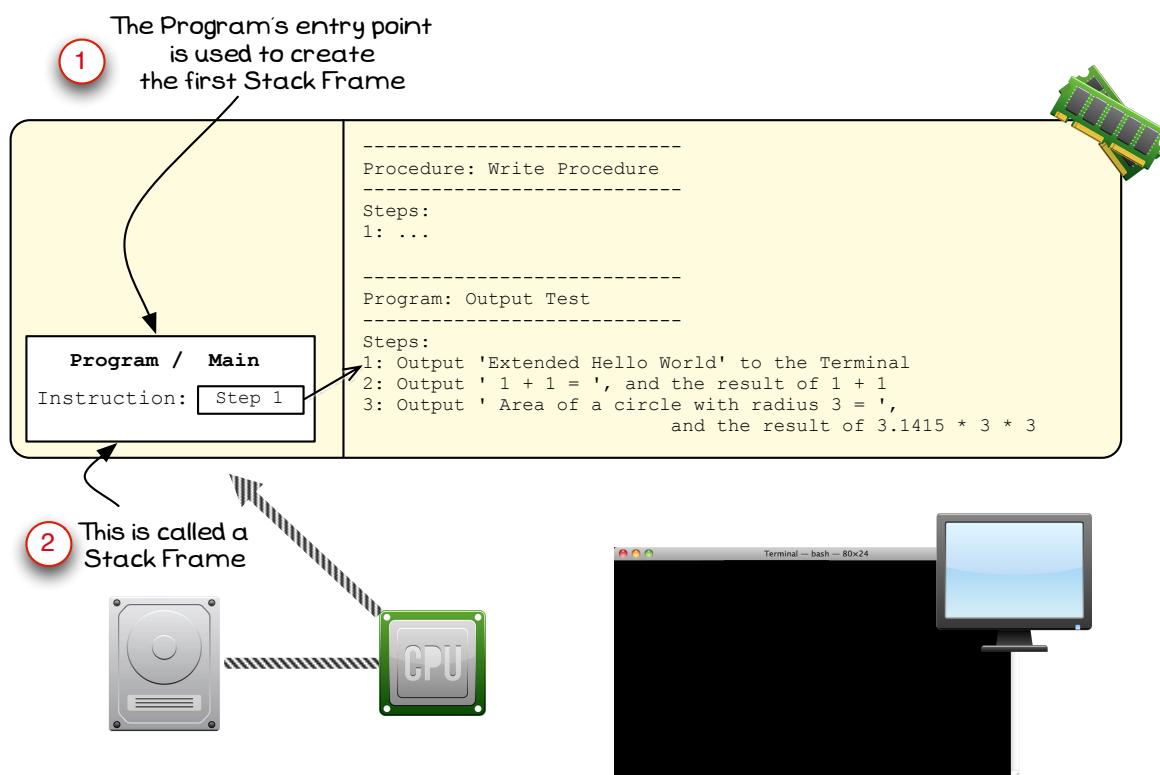


Figure 1.41: The current instruction is loaded onto the Stack

Note

- In Figure 1.41 the indicated areas show the following:
 1. The program's instruction is tracked on *The Stack*.
 2. Each Stack Frame keeps a record of the current instruction within a Procedure. This area is called the Stack as the Frames are *stacked* one on top of the other. The one on the top of the *stack* tells the computer which instruction is to be run.
- The Stack keeps track of the current instruction.
- The current instruction refers to the code loaded into the Code area.
- The compiler will have saved the details for which instruction is first into the executable file.
- In your code the program's **entry point** tells you which instruction will be first.

Running The First Instruction

The Operating System has finally finished loaded the Program, and can now start its instructions running. The CPU uses the **Current Instruction** that is on the top of the Stack, locates the Code, and runs the instruction. In this case this is a procedure call to a Procedure that writes output to the Terminal. When this instruction completes, the text *Output Test Program* will have appeared on the Terminal for the user to see. The results of this are shown in Figure 1.42.

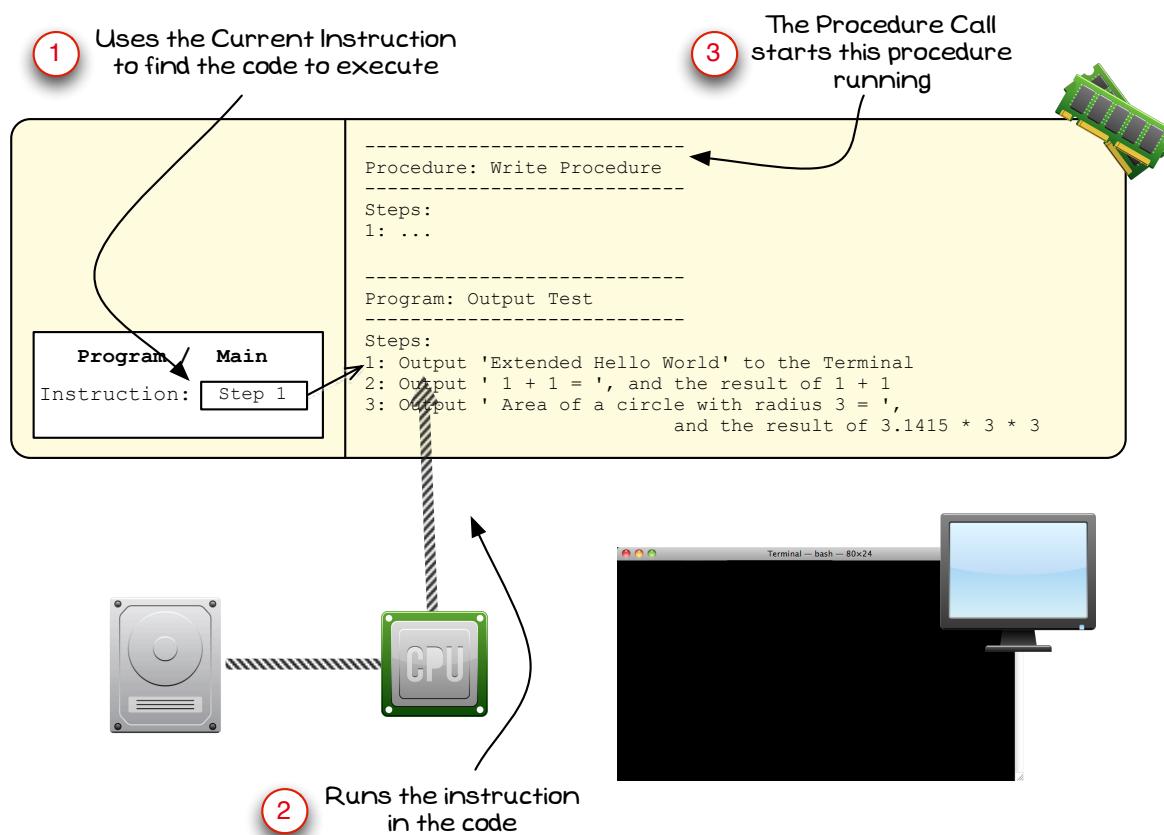


Figure 1.42: The Computer runs the first instruction, outputting details to the Terminal

Note

- In Figure 1.42 the indicated areas show the following:
 - The current instruction is read from the Frame on the top of the Stack.
 - The Computer runs the instruction from the code loaded into memory.
 - The instruction is a procedure call, which starts the execution of the Write Procedure.
- The computer runs the code **one** instruction at a time.
- When that instruction is finished it moves onto the next instruction. This is important, and means that are programs run the commands in **Sequence**.
- Writing data to the Terminal takes more than a single instruction, the procedure call gets the Computer to run the instructions in the called Procedure.

1.5.2 Calling the Write Procedure

At this point the Computer has been instructed to Call the Write Procedure. This procedure⁷ will output the data passed to it to the Terminal. In order to do this, the instructions within the called procedure need to be followed.

Calling Write for the first time

Each Procedure contains instructions that when followed get the Computer to perform a task. The procedure call sets up the Stack so that the instructions within the Write Procedure are executed.

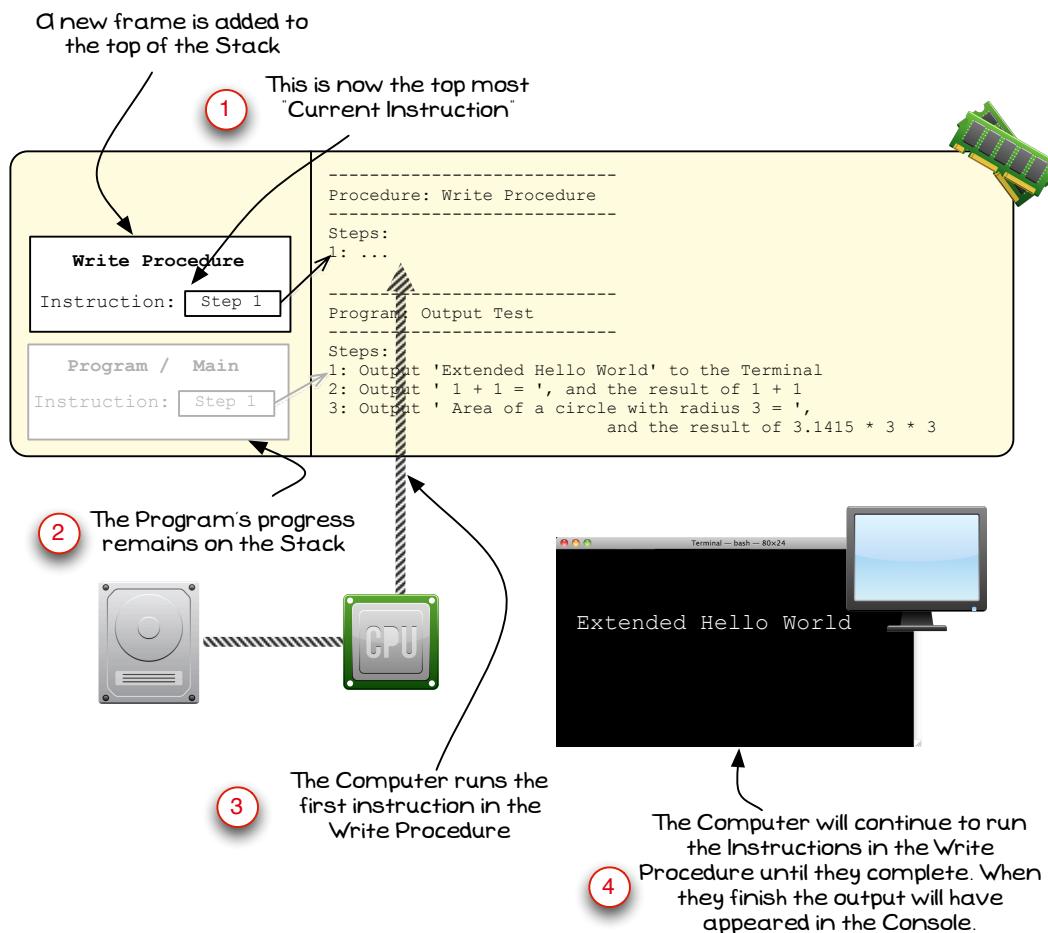


Figure 1.43: The Write Procedure is called, and has its instructions executed

Note

- In Figure 1.43 the indicated areas show the following:
 - A new Frame is added to the Stack for the Write Procedure.
 - The new Frame appears on top of the Frame that has the program's Current Instruction.
 - Now the Computer can run the instructions within the Write Procedure.
 - The instructions are run one at a time until the Write Procedure finishes. At this time the first output will have appeared on the Terminal.

⁷The printf procedure in C and the WriteLn procedure in Pascal.

The Write Procedure Ends

When the Write Procedure's instructions finish control needs to return to the code that called it, in this case the program's code.

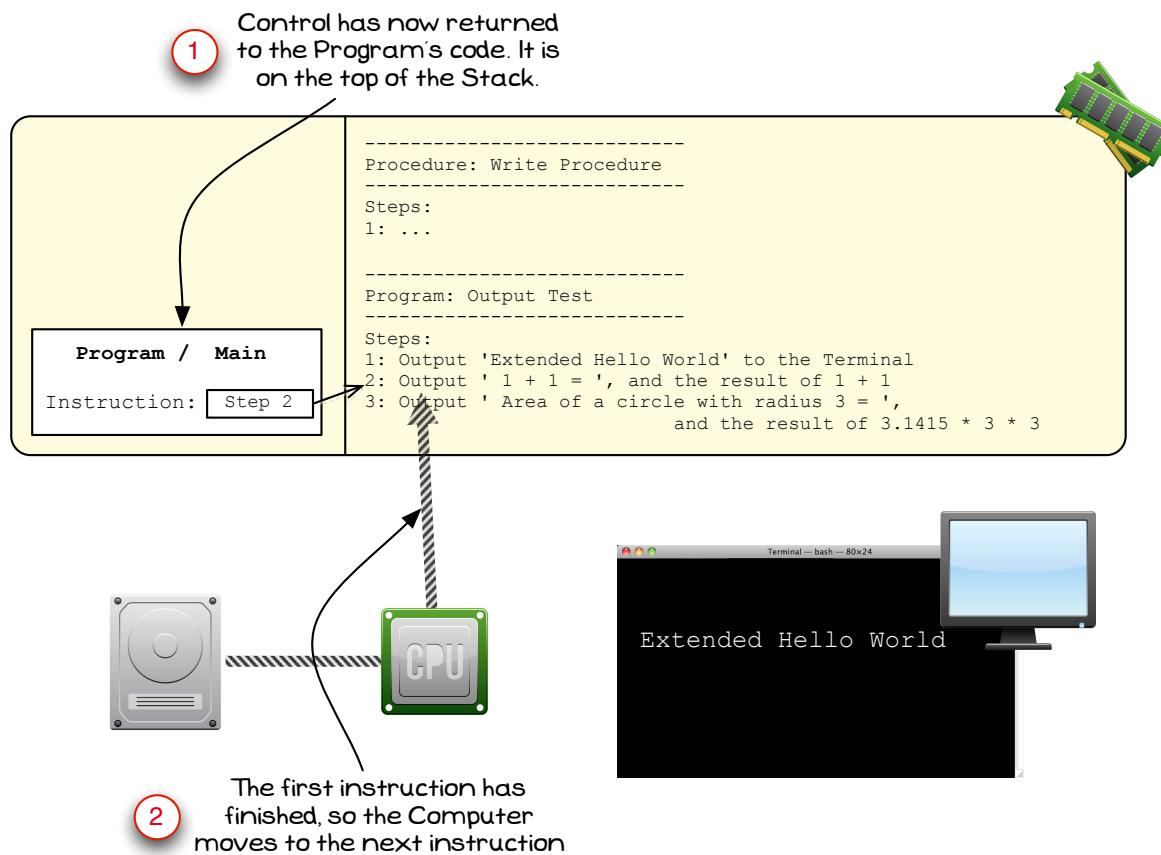


Figure 1.44: The Write Procedure is called, and has its instructions executed

Note

- In Figure 1.44 the indicated areas show the following:
 - When the Write Procedure finishes its Frame is removed from the Stack, and control returns to the program's code.
 - The first instruction on the program's code has now finished, so the Computer moves to the second instruction.

One way to visualise this is to picture the Program, and the Procedure, as a book of instructions. Imagine you are told to follow the instructions in a book. You get the book, place it on a table and read the first instruction which tells you to perform the *Write Procedure*. You leave the original book on the table, and fetch the *Write Procedure* book and place it on top of the book on the table, thereby creating a Stack of books. Now you can follow the instructions, one by one, from the book on top of the Stack. When you finish the last instruction you take the book off the top of the stack and return to the book beneath it. This will enable you to perform the steps within the Procedures without forgetting where you are up to in the earlier code.

The Second Call to Write

The second instruction in the program's code is another call to the Write procedure.

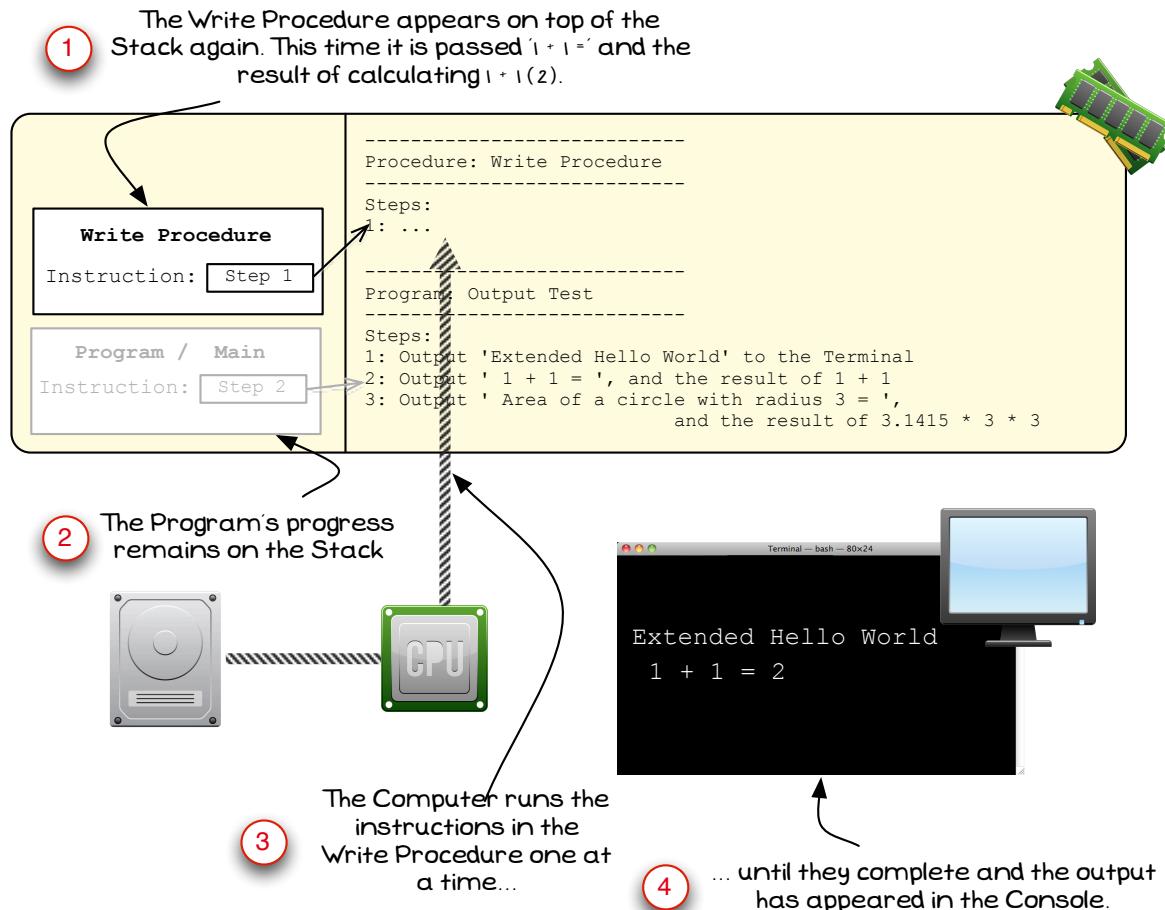


Figure 1.45: The Write Procedure is called a second time

Note

- In Figure 1.45 the indicated areas show the following:
 1. A new Frame is created for the Write Procedure. This allows the Computer to keep track of which statement is the current statement within this Procedure.
 2. The program's current instruction remains on the Stack for when the call to the Write procedure finishes.
 3. Each of the instructions in the Write procedure are executed to write the data to the Terminal.
 4. The output text appears on the Terminal.

The Second Write Call Ends

When the second call to Write ends control returns to the Program, which moves on to its final instruction.

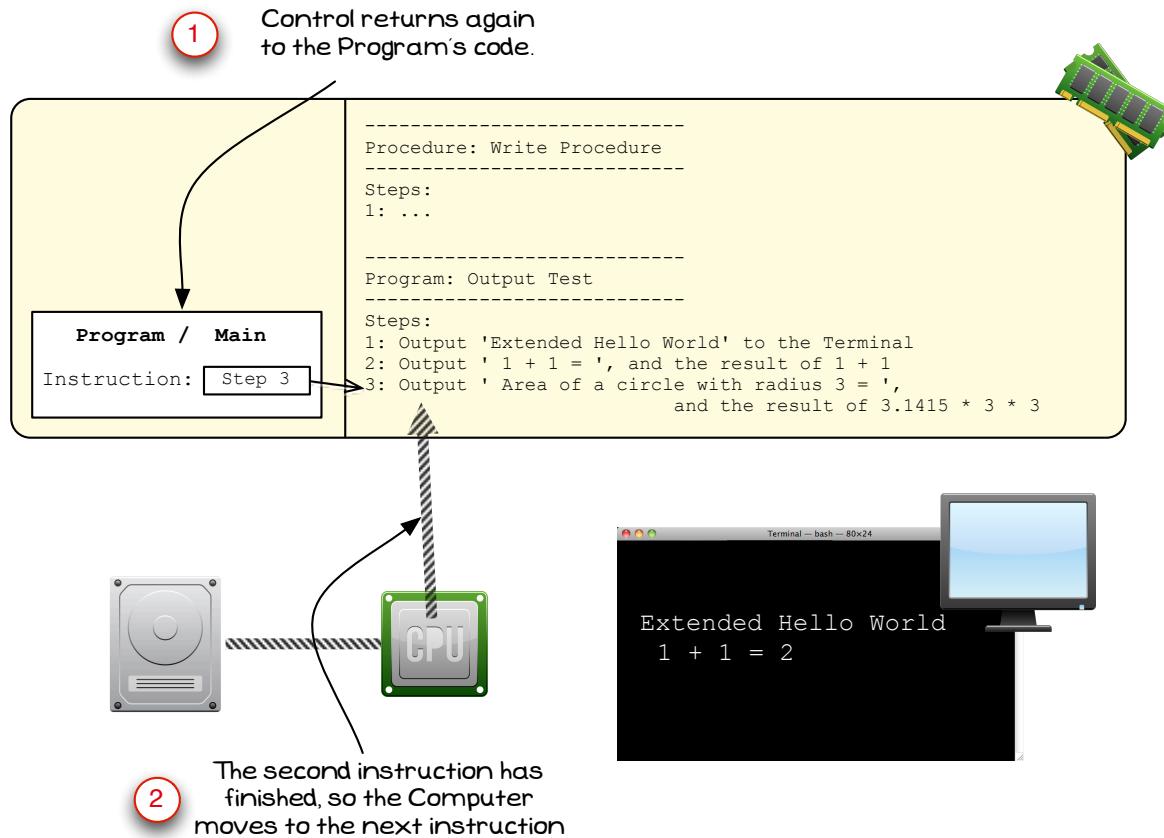


Figure 1.46: The second call to the Write Procedure ends, and control returns to the Program

Note

- In Figure 1.46 the indicated areas show the following:
 1. When the second call to the Write Procedure ends control returns to the program's code.
 2. As the second instruction has now finished the Computer moves to the third, and final, instruction in the program.

The Third Call to Write

The final instruction in the program is a third call to the Write Procedure.

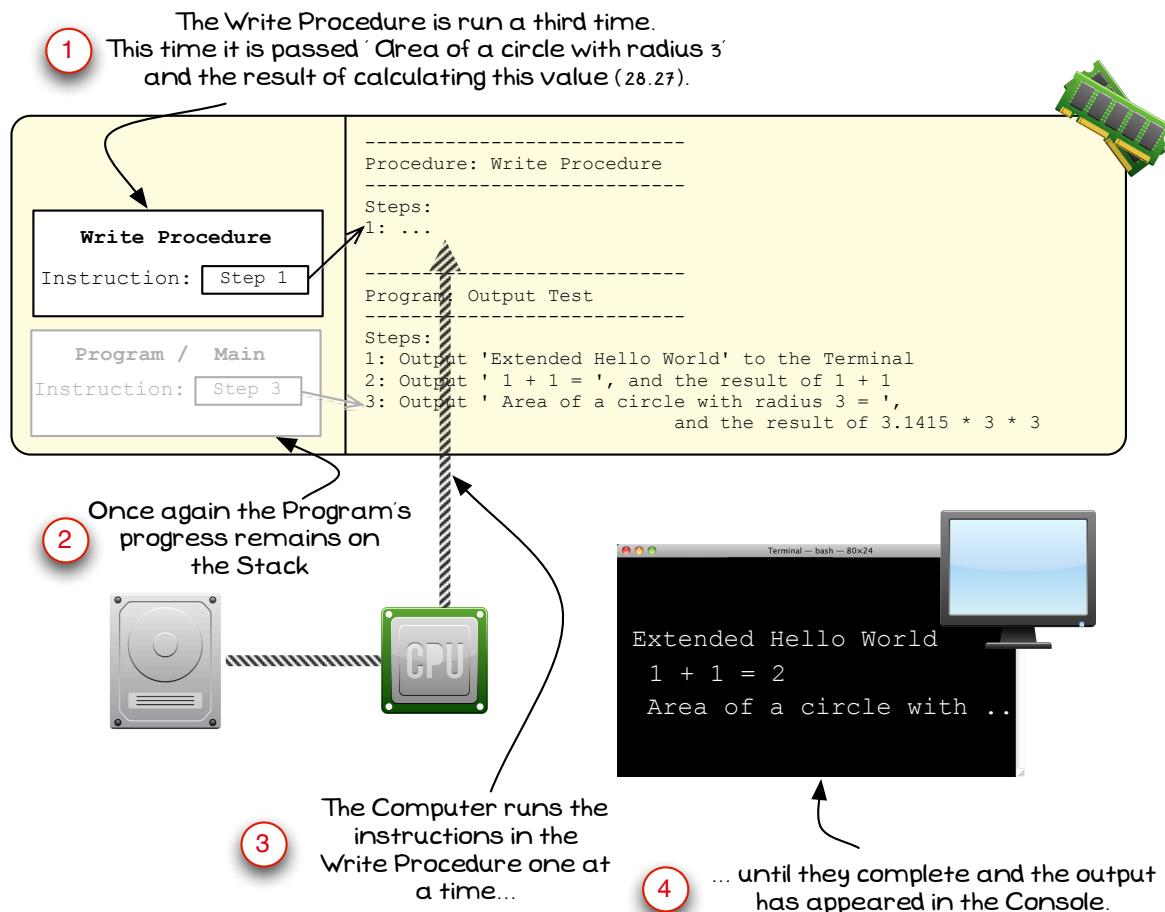


Figure 1.47: The Write Procedure is called a third can final time

Note

- In Figure 1.47 the indicated areas show the following:
 - Once again, a Stack Frame is created to keep track of the progress within the Write Procedure.
 - The program's current instruction remains on the Stack so that the Computer can return to it when the Write Procedure ends.
 - Each of the instructions in the Write Procedure are executed.
 - The Write Procedure writes the data passed to it to the Terminal.

The Program Ends

When the last instruction in the Write Procedure finishes control return back to the Program, which has no more instructions. This means that the program has finished.

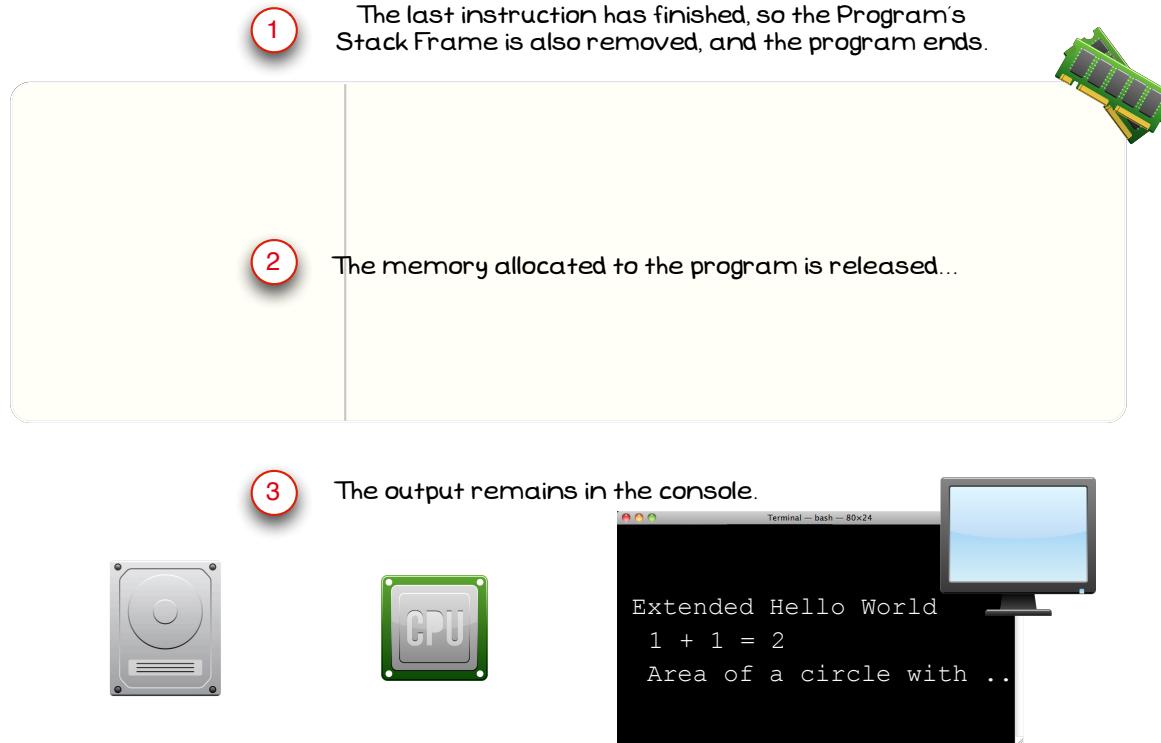


Figure 1.48: The Write Procedure ends, then the program ends

Note

- In Figure 1.48 the indicated areas show the following:
 1. The Write Procedure has finished its instructions, and so it ends. This was also the last instruction in the program's code, so it also ends. This tells the Operating System that Program has finished.
 2. The Operating System releases the memory used by the program so that it can be used by other programs.
 3. All of this will have happened in an instant, and after the program has finished all that remains is the output in the Terminal.

1.5.3 Summary

In this section you have seen the actions that occur behind the scenes when your program is executed. The most important aspects are the fact that the instructions run one at a time in **sequence**, and that a procedure call results in the instructions within the Procedure running until they end. Its also important to remember that the instructions within the Procedure must finish before control returns to where the call was made.

Note

- A Program is a sequence of instructions.
- These instructions are organised into Procedures that can be called.
- The program's instructions are loaded into memory when the program is launched.
- Instructions are run one at a time.
- The Stack is used to keep track of the current instruction.
- When the program starts a Frame is added to the Stack to set the entry point as the first instruction.
- When a Procedure is called, a Frame is added to the Stack to keep track of its current instruction.



1.6 Program Creation Examples

1.6.1 Seven Times Table

This program prints out the seven times table from 1×7 to 10×7 . The description of the program is in Table 1.15, the pseudocode in Listing 1.17, the C code in Listing 1.18, and the Pascal code in Listing 1.19.

Program Description	
Name	Seven Times Table
Description	Displays the Seven Times Table from 1×7 to 10×7 .

Table 1.15: Description of the Seven Times Table program

Pseudocode

```
-----
Program: SevenTimesTable
-----
Steps:
1: Output 'Seven Times Table' to the Terminal
2: Output '-----' to the Terminal
3: Output ' 1 x 7 = ', and 1 * 7 to the Terminal
4: Output ' 2 x 7 = ', and 2 * 7 to the Terminal
5: Output ' 3 x 7 = ', and 3 * 7 to the Terminal
6: Output ' 4 x 7 = ', and 4 * 7 to the Terminal
7: Output ' 5 x 7 = ', and 5 * 7 to the Terminal
8: Output ' 6 x 7 = ', and 6 * 7 to the Terminal
9: Output ' 7 x 7 = ', and 7 * 7 to the Terminal
10: Output ' 8 x 7 = ', and 8 * 7 to the Terminal
11: Output ' 9 x 7 = ', and 9 * 7 to the Terminal
12: Output '10 x 7 = ', and 10 * 7 to the Terminal
13: Output '-----' to the Terminal
```

Listing 1.17: Pseudocode for Seven Times Table program.



C++

```
/*
 * Program: seven_times.c
 * Displays the Seven Times Table from 1 x 7 to 10 x 7.
 */

#include <stdio.h>

int main()
{
    printf("Seven Times Table\n");
    printf("-----\n");
    printf(" 1 x 7 = %d\n", 1 * 7);
    printf(" 2 x 7 = %d\n", 2 * 7);
    printf(" 3 x 7 = %d\n", 3 * 7);
    printf(" 4 x 7 = %d\n", 4 * 7);
    printf(" 5 x 7 = %d\n", 5 * 7);
    printf(" 6 x 7 = %d\n", 6 * 7);
    printf(" 7 x 7 = %d\n", 7 * 7);
    printf(" 8 x 7 = %d\n", 8 * 7);
    printf(" 9 x 7 = %d\n", 9 * 7);
    printf("10 x 7 = %d\n", 10 * 7);
    printf("-----\n");

    return 0;
}
```

Listing 1.18: C Seven Times Table**Pascal**

```
//
// Program: SevenTimesTable.pas
// Displays the Seven Times Table from 1 x 7 to 10 x 7.
//
program SevenTimesTable;
begin
    WriteLn('Seven Times Table');
    WriteLn('-----');
    WriteLn(' 1 x 7 = ', 1 * 7);
    WriteLn(' 2 x 7 = ', 2 * 7);
    WriteLn(' 3 x 7 = ', 3 * 7);
    WriteLn(' 4 x 7 = ', 4 * 7);
    WriteLn(' 5 x 7 = ', 5 * 7);
    WriteLn(' 6 x 7 = ', 6 * 7);
    WriteLn(' 7 x 7 = ', 7 * 7);
    WriteLn(' 8 x 7 = ', 8 * 7);
    WriteLn(' 9 x 7 = ', 9 * 7);
    WriteLn('10 x 7 = ', 10 * 7);
    WriteLn('-----');
end.
```

Listing 1.19: Pascal Seven Times Table

1.6.2 Circle Area

This program prints out the area of circles with different radius. The description of the program is in Table 1.16, the pseudocode in Listing 1.20, the C code in Listing 1.21, and the Pascal code in Listing 1.22.

Program Description	
Name	<i>Circle Areas</i>
Description	Displays the Circle Areas for circles with radius from 1.0 to 5.0 with increments of 0.5.

Table 1.16: Description of the Circle Areas program

Pseudocode

```
-----
Program: CircleAreas
-----
Steps:
1: Output 'Circle Areas' to the Terminal
2: Output '-----' to the Terminal
3: Output ' Radius: 1.0 = ', 3.1415 * 1.0 * 1.0 to the Terminal
4: Output ' Radius: 1.5 = ', 3.1415 * 1.5 * 1.5 to the Terminal
5: Output ' Radius: 2.0 = ', 3.1415 * 2.0 * 2.0 to the Terminal
6: Output ' Radius: 2.5 = ', 3.1415 * 2.5 * 2.5 to the Terminal
7: Output ' Radius: 3.0 = ', 3.1415 * 3.0 * 3.0 to the Terminal
8: Output ' Radius: 3.5 = ', 3.1415 * 3.5 * 3.5 to the Terminal
9: Output ' Radius: 4.0 = ', 3.1415 * 4.0 * 4.0 to the Terminal
10: Output ' Radius: 4.5 = ', 3.1415 * 4.5 * 4.5 to the Terminal
11: Output ' Radius: 5.0 = ', 3.1415 * 5.0 * 5.0 to the Terminal
12: Output '-----' to the Terminal
```

Listing 1.20: Pseudocode for Circle Area program.



C++

```
/*
 * Program: circle_areas.c
 * Displays the Circle Areas for circles with radius
 * from 1.0 to 5.0 with increments of 0.5.
 */

#include <stdio.h>

int main()
{
    printf("Circle Areas\n");
    printf("-----\n");
    printf(" Radius: 1.0 = %4.2f\n", 3.1415 * 1.0 * 1.0);
    printf(" Radius: 1.5 = %4.2f\n", 3.1415 * 1.5 * 1.5);
    printf(" Radius: 2.0 = %4.2f\n", 3.1415 * 2.0 * 2.0);
    printf(" Radius: 2.5 = %4.2f\n", 3.1415 * 2.5 * 2.5);
    printf(" Radius: 3.0 = %4.2f\n", 3.1415 * 3.0 * 3.0);
    printf(" Radius: 3.5 = %4.2f\n", 3.1415 * 3.5 * 3.5);
    printf(" Radius: 4.0 = %4.2f\n", 3.1415 * 4.0 * 4.0);
    printf(" Radius: 4.5 = %4.2f\n", 3.1415 * 4.5 * 4.5);
    printf(" Radius: 5.0 = %4.2f\n", 3.1415 * 5.0 * 5.0);
    printf("-----\n");

    return 0;
}
```

Listing 1.21: C Circle Areas

Pascal

```
//
// Program: circle_areas.c
// Displays the Circle Areas for circles with radius
// from 1.0 to 5.0 with increments of 0.5.
//

program CircleAreas;
begin
    WriteLn('Circle Areas');
    WriteLn('-----');
    WriteLn(' Radius: 1.0 = ', (3.1415 * 1.0 * 1.0):4:2);
    WriteLn(' Radius: 1.5 = ', (3.1415 * 1.5 * 1.5):4:2);
    WriteLn(' Radius: 2.0 = ', (3.1415 * 2.0 * 2.0):4:2);
    WriteLn(' Radius: 2.5 = ', (3.1415 * 2.5 * 2.5):4:2);
    WriteLn(' Radius: 3.0 = ', (3.1415 * 3.0 * 3.0):4:2);
    WriteLn(' Radius: 3.5 = ', (3.1415 * 3.5 * 3.5):4:2);
    WriteLn(' Radius: 4.0 = ', (3.1415 * 4.0 * 4.0):4:2);
    WriteLn(' Radius: 4.5 = ', (3.1415 * 4.5 * 4.5):4:2);
    WriteLn(' Radius: 5.0 = ', (3.1415 * 5.0 * 5.0):4:2);
    WriteLn('-----');
end.
```

Listing 1.22: Pascal Circle Areas

1.6.3 Shape Drawing

This program draws some shapes to the screen using the **SwinGame** Software Development Kit (SDK). The SwinGame SDK is a library that provides a number of reusable code artefacts that you can use to create 2D games. This SDK is available for both C and Pascal, and work on Linux, Mac, and Windows.

The description of the program is in Table 1.17, the pseudocode in Listing 1.23, the C code in Listing 1.24, and the Pascal code in Listing 1.25.

Program Description	
Name	<i>Shape Drawing</i>
Description	Draws a number of shapes to the screen using SwinGame.

Table 1.17: Description of the Shape Drawing program

Pseudocode

```
-----
Program: Shape Drawing
-----
Steps:
1: Call OpenGraphicsWindow with 'Shape Drawing', 800, and 600
2: Call ClearScreen
3: Call FillRectangle with ColorWhite, 10, 10, 780, and 580
4: Call RefreshScreen
5: Call Delay with 500
6: Call FillCircle with ColorRed, 50, 50, and 25
7: Call FillCircle with ColorGreen, 80, 50, and 25
8: Call FillCircle with ColorBlue, 110, 50, and 25
9: Call RefreshScreen
10: Call Delay with 500
11: FillTriangle with ColorYellow, 100, 100, 150, 175, 210, and 115
12: Call RefreshScreen
13: Call Delay with 2000
14: Call ReleaseAllResources
```

Listing 1.23: Pseudocode for Shape Drawing program.



The Lines from the program will do the following:

- **OpenGraphicsWindow** opens a Window with the title ‘Shape Drawing’ that is 800 pixels wide by 600 pixels high.
- **ClearScreen** clears the screen to black.
- **FillRectangle** uses the color, the x, y location, and width and height to fill a rectangle.
- **RefreshScreen** updates the screen to show what has been drawn. All SwinGame drawing is done offscreen, and only drawn to the screen when RefreshScreen is called.
- **Delay** pauses the program for a number of milliseconds, so 500 will wait for half a second.
- **FillCircle** uses the color, given x, y location and radius to fill a circle.
- **FillTriangle** fills a triangle with the given x, y points (6 values for 3 points).

C++

```
/*
 * Program: shape_drawing.c
 * Draws a number of shapes to the screen using SwinGame.
 */
#include <stdio.h>
#include <stdbool.h>
#include "SwinGame.h"

int main()
{
    open_graphics_window("Shape Drawing", 800, 600);
    load_default_colors();

    clear_screen();

    fill_rectangle(ColorWhite, 10, 10, 780, 580);

    refresh_screen();
    delay(500);

    fill_circle(ColorRed, 50, 50, 25);
    fill_circle(ColorGreen, 80, 50, 25);
    fill_circle(ColorBlue, 110, 50, 25);

    refresh_screen();
    delay(500);

    fill_triangle(ColorYellow, 100, 100, 150, 175, 210, 115);
    refresh_screen();
    delay(2000);

    release_all_resources();
    return 0;
}
```

Listing 1.24: C Shape Drawing Code

The SwinGame procedure for C are named using the standard C naming scheme. The names are:

- **open_graphics_window** opens a Window with the title ‘Shape Drawing’ that is 800 pixels wide by 600 pixels high.
- **load_default_colors** loads default colors for use in your code.
- **clear_screen** clears the screen to black.
- **fill_rectangle** uses the color, the x, y location, and width and height to fill a rectangle.
- **refresh_screen** updates the screen to show what has been drawn. All SwinGame drawing is done offscreen, and only drawn to the screen when RefreshScreen is called.
- **delay** pauses the program for a number of milliseconds, so 500 will wait for half a second.
- **fill_circle** uses the color, given x, y location and radius to fill a circle.
- **fill_triangle** fills a triangle with the given x, y points (6 values for 3 points).

Pascal

```
//  
// Program: ShapeDrawing.pas  
// Draws a number of shapes to the screen using SwinGame.  
//  
program GameMain;  
uses  
  sgTypes, sgInput, sgAudio, sgGraphics, sgResources, sgUtils, sgText;  
  
begin  
  OpenGraphicsWindow('Shape Drawing', 800, 600);  
  
  ClearScreen();  
  
  FillRectangle(ColorWhite, 10, 10, 780, 580);  
  
  RefreshScreen();  
  Delay(500);  
  
  FillCircle(ColorRed, 50, 50, 25);  
  FillCircle(ColorGreen, 80, 50, 25);  
  FillCircle(ColorBlue, 110, 50, 25);  
  
  RefreshScreen();  
  Delay(500);  
  
  FillTriangle(ColorYellow, 100, 100, 150, 175, 210, 115);  
  RefreshScreen();  
  Delay(2000);  
  
  ReleaseAllResources();  
end.
```

**Listing 1.25:** Pascal Shape Drawing Code

1.7 Program Creation Exercises

1.7.1 Concept Questions

Read over the concepts in this chapter and answer the following questions:

1. What is a **Program**? What does it contain?
 2. A program is an artefact, something you can create in code. Why would you want to create a program your code?
 3. What is the entry point of a program?
 4. Do you have to write all of the code for your program, or are you able to use artefacts from elsewhere?
 5. What is a **Statement**?
 6. What statement was introduced in this chapter?
 7. What is a **Procedure**?
 8. Where can you find procedures you may be interested in using?
 9. What is an **Expression**?
 10. Where are expressions coded? Give an example of an expression being used in code.
 11. What are the three broad kinds of **Types** that a language will provide?
 12. What are the names of the types in the language you are using? Name the main type you are likely to work with for each of the broad kinds of types.
 13. What are the values of the following expressions? Which types could these values be used with (possibly multiple)? Note: some answers are dependent of the language you are using.

(a) 5	(d) 3.1415	(g) 2 + 1 / 2.0 * 6
(b) 3 + 5 * 2	(e) 1 / 2	(h) "Fred Smith" (C)
(c) (3 + 5) * 2	(f) 1.0 / 2.0	(i) 'Fred Smith' (Pascal)
 14. Where can statements be coded in your program?
 15. What is the role of an **Identifier**? What artefacts be be identified?
 16. What is a keyword?
 17. Which of the following are valid identifiers?

(a) hello	(d) my_name	(g) void	(j) a1
(b) _123	(e) my_name	(h) Main	(k) 3.1415
(c) fred	(f) begin	(i) 1234	(l) WOW_COOL
 18. What is a **Library**, and what does it contain?
 19. What happens to **Comments** when you code is compiled? Why do languages include these? Why are they considered important by good developers?
 20. What is the stack? What does it keep track of when your program is running?
 21. What is loaded into memory when your program is started?
 22. What happens in the computer when a procedure is called?
-

1.7.2 Code Writing Questions: Applying what you have learnt

Apply what you have learnt to the following tasks:

1. Write a program that prints the 5 times table from $1 * 5$ to $10 * 5$. See Table 1.18.
 - Think about the artefacts you will create, and use.
 - Write pseudocode for the program's instructions
 - Convert your pseudocode to either C or Pascal
 - Compile and Run your program, and check that the values are correctly calculated

Program Description	
Name	Description
<i>Five Times Table</i>	Displays the 5 times table from 1×5 to 10×5 .

Table 1.18: Description of the *Five Times Table* program.

2. Write a program that prints the powers⁸ of 2 from 2^1 to 2^8 . See Table 1.19.
 - Think about the artefacts you will create, and use.
 - Write pseudocode for the program's instructions
 - Convert your pseudocode to either C or Pascal
 - Compile and Run your program, and check that the values are correctly calculated

Program Description	
Name	Description
<i>Powers of Two</i>	Displays the powers of 2 from 2^1 to 2^8 .

Table 1.19: Description of the *Five Times Table* program.

3. Write a program that prints the 73 times table from $1 * 73$ to $10 * 73$. See Table 1.20.
 - Think about the artefacts you will create, and use.
 - Write pseudocode for the program's instructions
 - Convert your pseudocode to either C or Pascal
 - Compile and Run your program, and check that the values are correctly calculated

Program Description	
Name	Description
<i>Seventy Three Times Table</i>	Displays the 73 times table from 1×73 to 10×73 .

Table 1.20: Description of the *Seventy Three Times Table* program.

⁸In the code you will need to calculate these manually using times ($2^1 = 2$, $2^2 = 2*2$, $2^3 = 2*2*2$, etc.)

4. Write a program that prints a table showing calculations of circle dimensions. This should output the radius, circle area, diameter, and circumference of circles with a radius of 1cm, 1.5cm, and 2cm. See Table 1.21.
- Find the necessary calculations and think about the artefacts you will use and create.
 - Write pseudocode for the program's instructions.
 - Convert your pseudocode to either C or Pascal.
 - Compile and Run your program, and check that the values are correctly calculated.

Program Description	
Name	Circle Dimensions
Description	Displays a table of circle dimensions for circles with a radius of 1cm, 1.5cm, and 2cm. This will output the radius, circle area, diameter, and circumference of circles.

Table 1.21: Description of the *Circle Dimensions* program.

5. Write a program with SwinGame that draws a face using primitive shapes. See Table 1.22.
- Draw up an outline of the program. Work out the coordinates of the circles for the face, circles. Determine three points of the triangle for the mouth.
 - Write up the pseudocode for the program's instructions. Remember to use the RefreshScreen and Delay procedures to see the results.
 - Convert your pseudocode to either C or Pascal
 - Compile and Run your program, and check that the values are correctly calculated

Program Description	
Name	Face Shape
Description	Displays face to the screen using SwinGame.

Table 1.22: Description of the *Face Shape* program.

1.7.3 Extension Questions

If you want to further your knowledge in this area you can try to answer the following questions. The answers to these questions will require you to think harder, and possibly look at other sources of information.

1. C and Pascal are imperative programming languages. What does this mean, and how does it relate to the way you think about code?
2. Artefacts are things that you can create. What artefacts were introduced in this chapter? If you could create these artefacts, what would you do with them?

2

Storing and Using Data

How it is time to turn your attention to a the finer details of spell craft. So far the magic you are working has not made use of material components. fetch the Dragon scales from off the chest in the corner, and place them in that goblet. Prepare you wand, and.....

So far our programs have only been able to work with Literal values. This Chapter introduces new artefacts you can use to store data, and from which you can read the value stored called Variables and Constants. It will also introduce a new artefact that can be used to calculate values called a Function. Using these artefacts you will see how you can store and manipulate values in your code.

When you have understood the material in this chapter you will be able to write code that uses Variable and Constants to stores and manipulates data. You will be able to share data between different Procedures, and create and use Functions to calculate values.

Contents

2.1 Concepts Related to Storing and Using Data	83
2.1.1 Variable	85
2.1.2 Constant	86
2.1.3 Local Variable	87
2.1.4 Global Variable	88
2.1.5 Parameter	89
2.1.6 Pass by Value and Pass by Reference	90
2.1.7 Statement (with Assignment)	91
2.1.8 Assignment Statement	92
2.1.9 Function	93
2.1.10 Function Call	94
2.1.11 Expressions (with Function Calls, Variables, and Constants)	95
2.1.12 Program (with functions)	96
2.1.13 Summary	97
2.2 Using these Concepts	98
2.2.1 Designing Change	98
2.2.2 Understanding the Change Calculator	99
2.2.3 Choosing Artefacts for the Change Calculator	100
2.2.4 Writing the Code for the Change Calculator	108
2.2.5 Compiling and Running the Change Calculator	109
2.3 Storing and Using Data in C	111
2.3.1 Implementing Change Calculator in C	111
2.3.2 Implementing Change Calculator using C++	113

2.3.3 C Variable Declaration	115
2.3.4 C Program (with Global Variables and Constants)	117
2.3.5 C Procedure Declaration (with Local Variables)	119
2.3.6 C++ Assignment Statement	120
2.3.7 C Procedure Declaration (with Parameters)	122
2.3.8 C Procedure Call (with pass by reference)	124
2.3.9 C Terminal Input	126
2.3.10 C Program (with Functions)	129
2.3.11 C Function Declaration	130
2.3.12 C Procedure Declaration (as Function)	131
2.3.13 C Function Call	132
2.3.14 Return Statement	133
2.3.15 C Statement (with Return Statement)	134
2.4 Storing and Using Data in Pascal	135
2.4.1 Implementing Change Calculator in Pascal	135
2.4.2 Pascal Program (with Global Variables and Constants)	137
2.4.3 Pascal Procedure Declaration (with Local Variables)	139
2.4.4 Pascal Assignment Statement	140
2.4.5 Pascal Procedure Declaration (with Parameters)	142
2.4.6 Pascal Terminal Input	144
2.4.7 Pascal Program (with Functions)	145
2.4.8 Pascal Function Declaration	147
2.4.9 Pascal Function Call	148
2.5 Understanding Data	149
2.5.1 Variables	149
2.5.2 Parameters, Locals and Globals	154
2.5.3 Function Return Values	161
2.5.4 Pass by Reference	166
2.5.5 Hand Execution with Variables	171
2.5.6 Summary	177
2.6 Data Examples	178
2.6.1 Times Table	178
2.6.2 Circle Area	181
2.6.3 Bicycle Race	189
2.7 Data Exercises	194
2.7.1 Concept Questions	194
2.7.2 Code Writing Questions: Applying what you have learnt	196
2.7.3 Extension Questions	197

2.1 Concepts Related to Storing and Using Data

Chapter ??, ??, showed you how you can create your own procedures, with each procedure performing a task for the program. The procedures that you created did use some data, but in each case the data was entered directly into the code of the program, as a Literal.

This next step introduces the idea of creating your own artefacts that can be used to **store**, or **calculate** a value. Using these artefacts you can start to work with values in a more dynamic way, allowing you to get the computer to perform calculations, and to store and manipulate values.

In this Chapter you will learn how to create the following programming **artefacts**:

- **Variable**: You can **store** a value in a Variable, and **retrieve** the value from the Variable.
- **Constant**: Is similar to a Variable, except that its value cannot change after it is declared.
- **Function**: Is similar to a **Procedure**, but is used to calculate a value rather than to produce a side effect.

You will learn about the following **terminology**:

- **Global Variable**: Variables declared within the program's code are called Global Variables.
- **Local Variable**: Variables declared within a Function or Procedure are called Local Variables.
- **Parameter**: Parameters are variables that allow values to be passed into a Function or Procedure.
- **Expressions (with Function Calls, Variables, and Constants)**: See how Functions, Constant, and Variables can be used in Expressions.

Additionally, you will learn how to perform the following **actions**:

- **Assignment Statement**: You use an Assignment Statement to store a value in a Variable.
- **Function Call**: This is part of an Expression, and is used to call the Function and to read the returned result.

You may need to revise the following programming artefacts:

- **Program**: The idea of building your own Programs.
- **Procedure**: Creating your own Procedure, as well as calling Procedures from libraries.

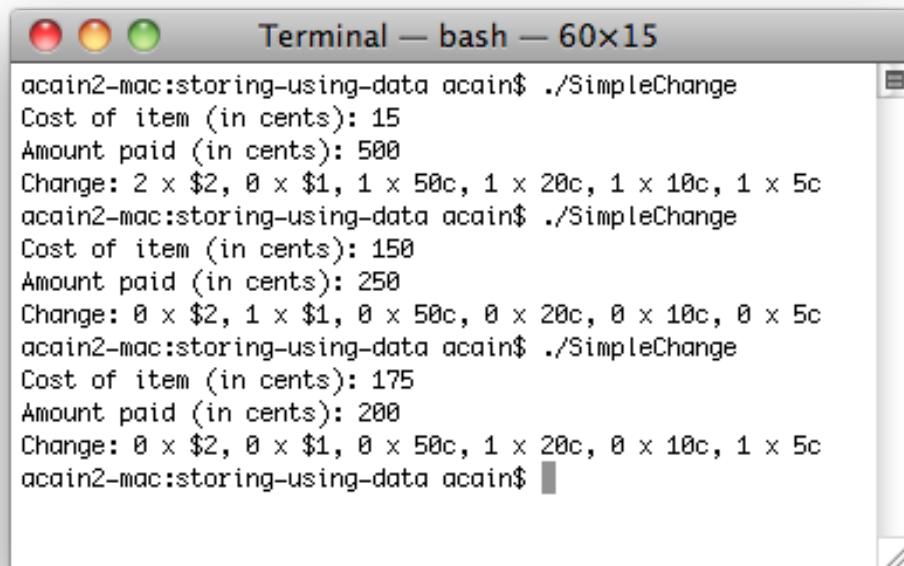
The following programming terminology will also be used in this Chapter:

- **Statement**: An instruction performed in your code.
- **Identifier**: The name of an artefact, or the text used to identify something meaningful to the language.

This material also requires that you have a good understanding of the following actions:

- **Procedure Call**: A procedure call is an instruction to run a Procedure.

By the end of this material we will have worked through an example where you create a program that calculates change for a vending machine. This program will read the cost and amount paid from the user, and will then output the number of coins that need to be returned. The output of several runs of this program are shown in Figure 2.1.



The image shows a Mac OS X terminal window titled "Terminal — bash — 60x15". The window contains three separate runs of a program named "SimpleChange". Each run prompts for the cost of an item and the amount paid, then calculates the change using US coin denominations (\$2, \$1, 50¢, 20¢, 10¢, 5¢). The first run costs 15 cents and is paid with 500 cents. The second run costs 150 cents and is paid with 250 cents. The third run costs 175 cents and is paid with 200 cents. The terminal window has a standard OS X look with red, yellow, and green close buttons at the top left.

```
acain2-mac:storing-using-data acain$ ./SimpleChange
Cost of item (in cents): 15
Amount paid (in cents): 500
Change: 2 x $2, 0 x $1, 1 x 50c, 1 x 20c, 1 x 10c, 1 x 5c
acain2-mac:storing-using-data acain$ ./SimpleChange
Cost of item (in cents): 150
Amount paid (in cents): 250
Change: 0 x $2, 1 x $1, 0 x 50c, 0 x 20c, 0 x 10c, 0 x 5c
acain2-mac:storing-using-data acain$ ./SimpleChange
Cost of item (in cents): 175
Amount paid (in cents): 200
Change: 0 x $2, 0 x $1, 0 x 50c, 1 x 20c, 0 x 10c, 1 x 5c
acain2-mac:storing-using-data acain$
```

Figure 2.1: The Change Calculator running in the Terminal

2.1.1 Variable

A Variable is a **container** into which you can store a value, which can then be retrieved later. The Variable allows you to store values you want to work with in your program, you store values in the variable so that you can read them back later. The variable's themselves are either a **Global Variable**, **Local Variable**, or **Parameter**.

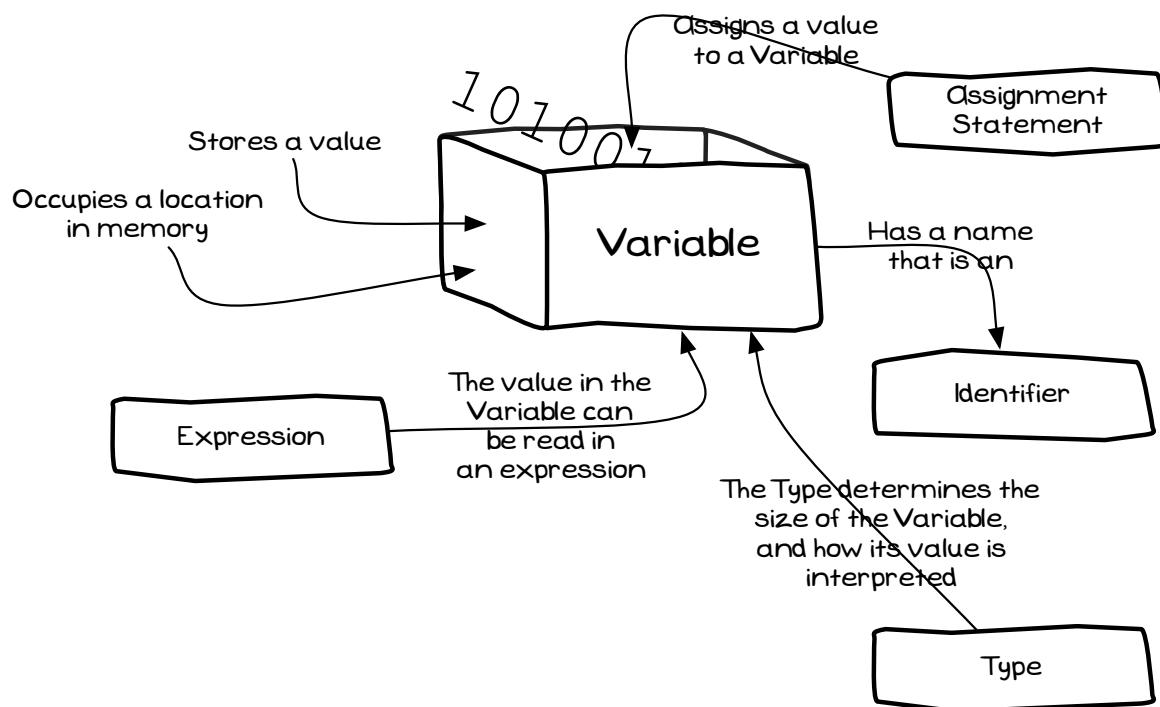


Figure 2.2: Variables store a value that can be read and changed

Note

- A Variable is an **artifact**, you can create variables to store values in your programs.
- You can think of a Variable like a "box with an item in it". The Variable is the box, its value is the item within it.
- Each Variable has a ...
 - **Name** that can be used to refer to it.
 - **Value** that it is storing.
 - **Type** that determines the size of the Variable and how its value is interpreted.
- You use an **Assignment Statement** to store a *value* into the Variable.
- You can **read** the *value* from Variable in Expressions.
- The Variable is **different** to its value:
 - The Variable is a container into which a value can be stored.
 - You can read the *value* from the Variable.
 - The Variable **is not** the value, it is a container into which the value is stored.



2.1.2 Constant

A Constant is just like a [Variable](#), but its value cannot be changed. Constants are declared within the Program, and given a value when they are created. Once they are created the value within the Constant cannot be changed. This is useful for data where you do not want the value changing during the program's execution.

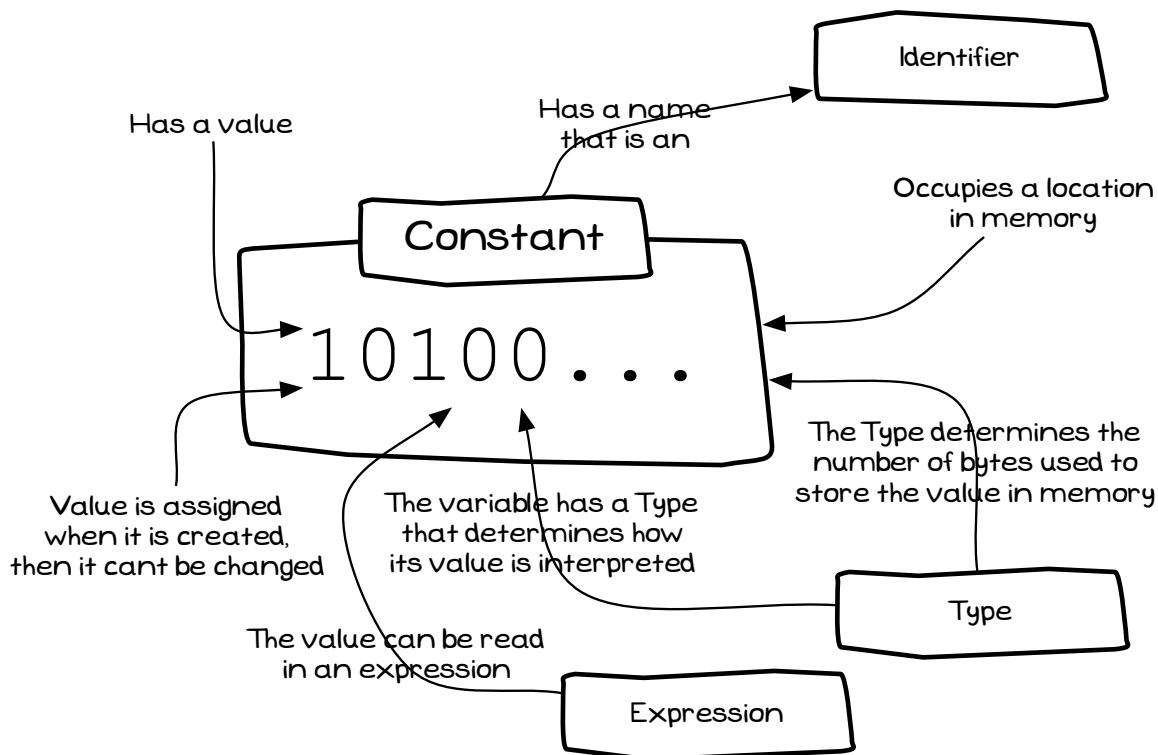


Figure 2.3: Constants have a value that cannot be changed

Note

- A Constant is an **artefact**. You can create Constants in your Program to store values that must not change.
- A Constant is similar to a [Variable](#), they have a...
 - **Name** that is used to access them.
 - **Value** that can be read in an Expression.
 - **Type** that determines how their data is interpreted.
- You **read** *values* of Constants in Expressions.
- Constants are useful for data you do not want to change during the program.
- The name of the Constant is an [Identifier](#).
- The Constant's name should reflect the value it is storing.

2.1.3 Local Variable

Variables can be declared at a number of different places in your code. Variables that are declared within Procedures are called **Local Variables**. Most of the variables in your code will be Local Variables.

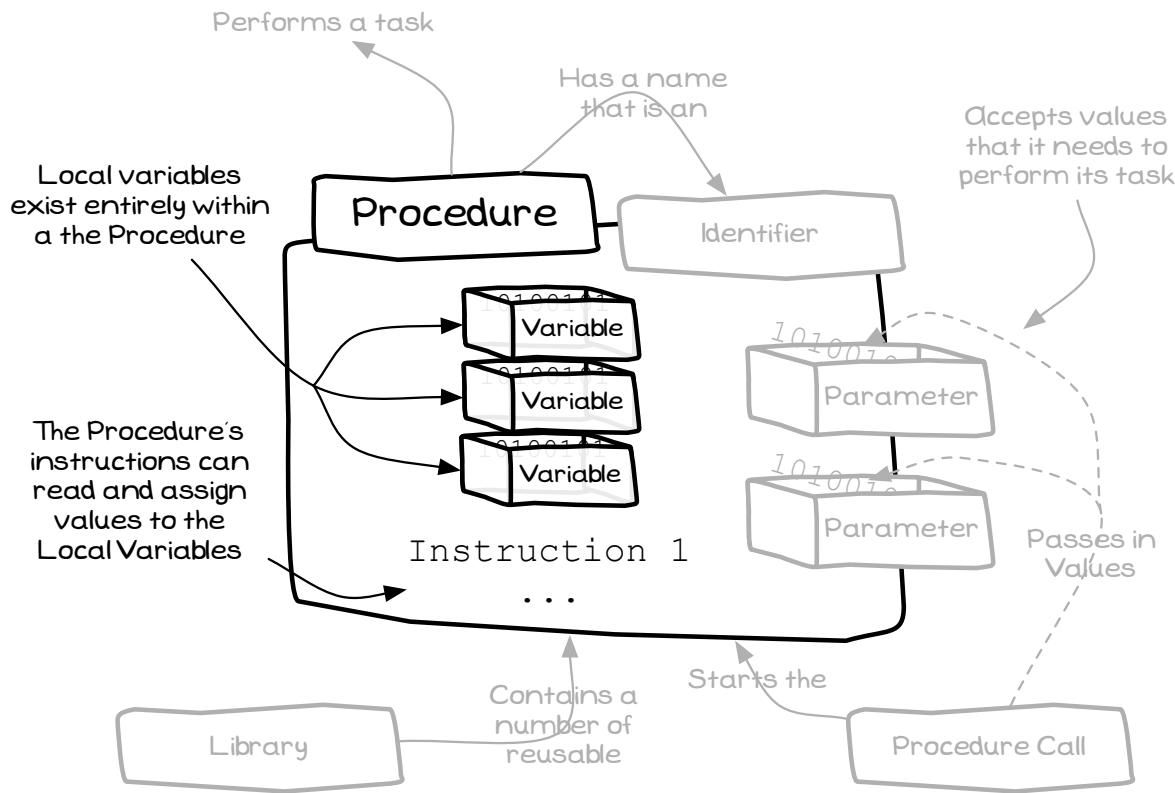


Figure 2.4: Variables declared within a Procedure are Local Variables

Note

- Local Variable is the **term** given to a Variable that is declared within a Procedure.
- Variables that are declared within Procedures are called **Local Variables**.
- Local Variables are located **within** the Procedure they are declared in.
- They can only be accessed by instructions in the Procedure.
- It is **good practice** to use Local Variables to store values. These variables can only be accessed from the instructions within the Procedure, this makes it easier to understand how the variable is being used and where it is being changed.
- Space is allocated for the Local Variables when the Procedure is called.
- When the call ends, the Local Variables for that call are destroyed.



2.1.4 Global Variable

Variables and Constants can be declared within a Program. Variables declared in this way are called Global Variables. It seems tempting to use Global Variables to share values between procedures, but this is a bad idea. Global Variables should be avoided, and for many programs are unnecessary. The issue with Global Variables is that their values can be changed from anywhere within the program's code. This can make it difficult to locate the source of errors when globals are used.

While Global Variables should be avoided, Constants should be declared globally. As these values can not change, the issues with Global Variables do not affect Constants.

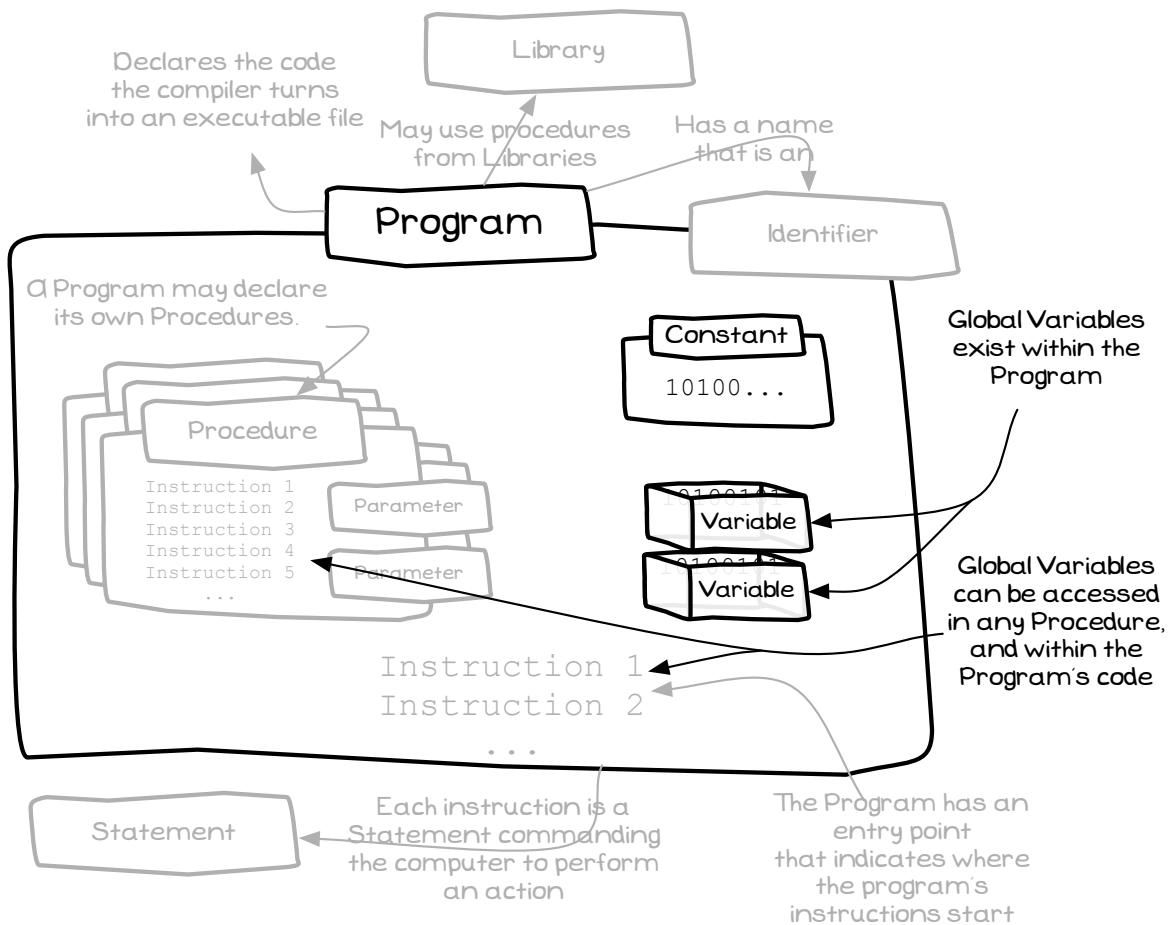


Figure 2.5: Variables declared within a Program are Global Variables

Note

- Global Variable is the **term** given to a Variable that is declared within the program.
- Variables that are declared within a Program are called **Global Variables**.
- Global Variables can be accessed by the program's instructions, and by the instructions in any of the Procedures.
- You should **avoid** using Global Variables. These variables can be accessed anywhere within the Program, making it difficult to locate errors.
- Using Global Variables introduces hidden dependencies between Procedures, breaking the isolated nature of the Procedures.
- Constants **should** be declared globally, and used to give meaning to values entered into your code.

2.1.5 Parameter

The instructions within a Procedure define the actions that occur when that procedure is called. In most cases these instructions need to be given values to work with. These values can be passed to the Procedure using Parameters. The Parameter is a Variable that has its value set in the procedure call.

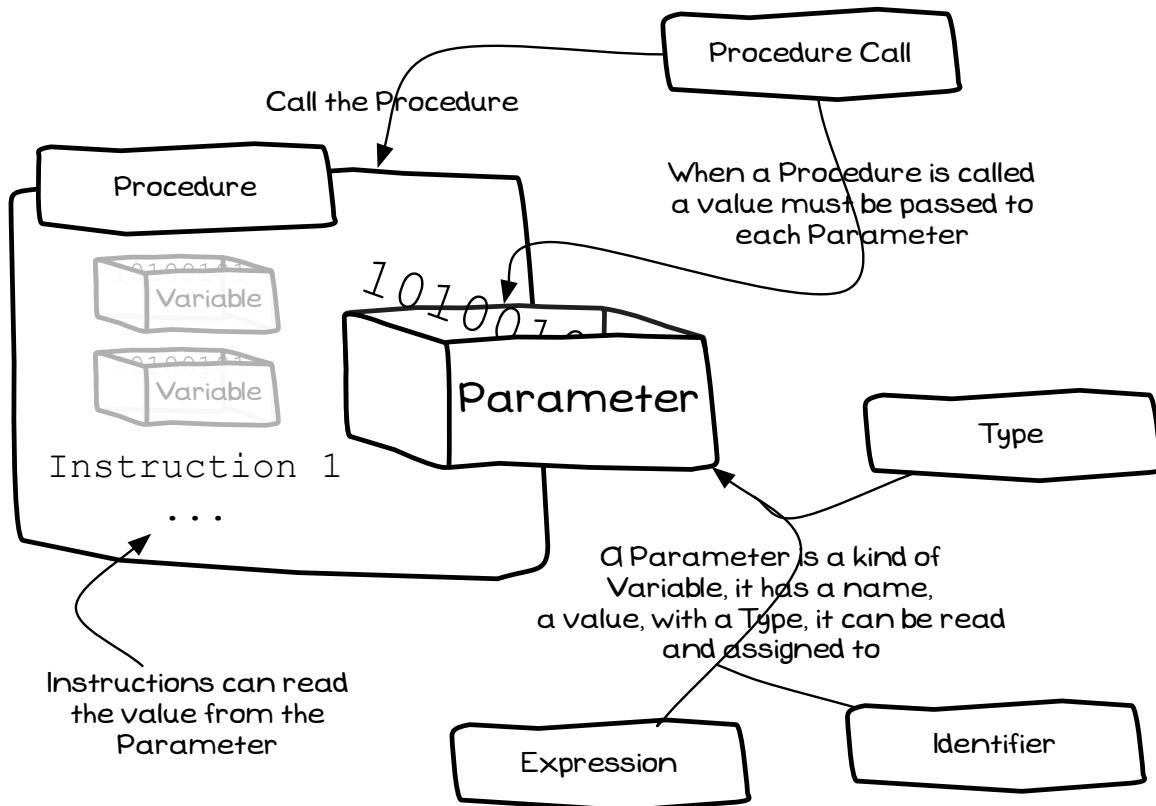


Figure 2.6: Parameters allow data to be passed to Procedures

Note

- Parameter is the **term** given to the Variables declared to accept values passed to Procedures.
- The **procedure call** assigns values to each of the Procedure's Parameters.
- Parameters allow you to pass values into a Procedure.
- Within the Procedure the Parameters can be used in the same way as any other Variable.
- It is **good practice** to use Parameters to pass values into a Procedure.



2.1.6 Pass by Value and Pass by Reference

There are actually two ways that values can be passed to Parameters. This relates back to the fact that Variables have two aspects: the Value within the Variable, and the Variable itself. These two means of passing parameters allow you to either pass a value, or pass a Variable.

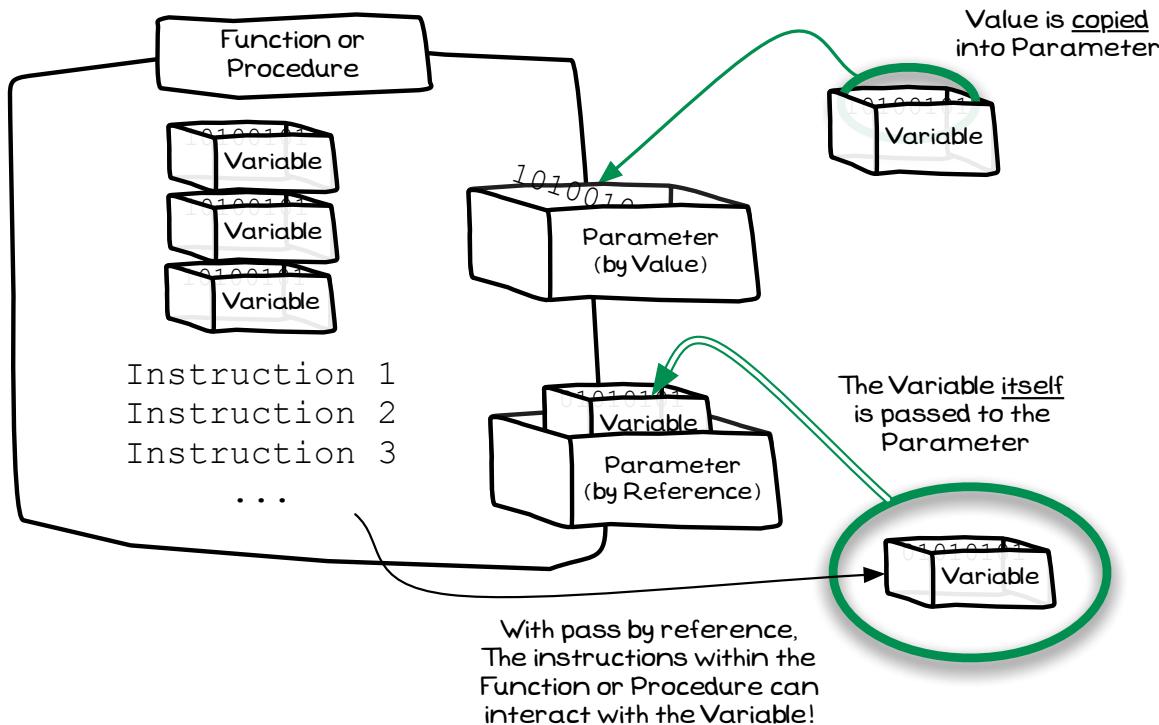


Figure 2.7: Parameters can accept data by reference or by value

Note

- Pass by Reference and Pass by Value are **terms** that explain how data is passed to a Parameter.
- Most parameters are passed by value.
- Pass by value copies the value to the parameter. This means pass by value can work with any Expression.
- Pass by reference allows you to pass the Variable itself to the parameter.
- The main use for pass by reference is to allow the Procedure or Function to store a value in the Variable passed to the parameter.
- It is called pass by reference due to the way it is implemented, with the parameter receiving a reference to the Variable. Section 2.2 will cover this in more detail, conceptually the Variable itself is passed to the Parameter.

2.1.7 Statement (with Assignment)

Statements are the actions that we can get the computer to perform. At this stage we have covered the statements that run procedures, the [Procedure Call](#), and the statement to assign values to variables, the [Assignment Statement](#).

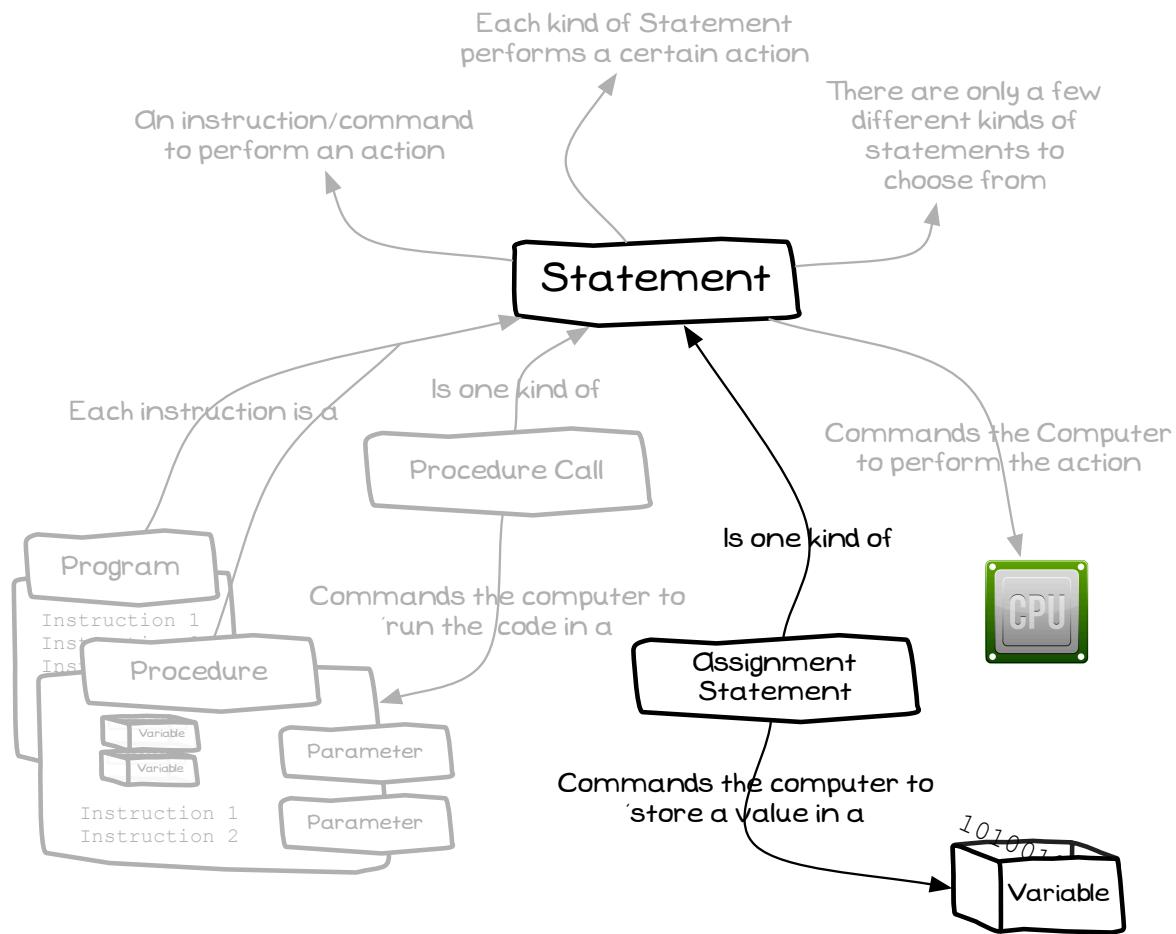


Figure 2.8: A Statement may be an Assignment statement

Note

- Statement is the **term** given to the instructions in our code.
- Statements can be either:
 - [Procedure Call](#) used to run the code in a Procedure, as covered in Chapter 1.
 - [Assignment Statement](#) used to calculate a value and store it in a Variable.
- All instructions in your code are Statements, these include the instructions in your Program as well as the instructions in your Procedures and Functions.

2.1.8 Assignment Statement

The Assignment Statement calculates a value, and stores it in a Variable. You use an assignment statement to store values in variables.

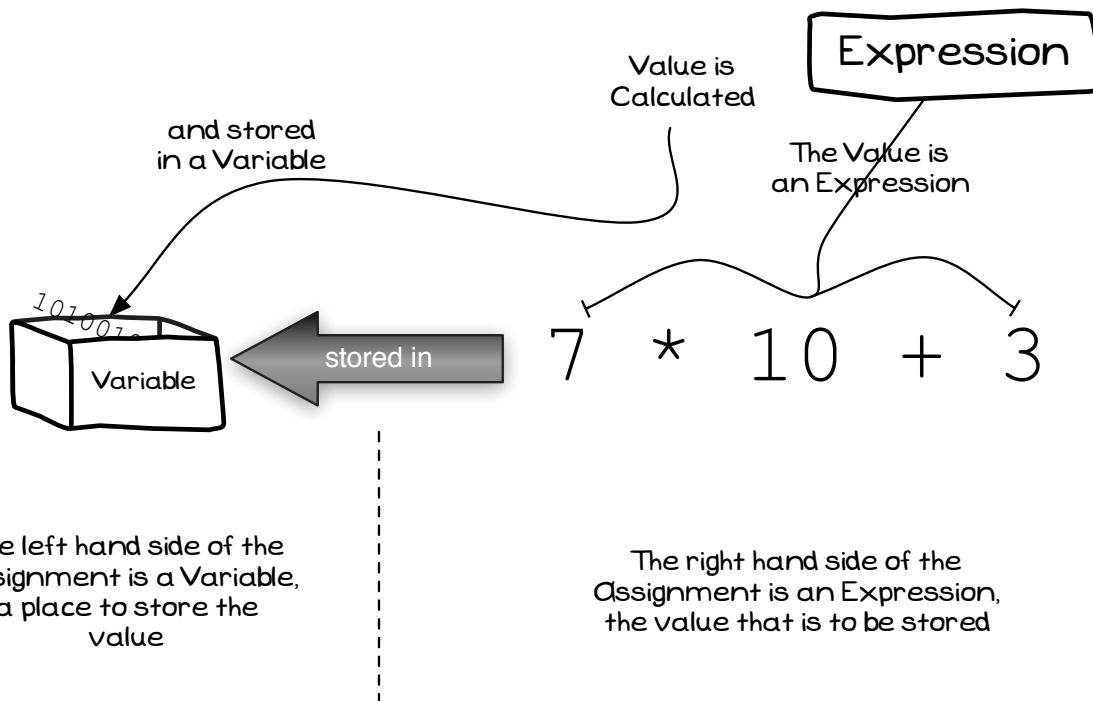


Figure 2.9: Assignment Statements assign values to Variables

Note

- An Assignment Statement is an **action** you can get the computer to perform.
- The *right hand side* of the Assignment is an **Expression** that calculates the value to be stored.
- The *left hand side* of the Assignment is a **Variable** into which the value is stored.
- When the Assignment Statement is executed the Expression is evaluated first, and then the resulting value is stored in the variable.
- It's important to remember that the Variable is a location at which to store a value. When the Variable appears on the left hand side of an assignment it is being used to store the resulting value. If the variable appears on the right hand side its value is being used as part of the expression.

2.1.9 Function

Functions are used to calculate values. In many ways a Function is just like a Procedure, it has a name, can be called, can accept parameters, can have local variables, and performs a number of instructions when it is called. Unlike a Procedure, however, Functions are used to calculate values. When the function you called ends it returns back to you with a value.

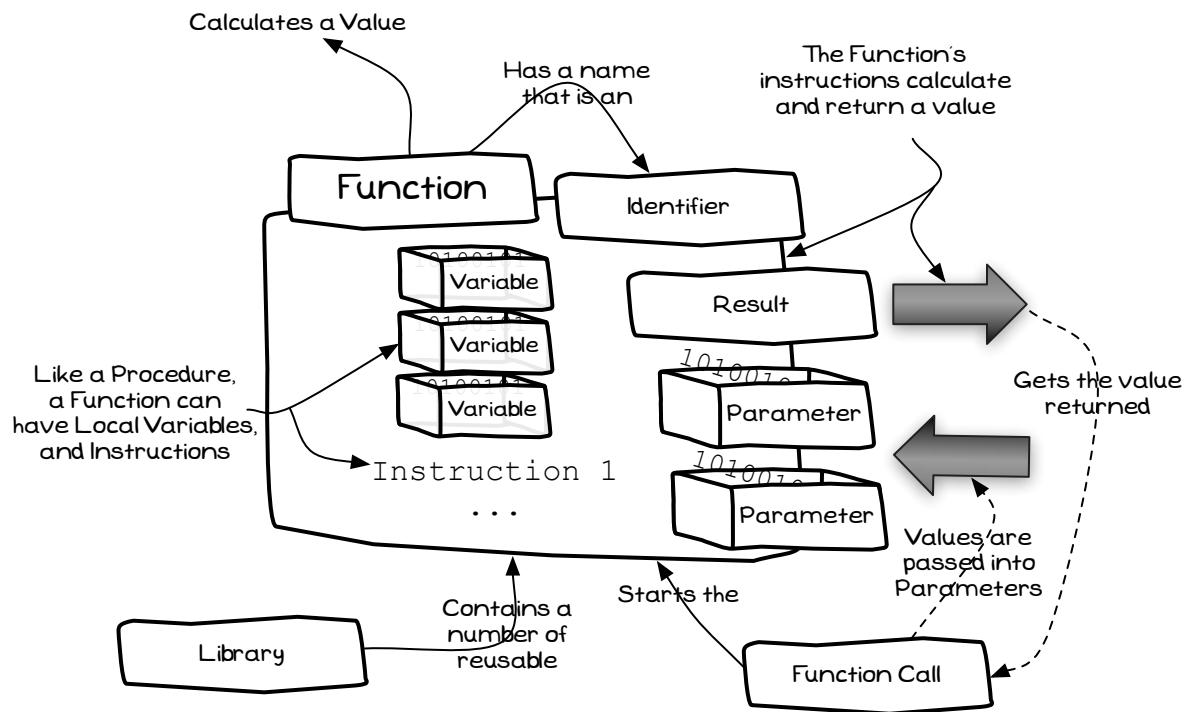


Figure 2.10: A Function is just like a Procedure, except it calculates and returns a Value

Note

- A Function is an **Artefact**. Something that you can create and use in your program's code.
- A Function is just like a Procedure in that it ...
 - Has a name that is used to call it.
 - Performs instructions when it is executed.
 - Can accept Parameters to allow the caller to pass in values.
 - Is allowed to create its own local variables.
- Unlike a Procedure, a Function...
 - Should **not** have any side effects.
 - Calculates and returns a value.
 - Is called as part of an Expression.
- You use Functions to calculate values.
- You use a **Function Call** to call a function as part of an Expression.



2.1.10 Function Call

A Function Call is used to execute a **Function**, and to read the value that is returned. This is similar to a **Procedure Call**, but unlike a procedure call it must be done as part of an Expression.

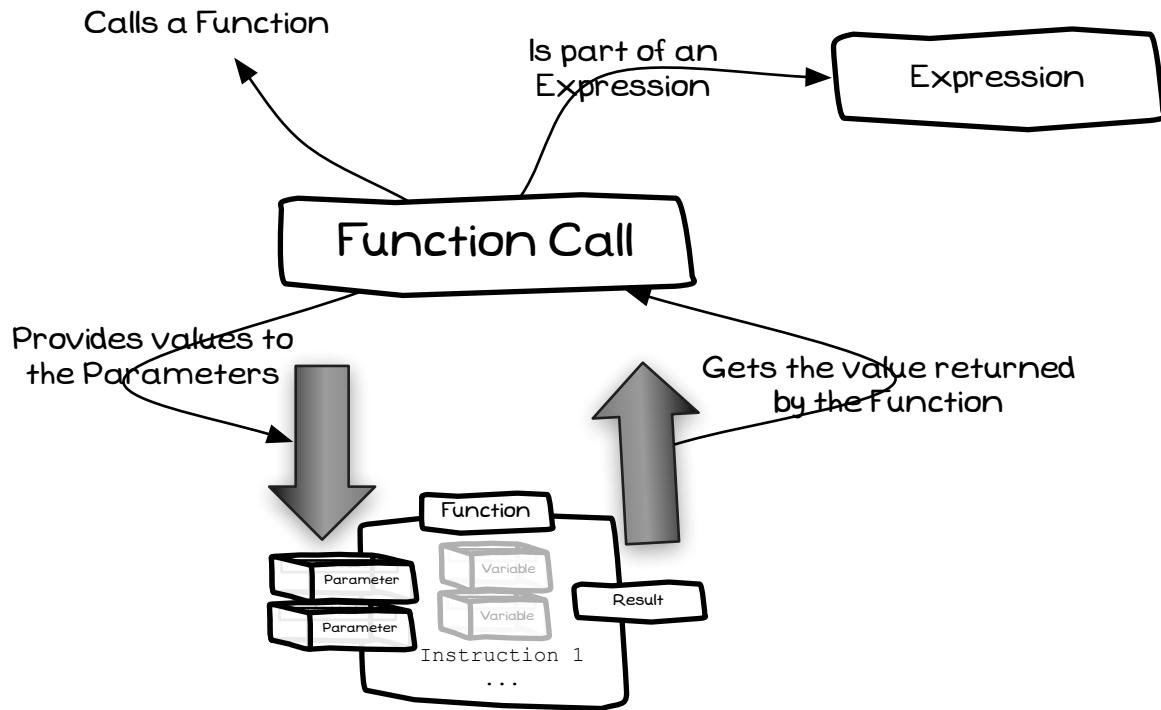


Figure 2.11: A Function Call is part of an Expression where the value is calculated

Note

- A Function Call is an **action**, but one that is performed as part of an Expression.
- Function calls can appear in *any* expression. For example, you can use a Function Call to calculate the value in an **Assignment Statement**. You can use a Function Call to calculate the argument values for a procedure call.

2.1.11 Expressions (with Function Calls, Variables, and Constants)

You can **read** the values from Variables and Constants within Expressions. The value of the expression is calculated by **reading** the values from the Variables and Constants when the expression is calculated¹.

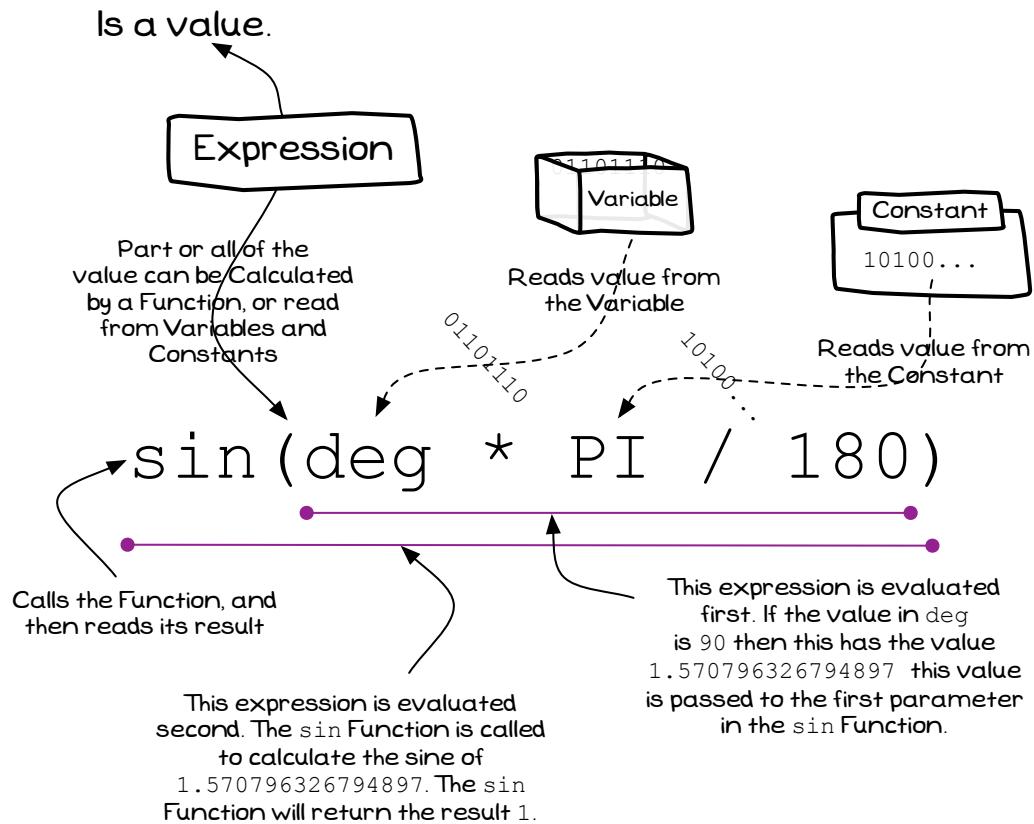


Figure 2.12: Expressions can read values from Function Calls, Variables, and Constants

Note

- Expression is the **term** given to the code that calculates values within your Statements.
- You can read values from Function Calls, Variables, and Constants.
- You use the Variable or Constant's name to access its value within an Expression.
- The **Function Call** runs the code in the Function, and then reads the result returned.
- There are actually **two expressions** in Figure 2.12:
 1. The first Expression is the value passed to the sin function ($\text{deg} \times \text{PI} \times 180$). This value is calculated by reading the values from the deg variable and the PI constant. These values are then used in the Expression to determine that value that is passed to the Parameter in sin.
 2. The second Expression is the result returned from the call to the sin function. This will calculate the sine of the value calculated in the first expression.
- The Expression reads the value of the Variable **at the time** it is executed.
- Expressions are used to calculate values that are...
 - Passed to Parameters within **Procedure Calls**.
 - Assigned to Variables within **Assignment Statements**.

¹This means that the value will be affected by the statements that occurred before the expression was calculated.

2.1.12 Program (with functions)

You can declare your own Functions within the program's code.

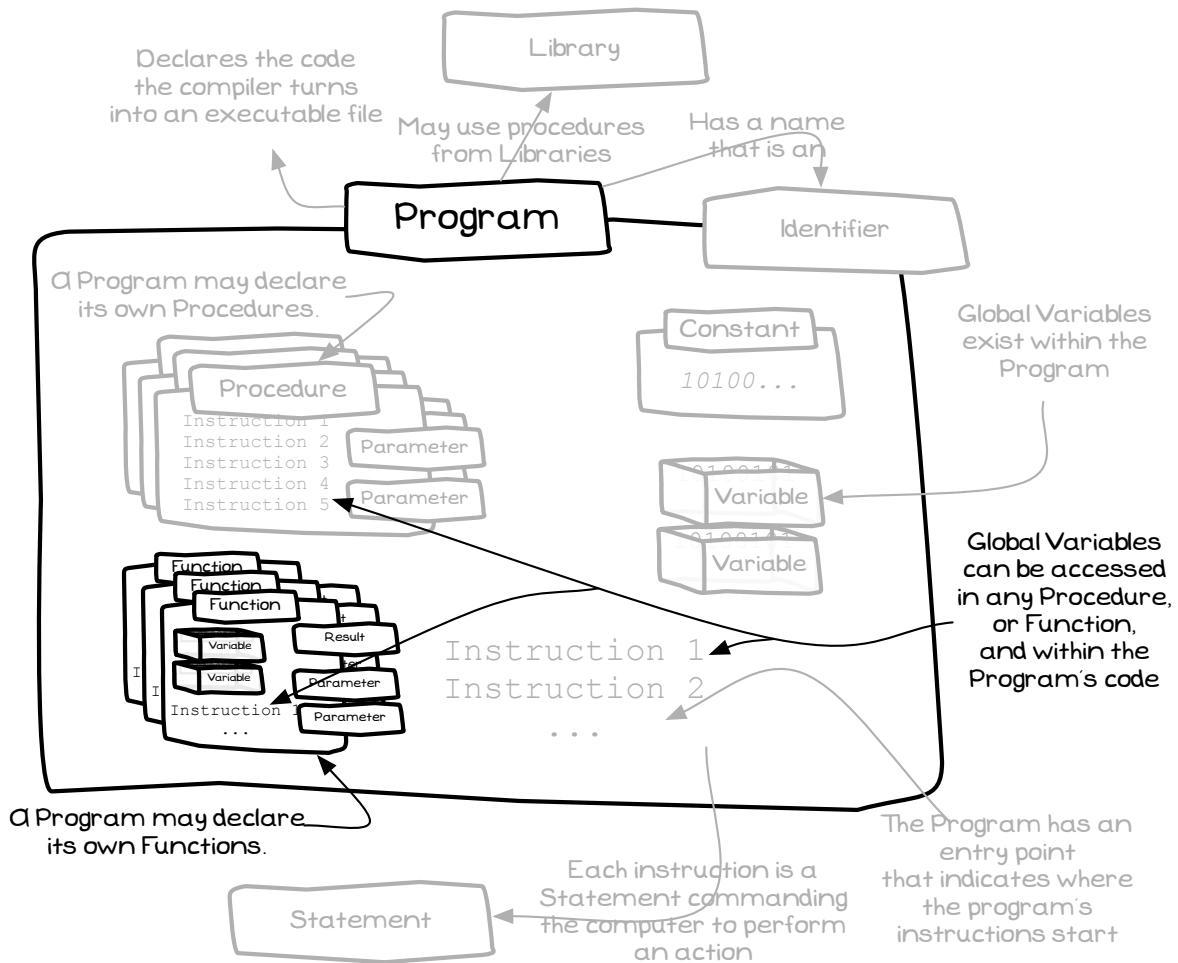


Figure 2.13: You can declare your own Functions in your program's code

Note

- A Program is an **Artefact**, you create Programs that the user can execute. Internally these programs contain other artefacts such as Procedures, Functions, and Variables.
- You can declare your own **Functions** within your program's code.
- With C and Pascal the Function must be declared before it is used.

2.1.13 Summary

This section has introduced a number of programming artefacts, some programming terminology, and one kind of instruction. An overview of these concepts is shown in Figure 2.14. The next section will look at how you can use these concepts to design some small programs.

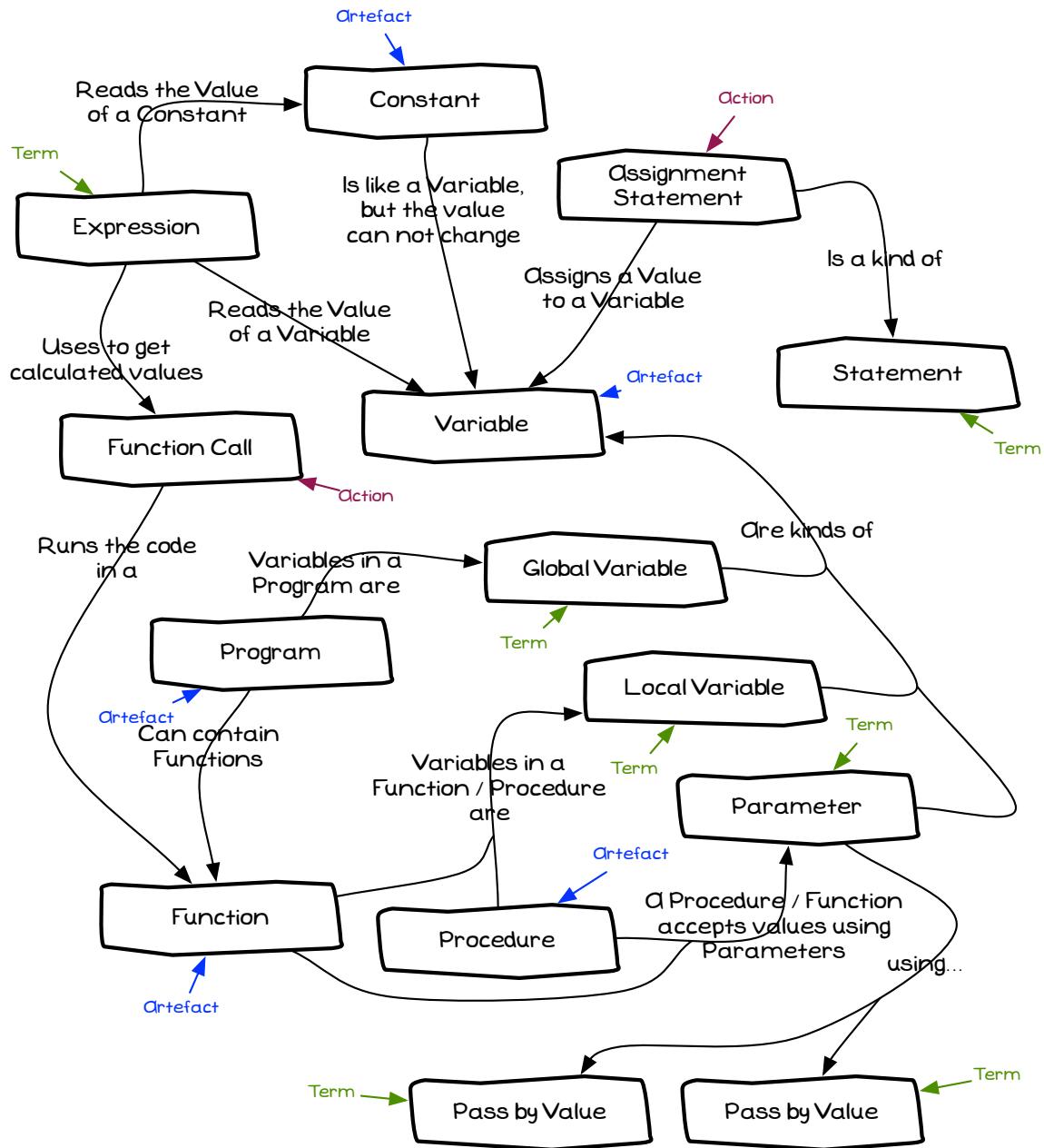


Figure 2.14: Key Concepts introduced in this Chapter

Note

- **Artefacts** are things you can *create* and *use*.
- **Terms** are things you need to *understand*.
- **Actions** are things you can *command* the computer to perform.

2.2 Using these Concepts

Variables, Constants, and Functions give us the ability to work with data in new ways in our programs. Previously the data within the program was limited to values entered directly into the code. Now with these concepts we can interact more meaningfully with data, performing calculations, and storing and manipulating values.

2.2.1 Designing Change

Table 2.1 contains a description of the next program we are going to design. This program will calculate and output the ideal change for a given transaction from a Vending Machine. In designing this program we will make use of the concepts introduced in this chapter; we will use a Function to calculate the coins to give, Variables to store values such as the amount paid, Constants for the values of the different coins, and Parameters to pass values to the Function.

Program Description	
Name	Change Calculator
Description	Calculates the idea change for a given transaction in a Vending Machine. The transaction involves reading the cost of the item purchased and the amount paid, and then outputting the number of each type of coin to give as change.

Table 2.1: Description of the Change Calculator program.

To design and implement this program we need to follow a number of steps:

1. Understand the problem, and get some ideas on the tasks that need to be performed.
2. Choose the artefacts we will create and use
3. Map these artefacts to code
4. Compile and run the program

In step 1 you need to understand what it is that you want the program to do. This will involve determining the tasks to be performed, the steps involved in those tasks, and any data associated with them. Once you have a good understanding of what you want to achieve you can start to build a solution. In this step you determine the artefacts you want to create, and try to locate those you could use from the available libraries. Step 3 then turns your plans into source code that can be compiled and run in Step 4.

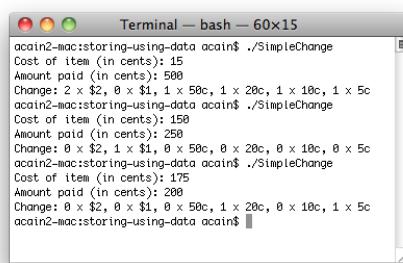


Figure 2.15: The Change Calculator running in the Terminal, from Figure 2.1

2.2.2 Understanding the Change Calculator

Receiving change from a transaction is something that you should be familiar with, and determining the ideal change is not an overly complex task. While the task itself is common, it does not mean that you can skip thinking about it. If you had to give \$6.50 in change you know without thinking that you should give three \$2 coins, and one 50c coin. What you need to do now is think through the steps that you perform intuitively.

Think about all the steps for how you actually calculated the change to be given. The first *secret* is to realise that you need to start with the coin with the largest value, and then work down from there to the coin with the smallest value. The number of coins you give each time can then be calculated by dividing the amount of change to be given by the value of the current coin. Lastly you need to reduce the amount of change that remains to be given. These steps are shown in the pseudocode in Listing 2.1.

Pseudocode

```

Program: Change Calculator
-----
Constants:
- TWO_DOLLARS with value 200
- ONE_DOLLAR with value 100
- FIFTY_CENTS with value 50
- TWENTY_CENTS with value 20
- TEN_CENTS with value 10
- FIVE_CENTS with value 5
-----
Functions:
- Get Change Value(), returns change value in cents
- Coins To Give ( Change, Coin Value ), returns number of coins to give
-----
Procedures:
- Give Change( Change Value, Coin Value, Coin Description )
-----
Entry Point (Main):
-----
Variables:
- Change Value (Integer)
Steps:
1: Assign to Change Value, the result of calling Get Change Value()
2: Output 'Change: ' to the Terminal
3: Call Give Change ( Change Value, TWO_DOLLARS, '$2' )
4: Call Give Change ( Change Value, ONE_DOLLAR, '$1' )
5: Call Give Change ( Change Value, FIFTY_CENTS, '50c' )
6: Call Give Change ( Change Value, TWENTY_CENTS, '20c' )
7: Call Give Change ( Change Value, TEN_CENTS, '10c' )
8: Call Give Change ( Change Value, FIVE_CENTS, '5c' )
9: Output a new line to the Terminal

```

Listing 2.1: Pseudocode for Change Calculator program.



2.2.3 Choosing Artefacts for the Change Calculator

The design for any program should involve thinking about the data as well as the tasks involved. You can start by dividing the program's tasks into functions and procedure, and then look at the data that will be needed by these in order to achieve their tasks.

Designing the tasks for the Change Calculator

Think about the process of calculating change. At the start you need to determine the amount of change that needs to be given. Next you need to determine how many of each kind of coin you are going to give. These steps can be coded into their own functions and procedures. These functions and procedures are outlined in the following list, and shown in the structure chart in Figure 2.16.

- Get Change Value: This **Function** will be responsible for asking the user to enter the cost of the item, and the amount paid. It will then calculate the amount of change that needs to be given, in cents.
- Give Change: This **Procedure** will be responsible for the calculations related to giving one kind of coin as change. This will involve getting the number of coins to give, updating the amount of change remaining, and outputting the details (for that coin) to the Terminal.
- Coins to Give: This **Function** will be responsible for calculating the number of coins to give as change, given an amount of change and the value of the coin.

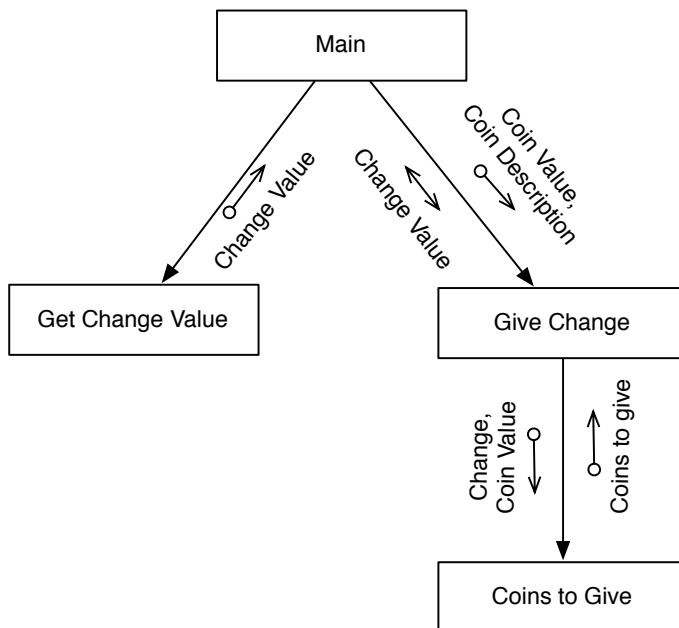


Figure 2.16: Structure Chart for the Change Calculator

To understand how this is going to work you now need to think about how the kinds if data these different tasks are going to need.

Designing the data for the Change Calculator

Designing data is much like designing the procedures in your program. You need to think about the solution and try to identify the different values that are being used. Each of these values can then become either a Variable, a Parameter, or a Constant. Looking over the steps for calculating change you should be able to identify several different values you will need to work with. You need to be able to store things like the cost of the item being purchased, the amount of money paid, and the amount of change you need to give. A good way to approach this

is to think about the values the program will output, as well as the values the user will need to input and any intermediate values you will need to perform the required calculations.

You can use the input/output/processing ideas to help you think about the data that will be required for each function or procedure. Let us examine each of the functions and procedures in turn.

Get Change Value The first task we can examine is the Get Change Value [Function](#). This task will be responsible for determining how much change needs to be given to the user. To design the data needed for this task you can think about its inputs, outputs, and temporary values.

When you are designing a function or procedure it can be given input by the calling code in the [Function Call](#) or [Procedure Call](#). Inputs provided via the function or procedure call are coded using [Parameters](#). To determine the parameters you need, think about the information that this function or procedure will need to be given to fulfil its responsibilities. In the case of the Get Change Value, it does not require any input from the caller.

- Get Change Value does not require any input from the calling code, so there is no need for any parameters with this function.

The Get Change Value task will be coded as a function because it has some output. The result returned by a function is an output, it is what makes it different from a procedure. When you call a function it runs some steps and then returns a value, the value returned is the output from the function. Get Change Value needs to return back the amount of change, this is why it is a function in this design.

- Get Change Value returns a number. The number returned is the value of the change to be given in cents.

Within the Get Change Value function there will need to be some values that it uses to calculate its output. Get Change Value will need to read the cost of the item from the user, and the amount paid. These two values need to be stored somewhere, so the design of this function will require two [Local Variables](#): they can be called Cost of Item and Payment. These will exist entirely within the function, being the values the function requires to calculate its output.

- Get Change Value will have two local variables: Cost of Item and Payment.

The pseudocode for this function is shown in Listing 2.2.

Pseudocode

```
Function: Get Change Value
-----
Returns: An Integer - the value of the change to give in cents
Variables:
- Cost of Item (Integer)
- Payment (Integer)
Steps:
1: Output 'Cost of item (in cents): '
2: Read a number into Cost of Item
3: Output 'Amount paid (in cents): '
4: Read a number into Payment
5: Return the result, Payment - Cost of Item
```

Listing 2.2: Pseudocode for the Get Change Value function



Give Change The next task to consider is the Give Change [Procedure](#). This procedure will be responsible for coordinating the actions for giving a certain coin in change. Once again you

need to think about the inputs it requires (parameters), its outputs, and any values it will work with internally.

Put yourself in the place of the computer.² You (as the computer) have been *asked* to Give Change. What data will you need to complete this request? At a minimum you will need to know the total change value that is being given, and the value of the coin that you are issuing.³

There is one more input that you can only find by thinking about the tasks the computer needs to perform in this code. A part of giving the change will be to output a message, something like ‘3 x 20c’ or ‘1 x \$2’. The number of coins can be calculated within the procedure, but the text is another issue. The computer will not know what text to output. This data must, therefore, come as input into the procedure.

- Give Change needs to be *given* the Change Value variable, and it needs to be told value of the coin and the coin’s description. These will be coded as parameters, with Change Value being passed by reference.

Give Change will output text to the Terminal, but it also needs to update the Change Value variable it is given. This can be considered as output from the procedure. As the Change Value variable is passed by reference, you can picture this as receiving the variable from the caller. Any changes this procedure makes on its Change Value parameter will actually be changing the value in the variable passed to the Give Change procedure. In this way the procedure is able to output a value.⁴

- Give Change will output the updated Change Value, as this is passed by reference.

Within Give Change you will need to store the number of coins to give in change. This will become a local variable that can be used to calculate the updated Change Value and to output the details to the Terminal.

- Give Change will have one local variable: *to Give* used to store the number of this coin to be given in the change.

The pseudocode for this procedure is shown in Listing 2.3.

Pseudocode

```
Procedure: Give Change
-----
Parameters:
1: Change Value (Integer - by ref)
2: Coin Value (Integer)
2: Coin Description (String)
Variables:
- To Give (Integer)
Steps:
1: To Give = Coins to Give ( Change Value, Coin Value )
2: Change Value = Change Value - To Give * Coin Value
3: Output Number Given, ' x ', and the Coin Description
```

Listing 2.3: Pseudocode for the Give Change function



Coins to Give The Coins to Give function is used to calculate the number of coins to give. This code could have been written within the Give Change procedure, but it was decided to code this in its own function.

²Remember the computer is unintelligent. You cannot rely upon your knowledge. Try to think about the information you are using and the steps you are performing to make sure you can capture what needs to be done in your code.

³In this design the Give Change procedure will be called once for each coin.

⁴An alternative implementation could have been to code this as a **Function**, with the new change value being returned. It would then be the responsibility of the caller to assign this into their change variable.

Coins to Give will need to be told the value of the change, and the value of the coin. Both of these parameters will be passed by value, as it is only the values that are required. Internally Coins to Give will not require any additional data, as it can calculate its output from the two input values.

- Coins to Give needs to be *told* the value of the change and the value of the coin, it can then use these values to calculate the number of these coins that need to be given in the change.
- Coins to Give will output the number of the indicated coin that needs to be given in the change.

The pseudocode for this procedure is shown in Listing 2.4.

Pseudocode

```
Function: Coins to Give
-----
Returns: An Integer - the number of coins to give in change
Parameters:
 1: Change (Integer)
 2: Coin Value (Integer)
Steps:
 1: Return the result of Change divided by Coin Value
```

Listing 2.4: Pseudocode for the Coins to Give function



Entry Point (Main) The program's code always performs the same task. It coordinates the actions the program is performing. The pseudocode for this was shown in Listing 2.1. This code will need to store the value of the change. It will get the value to store in this by calling Get Change Value. It will call Give Change for each of the coin values, and pass this variable to the procedure for it to update. These actions are shown in Figure 2.17.

- The Entry Point (Main) needs a local variable to store the Change Value.

Thinking further about the program there are also the constant values of the different coins. These values are almost taken for granted when you think about giving change yourself, but remember the computer is unintelligent so you need to specify *everything* for it. The values of the coins can be coded using a constant for each coin. Using constants is a better option than hard coding these values directly in the program, as the name helps provide a context for the value when it is used.

Note

- Notice that the functions and procedures are isolated from each other. They have defined inputs and outputs, but their local variables are hidden within their code.
- This means you can focus on the function/procedure you are working on, and do not have to think much about the details of the other functions/procedures in your code.
- Parameters allow you to pass data into a function/procedure.
- Passing a parameter by reference allows you to output a value by updating the variable you are passed. This means a procedure can output values by having parameters passed by reference, and functions can also use this mechanism to update values in variables as well.



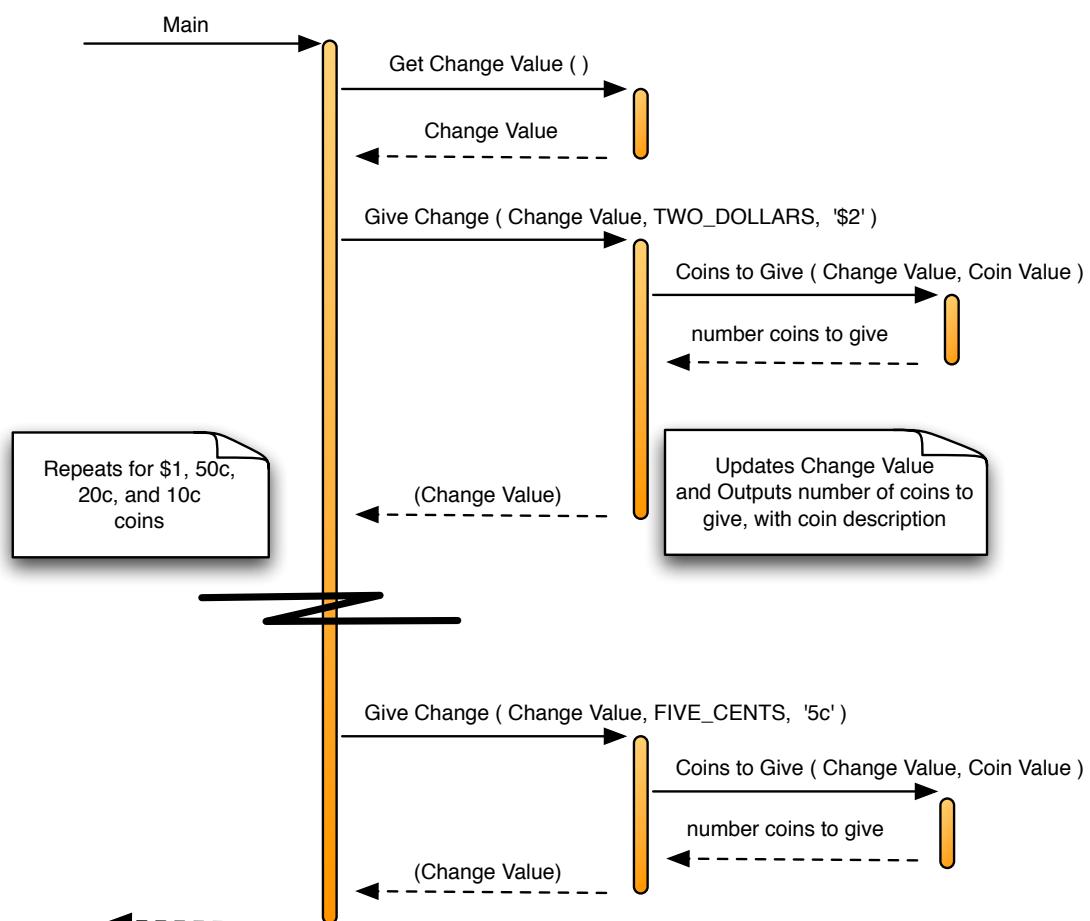


Figure 2.17: Sequence Diagram for the Change Calculator

Change Calculator Design Overview

The Change Calculator program contains the following artefacts:

- **Constants:**

- **TWO_DOLLARS** with value 200, represents the value of a \$2 coin
- **ONE_DOLLAR** with value 100, represents the value of a \$1 coin
- **FIFTY_CENTS** with value 50, represents the value of a 50c coin
- **TWENTY_CENTS** with value 20, represents the value of a 20c coin
- **TEN_CENTS** with value 10, represents the value of a 10c coin
- **FIVE_CENTS** with value 5, represents the value of a 5c coin

- **Functions:**

- **Get Change Value:** Calculates and returns the amount of change that needs to be given by the program. This function has the following:

- * **Local Variables:**

- **Cost of Item:** Stores the value of the item being purchased. This design uses cents as its base to make the calculations easier, avoiding rounding issues involved in using floating point values.
 - **Payment:** This is used to store the value of the payment being made. Once again this will be in cents.
 - **Coins to Give:** Calculates the number of coins to give, returning how many of these coins should be dispensed as part of giving this change. This has the following:

- * **Parameters:**

- **Change:** The value of the change that is to be given.
 - **Coin Value:** The value of the coin that is being dispensed.

- **Procedures:**

- **Give Change:** Calculates the change, and outputs the coin details to the Terminal. This shows the number and description of the change given. This requires the following:

- * **Parameters:**

- **Change Value** (by reference): The variable that is storing the amount of change to be given.
 - **Coin Value:** The value of the coin to issue.
 - **Coin Description:** The description of the coin.

- * **Local Variables:**

- **To Give:** Stores the number of the coin to give in change, used to update the Change Value and to output the message.
 - **Entry Point (Main):** this coordinates the overall activity of the program. Getting the amount of change using Get Change Value, and using Give Change to issue the change for each coin.

- * **Variables:**

- **Change Value:** Keeps track of the current amount of change that has to be given to the user.

In addition to these artefacts the program will make use of some procedures from the available libraries. This will include the following:

- **Output:** You need to use your language's procedures to write output to the Terminal.
- **Input:** Languages also provide a means of reading values from the user. In C and Pascal these input procedures need to be passed the variables that you want the value assigned to. This uses pass by reference to enable the input procedure to store the values read into a variable for you.

C++

In C the input procedure to read values from the Terminal is called `scanf()`, see [C Terminal Input](#).

**Pascal**

In Pascal the input procedure to read values from the Terminal is called `ReadLn()`, see [Pascal Terminal Input](#).



Reviewing the design diagrams

Figure 2.16 shows the Structure Chart for the Change Calculator. This shows the structure of the solution, visually showing the functions and procedures and the calls between them. This diagram has been enhanced to show the parameter values being passed into the functions and procedures, and the values being returned. These data flows are shown alongside the call arrow, with their own smaller arrow to indicate the direction of the flow.

By reading Figure 2.16 you can tell that `Coins to Give` is going to be a function as it is returning data to `Main`. `Give Change` could be a function or a procedure as it is accepting and returning the `Change Value`, in this design it is a procedure, and updates the value in the `Change Value` variable using pass by reference. You can also see that `Coins to Give` and `Give Change` both require parameters to accept the data being passed into them.

The Structure Chart shows the static structure of the code, indicating the calls between the functions and procedures, but not communicating when these are called, or how many times. This dynamic information is captured in the Sequence Diagram shown in Figure 2.17. This diagram shows the sequence in which the function and procedure calls are performed. Notice that this diagram has also been enhanced to show the data that flows into and out of the functions and procedures.

The Sequence Diagram indicates how values are being returned. The return of a function's value is shown by indicating the value on the returning arrow. Notice the indication of this value on the arrow returning from the call to `Get Change Value`, this indicates that `Get Change Value` is a function. On the other hand, the brackets around the return value from `Give Change` indicates that this is being performed by updating a parameter that was passed using pass by reference. You can use this information to determine how the design intends these to be coded. As `Get Change Value` is directly returning a value it must be a function, `Give Change` updates a parameter and does not return a value itself so it must be a procedure.

Note

- A structure chart shows the static structure of the program. This tells you which procedure/functions call which, and the data that is passed between them. It shows you what is written in the code.
- A sequence diagram shows the dynamic operation of a program. This tells you how the functions and procedures interact to achieve some goal. It shows you what is happening when the code is run.



Designing data in general

When you are designing a program you need to think about each function or procedure, and determine if it requires data to be given to it to enable it to perform its action. Any data it requires must then be passed to it using Parameters. The key to understanding parameters is to remember that each function/procedure is its own isolated domain. It can have its own data, using local variables, and has its own instructions. In most cases these instructions will need to be given some starting data, some information upon which to act.

One strategy you can use to picture this is to put yourself in the place of the function, or procedure. For example, what would you need to be told in order to determine the number of coins to give in change for a single coin type. You can not work out the answer without being told the value of the change to be given, and the value of the coin. With these two pieces of information you now have enough to calculate the required output. For example, how many \$2 coins should be given for \$6.50 in change. These two pieces of data can be used to calculate the answer 3. This is exactly what is being coded in the Coins to Give function.

In this respect Procedures are the same. In order to give change, the Give Change procedure needs some information. It can not act without this information. If you needed to *give change*, you need to be told something about the amount of change you are giving and the value of the coin you are issuing. In this case, Give Change needs to be told the change value, the coin value, and the description of that coin. With these details the procedure can then code the steps needed to produce its desired side effects.

Parameters are mostly used to pass a value into a function or procedure. However, in some cases it is useful to pass in the *variable* rather than the *value*. This is the case when you want the function or procedure to *change* the value in the variable passed to it. Take the change value parameter in Give Change as an example. When this procedure issues some coins, the amount of change still to be given must be updated. For example, when you issue '3 x \$2' for the \$6.50 in change, the new change value will be 50c ($650 - (3 \times 200)$). By passing the *variable* into Give Change it is possible for this code to change the value in the callers variable, ensuring that the correct change is given.

Modelling and Abstraction

One of the main keys to understanding programs, and to creating good designs, is to see how each artefact you create models something from the problem. The procedures that you create model the processes, performing actions that relate to the task at hand. The functions model calculations that need to be performed, such as determining how many coins should be given. In the same way the variables that you create will store values related to the problem. Each variable **represents** something, some information related to your program. The *thing* it represents should be reflected in the name given to the variable.

The act of trying to model the problem in software is called **abstraction**. It is the process of determining the features of the problem that are essential to the solution, and capturing these in a way that models reality but keeps only the details we care about. The ability to create your own model is a skill that you learn with experience, taking a lot of practice to really master.

Note

- A good design will closely model the program's domain. You should be able to relate the artefacts you are creating with things related to the program.

2.2.4 Writing the Code for the Change Calculator

The pseudocode from Listing 2.1 shows the instructions, and how these should be divided between the Coins to Give Function and Output Change Data Procedure. At this stage these instructions need to be translated into source code, so that they can be compiled and the resulting program tested.

The following two sections, Section 2.3 Storing and Using Data in C and Section 2.4 Storing and Using Data in Pascal, contain a description of the syntax needed to create programs in the C and Pascal programming languages that include Variable and Function declarations.

Note

Remember the basic process for reading the Syntax Diagrams is to:

1. Find the page with the Syntax rule you are interested in knowing about.
2. Have a quick look at the Syntax Diagram and the rules it contains. Read each rule, and get a basic feel for how it is going to come together for your program.
3. Read the example to see one way of using the Rule. The Syntax Diagram can be used to create any number of variations of the rule, the example gives you at least one way these rules can be coded.
4. Return to the diagram and make sure you can match each part of the example back to the rule that created it.
5. Look up any related rules that are not explained on this rule's page.



2.2.5 Compiling and Running the Change Calculator

Once you have completed your program, you need to compile and test it.

1. Open the **Terminal**⁵ program for your Operating System
2. Use the **cd** command to move to the directory with your code, for example
`cd /Users/acain/Documents/Code`
3. Run the compiler with your program's code. See the language specific details below.
4. Fix any compiler errors, using the tips from Section [1.2.5 Compiler Errors](#).
5. Execute the program using `./SimpleChange` and check the results

C++

The C compiler is called **gcc**. To compile your *Change Calculator* program you will need to run the following:

```
gcc -o SimpleChange simple-change.c
```



C++

C does not have support for pass by reference, though it can be achieved using other means (see Chapter [6](#)). The C++ language is an extension to C that fixed a number of issues, as well as added some new features. One of the new features was built in pass by reference. To compile the C++ version of the Change Calculator you need to use the C++ compiler called **g++**. To compile your *Change Calculator* program you will need to run the following:

```
g++ -o SimpleChange simple-change.cpp
```



Pascal

The Pascal compiler is called **fpc**. To compile your *Change Calculator* program you will need to run the following:

```
fpc -S2 SimpleChange.pas
```



⁵The **MinGW Shell** on Windows.

Generating Test Data

Now that our program is making greater use of data it becomes more important to think about how to test your program. Testing is the process of trying to locate issues with your code. There are two main classifications for issues, **syntactic errors** and **semantic errors**.

Syntactic errors indicate places in your code where you have not correctly followed the syntax of the language. These are the easiest kind of error to find as the compiler will not be able to compile the program if its syntax is not correct. The errors that the compiler report are all syntax errors. As you gain experience with a Language you will find that you make fewer and fewer of these kinds of errors.

Semantic errors, on the other hand, will not be found by the compiler. These are errors in the logic within the program. They are cases where you have correctly structured the code, but the code itself does not get the computer to perform the actions you require. These are the kinds of errors that you need to learn to be able to detect yourself. Selecting the right kind of test data is an important task when you start to think about testing programs.

For the Change Calculator there are two inputs provided by the user. In order to test this program you need to determine the values that are passed to the Cost of the Item and the Payment inputs. You want to choose values that can help you uncover any unexpected issues with the programs code.

Cost of Item	Payment	Expected Output	Reason
\$2.50	\$5.00	1 x \$2, 1 x 50c	Basic test to check that the program works for a simple data input.
\$0.15	\$4.00	1 of each coin	Generates \$3.85 in change, checking that each coin is used.
\$0.05	\$0.10	1 x 5c	Check the output can be a single coin. Could also add tests to check other individual coins.
\$0.60	\$1.00	2 x 20c	Check that 2 coins can be given.
\$0.00	\$5.00	2 x \$2, 1 x \$1	Can it accept no cost of item, and give back payment?
\$3.85	\$0.00	-1 of each coin	Check what happens if insufficient funds are provided.

Table 2.2: Test Data for the Change Calculator

The data we use to test an execution of a program is called a **Test Case**. Each test case provides a set of input values, and the expected results given this input. Example test cases for the Change Calculator are shown in Table 2.2. This table shows the input values, the expected output values, and some rational for why this test case should be run. To perform the testing you run the program once for each test case and check the output of the program against the expected output. If there are any differences you know that their is a problem, and need to check the program's code to find the source of the logic errors.

2.3 Storing and Using Data in C

2.3.1 Implementing Change Calculator in C

Section 2.2 of this chapter introduced the ‘Change Calculator’ program, and its design. Its implementation requires the definition of functions as well as procedures. These functions and procedures accepted parameters and use local variables.

This section of the chapter introduces the C syntax rules for implementing these concepts using the C language. The C implementation of the Change Calculator is shown in Listing 2.5. To get support for pass by reference you need to use C++, an extended version of the C language. This means that the C version is an alternate design, with Give Change becoming a function so that it can return the new change value. The C++ implementation is shown in Section 2.3.2.

```
/*
* Program: simple-change.c
* Calculate the ideal change for a given transaction.
*/
#include "splashkit.h"

#define TWO_DOLLARS 200
#define ONE_DOLLAR 100
#define FIFTY_CENTS 50
#define TWENTY_CENTS 20
#define TEN_CENTS 10
#define FIVE_CENTS 5

int coins_to_give(int change, int coin_value)
{
    return change / coin_value;
}

int give_change(int change_value, int coin_value, string coin_desc)
{
    int to_give;

    to_give = coins_to_give(change_value, coin_value);
    write(to_string(to_give) + " x " + coin_desc + ", ");

    return change_value - to_give * coin_value;;
}

int get_change_value()
{
    string line;
    int cost_of_item;
    int payment;

    write("Cost of item (in cents): ");
    line = read_line();
    cost_of_item = convert_to_int(line);

    printf("Amount paid (in cents): ");
    line = read_line();
    payment = convert_to_int(line);

    return payment - cost_of_item;
}
```

```
int main()
{
    int change_value;
    change_value = get_change_value();

    write("Change: ");
    change_value = give_change(change_value, TWO_DOLLARS, "$2");
    change_value = give_change(change_value, ONE_DOLLAR, "$1");
    change_value = give_change(change_value, FIFTY_CENTS, "50c");
    change_value = give_change(change_value, TWENTY_CENTS, "20c");
    change_value = give_change(change_value, TEN_CENTS, "10c");
    change_value = give_change(change_value, FIVE_CENTS, "5c");
    write_line();

    return 0;
}
```

Listing 2.5: C code for the Change Calculator

Note

- Save the C code in a file named `simple-change.c`.
- Compile this using `gcc -o SimpleChange simple-change.c`.
- Run the resulting program using `./SimpleChange`.
- Perform each of the Test Cases from Table 2.2 and check that the output matches the expected values.
- Look over the code and examine how the Variables, Parameters, Constants, and Function are being used.
- The c-string parameter should be coded as `const char *coin_desc` to allow you to pass literal values to it.
- Notice how the indentation makes it easy to see where each Function and Procedure starts and ends. Always lay your code out so that it is easy to see its structure.
- See how the Function and Procedure are declared before they are used. This is important as the C compiler reads the code from the start, and must know about the artefacts before you use them.

2.3.2 Implementing Change Calculator using C++

C++ contains a number of extensions that can make programming in C easier. One of these extensions is the ability to use pass by reference. The code in Listing 2.6 shows the C++ implementation of the Change Calculator. In this code Give Change is a procedure with the change_value parameter being passed by reference.

```
/*
* Program: simple-change.cpp
* Calculate the ideal change for a given transaction.
*/
#include <stdio.h>

#define TWO_DOLLARS 200
#define ONE_DOLLAR 100
#define FIFTY_CENTS 50
#define TWENTY_CENTS 20
#define TEN_CENTS 10
#define FIVE_CENTS 5

int coins_to_give(int change, int coin_value)
{
    return change / coin_value;
}

void give_change(int &change_value, int coin_value, const char *coin_desc)
{
    int to_give;

    to_give = coins_to_give(change_value, coin_value);
    change_value = change_value - to_give * coin_value;

    printf("%d x %s, ", to_give, coin_desc);
}

int get_change_value()
{
    int cost_of_item;
    int payment;

    printf("Cost of item (in cents): ");
    scanf("%d", &cost_of_item);

    printf("Amount paid (in cents): ");
    scanf("%d", &payment);

    return payment - cost_of_item;
}

int main()
{
    int change_value;
    change_value = get_change_value();

    printf("Change: ");
    give_change(change_value, TWO_DOLLARS, "$2");
    give_change(change_value, ONE_DOLLAR, "$1");
    give_change(change_value, FIFTY_CENTS, "50c");
    give_change(change_value, TWENTY_CENTS, "20c");
    give_change(change_value, TEN_CENTS, "10c");
    give_change(change_value, FIVE_CENTS, "5c");
```

```
    printf("\n");
    return 0;
}
```

Listing 2.6: C++ code for the Change Calculator

Note

- Save the C++ code in a file named `simple-change.cpp`.
- Compile this using `g++ -o SimpleChange simple-change.cpp` .
- Run the resulting program using `./SimpleChange` .
- Perform each of the Test Cases from Table 2.2 and check that the output matches the expected values.
- Look over the code and examine how the Variables, Parameters, Constants, and Function are being used.

2.3.3 C Variable Declaration

A Variable Declaration allows you to create a Variable in your Code. In C you can declare variables in the program's code, and in Functions and Procedures.

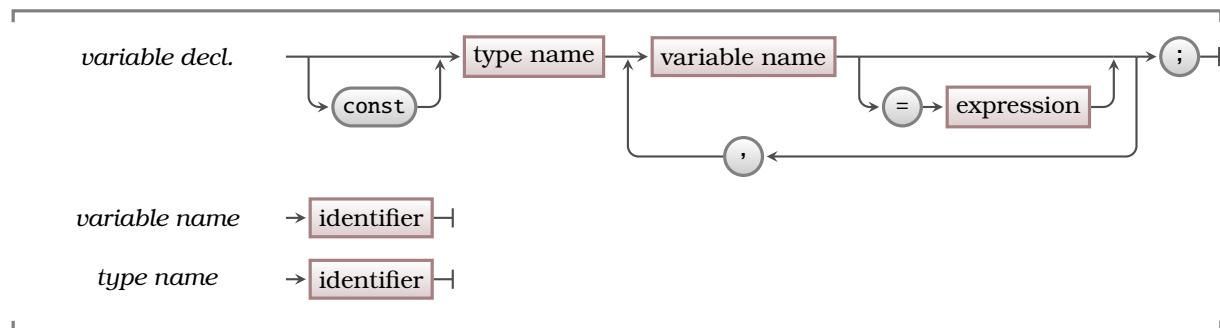


Figure 2.18: C++ Syntax for Variable Declaration

C++

```

/*
 * Program: variable_test.c
 * This program demonstrates some variable declarations.
 */
#include <stdio.h>

const float PI = 3.1415;
float global_float = 12.3;
int global_int = 73;

void test(int param_int, float param_float)
{
    int my_local = 37, another_local = 42;
    printf("my local int = %d, another_local = %d\n", my_local, another_local);
    printf("param int = %d, param float = %f\n", param_int, param_float);
    printf("globals are %f and %d\n", global_float, global_int);
}

int main()
{
    int local_int;
    local_int = 21;

    test(local_int, PI * local_int * local_int);

    printf("local int = %d\n", local_int);
    printf("globals are %f and %d\n", global_float, global_int);
    printf("PI is a constant with value %f\n", PI);
    return 0;
}
  
```

Listing 2.7: Variable Declaration Tests

Note

- This is the C Syntax for creating your own [Variable](#).
- This syntax can be used to declare ...
 - Local Variables within Functions and Procedures.
 - Global Variables within the program.
- In C the Variable Declaration starts with the [Type](#) name indicating the kind of data that will be stored.
- Following the Type is a list of the identifiers for the Variables that are being created. You can create one or more variables in a single Variable Declaration, but all of these Variables will have the same type.
- Each variable can be assigned a value when it is declared.
- The **const** modifier can be added to the start of a Variable declaration to create a Constant.
- See [C Procedure Declaration \(with Local Variables\)](#) for details on declaring Local Variables within Functions and Procedures.
- See [C Program \(with Global Variables and Constants\)](#) for details on declaring Global Variables within the program itself.
- The syntax for declaring Parameters is very similar, see [C Procedure Declaration \(with Parameters\)](#).



2.3.4 C Program (with Global Variables and Constants)

You can declare global variables and constants within a C Program file.

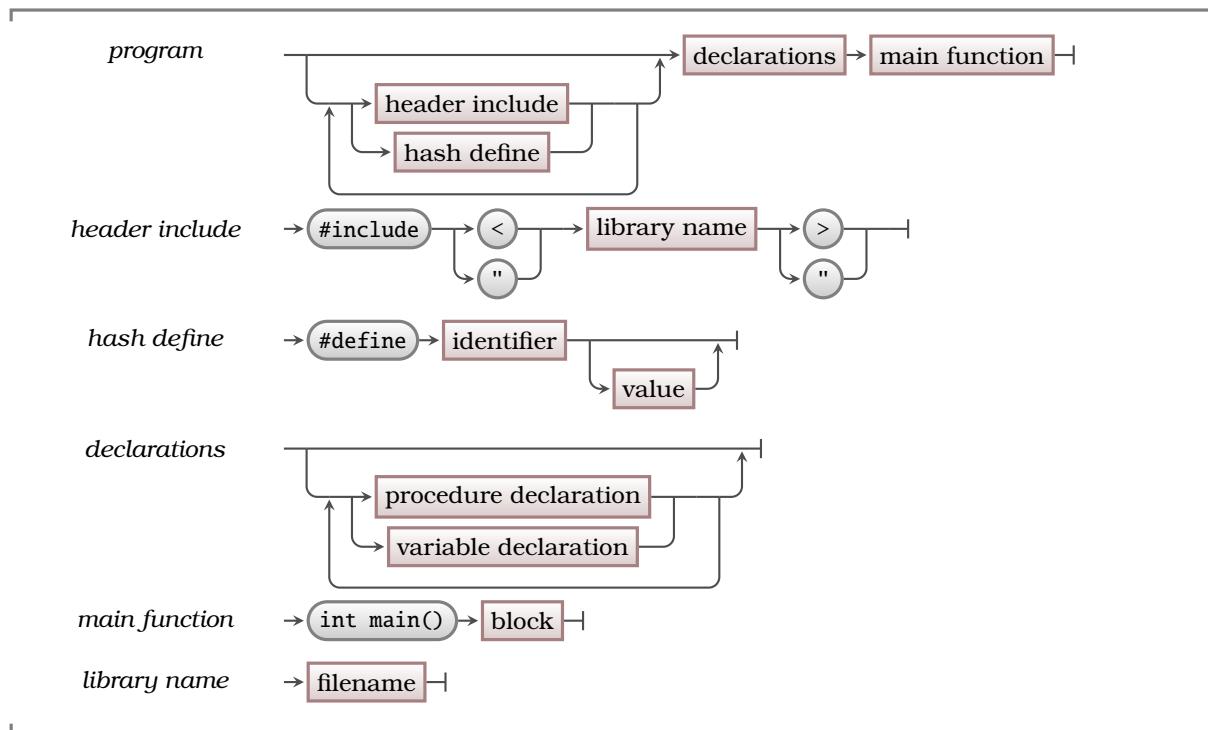


Figure 2.19: C++ Syntax for a program (with global variables and constants)

Note

- This syntax allows you to declare [Global Variables](#) and Constants.
- See Listing 2.7 for an example of declaring Global Variable and Constants.
- In Listing 2.7 ...
 - `global_float` and `global_int` are Global Variables. These can be accessed in both the test procedure and `main`.
 - `PI` is a Global Constant, with the value 3.1415. This can be read in both the test procedure and `main`.
- Global variable should be avoided.
- Global constants can be declared in two ways, using a [C Variable Declaration](#) with the `const` modifier or using `#define`.
- See Listing 2.8 for an example of `#define` and constants.
- With `#define` you are defining a 'value' for the identifier, this identifier is then substituted with the 'value' throughout your code. This means you have to pay particular attention to what 'value' you use.
- There are a number of conventions, called coding standards, that describe how your code should appear for a given language. In this text we will use a common C convention of having all *Constants* in **UPPER CASE**, with underscores (_) used to separate words. So the *Maximum Height* constant becomes `MAXIMUM_HEIGHT`.

C++

```
#include "splashkit.h"

// This defines that the identifier PI should always be replaced
// with the "value" 3.1415
// This is like "replace all" PI with 3.1415 (Notice no =)
#define PI 3.1415

// This defines a global constant that store the value
// 3.1415 / 180 ... which is 0.01745277778
// Notice the = ... this means "assign to"
// In this case there is a DEG_TO_RAD "constant"
const float DEG_TO_RAD = PI / 180;

int main()
{
    write_line(PI);
    write_line(DEG_TO_RAD);

    return 0;
}
```

Listing 2.8: C program with a defined constant, and constant variable



2.3.5 C Procedure Declaration (with Local Variables)

The Functions and Procedures in C can contain declaration for [Local Variables](#).

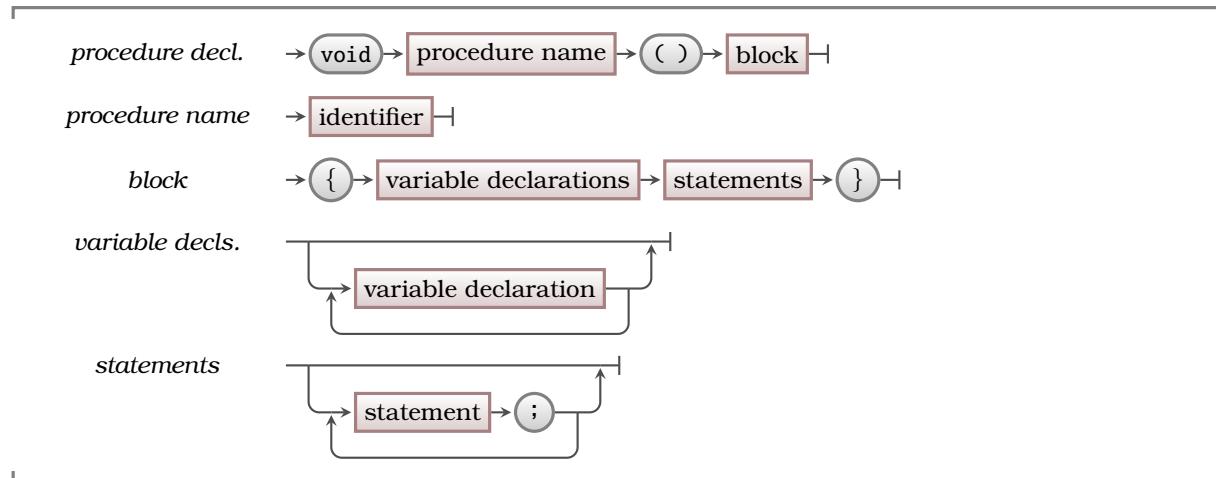


Figure 2.20: C++ Syntax for Procedure Declaration (with Local Variables)

Note

- This is the syntax for declaring [Local Variables](#) in a Procedure.
- See Listing 2.7 for an example of declaring Local Variables.
- In Listing 2.7 ...
 - The test procedure has two local variables: `my_local` and `another_local`.
 - The `main` function has one local variable called `local_int`.
- The Local Variables must be declared before the statements within the Function and Procedure's block.
- In C you cannot declare variables after the first statement in the block.
- In this text we will use a common C convention of having all *Local Variables* in **lower case**, with underscores (_) used to separate words. So the *My Name* local variable becomes `my_name`.

2.3.6 C++ Assignment Statement

The assignment statement is used to store a value in a variable.

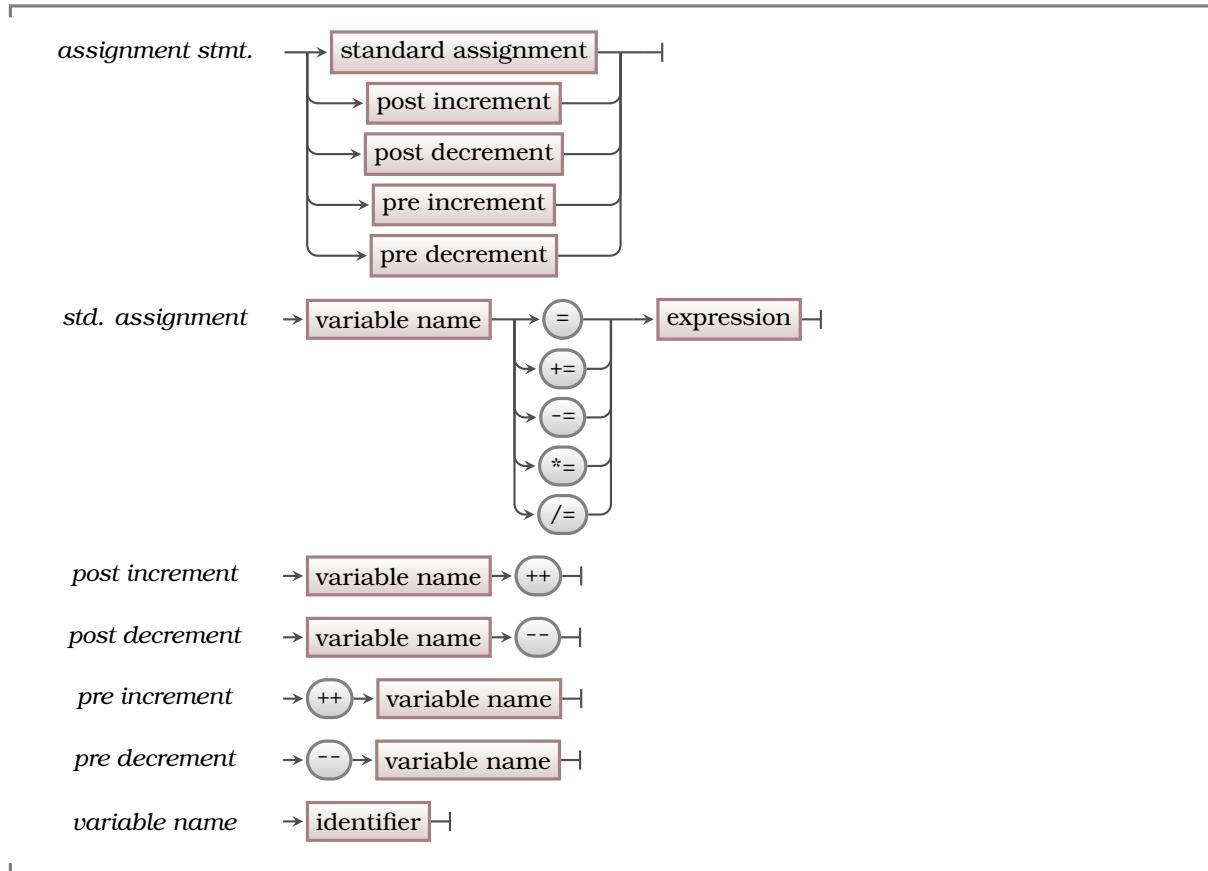


Figure 2.21: C++ Syntax for an Assignment Statement

Note

- This is the C++ syntax for the [Assignment Statement](#).
- In C++ assignment is indicated by the equals sign (=).
- The *left hand side* of the assignment must be a valid variable, this is where the value is to be stored.
- The *right hand side* of the assignment is an expression, this calculates the value that will be stored in the Variable.
- There are multiple versions of the assignment, giving short hand ways of using the current value.
 - = is the standard assignment, this stores the value of the expression in the Variable.
 - += increments the variable's value,
a += n; is equivalent to a = a + n;
 - -= decrements the variable's value,
a -= n; is equivalent to a = a - n;
 - *= multiplies the value in the variable by a given factor.
a *= n; is equivalent to a = a * n;
 - /= divides the value in the variable by a factor.
a /= n; is equivalent to a = a / n;
- The ++ and -- operators allow a variables value to be incremented or decremented.

C++

```
#include "splashkit.h"

/* Program assignment_tests.c
 * Demonstrates assignment to a variable.
 */
int main()
{
    int my_data = 10, days_in_month, days_remaining;
    write("my_data is ");
    write_line(my_data);

    my_data = my_data + 1;      //add 1 to my_data and store in my_data
    write("my_data is ");
    write_line(my_data);

    my_data++;     //add 1 to my_data and store in my_data
    write("my_data is ");
    write_line(my_data);

    my_data *= 2;    //double my_data and store in my_data
    printf("my_data is %d\n", my_data);

    days_in_month = 365 / 12; //assign days_in_month a calculated value
    printf("On average there are " + to_string(days_in_month) + " days in a month.");

    //assign days_remaining a calculated value
    days_remaining = 365 - days_in_month * 12;
    write_line("The remaining " + to_string(days_remaining) + " days are distributed to a few months");

    return 0;
}
```

Listing 2.9: Assignment Tests

2.3.7 C Procedure Declaration (with Parameters)

In C [Parameters](#) can be declared in any Function or Procedure declaration.

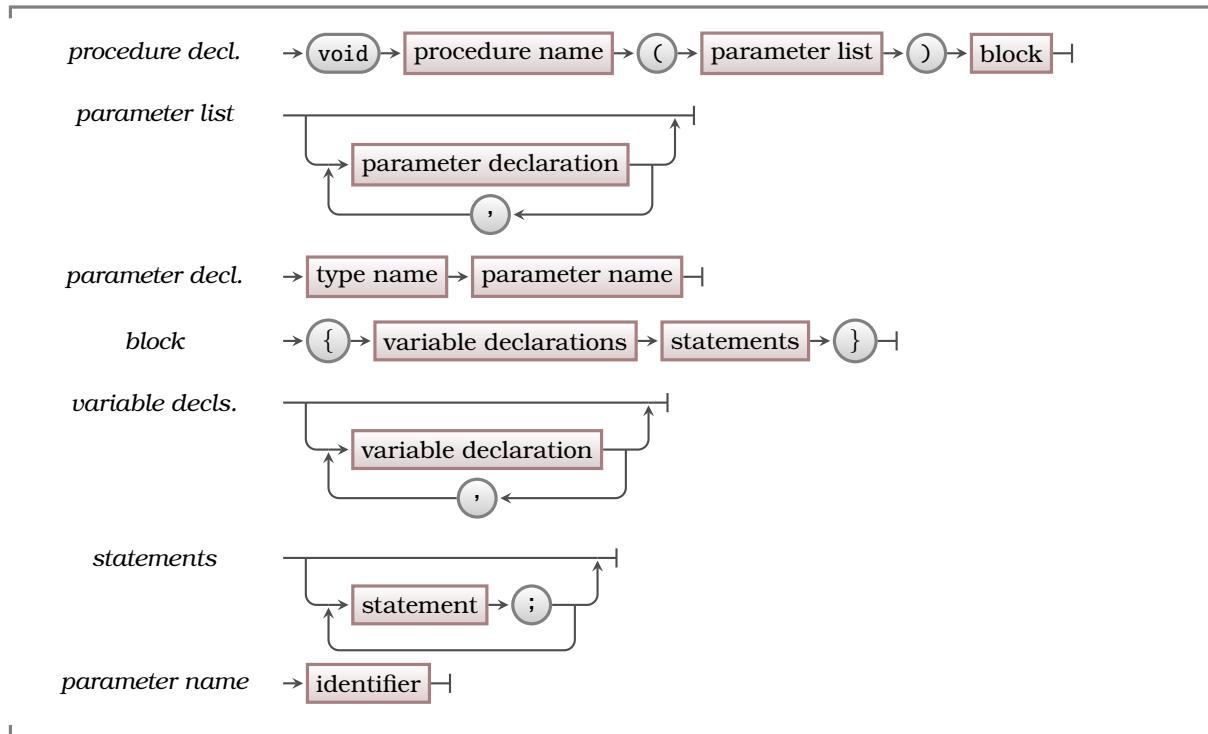


Figure 2.22: C++ Syntax for Procedure Declarations (with Parameters)

Note

- The syntax in Figure 2.22 shows the C code for declaring Procedures with Parameters.
 - Parameters in C are declared in a similar way to other Variables, with the Type name appearing first followed by the Parameter's name.

C++

```
/* Program: parameter-test.c */

void print_equation(int m, double x, int c)
{
    write_line(to_string(m) + " x " + to_string(x) " + " + to_string(c) + " = " + to_string(m * x + c));
}

int main()
{
    print_equation(2, 5.1, 3);
    print_equation(7, 2.74, -8);
    return 0;
}
```

Listing 2.10: Assignment Tests

C++ Reference Parameters

C has limited support for pass by reference, but this feature was added with the extensions in the C++ language. The following syntax shows how to declare a parameter that will be passed by reference. Please note that this is not standard C code, and will require you to use a C++ compiler.

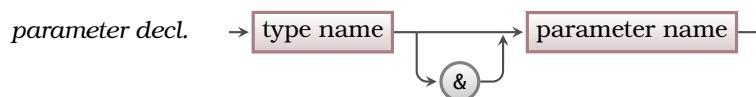


Figure 2.23: C++ Syntax for by-ref Parameters

C++

```

/* Program: test-byref.cpp */
#include <stdio.h>

void double_it(int &data)
{
    printf("Data passed in was %d, about to double it...\n", data);
    data = data * 2;
    printf("In double_it data is now %d\n", data);
}

int main()
{
    int val = 3;

    printf("In main val is %d\n", val);
    double_it(val);
    printf("Back in main val is now %d\n", val);
}
  
```

Listing 2.11: Example of passing by reference using C++

Note

- This requires a **C++ compiler**, such as the g++ compiler.
- C++ is an extension of the C compiler, so your C code should be able to be compiled with the C++ compiler.
- The C++ compiler is more strict on some rules than the C compiler. This means the compiler will give you more help in avoiding some poor programming practices.
- Notice that with pass by reference you do not need to do anything special in the call, the compiler takes care of the necessary details needed to achieve this.

2.3.8 C Procedure Call (with pass by reference)

Many languages support pass by reference in the compiler. This is where the compiler manages the passing of the reference for you in the background. Unfortunately C does not do this transparently and you need to manually pass the reference yourself. Its good to know that the concept remains the same, but it does mean that you must manually add code to achieve this.

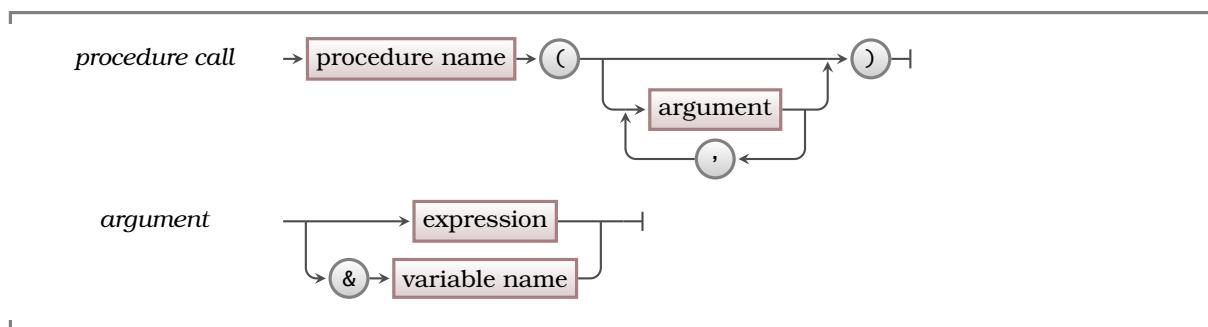


Figure 2.24: C++ Syntax for Procedure Call (with pass by reference)

Note

- With C you must manually pass parameters by reference using the ampersand (&) operator.
- C++ does have support for pass by reference, see [C++ Reference Parameters](#).
- The ampersand (&) operator gets the address of the variable given to it, in effect you manually fetch the reference to the variable and pass that as the argument.
- Passing the address of a variable allows the called code to use that address to find your Variable's value. The code can read and store values in this Variable for you.
- Reading input from the Terminal is one of the common use of pass by reference.



C++

```
/* Program: input-test.c */

int main()
{
    int age = 0;
    double num = 0.0;
    string line;

    write("Please enter your age: ");
    // pass age by reference, allowing scanf to store the value read
    // into this variable
    line = read_line();
    age = convert_to_integer(line);

    write("What is your favourite number: ");
    // pass num by reference
    line = read_line();
    num = convert_to_double(line);

    write_line("Thanks, your age is " + to_string(age));
    write_line("Your favourite number is " + to_string(num));

    return 0;
}
```

Listing 2.12: Testing Pass by Reference in C

2.3.9 C Terminal Input

C comes with a range of [Librarys](#) that provide reusable programming artefacts, including reusable [Function](#) and [Procedures](#). The stdio.h refers to the Standard Input/Output library, and including code to read input from the Terminal. The scanf function is used to read data from the Terminal.

Function Prototype	
<code>int scanf(char *format, ...)</code>	
Returns	
int	The number of values read by scanf.
Parameter	Description
format	The format specifier describing what is to be read from the Terminal. See Table 2.4.
...	The variables into which the values will be read. There must be at least as many variables as format tags in the format specifier.

Table 2.3: Parameters that must be passed to scanf

The scanf function is controlled by the format parameter. This parameter tells scanf what it must read from the input. Details of how to construct the format String are shown in Table 2.4.

	Description	Example Usage
<i>white space</i>	Skips white space at this point in the input.	<code>scanf(" %d", &age);</code>
<i>non white space</i> ^a	Matches the input against characters, skipping this text in the input. Fails if input does not match. The example looks for 'age: ' and reads the following integer.	<code>scanf("age: %d", &age);</code>
<i>format tag</i>	The tag indicates the kind of data to read and store in a Variable. This starts with a percent character. See Table 2.5 and Figure 2.25 for the syntax of the format tags.	<code>scanf("%d", &age);</code>

Table 2.4: Elements of the Format String in scanf

^aExcept for the percent character which is used in the Format Tag.

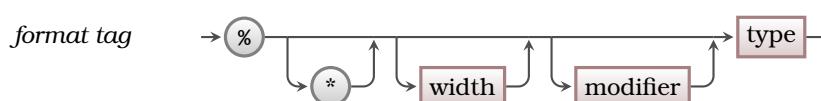


Figure 2.25: C++ Syntax for Format Tags for scanf

C++

```
/*
 * Program: test-scanf.c
 * Tests some uses for scanf
 */

int main()
{
    int age;
    long int count;
    float val;
    double height;
    long double range;

    printf("Please enter age, count, val, height, then range.\nInput: ");
    scanf("%d %ld %f %lf %Lf", &age, &count, &val, &height, &range);

    printf("You entered: %d %ld %f %f %Lf\n", age, count, val, height, range);

    return 0;
}
```

Listing 2.13: Example of reading data using scanf.

	Description	Example Usage
*	Read the data, but ignore it. Does not store the value in a Variable.	<code>scanf("%*d");</code>

Width	Description	Example Usage
<i>number</i>	The maximum number of characters to read in the current operation.	<code>scanf("%3d", &age);</code>

Modifier	Description	Example Usage
h	Reads a <code>short int</code> for the d or i Types.	<code>scanf("%hi", &age);</code>
l	Reads a <code>long int</code> for the d or i Types, or a <code>double</code> for f.	<code>scanf("%lf %li", &height, &count);</code>
L	Reads a <code>long double</code> for f.	<code>scanf("%Lf", &range);</code>

Type	Data Read	Example Usage
c	A single character.	<code>scanf("%c", &ch);</code>
d or i	Decimal integer. This is able to read a starting + or - if present.	<code>scanf("%d", &height);</code>
f	Decimal floating point number. Can be signed, or in scientific notation.	<code>scanf("%f", &radius);</code>
s	Text data. Should be preceded by the number of characters to read. The c-string must have sufficient space to store the data read ^a .	<code>scanf("%40s", name);</code>
[pattern]	Text data. As with %s, but this allows you to specify the pattern of characters that can be read.	<code>scanf("%7[1234567890]", num_text);</code>
[^pattern]	Text data. As with %s, but this allows you to specify the pattern of characters that can not be read.	<code>scanf("%40[^\\n]", name);</code>

^aThis will be covered in future chapters.

Table 2.5: Details for `scanf`'s Format Tag type, specifiers, modifiers, and width

2.3.10 C Program (with Functions)

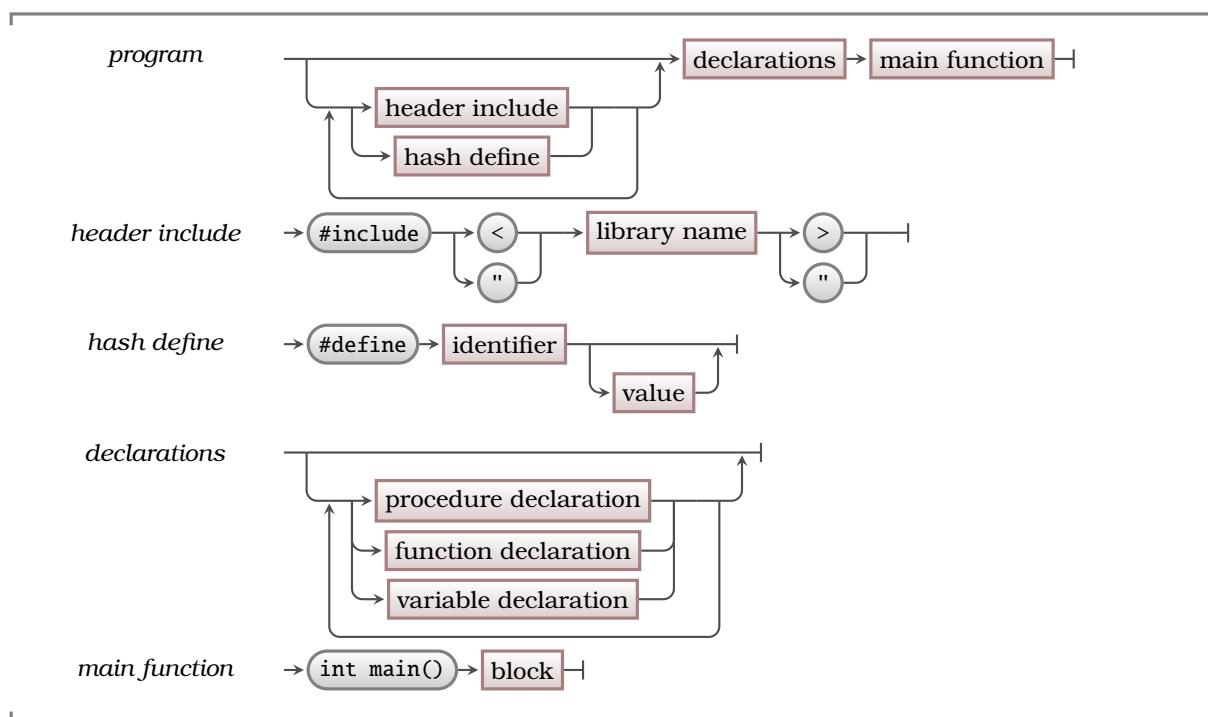


Figure 2.26: C++ Syntax for a Program (with Functions)

C++

```
// ... header missing

double fahrenheit_to_celsius(double fahrenheit)
{
    return (5.0 / 9.0) * (fahrenheit - 32);
}

int main()
{
    double input_temp;
    string line;

    write("Please enter temperature in Fahrenheit: ");
    line = read_line();
    input_temp = convert_to_double(line);

    write_line("This is " + to_string(fahrenheit_to_celsius(input_temp)) + " in Celsius");
    return 0;
}
```

Listing 2.14: Example of declaring a Function in a C Program.

Note

- A C program may contain declarations for custom **Functions**.
- The Function must be declared before it is used.

2.3.11 C Function Declaration

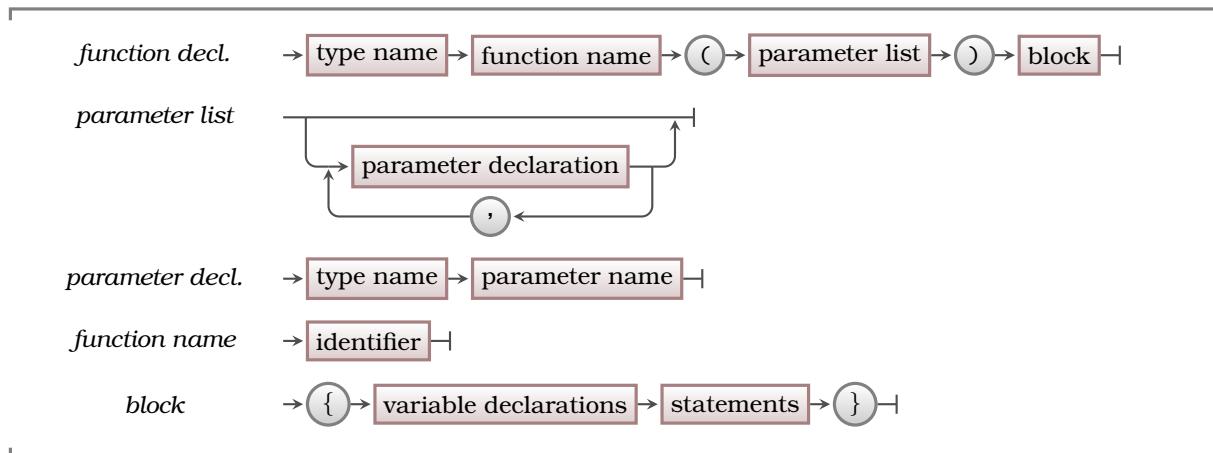


Figure 2.27: C++ Syntax for a Function

C++

```

/* Program: test-square.c */
#include <stdio.h>

int square(int val)
{
    return val * val;
}

int main()
{
    printf("5 squared is %d\n", square(5));
    return 0;
}
  
```

Listing 2.15: Example Function Declaration of a square Function.

Note

- In C [Function](#) and [Procedure](#) declarations are very similar.
- In C, a Function's declaration starts with the [Type](#) of data the Function will return.
- This is followed by the name of the Function, and its Parameters. In the same way as is done in the [C Procedure Declaration \(with Parameters\)](#).
- The body of the function is a block, in the same was as a [C Procedure Declaration \(with Parameters\)](#).
- See [C Function Call](#) for the syntax needed to call your functions.
- See the [Return Statement](#) to see how to return a result from a function in C.
- void is a type, so in C functions and procedures are identical. See [C Procedure Declaration \(as Function\)](#) to see how C handles procedures.
- The entry point of the program is the main function. It returns a number to the Operating System that can be used to indicate the success or failure of the program. You can read the value returned from the last program to execute in the Terminal using `echo $?`.

2.3.12 C Procedure Declaration (as Function)

C does not have a strong separation of Functions and Procedures. Instead, in C all Procedures are Functions that return a special void [Type](#). This means that the standard C Syntax does not include a separate definition for Procedure declarations. Even though C does not have direct syntax for Procedures, the concept is still very important.

procedure decl. → function declaration

Figure 2.28: C++ Syntax for a Procedure (as a Function)

Note

- In C all Procedures are Functions that return a void type.
- The void type indicates an *empty* type. This indicates that Procedures return no data to the caller.

2.3.13 C Function Call

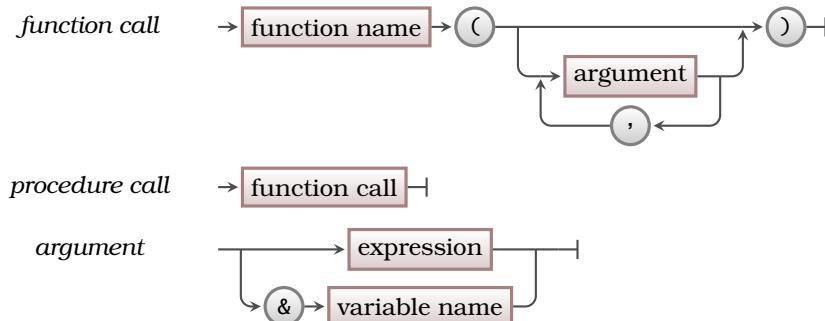


Figure 2.29: C++ Syntax for a Function Call

C++

```

/* Program: test-fn-calls.c */
#include <stdio.h>

int square(int x) { return x * x; }
int sum(int a, int b) { return a + b; }

int main()
{
    int answer = sum(square(5), square(4));
    printf("5 squared + 4 squared is %d\n", answer);

    printf("(1 + 2) + (3 + 4) = %d\n", sum(sum(1, 2), sum(3, 4)));
    printf("2 squared, squared = %d\n", square(square(2)));

    return 0;
}
  
```

Listing 2.16: Example of Function Calls.

Note

- A C function call is similar to a [Procedure Call](#).
- You use the name of the [Function](#), its identifier, to indicate which function is called.
- Following the Function's name is the list of *arguments*, these are the values (or variables) that are being passed to the called Function.
- The return type of the Function determines where the Function may be called.
 - Procedure, void Functions, can only be called in a [Procedure Call Statement](#).
 - Other Functions can be used in Expressions, see [Expressions \(with Function Calls, Variables, and Constants\)](#). In these cases the type of data returned by the Function will determine the type of the Function Call.
- In Listing 2.16 the values of the inner function calls are passed to the arguments of the outer calls. This means that `square(5)` is calculated first then `square(4)`. The results of these two Function Calls are then passed as the *arguments* into the call to the `sum` Function. In this case the Function call will be `sum(25, 16)` with 25 being the result returned by `square(5)` and 16 being the result returned by `square(4)`.

2.3.14 Return Statement



Figure 2.30: C++ Syntax for a Return Statement

C++

```

/* Program: test-return.c */
#include <stdio.h>

int test_return()
{
    printf("test-return started\n");
    return 3;
    printf("Cannot be run as code returned above!");
}

int main()
{
    printf("Calling test_return - the value %d is returned!\n", test_return());
    return 0;
}
  
```

Listing 2.17: Example illustrating return in action.

Note

- The Return Statement is used to end a Function, or Procedure, and to return a value.
- Procedures can end completing all of their instructions.
- Functions that return a value must have a return to indicate the value to be returned by the caller.
- The *Expression* in the Return Statement is optional so that you can use the statement to end a Procedure.
- When the Return Statement is executed the current Function or Procedure ends, and the value of the *Expression* is returned to the Function Call.
- Listing 2.17 illustrates the point that return ends the current function, with only the first printf call being run in the test_return() Function.

2.3.15 C Statement (with Return Statement)

The different statements of a Programming Language allow you to command the Computer to perform different actions. We have covered three kinds of statements so far.

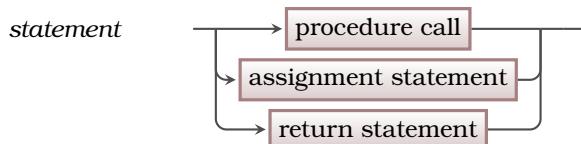


Figure 2.31: C++ Syntax for Statements (with Return Statement)

Note

- At this point you have seen how to command the computer to:
 - Call a Procedure (void Function) using a **procedure call**.
 - Assign a value to a Variable using an **Assignment Statement**.
 - End the current Function or Procedure, and return a value to the caller using a **Return Statement**.
- These Statements can be placed within the *blocks* of the Functions and Procedures of your Programs.



2.4 Storing and Using Data in Pascal

2.4.1 Implementing Change Calculator in Pascal

Section 2.2 of this chapter introduced the ‘Change Calculator’ program, and its design. Its implementation requires the definition of functions as well as procedures. These functions and procedures accepted parameters and use local variables.

This section of the chapter introduces the Pascal syntax rules for implementing these concepts using the Pascal language. The Pascal implementation of the Change Calculator is shown in Listing 2.18.

```
// Calculate the ideal change for a given transaction.
program SimpleChange;

const
  TWO_DOLLARS = 200;
  ONE_DOLLAR = 100;
  FIFTY_CENTS = 50;
  TWENTY_CENTS = 20;
  TEN_CENTS = 10;
  FIVE_CENTS = 5;

function CoinsToGive(change, coinValue: Integer): Integer;
begin
  result := change div coinValue;    // integer division... ignore remainder
end;

procedure GiveChange(var changeValue: Integer; coinValue: Integer; coinDesc: String);
var
  toGive: Integer;
begin
  toGive := CoinsToGive(changeValue, coinValue);
  changeValue := changeValue - toGive * coinValue;

  Write(toGive, ' x ', coinDesc, ', ');
end;

function GetChangeValue(): Integer;
var
  costOfItem, payment: Integer;
begin
  Write('Cost of item (in cents): ');
  ReadLn(costOfItem);

  Write('Amount paid (in cents): ');
  ReadLn(payment);

  result := payment - costOfItem;
end;

procedure Main();
var
  changeValue: Integer;
begin
  changeValue := GetChangeValue();

  Write('Change: ');
  GiveChange(changeValue, TWO_DOLLARS, '$2');
  GiveChange(changeValue, ONE_DOLLAR, '$1');
```

```
GiveChange(changeValue, FIFTY_CENTS, '50c');
GiveChange(changeValue, TWENTY_CENTS, '20c');
GiveChange(changeValue, TEN_CENTS, '10c');
GiveChange(changeValue, FIVE_CENTS, '5c');
WriteLn();
end;

begin
    Main();
end.
```

Listing 2.18: Pascal code for the Change Calculator

Note

- Save the Pascal code in a file named SimpleChange.pas.
- Compile this using `fpc -S2 SimpleChange.pas`.
- Run the resulting program using `./SimpleChange`.
- Perform each of the Test Cases from Table 2.2 and check that the output matches the expected values.
- Look over the code and examine how the variables, parameters, constants, and function are being used.
- Notice how the indentation makes it easy to see where each function and procedure starts and ends. Always lay your code out so that it is easy to see its structure.
- See how the function and procedure are declared before they are used. This is important as the Pascal compiler reads the code from the start, and must know about the artefacts before you use them.



2.4.2 Pascal Program (with Global Variables and Constants)

You can declare global variables and constants within a Pascal Program file.

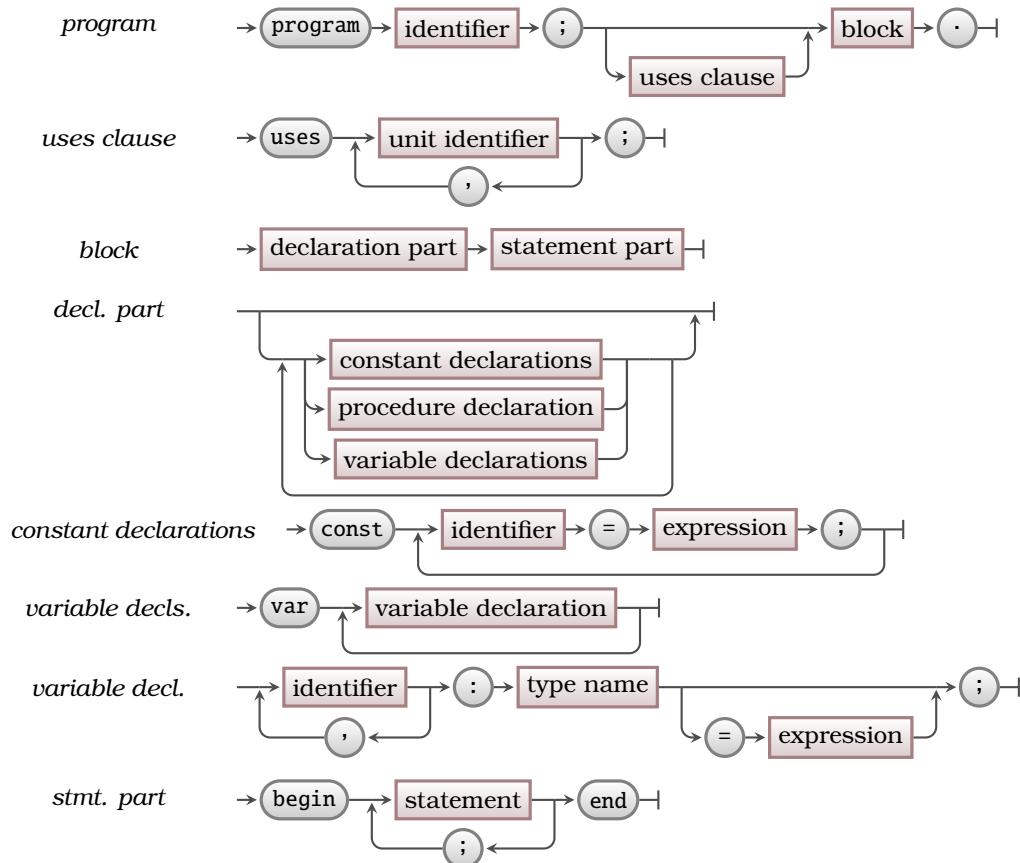


Figure 2.32: Pascal Syntax for a program (with global variables and constants)

Note

- This syntax allows you to declare **Global Variables** and Constants.
- See Listing 2.19 for an example of declaring Global Variable and Constants.
- In Listing 2.19 ...
 - `globalFloat` and `globalInt` are global variables. These can be accessed in both the `test` procedure and `main` procedure.
 - `PI` is a global constant, with the value 3.1415. This can be read in both the `test` procedure and `main` procedure.
- Global variable should be avoided.
- There are a number of conventions, called coding standards, that describe how your code should appear for a given language. In this text we will use a common Pascal convention of having all **Constants** in **UPPER CASE**, with underscores (`_`) used to separate words. So the `Maximum Height` constant becomes `MATRIX_HEIGHT`.

Pascal

```
// This program demonstrates some variable declarations.  
program VariableTest;  
  
const PI = 3.1415;  
var  
    globalFloat: Single = 12.3;  
    globalInt: Integer = 73;  
  
procedure Test(paramInt: Integer; paramFloat: Single);  
var  
    myLocal: Integer = 37;  
    anotherLocal: Integer = 42;  
begin  
    WriteLn('my local int = ', myLocal, ', anotherLocal = ', anotherLocal);  
    WriteLn('param int = ', paramInt, ', param int2 = ', paramFloat:4:2);  
    WriteLn('globals are ', globalFloat, ' and ', globalInt);  
end;  
  
procedure Main();  
var  
    localInt: Integer;  
begin  
    localInt := 21;  
  
    Test(localInt, PI * localInt * localInt);  
  
    WriteLn('local int = ', localInt);  
    WriteLn('globals are ', globalFloat, ' and ', globalInt);  
    WriteLn('PI is a constant with value ', PI);  
end;  
  
begin  
    Main();  
end.
```



Listing 2.19: Example variable and constant declarations

2.4.3 Pascal Procedure Declaration (with Local Variables)

The functions and procedures in Pascal can contain declaration for [Local Variables](#).

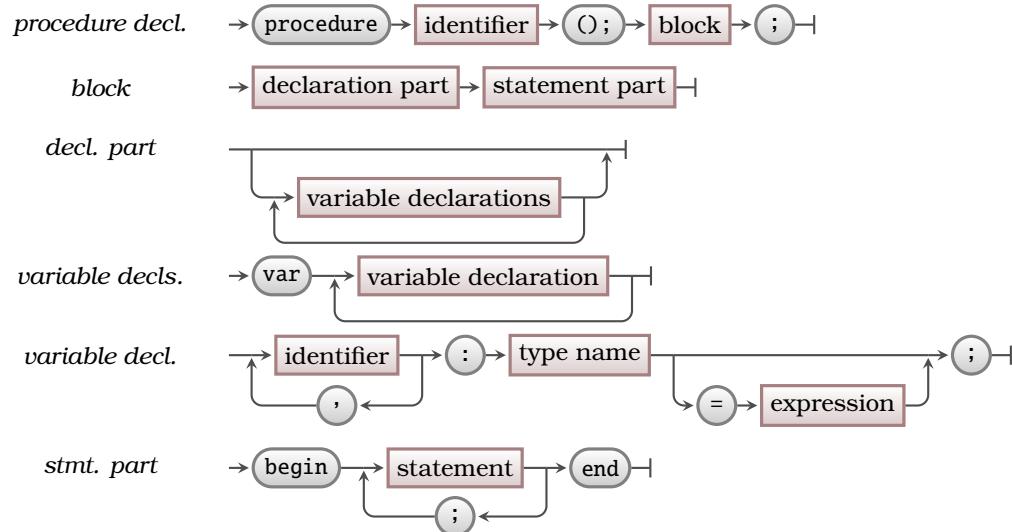


Figure 2.33: Pascal Syntax for Procedure Declaration (with Local Variables)

Note

- This is the syntax for declaring [Local Variables](#) in a procedure.
- See Listing 2.19 for an example of declaring local variables.
- In Listing 2.19 ...
 - The test procedure has two local variables: `myLocal` and `anotherLocal`.
 - The `main` function has one local variable called `localInt`.
- The local variables are declared before the procedure's statements.
- In this text we will use a common Pascal convention of having all *local variables* in **camel Case**, where the first character is lower case but subsequent words in the identifier start with an upper case character. So the `My Name` local variable becomes `myName`.

2.4.4 Pascal Assignment Statement

The assignment statement is used to store a value in a variable.

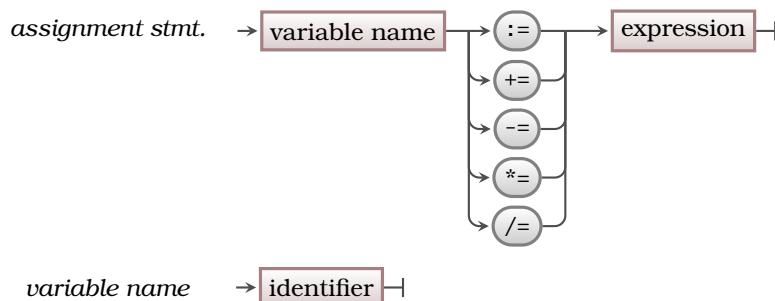


Figure 2.34: Pascal Syntax for an assignment statement

Note

- This is the Pascal syntax for the [Assignment Statement](#).
- In Pascal assignment is indicated by a colon followed by an equals sign (`:=`).
- The *left hand side* of the assignment must be a valid variable, this is where the value is to be stored.
- The *right hand side* of the assignment is an expression, this calculates the value that will be stored in the variable.
- There are multiple versions of the assignment, giving short hand ways of using the current value.
 - `=` stores the value of the expression in the variable.
 - `+=` increments the variable's value,
`a += n;` is equivalent to `a := a + n;`
 - `-=` decrements the variable's value,
`a -= n;` is equivalent to `a := a - n;`
 - `*=` multiplies the value in the variable by a factor.
`a *= n;` is equivalent to `a := a * n;`
 - `/=` divides the value in the variable by a factor.
`a /= n;` is equivalent to `a := a / n;`

Pascal

```

program AssignmentTest;

procedure Main();
var
  myData: Integer = 10;
  daysInMonth, daysRemaining: Integer;
begin
  WriteLn('myData is ', myData);

  myData := myData + 1;      //add 1 to myData and store in myData
  WriteLn('myData is ', myData);

  myData += 1;      //add 1 to myData and store in myData
  WriteLn('myData is ', myData);

  myData *= 2;      //double myData and store in myData
  WriteLn('myData is ', myData);

  daysInMonth := 365 div 12; //assign daysInMonth a calculated value
  WriteLn('On average there are ', daysInMonth, ' days in a month.');

  //assign daysRemaining a calculated value
  daysRemaining := 365 - daysInMonth * 12;
  WriteLn('The remaining ', daysRemaining,
          ' days are distributed to a few months.');
end;

begin
  Main();
end.

```

Listing 2.20: Sample assignment statements

2.4.5 Pascal Procedure Declaration (with Parameters)

In Pascal **Parameters**s can be declared in any function or procedure declaration.

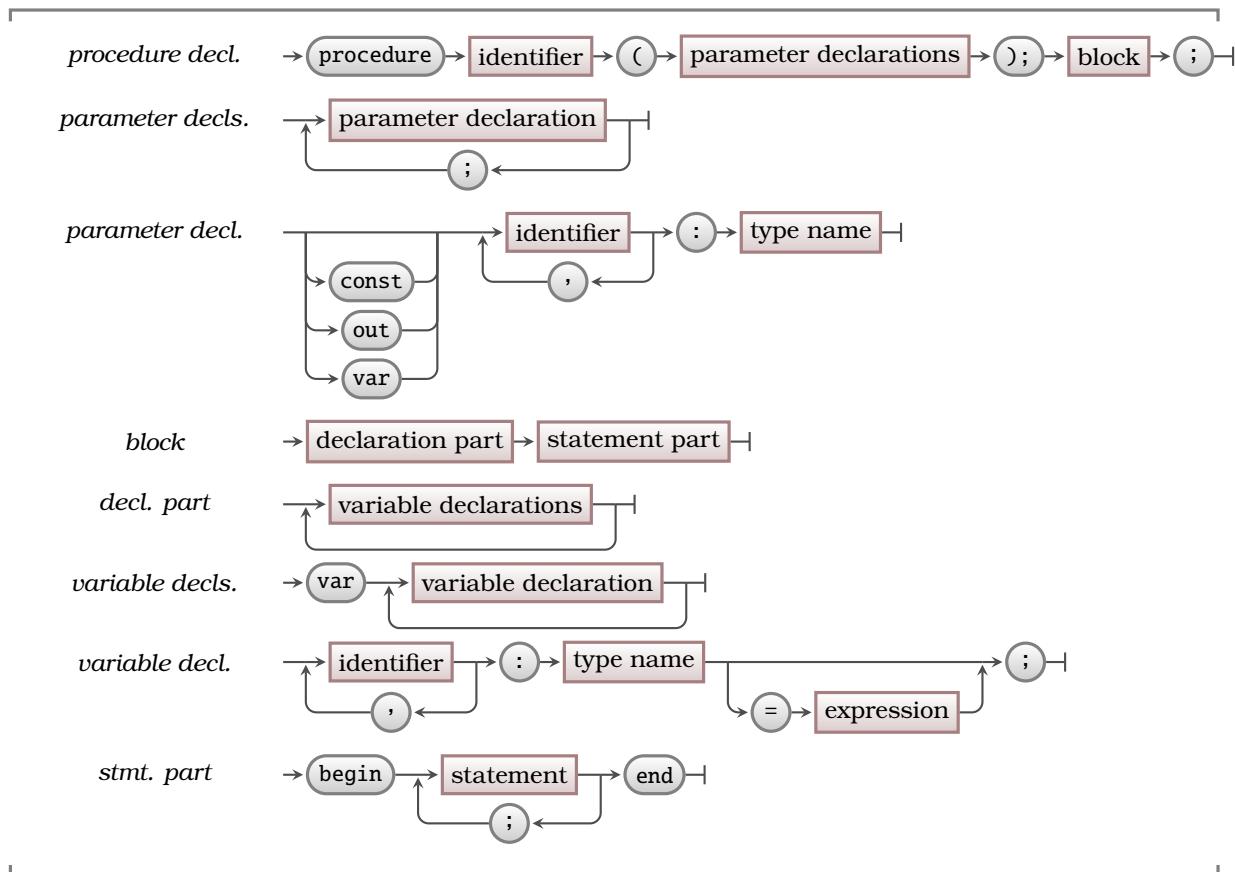


Figure 2.35: Pascal Syntax for procedure declarations (with parameters)

Note

- The syntax in Figure 2.22 shows the Pascal syntax for declaring procedures with **Parameters**.
 - Parameters in Pascal are declared in a similar way to other variables.
 - Pascal supports passing parameters by reference and by value. The parameters in Listing 2.21 are all passed by value.
 - There are three ways of passing references by reference in Pascal (see Listing 2.22 for examples of these):
 - **const**: The parameter is passed a reference to the variable, but cannot change its value. This can be used to pass a value **in** to the procedure by reference.
 - **out**: The parameter is passed a reference to the variable, but cannot assume it contains a meaningful value. This can be used to pass a value **out** to the procedure by reference.
 - **var**: The parameter is passed a reference to the variable. This can be used to pass a value **in** and **out** of the procedure by reference.

Pascal

```

program ParameterTest;

procedure PrintEquation(m: Integer; x: Double; c: Integer);
begin
    WriteLn(m, ' x ', x:4.2, ' + ', c, ' = ', m * x + c);
end;

begin
    PrintEquation(2, 5.1, 3);
    PrintEquation(7, 2.74, -8);
end.

```

Listing 2.21: Example procedure with parameters**Pascal**

```

program ParameterTypes;

procedure TestInByRef(const xIn: Integer);
begin
    WriteLn('I have a reference to the x variable');
    WriteLn('but I can not change it... ', xIn);
end;

procedure TestOutByRef(out xOut: Integer);
begin
    WriteLn('I have a reference to the x variable');
    WriteLn('It does not have a meaningful value');
    WriteLn('but I can store a value in it... ');
    xOut := 10;
end;

procedure TestInOutByRef(var xInOut: Integer);
begin
    WriteLn('I have a reference to the x variable');
    WriteLn('I can read its value ', xInOut);
    WriteLn('and I can store a value in it... ');
    xInOut := 20;
end;

procedure Main();
var
    x: Integer;
begin
    TestOutByRef(x);
    TestInByRef(x);
    TestInOutByRef(x);
    WriteLn('At the end of main x is ', x);
end;

begin
    Main();
end.

```

Listing 2.22: Example of the different parameter types

2.4.6 Pascal Terminal Input

Pascal comes with a range of [Libraries](#) that provide reusable programming artefacts, including reusable [Function](#) and [Procedures](#). The [System](#) unit includes procedures to read input from the Terminal. The [ReadLn](#) procedure is used to read data from the Terminal.

Procedure Prototype	
procedure ReadLn(...)	
Parameter	Description
...	The ReadLn procedure takes a variable number of parameters. A value is read from the Terminal for each parameters.

Table 2.6: Parameters that must be passed to [ReadLn](#)

Pascal

```
program HelloName;

procedure Main();
var
  name: String;
  x, y, z: Integer;
begin
  Write('Enter your name: ');
  ReadLn(name);
  WriteLn('Hello ', name);

  Write('Enter three numbers separated by spaces:');
  ReadLn(x, y, z);
  WriteLn('x = ', x, ' y = ', y, ' z = ', z);
end;

begin
  Main();
end.
```

Listing 2.23: Example of reading data from the Terminal



2.4.7 Pascal Program (with Functions)

Your Pascal program can contain definitions for your own [Functions](#).

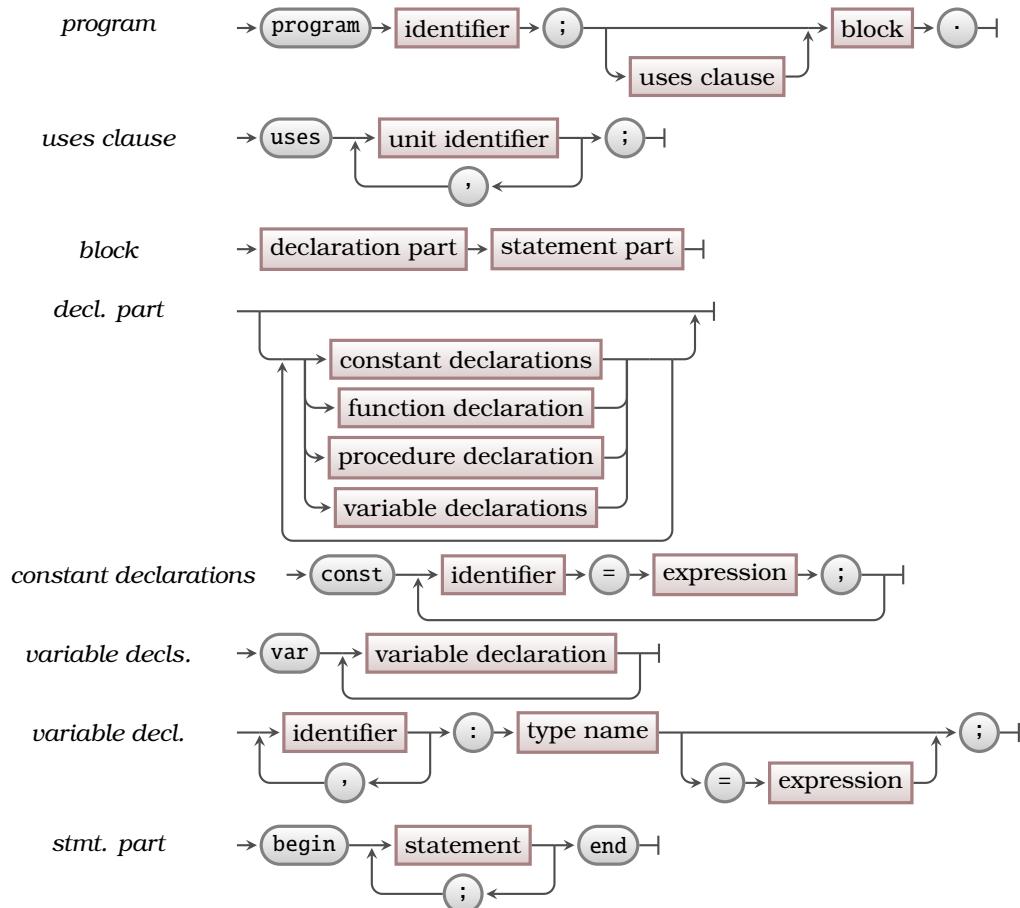


Figure 2.36: Pascal Syntax for a program (with functions)

Note

- A Pascal program may contain declarations for custom [Functions](#).
- The function must be declared before it is used.
- See [C Function Declaration](#) for the Syntax for declaring your own Functions.

Pascal

```
//  
// Program: FtoC.pas  
// Converts a temperature from Fahrenheit to Celsius  
//  
program FtoC;  
  
// Convert the tempF parameter to Celsius and return the result  
function ConvertFtoC(tempF: Double): Double;  
begin  
    result := (5/9) * (tempF-32);  
end;  
  
// Read a temperature in Fahrenheit, and output the value  
// converted to Celsius  
procedure Main();  
var  
    tempF, tempC: Double;  
begin  
    Write('Please enter temperature in Fahrenheit: ');  
    ReadLn(tempF);  
    tempC := ConvertFtoC(tempF);  
    WriteLn('This is ', tempC:4:2, ' in Celsius');  
end;  
  
begin  
    Main();  
end.
```



Listing 2.24: Example of declaring a function in a Pascal program

Note

- Listing 2.24 contains a program with a custom function that converts temperature from Fahrenheit to Celsius.



2.4.8 Pascal Function Declaration

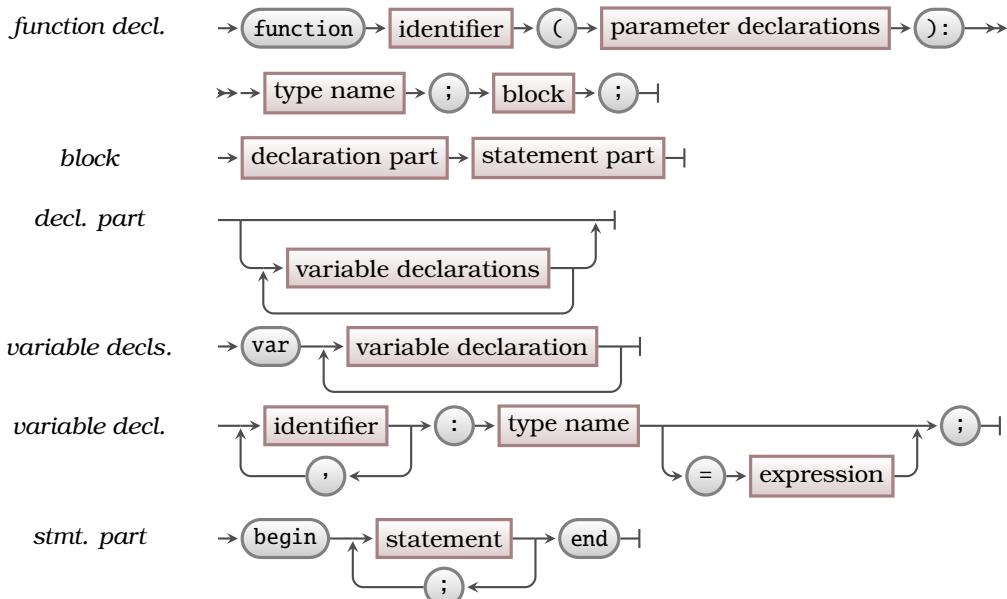


Figure 2.37: Pascal Syntax for a function

Pascal

```

program TestSquare;

function Square(val: Integer): Integer;
begin
  result := val * val;
end;

begin
  WriteLn('5 squared is ', square(5));
  WriteLn('73 squared is ', square(73));
end.
  
```

Listing 2.25: Example function declaration of a square function.

Note

- In Pascal **Function** and **Procedure** declarations are very similar.
- In Pascal, a function's declaration includes the **Type** of data the function will return.
- This is followed by the name of the Function, and its Parameters. In the same way as is done in the [C Procedure Declaration \(with Parameters\)](#).
- The body of the function is the same as a [Pascal Procedure Declaration \(with Parameters\)](#).
- See [Pascal Function Call](#) for the syntax needed to call functions.
- Each Pascal function has a special **result** variable, the value of this variable is returned to the caller when the function ends.

2.4.9 Pascal Function Call

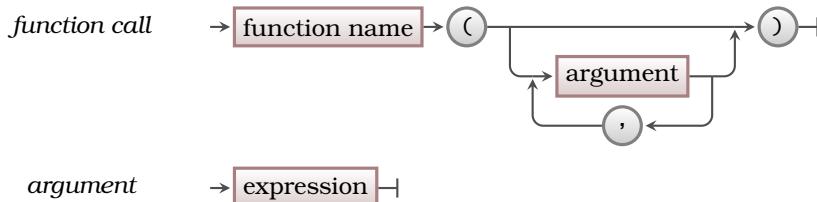


Figure 2.38: Pascal Syntax for a function call

Pascal

```
program TestFnCalls;

function Square(x: Integer): Integer;
begin
    result := x * x;
end;

function Sum(a, b: Integer): Integer;
begin
    result := a + b;
end;

procedure Main();
var
    answer: Integer;
begin
    answer := sum(square(5), square(4));
    WriteLn('5 squared + 4 squared is ', answer);

    WriteLn('(1 + 2) + (3 + 4) = ', Sum(Sum(1, 2), Sum(3, 4)));
    WriteLn('2 squared, squared = ', Square(Square(2)));
end;

begin
    Main();
end.
```



Listing 2.26: Example function calls

Note

- A function call can be used to calculate a value within an expression.
- A Pascal function call is similar to a [Procedure Call](#).
- You use the name of the **Function**, its identifier, to indicate which function is called.
- Following the function's name is the list of *arguments*, these are the values (or variables) that are being passed to the called function.
- In Listing 2.26 the values of the inner function calls are passed to the arguments of the outer calls. This means that `Square(5)` is calculated first then `Square(4)`. The results of these two function calls are then passed as the *arguments* into the call to the `Sum` function. In this case the function call will be `Sum(25, 16)` with 25 being the result returned by `Square(5)` and 16 being the result returned by `Square(4)`.



2.5 Understanding Data

This Chapter has introduced the idea of creating Variables within your program's code to store values. These variables can be stored in Global or Local Variables, and passed to Functions or Procedures using Parameters.

This Section will help you answer the following questions:

- How do local and global variables work?
- What happens with Parameters when Functions and Procedures are called?
- How do Function's return a value?
- How does Pass by Reference work?
- How can I demonstrate or check the execution of a program with Variables?

2.5.1 Variables

To get started let us have a look at the basic concept of Variables. Rather than working through the Change Calculator program we can use a simpler program to start with. Understanding this will then help you understand what is happening in the Change Calculator. The program we will use is in Listing 2.27. This program will contain two variables that are local to the main function in C or the Main procedure in Pascal: val and range.

Pseudocode

```
Program: Test Assignment
-----
Variables:
* val (Integer)
* range (Double)
Steps:
1: Assign to val, 23
2: Assign to range, 1 + val / 100
3: Output val and range to the Terminal
```

Listing 2.27: Pseudocode illustrating variable use



Visualising Variables in the Computer

To get started understanding this let us return to the Conceptual Computer we have been using to explain how these concepts work.

- Test Assignment's Program Launches, putting Main on the Stack.
- The assignment statement stores a value in the val Variable.
- The second assignment statements stores a value in the range Variable.
- The values of the Variables are output to the Terminal.

Test Assignment's Program Launches, putting Main on the Stack.

When the program starts space is allocated on the Stack for the program's Main Function or Procedure. When a Function or Procedure has **Local Variables**, these are also allocated onto the Stack.

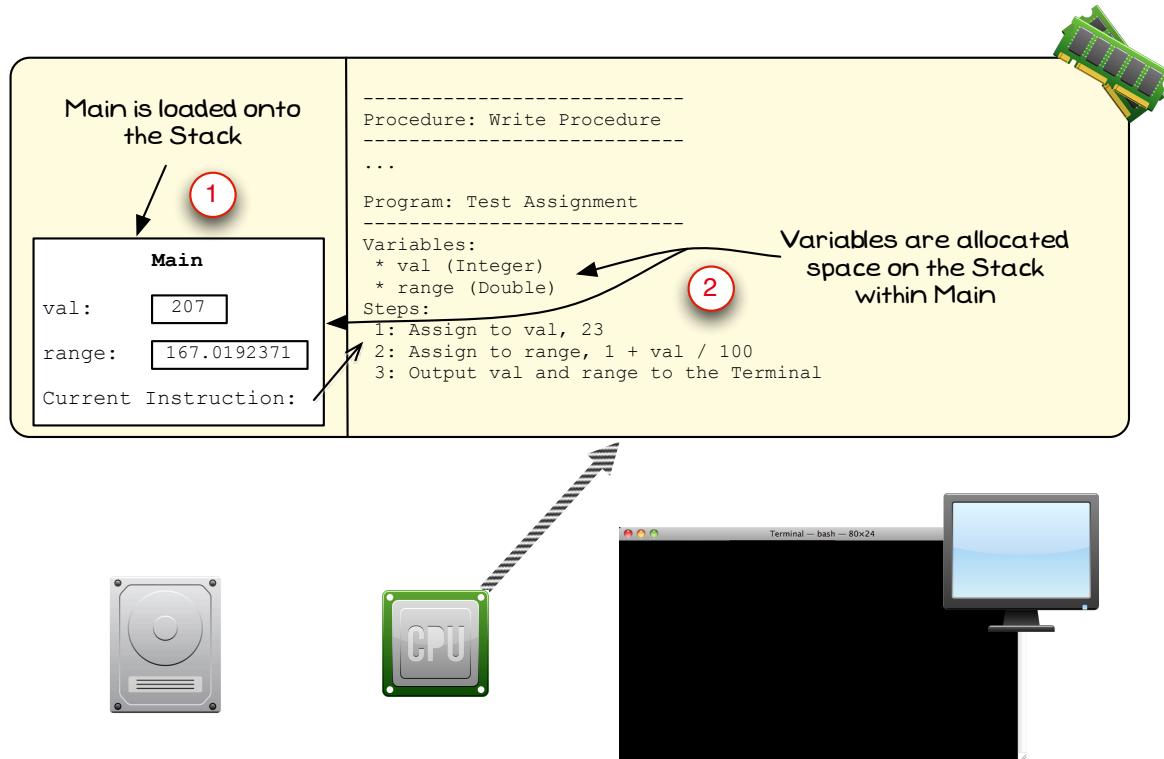


Figure 2.39: Program's entry point is loaded onto the Stack

Note

- The program's instructions are loaded into memory when the program is executed.
- In Figure 2.39 the indicated areas show the following:
 1. When the program starts, Main is loaded onto the Stack. This keeps track of all of the details related to the execution of Main.
 2. The Variables that are declared in Main are allocated space on the Stack.
- The two aspects of the Variable are shown in the illustration. The **variable** is the box drawn next to its name. The **value** is then written within the Variable.
- Notice that the variables each have a value to start with, even though the code has not yet assigned a value. The variable is allocated memory into which its value will be stored. **Memory is not automatically cleared** after its use, so the variables will get whatever value happened to be at that location previously.
- Each Variable is allocated enough space to stores its value. The val Variable needs 32 bits to store an Integer value, whereas the range Variable needs 64 bits to store its Double value.

The assignment statement stores a value in the val Variable.

The first action of this program is to store the value 23 in the val Variable.

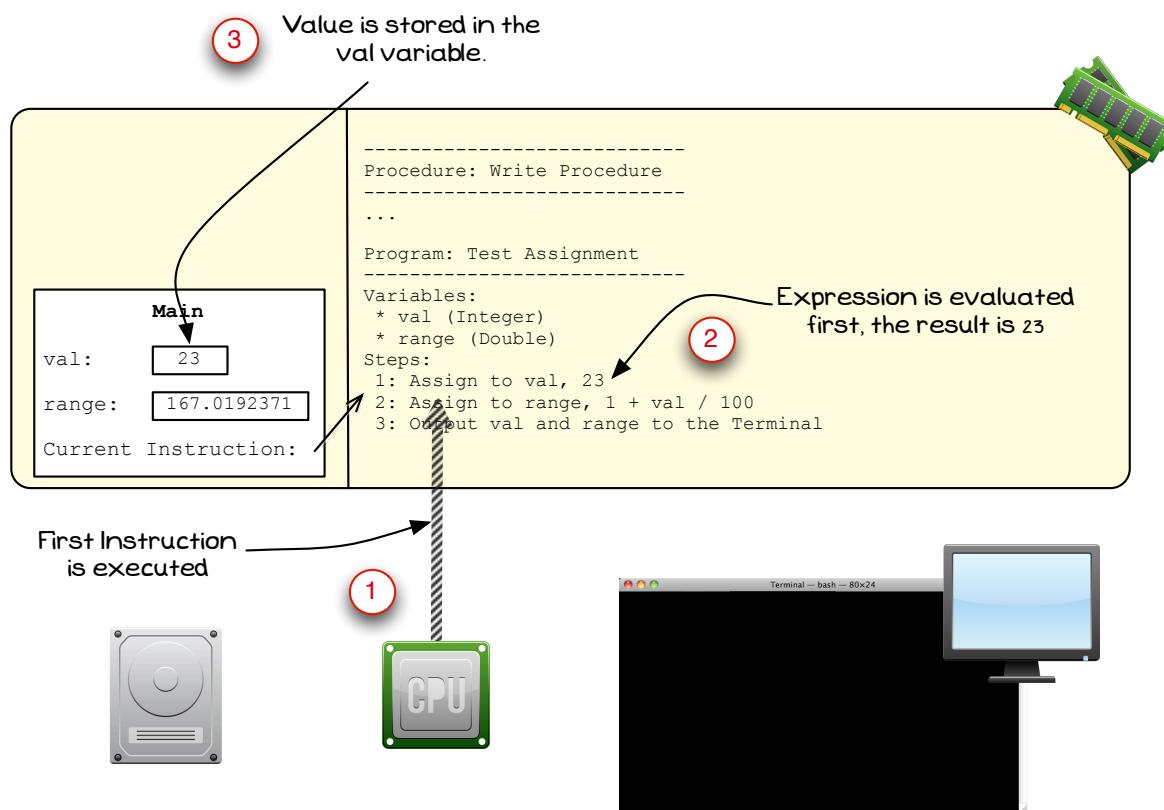


Figure 2.40: The Assignment Statement assigns a value to the val Variable

Note

- In Figure 2.40 the indicated areas show the following:
 - The **Assignment Statement** is executed. This code has two sides. The right hand side executes first and needs to evaluate the Expression. Next the left hand side is used to determine where the resulting value needs to be stored.
 - The value of the Expression is the Literal value 23.
 - 23 is stored in the val Variable.
- Once this line of the program has executed the val variable now has the value 23.

The second assignment statements stores a value in the range Variable.

The next step in the program is to store the value $1 + \text{val} / 100$ in the range Variable. This must read the value from the val Variable to determine the value to be stored.

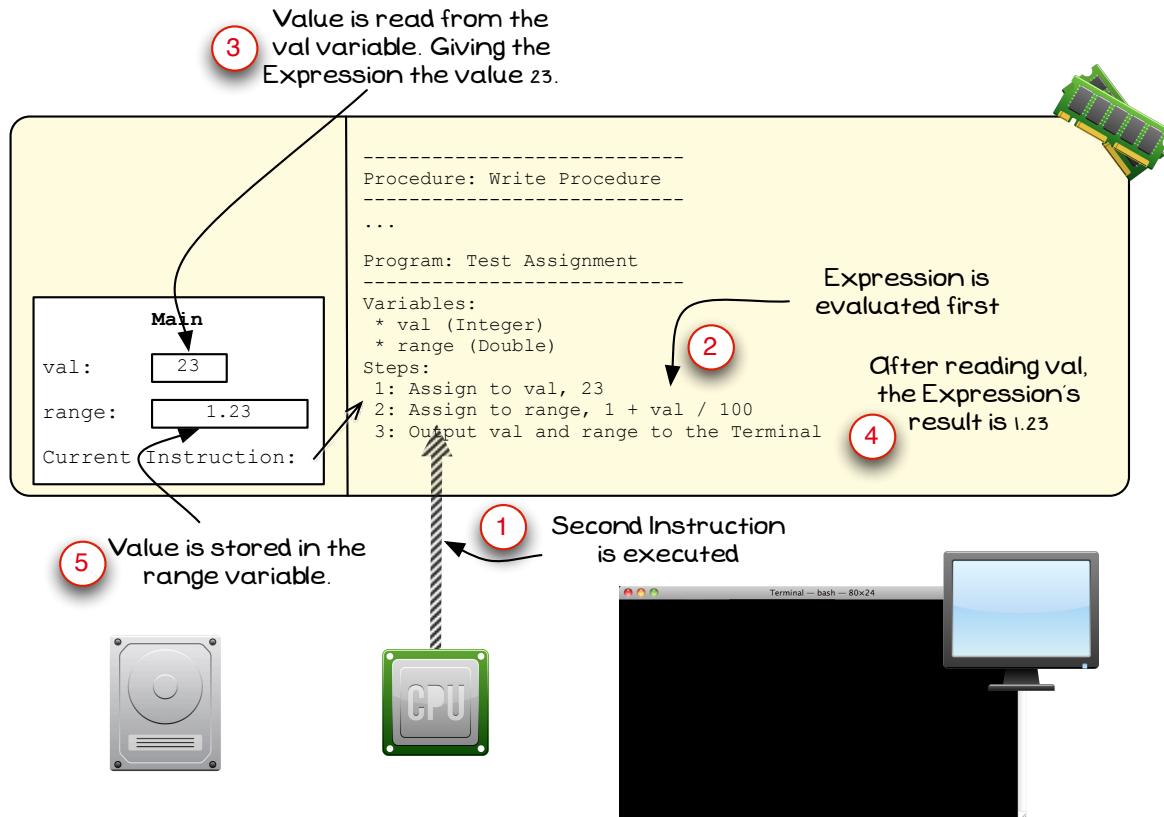


Figure 2.41: The Assignment Statement assigns a value to the `val` Variable

Note

- In Figure 2.41 the indicated areas show the following:
 - The **Assignment Statement** is executed. As before, this needs to evaluate the Expression on the right hand side and store the result in the Variable on the left hand side.
 - The first step to performing the assignment is to evaluate the expression. This requires the value from the `val` Variable.
 - Within the Expression the value of `val` is read. At this point in the program `val` has the value 23.
 - The Expression is now $1 + 23 / 100$, giving the result 1.23.
 - The calculated value is then stored in the `range` Variable.

The values of the Variables are output to the Terminal.

The third, and final, instruction in the program's main function or procedure is to output the values of each of these variables to the Terminal. This uses the values from within the Variables to determine what is output.

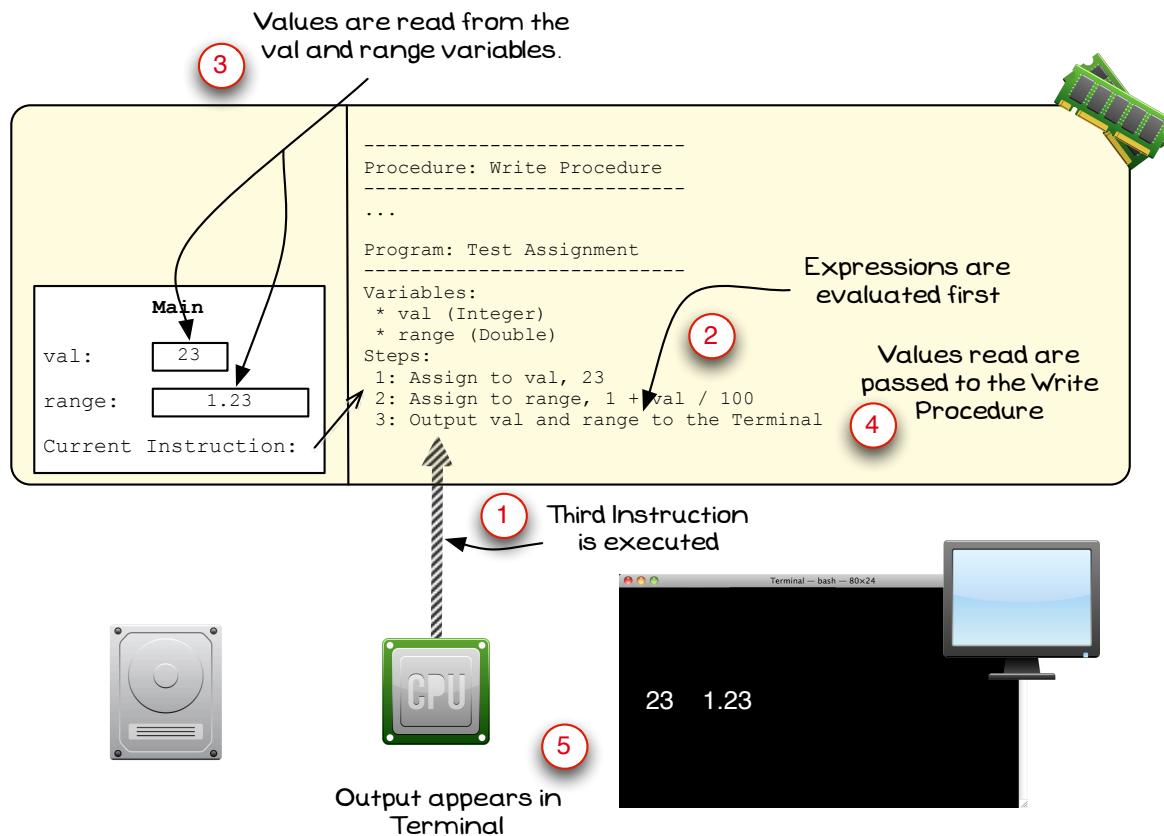


Figure 2.42: The Assignment Statement assigns a value to the val Variable

Note

- In Figure 2.42 the indicated areas show the following:
 - The final instruction is a call to the output procedure of the language. This must be passed the data to output.
 - The Write Procedure is passed a number of values to output.
 - The values of the Variables will be read to get the argument values to pass.
 - The values read from the Variables are then passed to the Write Procedure (`printf` in C, or `WriteLn` in Pascal).
 - When the Write Procedure finishes its task the values will have been written to the Terminal.

2.5.2 Parameters, Locals and Globals

There are three locations at which you can declare variables. *Global Variables* are declared within the program, *Local Variables* are declared within Functions and Procedures, and *Parameters* are used to pass values into Functions and Procedures.

Pseudocode

```
Program: Test Procedures
-----
Global Variable:
* evil (Integer)

Procedure: Test Params
-----
Parameters:
1: val (Integer)
Variables:
*: local (Integer)
Steps:
1: Assign local, val - evil
2: Output val, local, and evil
3: Decrease val by 1
4: Decrease local by 1
5: Decrease evil by 1

Variables:
* good (Integer)
Steps:
1: Assign to evil, 1
2: Assign to good, 1
3: Call Test Params ( good )
4: Output good and evil to the Terminal
```



Listing 2.28: Pseudocode used to examine how Global Variables, Local Variables, and Parameters work

Listing 2.28 is the code that we will use to demonstrate how these Variables work when code is executed by the Computer. This program does not have any meaningful purpose beyond interacting with Global Variables, Local Variables, and Parameters, so do not read too much into the actions being performed. What is important is to examine how the different variables work when the code is executed.

This Section will examine these Variables by looking at the following:

- Memory Partitions include space for Global Variables
- Program is loaded into memory
- Initial Commands Occur
- Passing Parameters in a Procedure Call
- Test Params Commands are Executed
- Control returns to Main when Test Params ends

Memory Partitions include space for Global Variables

When the program is executed the Operating System allocates it space in memory. This space is partitioned into different areas, with each area storing an aspect of the program's data or instructions.

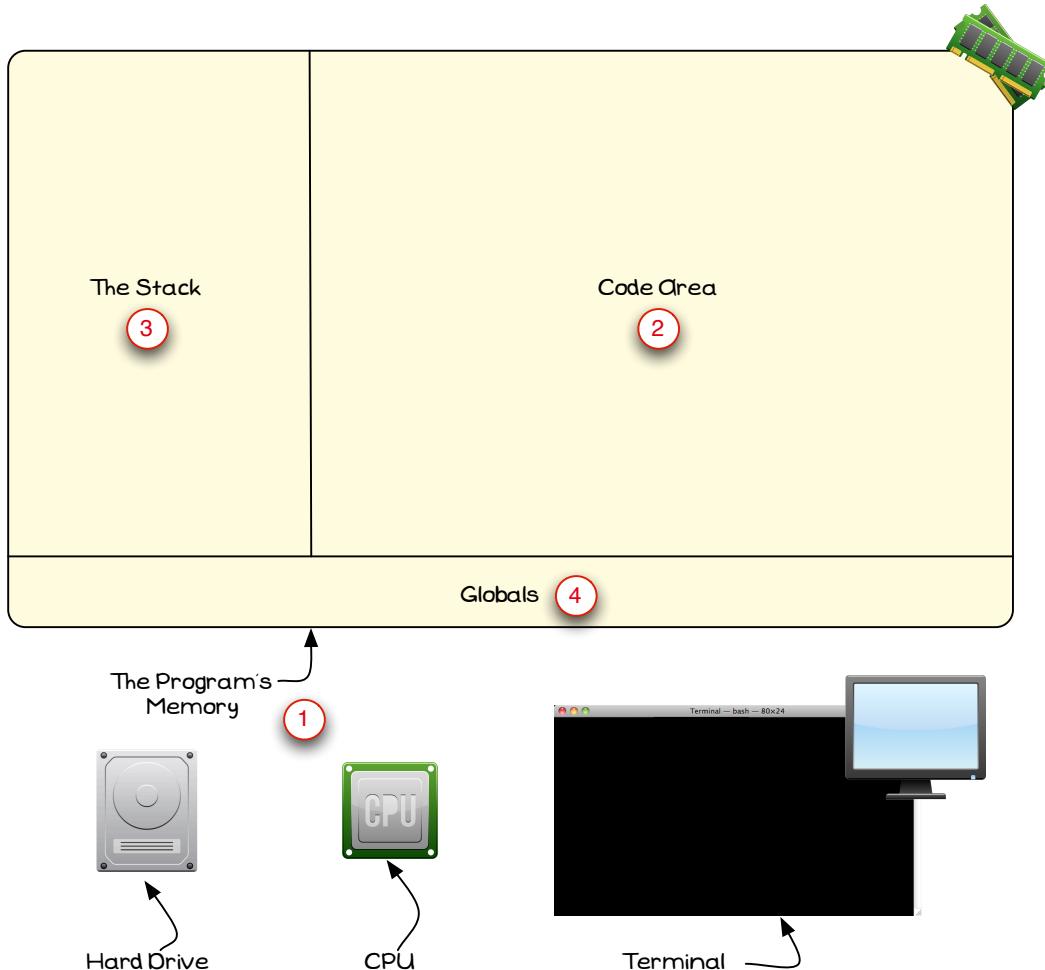


Figure 2.43: Memory is partitioned into spaces for the Code, Stack, and Global Variables

Note

- In Figure 2.43 the indicated areas show the following:
 1. The program's memory is allocated by the Operation System, and divided into areas to store different kinds of values.
 2. The program's instructions are loaded into the **Code Area**.
 3. The **Stack** is used to manage the execution of Functions and Procedures.
 4. **Global** Variables are stored in their own area of memory.



Program is loaded into memory

When the program is executed its global variables are allocated space, and then the main code is executed.

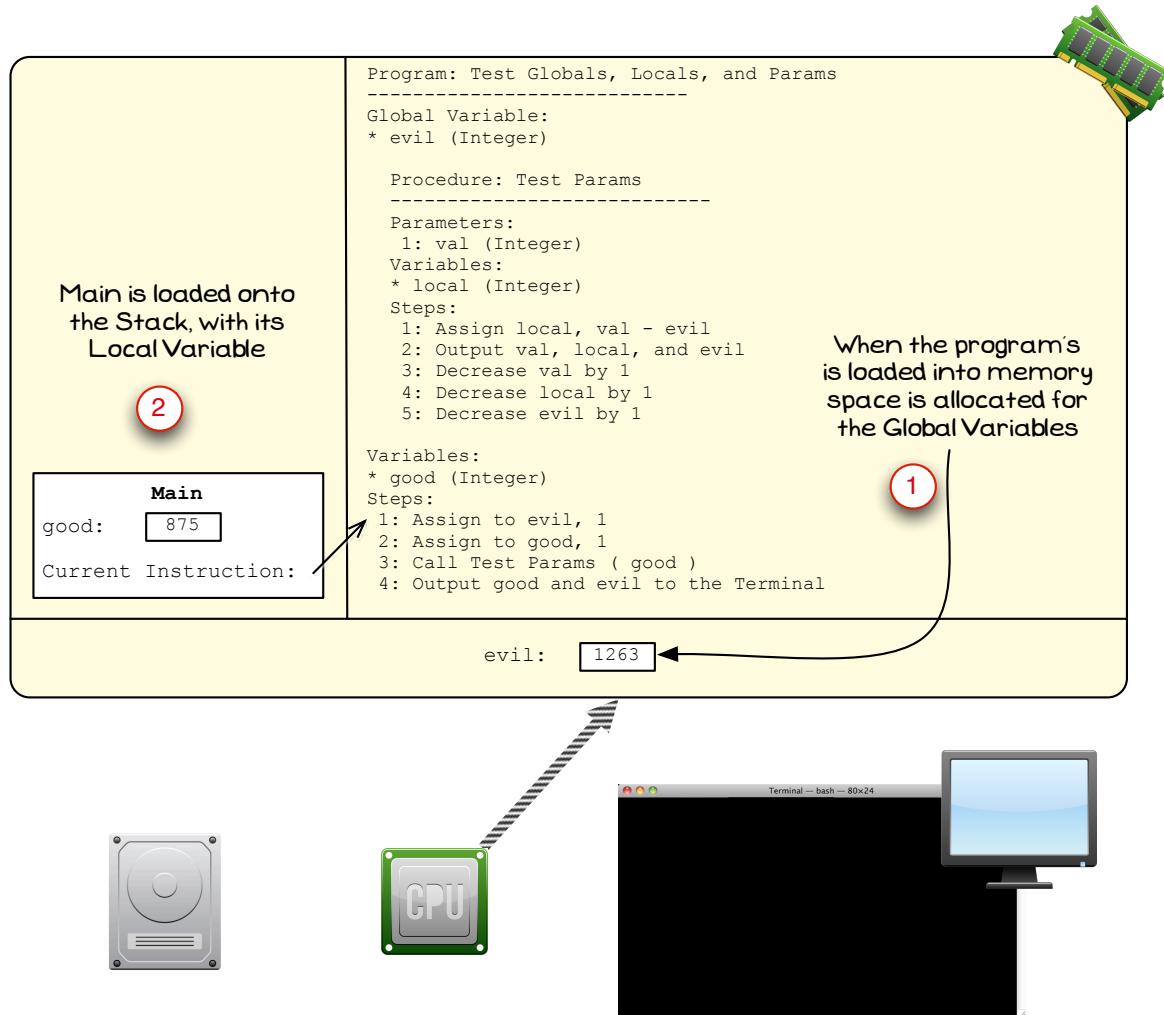


Figure 2.44: The Global Variables are allocated space, then the Entry Point is loaded onto the Stack

Note

- In Figure 2.44 the indicated areas show the following:
 - The global variables are allocated space when the program is loaded. Some Operating Systems ensure that this memory is cleared by zeroing each value, but others do not so you cannot rely upon the value these variables will have when they are loaded unless you initialise it yourself with an Assignment Statement.
 - The main code is then loaded onto the Stack, with space for its local variable.

Initial Commands Occur

The program executes each of the statements, one at a time. Figure 2.45 shows the values in memory by the time the Computer has completed the execution of the first two commands.

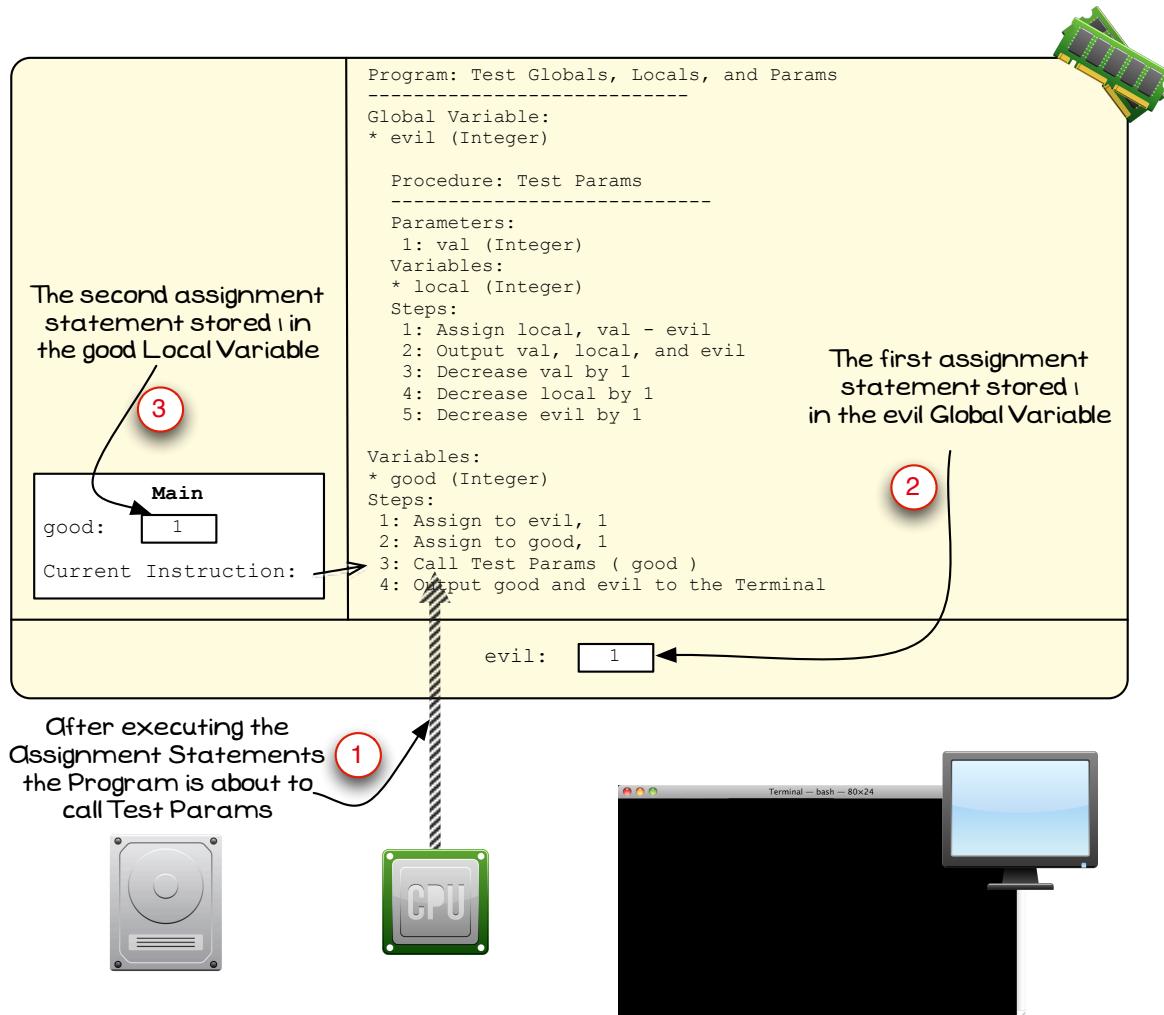


Figure 2.45: The instructions store values in the different variables

Note

- In Figure 2.45 the indicated areas show the following:
 - This picture is showing the computer at the point just before it executes the call to Test Params.
 - The first assignment statement assigned the value 1 to the evil Global Variable.
 - The second assignment statement assigned the value 1 to the good Local Variable.

Passing Parameters in a Procedure Call

At this point the Computer is executing the call to `Test Params`. This involves passing a value to the `val` parameter.

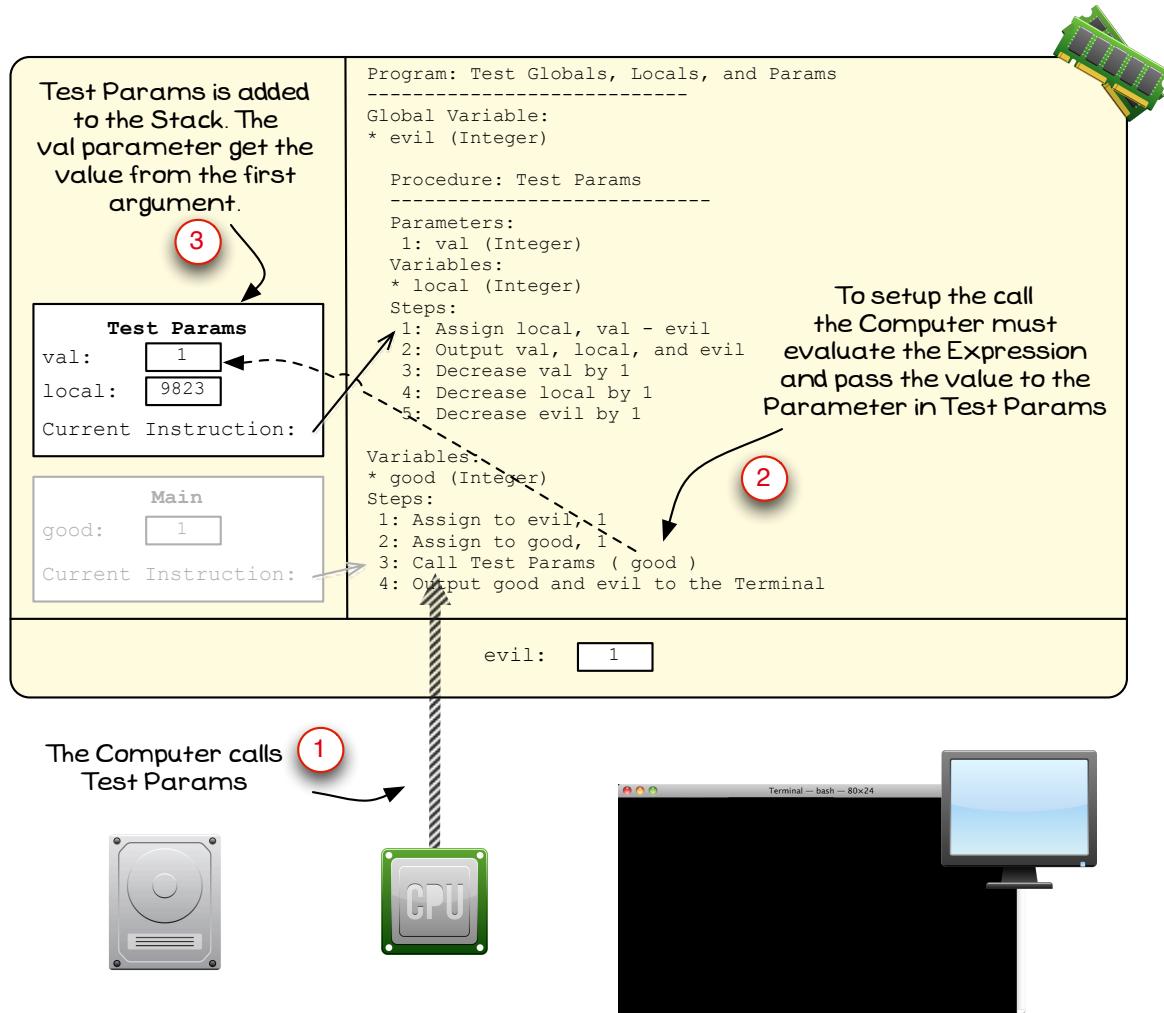


Figure 2.46: The *value* of the Argument is passed to the Parameter when the Procedure is called

Note

- In Figure 2.46 the indicated areas show the following:
 - The Computer calls the `Test Params` procedure. This must be passed a single value that will be copied into the `val` Parameter.
 - The first part of the call requires the Computer to evaluate the Expression being passed to the Procedure. In this case that involves reading the `good` Local Variable and passing its *value* to the Parameter.
 - Once the values for the Parameters are determined, the next step is to allocate space for `Test Params` on the Stack. Within `Test Params` there will be space allocated for the `val` parameter, as well as space allocated for the `local` Local Variable. The value of the *argument* is copied into the *parameter*, and the called Function or Procedure's code can now execute. In this case the value 1 is stored in `val`.

Test Params Commands are Executed

The commands in Test Params are executed one at a time. These instructions will interact with the Variables that are visible to the Test Params Procedure, which include its Local Variables as well as the Global Variables.

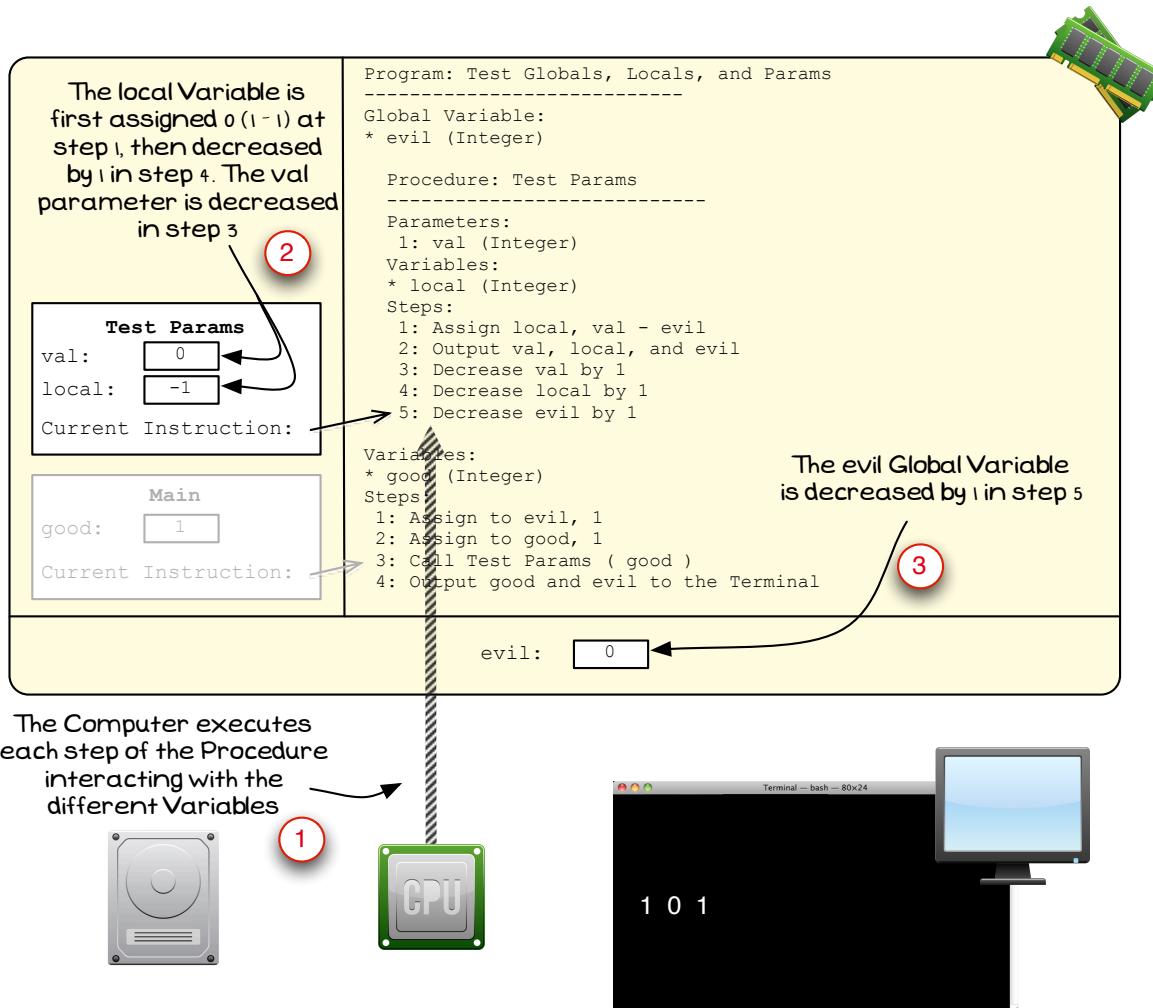


Figure 2.47: Steps in Test Params execute, altering values in the Variables

Note

- In Figure 2.47 the indicated areas show the following:
 - Each instruction is executed one at a time. This picture shows the instructions at the end of the last instruction in Test Params before it ends.
 - The steps of the Procedure are able to read and write values to the Function or Procedure's Parameters and Local Variables, as shown here.
 - The Procedure is also able to read and write values to the Global Variables.
- Notice that Test Params cannot see the good Variable. It exists within the Main code and cannot be accessed from other Functions or Procedures.
- You can decrease the value of a Variable using an Assignment Statement. For example 'Assign to val, the value val - 1'. This will read the current value from the val Variable, subtract one from that, and then store the result back in the val variable.

Control returns to Main when Test Params ends

When Test Params ends control returns to the program's main code. The space allocated to Test Params is now released, including the space allocated to the val Parameter and the local Variable.

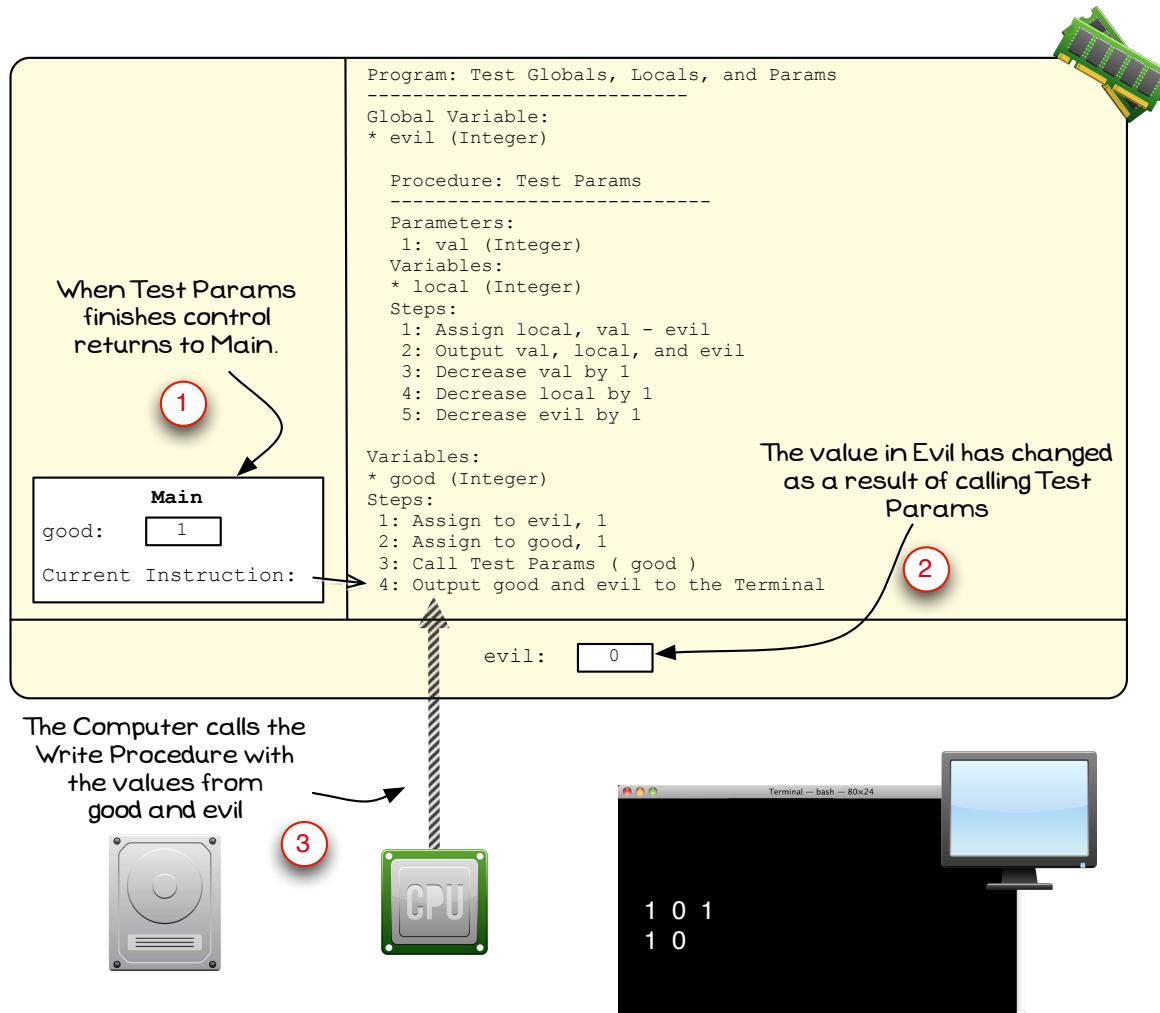


Figure 2.48: Control returns to Main when Test Params ends

Note

- In Figure 2.48 the indicated areas show the following:
 - Control returns to Main when Test Params ends.
 - Notice that the value of the evil Global Variable has changed as a side effect of running Test Params. This is easy to follow in a small program, but can cause difficulties when you start building larger programs. As a result, Global Variables should be avoided.
 - The last instruction in the main code outputs the values of good and evil to the Terminal.
- The value within good has been passed to the val Parameter in Test Params. This copies the *value*, not the Variable, as it is using the standard *Pass by Value* mechanism. This means the code in Test Params has no way of affecting the value in the good Variable.

2.5.3 Function Return Values

The next kind of data is the result returned by a call to a Function. A Function is just like a Procedure, except that it calculates a value and is therefore called from within an Expression. To examine how this works within the Computer we will see how the code in Listing 2.29 executes. This code declares a Square function that calculates the square of a number (val passed in as a Parameter).

Pseudocode

```
Program: Test Functions
-----
Function: Square
-----
Returns: An Integer - val squared
Parameters:
  1: val (Integer)
Steps:
  1: Returns the result, val * val

Variable:
  *: answer (Integer)
Steps:
  1: Assign to answer, Square(5)
  2: Output answer to the Terminal
```

Listing 2.29: Function example, to illustrate how data is returned from a Function



This Section will examine how Functions return values by looking at the following:

- Test Functions is loaded into memory
- Square Function is Called
- Square's result is calculated
- Returned value is used

Test Functions is loaded into memory

When the program is executes the Operating System loads its code into memory, and starts the main code running on the Stack.

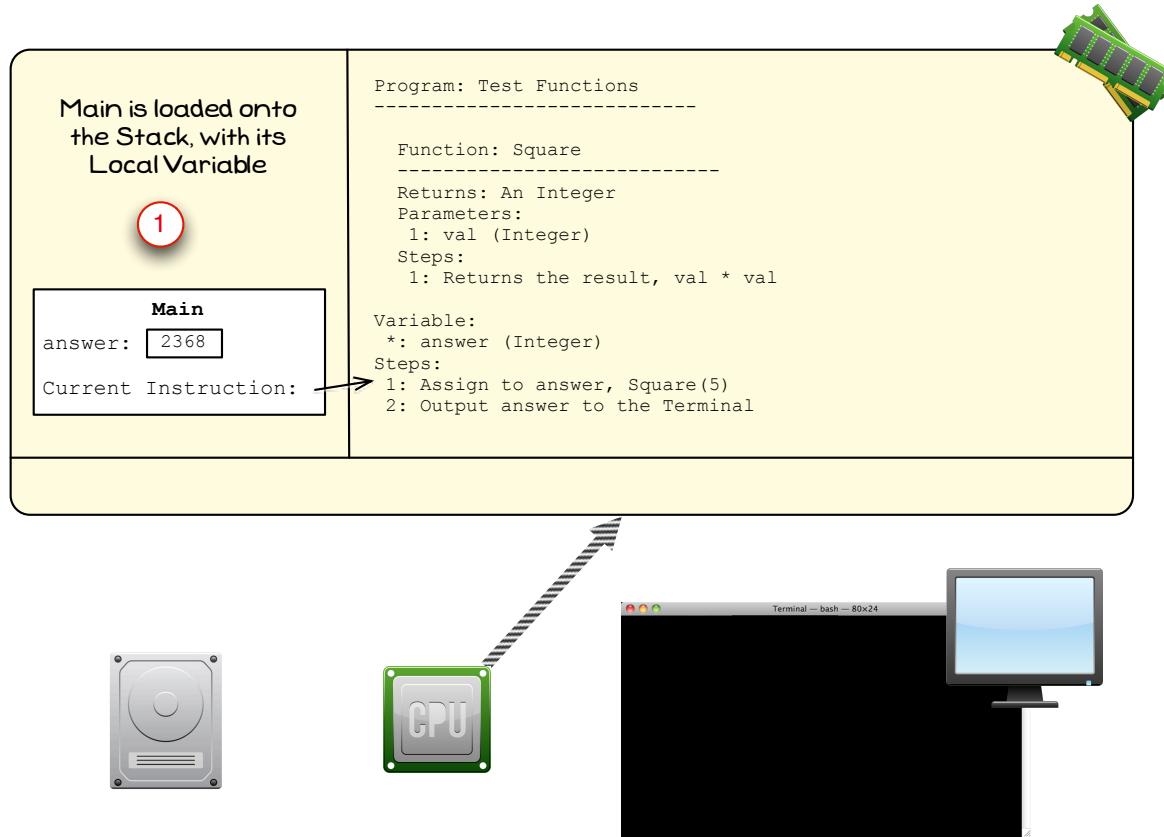


Figure 2.49: Code is loaded into Memory and Main is loaded onto the Stack

Note

- In Figure 2.49 the indicated areas show the following:
 1. Main is loaded onto the Stack, with space for its answer local variable.
- Notice the Global Variables section is empty as this program does not use Global Variables.

Square Function is Called

The Square function is called in the same way that a Procedure would be called. It is given space on the Stack, and its instructions are run.

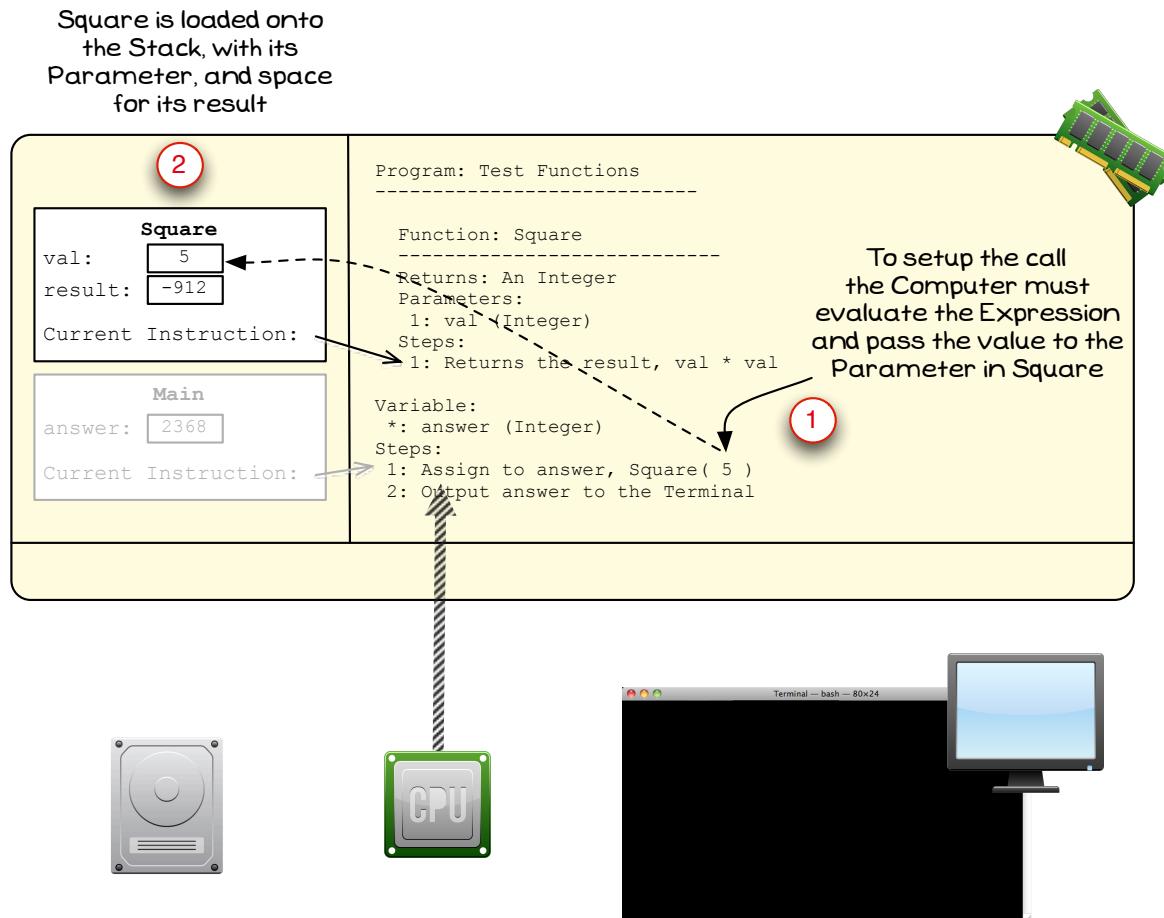


Figure 2.50: Square is called, with the value 5 passed to its val Parameter

Note

- In Figure 2.50 the indicated areas show the following:
 - When the main code calls Square it passed in the value 5, calculated simply by reading the Literal value.
 - Square is added to the top of the Stack, and its val parameter is assigned the value passed in from the Function Call. In addition to any Parameter and Local Variables the Function also has space for its result. This will be the value returned to the caller when the Function ends.
- The value of the result will initially be whatever happens to be at that location in memory the last time it was used. You must make sure you assign a value to this within the Function.

C++

In C the **Return Statement** assigns the value returned by the Function. There is no special result Variable that you can interact with. Conceptually, however, the value must be stored in order to be returned by the Function. In C the compiler takes care of these details for you.

Square's result is calculated

Within the Square function its result is calculated and returned.

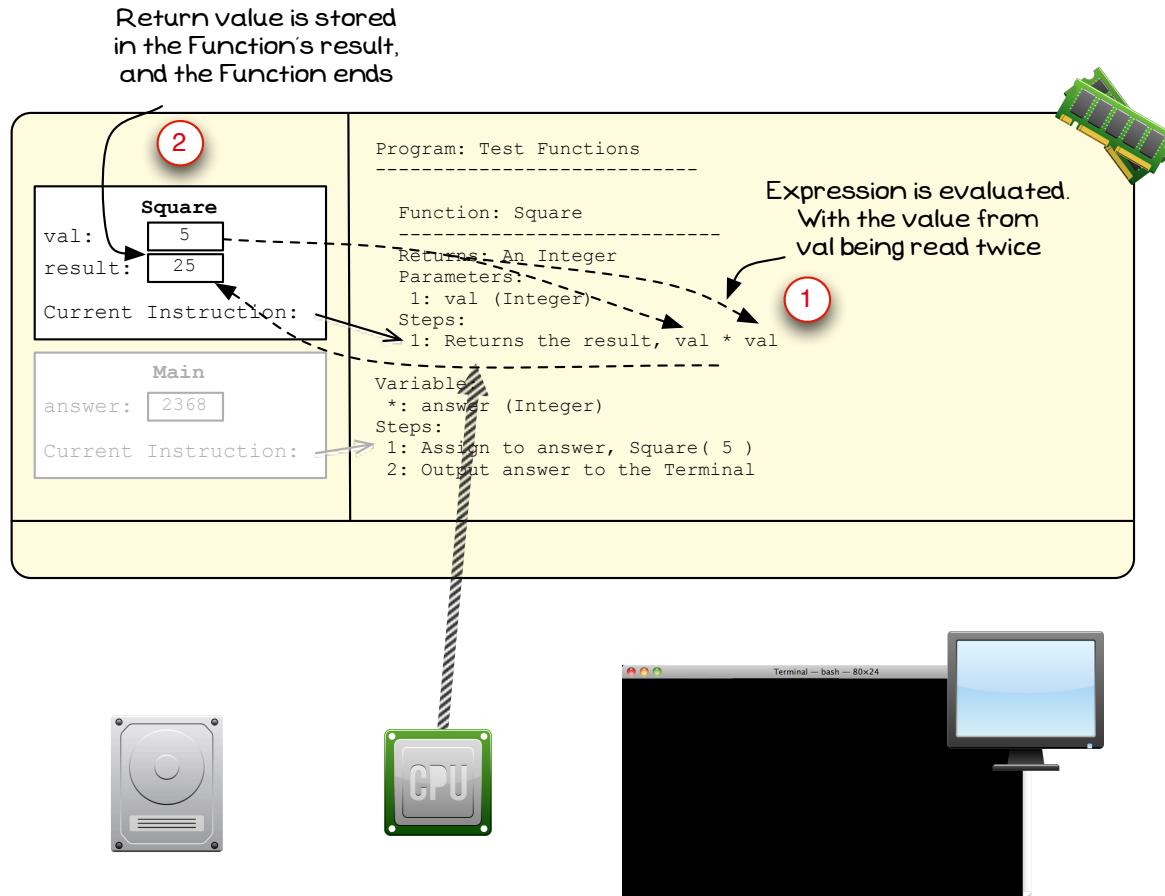


Figure 2.51: Square prepares the value that will be returned to the caller

Note

- In Figure 2.51 the indicated areas show the following:
 - The expression 'val * val' is evaluated. This reads the value of the val variable twice and multiplies the two values read. In this case this will calculate '5 * 5', giving 25.
 - The result of the expression is returned, setting the Function's result.

C++

In C this is accomplished with the [Return Statement](#). It assigns the result for the function and ends the Function at the same time.

Pascal

In Pascal you directly store the value in the special result Variable. This will not end the Function directly, but does store the value to be returned by the Function.

Returned value is used

When the Function ends, control is returned to the main code. This called the Function in order to evaluate an Expression. That expression will now continue to be evaluated using the value returned by the Function.

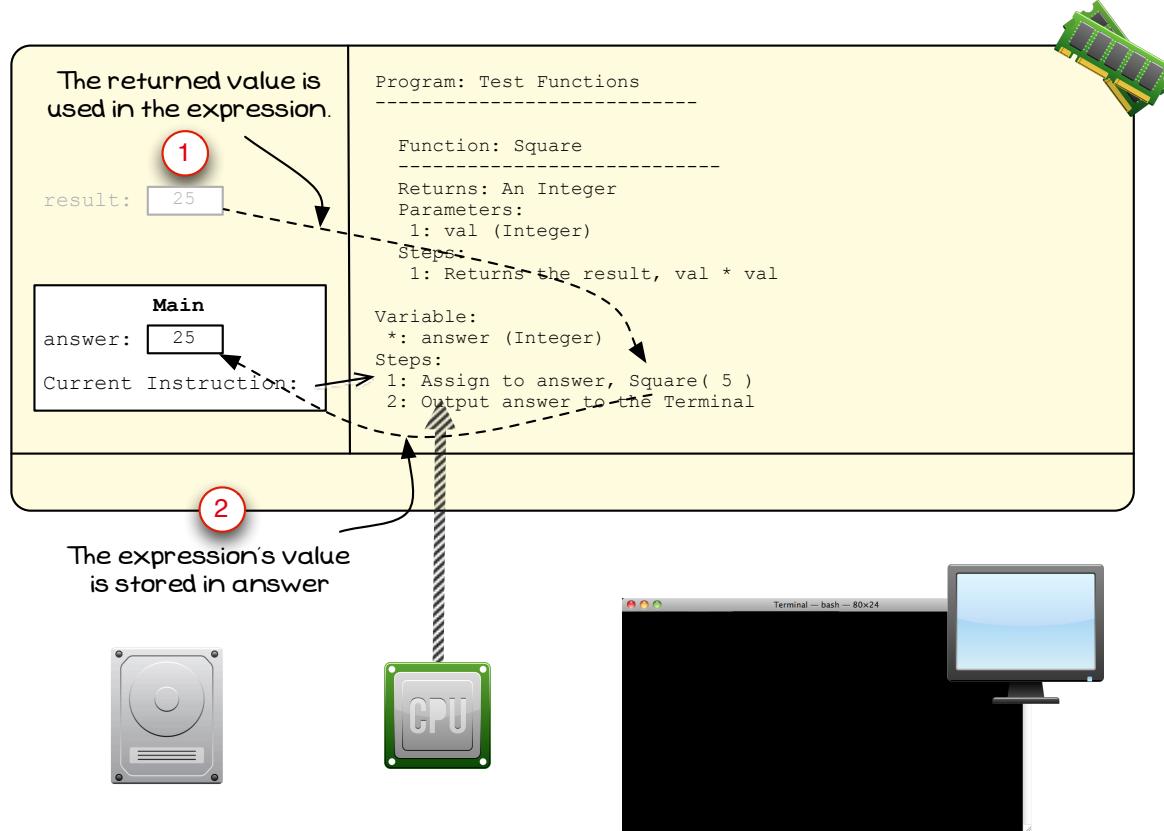


Figure 2.52: The result of calling `Square` is used within the Expression, and the value is stored in `answer`

Note

- In Figure 2.52 the indicated areas show the following:
 - The result of the Function is made available to the caller. This will be read as part of the Function Call, and is then used within the Expression.
 - In this case the Expression does not alter the value returned by the Function, and the result is stored in the `answer` variable.
- The expression that contains the Function Call can operate on the result returned from the function in the same way that it uses any other value. For example, the following expression is valid: $3 + \text{Square}(-7) * 5 + \text{Square}(\text{Square}(2))$.
- The key is to think of each Function Call as a value. So nested Function Calls like `Square(Square(2))` can be broken down into their individual values. In this example `Square(2)` is evaluated first, and returns the value 4. This is then passed to the outer Function Call giving `Square(4)` and a final result of 16.

2.5.4 Pass by Reference

Working with data involves working with Variables of one form or another. When thinking about Variables it is important to realise the two aspects of each Variable: the *Variable* and its *value*. The *Variable* is a location in memory, it is a place at which you can store a value. The *value* within the Variable is a separate concern. This value can be read from the Variable in Expressions.

Pseudocode

```
Program: Test Input
-----
Variables:
* age (Integer)
Steps:
1: Output 'Enter your age: ' to the Terminal
2: Read input into age
3: Output 'You entered ' and age to the Terminal
```

Listing 2.30: Example used to illustrate how Pass by Reference works with the standard input Functions and Procedures.



This Section will examine how Pass by Reference works by looking at the following:

- Main starts and a prompt is written for the user
- The Input Procedure is called to read a value into age
- The Input Procedure stores the value read from the user into age
- Control returns to Main and age has been updated with the value from the user

C++

In C the Read Input Procedure is the `scanf()` Function. This requires that you pass in a format string that indicates what will be read as well as passing in the Variables into which these values will be stored. You can read about this in [C Terminal Input](#). Step 2 from Listing 2.30 can be implemented in C using `scanf("%d", &age);`.



Main starts and a prompt is written for the user

The Program starts as normal with the Operating System allocating space for the Stack, Code, and Global Variables. Main is then loaded onto the Stack and its instructions are executed.

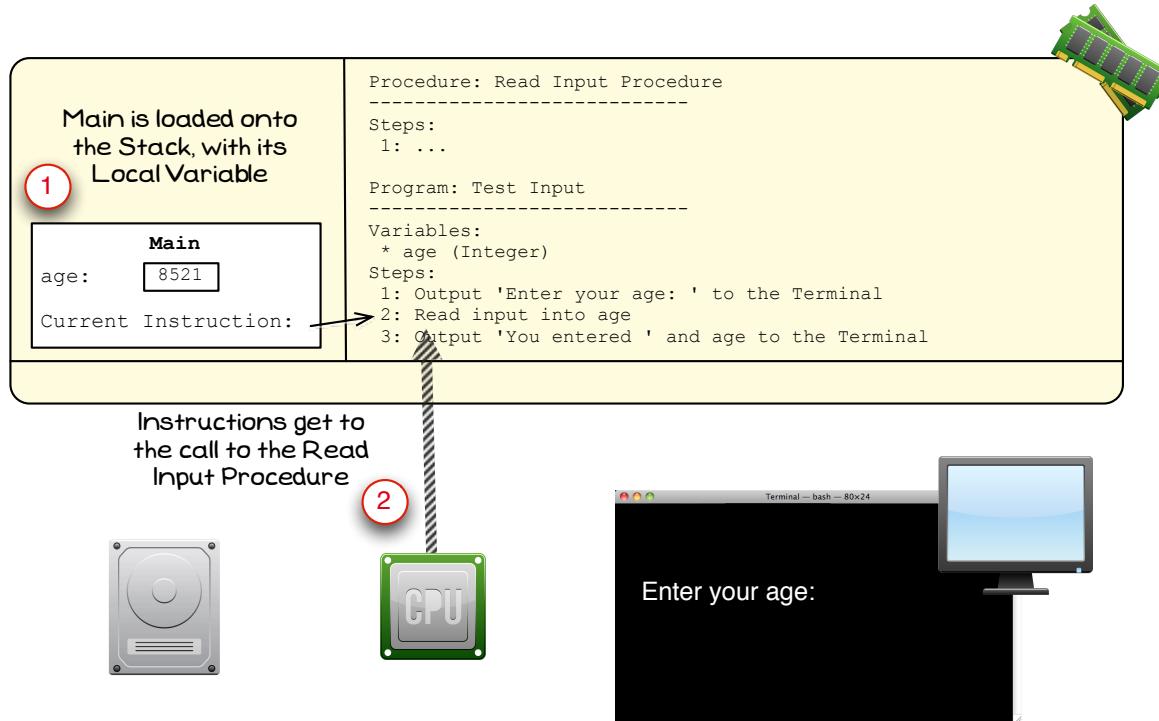


Figure 2.53: Code is loaded into Memory, Main is started, and the first instruction outputs a prompt

Note

- In Figure 2.53 the indicated areas show the following:
 - Main is loaded onto the stack, with space for the age Variable.
 - The first instruction outputs 'Enter your age: ' to the Terminal.
- When writing a prompt it is a good idea to keep the output on a single line. Just write the prompt without writing a new line.



The Input Procedure is called to read a value into age

The next instruction in the main code is to call the Language's input procedure to read a value from the user and to store that value in the age variable. In order to achieve this you must pass the age variable itself to this procedure. That will enable the input code to store the value it reads into the variable that you give it.

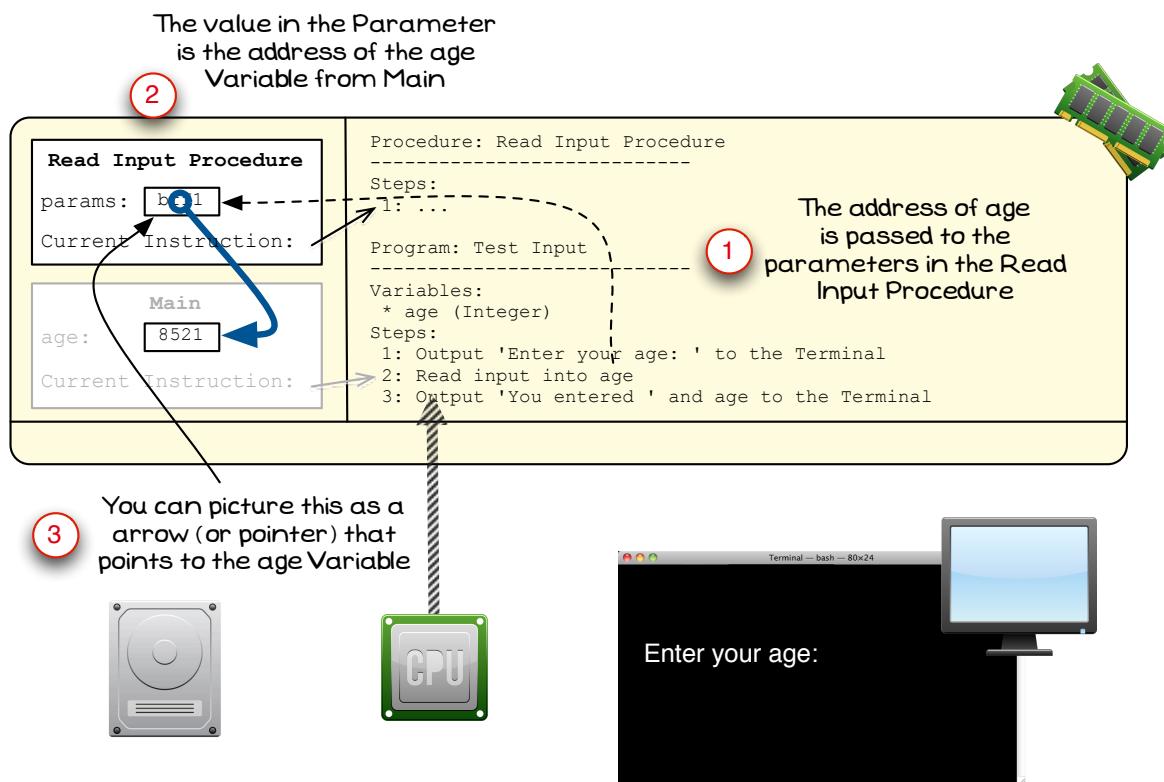


Figure 2.54: The input procedure starts, and the age Variable is passed to the Parameters

Note

- In Figure 2.54 the indicated areas show the following:
 - A variable is a location in memory, so it is not possible to actually move this location into the Parameter. What happens instead is that you pass the **address** of the Variable to the Parameter. This is why it is commonly called *Pass by Reference*, you are passing a reference to the Variable, in effect telling it where the Variable is that you want the data stored in.
 - The value passed to the Parameter is then the **address** of the Variable where the data should be stored. This Parameter is now storing the location of the age Variable.
 - An good way to picture this is as an arrow, or a pointer, that points from value within the Parameter to the age Variable. This helps visualise the fact that this Parameter is storing the location of the age Variable.

C++

In C you must manually get the address of the variable using the ampersand (&) operator. So the call `scanf("%d", &age);` in Main is passing the **address** of age. You can read `&age` as '*The address of age*'.

The Input Procedure stores the value read from the user into age

The language's input procedures are responsible for reading values from the user and storing these into Variables for you. The instructions in this code will do the work to convert the data from the text the user enters into the types required by the Variables the values are being read into.

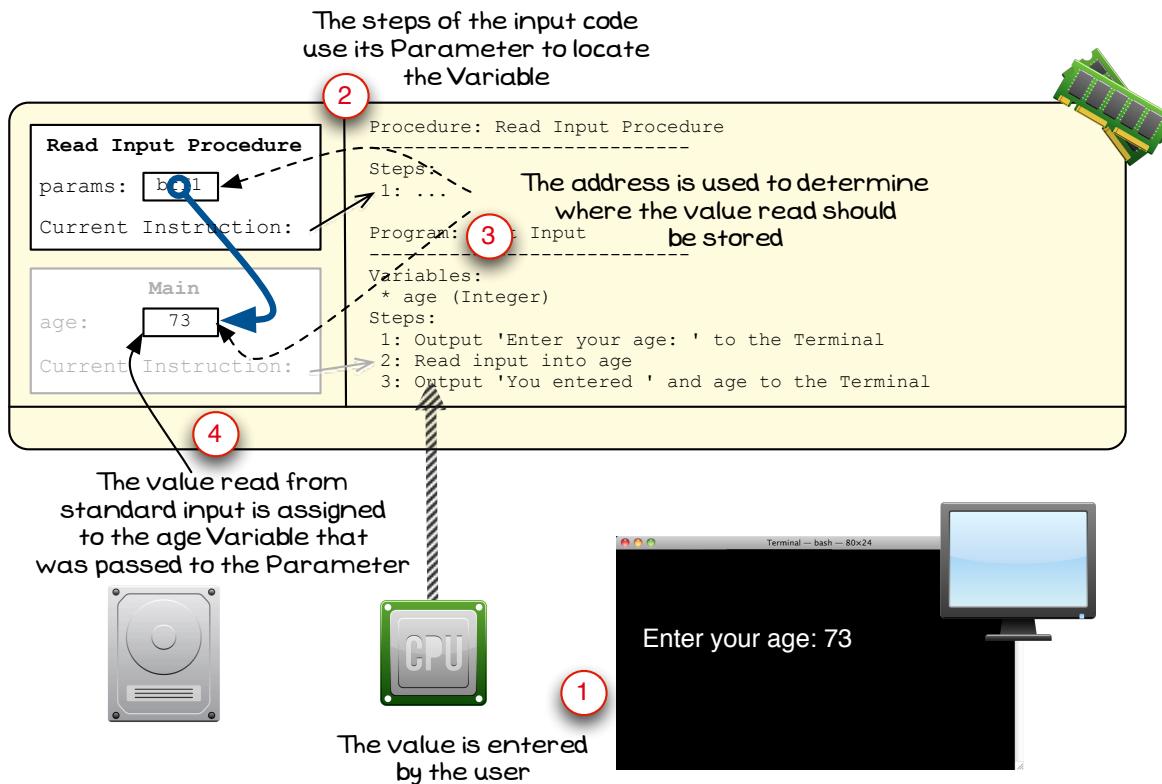


Figure 2.55: Within the input code the address is used to store the value read from the user

Note

- In Figure 2.55 the indicated areas show the following:
 - The input code waits for the user to enter a value at the Terminal. Once the value is entered it is scanned and stored in the Variables passed to the Input Procedure.
 - The input code needs to work out where the values read are to be stored. It does this by looking up the address stored in its Parameter.
 - The address is used to locate the Variable into which the value will be stored.
 - In this case the Parameter has the address of the age variable, and therefore the value read from the user is stored into Main's age Variable.
- The language's Input Procedures can be used to read in more than one value at a time. In these cases the inputs code will read values for each Variable one at a time.

C++

In C you are responsible for determining how the data read is interpreted. This is the purpose of the *format string* passed as the first parameter to `scanf`. In `scanf("%d", &age);` the "%d" indicates that the first Variable needs to be assigned a decimal integer value.

Control returns to Main and age has been updated with the value from the user

When control returns to the main code, the age variable has been updated to include the values read from the user.

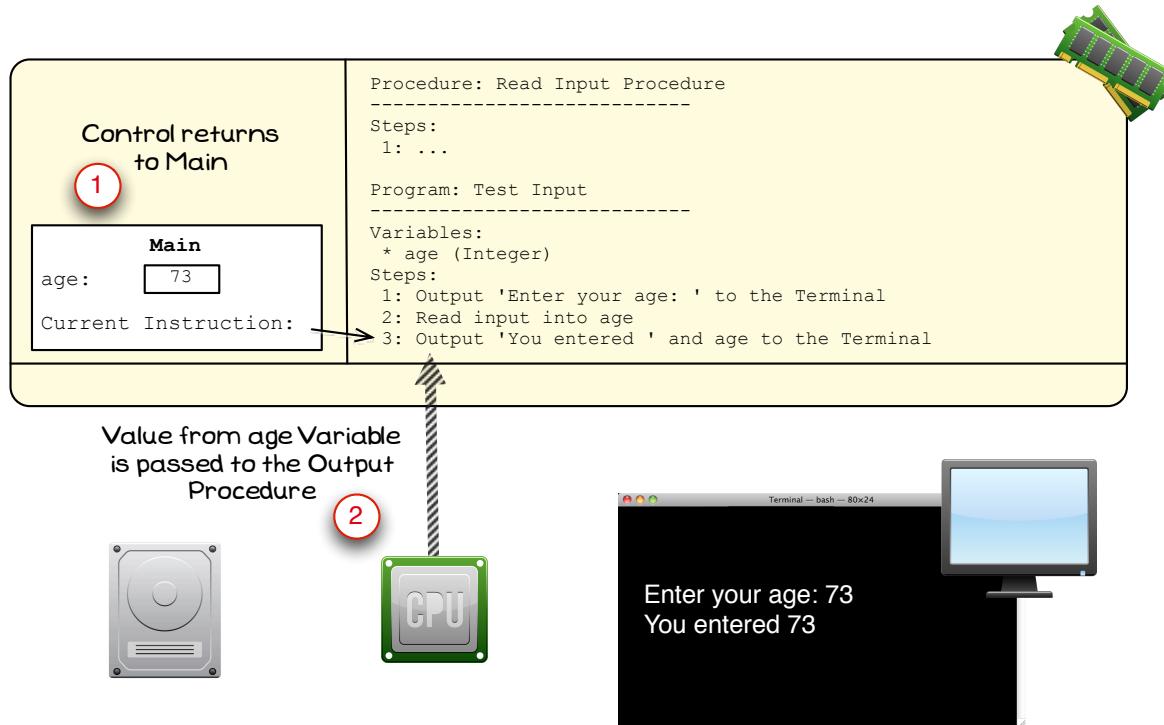


Figure 2.56: Control returns to Main, and the value of age has been updated by the input code

Note

- In Figure 2.56 the indicated areas show the following:
 1. Control returns to Main when the input code completes.
 2. The *value* from the age Variable is then passed to the Output procedure.
- Pass by Reference* is used when you want to pass the Variable, but in most cases you only want to pass the *value* so most code uses *Pass by Value*.

2.5.5 Hand Execution with Variables

One valuable skill you need to develop is the ability to step through a program in the same way the computer will. You will find that this skill will get more and more important as you progress to write larger programs. You will use this skill to help you detect and correct any **semantic errors**, in a process commonly referred to as **debugging**.

Semantic errors are errors in the logic of your program. This means that the program compiles, but does not do what you want it to. The tests that you use will help you locate these errors, but in many cases you will need to work out *why* the error is occurring so that you can make the required changes.

One of the better techniques to help you make these corrections is to reading your code, and to think about the actions the computer is performing. Often these errors only require one or two lines of code to change, but before you can make these changes you need to identify *where* the error is occurring, *why* it is occurring, and *how* you can correct the issue. By reading your code, and hand executing it you have the best chance of working out *where* the problem is, and *why* the problem is occurring.

These activities are very popular with Programming Tests, both in University subjects and job interviews. Being able to execute some code by hand means that you really understand what the code is doing. Once you have this level of understanding you just need to add some imagination and experience to become a very valuable software developer.

The steps you need to perform are:

- Locate the Function or Procedure to Test
- Turn off your brain
- Setup your 'memory'
- Run the steps one at a time

Locate the Function or Procedure to Test

Hand executing an entire program would take a very long time. As a result you want to narrow down your search to one or two Functions and/or Procedures. You can use your Structure Charts, or knowledge of the program's structure, to help you narrow down your options. Think about where the test failed, and where the code for this is located in your Program. With a small program you are likely to know immediately where the issue is, but as the program size grows it will become more difficult to know exactly where the problem is.

Listing 2.31 contains the pseudocode for a Procedure that will calculate and output the area of a Trapezoid. Unfortunately this code contains a number of errors. The following sections will demonstrate how to execute this procedure by hand. If this were a programming test you may be asked a question such as:

'What is the value in area when this program is run and the user enters 5 for base 1, 7 for base 2, and 4 for height?'

Executing this by hand will give you the necessary skills to be able to answer questions such as this.

Pseudocode

```
Procedure: Print Trapezoid Area
-----
Variables:
* base1, base2 (Double)
* height (Double)
* area (Double)
Steps:
1: Output 'Enter base 1: ' to the Terminal
2: Read input into base1
3: Output 'Enter base 2: ' to the Terminal
4: Read input into base1
5: Output 'Enter height: ' to the Terminal
6: Read input into height
7: Assign to area height * ( base1 + base1 / 2 )
8: Output 'Trapezoid area is ', and area
```



Listing 2.31: Pseudocode for a Procedure that calculates the area of a Trapezoid, with errors.

Turn off your brain

Once you have located the Function or Procedure that you need to test you can start to execute it. The main obstacle in most peoples way is, unfortunately, their brain. You are going to *think* about what you *wanted* the program to do. This is something the computer *cannot* do. The computer is unintelligent. It is a machine that does as you command. To run this program as the computer does you will need to **stop thinking** and start doing as commanded.

Rules:

- Do not think about what you *want* or *expect* the program to do.
- Perform the commands as they are, one at a time
- You cannot remember anything that is not written down.
- Focus only on the current command (Statement). Do not look at other Statements!
- Perform the actions you are commanded to perform, and only those actions. Do not do more than you are commanded to.
- Use the language's Statements to determine the actions you must perform:
 - An **Assignment Statement** will evaluate its expression, and assign a value to a variable.
 - A **Procedure Call** runs the code in a Procedure, a **Function Call** runs the code in a Function. To optimise this step you can perform the actions within called Functions or Procedures without stepping through their code in most cases.

Setup your ‘memory’

One of the rules when running your code by hand is that you can not remember anything. The values you use must come from the ‘memory’ that you setup for the Function or Procedure. So, the first step for executing this code performs the actions that called the procedure: you need to *allocate space* on the stack. You can simulate this using a piece of paper. This piece of paper will be the ‘memory’ used by the code as it executes. To set this memory up you will need to do the following:

1. Get a blank piece of paper.
2. Write `Print Trapezoid Area` to remember that is the procedure you are in.
3. Do the following for any Variables⁶ in the Function or Procedure:
 - (a) **Draw a box** to represent the Variable. Make sure that it is large enough for you to write values inside.
 - (b) **Write the name** of the Variable next to the box.
 - (c) If the Variable is a Parameter, copy the value from the argument into the box, otherwise leave it empty.

When you have finished these steps for the code in Listing 2.31 you should have a piece of paper that looks like the image in Figure 2.57. The empty boxes represent the different Variables, the fact we have not written a Value into these boxes tells us that at this point the value has not been set, and therefore should not be used.

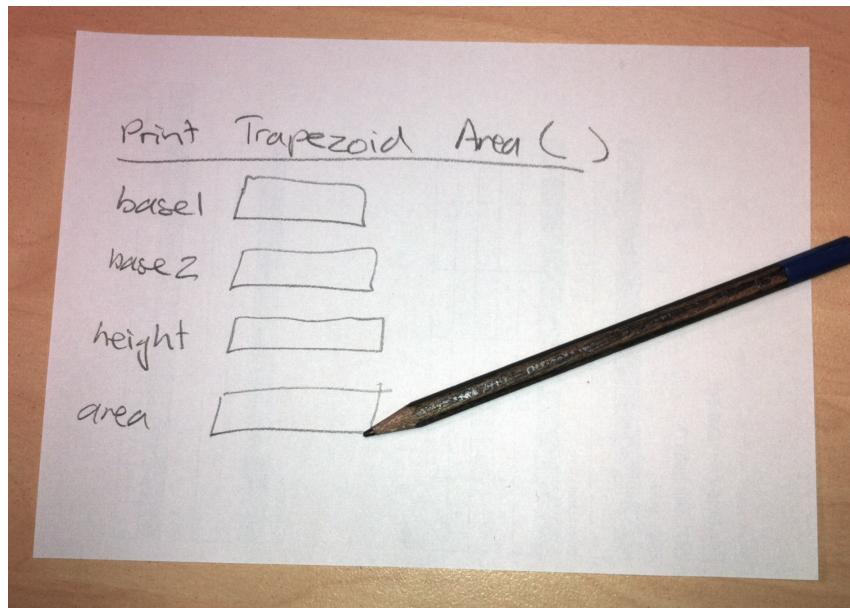


Figure 2.57: Draw space for all of the Function or Procedure’s Variables

⁶Draw boxes for Parameters, Local Variables, and Function results.

Run the steps one at a time

Now that the memory has been initialised it is time to run each instruction in the code. You need to do this **one instruction at a time**. It does not matter how large the code is, the computer will always run it one Statement at a time. Do not think, just read the Statement and perform the action.

1. Step 1 calls the output procedure of the Language. This will output the text '*Enter base 1:*' to the Terminal. You do not need to step into the internal actions of each Function or Procedure called, as long as you can replicate the actions that it would perform. If you had been asked '*What is output by the call to Print Trapezoid Area?*' it would be important to put this in the answer, but as we just want to know the final result you can skip this.
2. Step 2 reads a value from the user and stores it in **base1**. The user is responding to the prompt from step 1, and therefore enters 5, the value indicated in the question. As the **base1** variable is passed to this call you now write **5** into the **base1** box, representing the fact that this Variable now has the value 5. See Figure 2.58.
3. Step 3 outputs the next prompt, '*Enter base 2:*'.
4. Step 4 reads a value from the user, and stores it in **base1**. This is the source of the first error. This is most likely the result of a copy/paste error, where the developer has copied the instructions to prompt the user and read a value into **base1**, and then changed it to read a value into **base2**. When you copy and paste code you need to pay extra attention to making sure that the pasted code is changed correctly.

When this code is run a new value will be written into the **base1** Variable. Cross out the old value, and write in the new value. The value entered in this case is **7**, as indicated in the question. See Figure 2.59.

5. Step 5 outputs the prompt for **height**.
6. Step 6 reads **4** from the user and stores it into the **height** Variable. See Figure 2.60
7. Step 7 now reads the values from **height**, and **base1**. The equation for calculating the area of a Trapezoid is $height(\frac{base1+base2}{2})$, this has been coded incorrectly again, with several errors. Firstly **base1** is read twice, and **base2** is not used at all. Second, using BODMAS the current code is actually performing the equation $height(base1 + \frac{base1}{2})$. Performing this calculation you will find that the value stored in **area** is $4(7 + (7/2)) = 42$.
8. Step 8 outputs the area to the Terminal, displaying '*Trapezoid area is 42*'. See Figure 2.61.

At this point the Procedure has ended, and you have collected all of the details about how the program actually works. You need to use this information to determine where the error is. If this failed to find the problem then the problem may be located elsewhere in the program's code, or you have misread something. Getting someone else to help you debug your programs is always a good option, they are likely to see things that you may miss as the code's developer.

Note

- Pay careful attention to exactly what the statement is getting the computer to do.
- Proceed slowly and carefully, as mistakes can be as small as a single character in the wrong place.
- Make sure that you are aware of exactly what the different Statements do, and how they work.
- Always use the values of the Variables from '*memory*', i.e. read/write them on to your piece of paper.

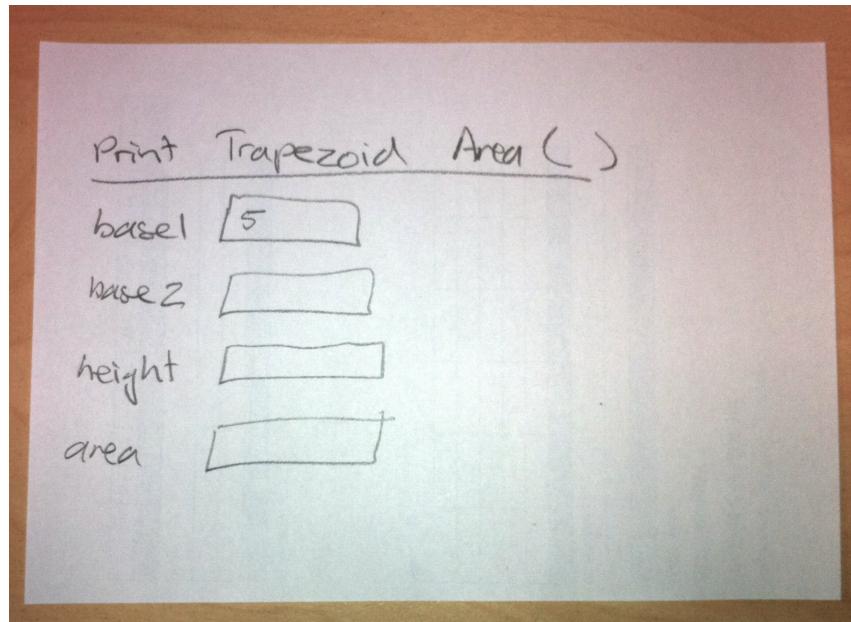


Figure 2.58: The value 5 is stored in the base1 Variable



Figure 2.59: The value 7 is stored in the base1 Variable

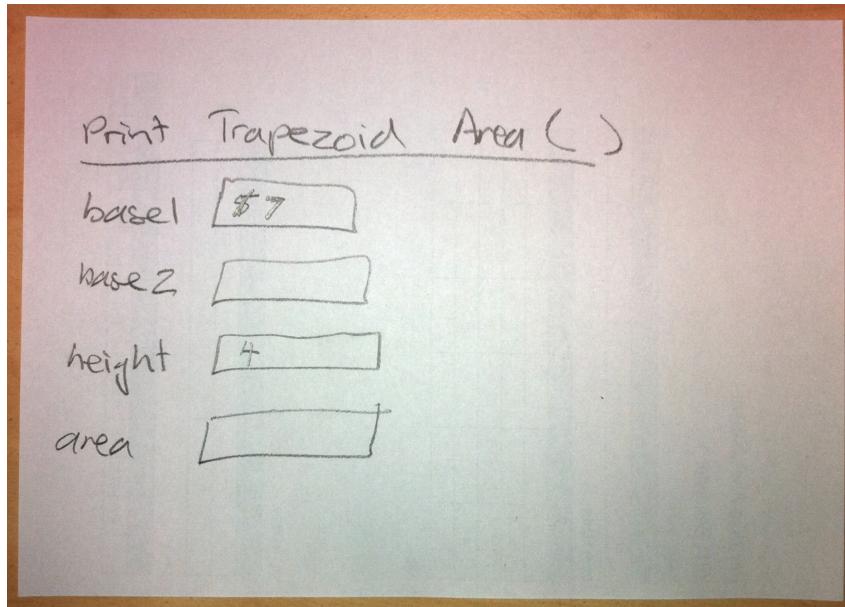


Figure 2.60: The value 4 has been stored in height

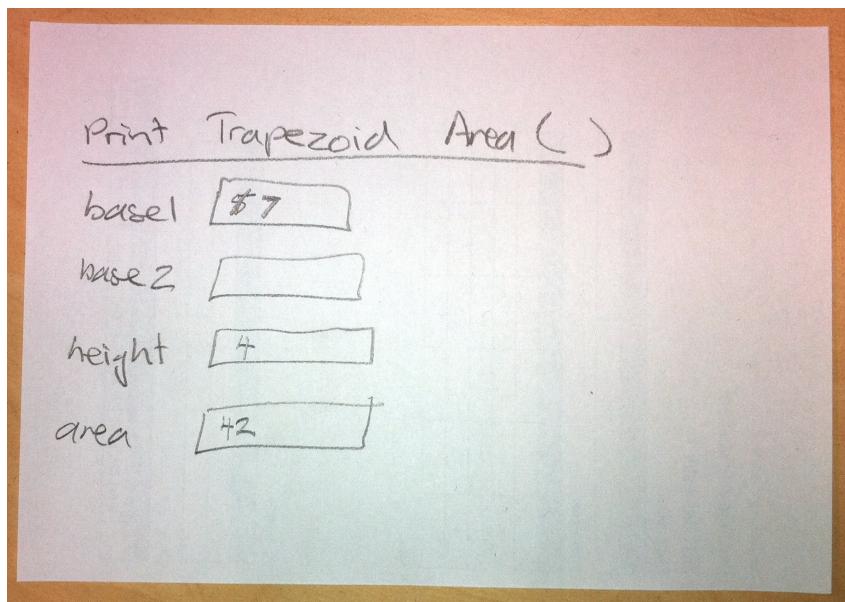


Figure 2.61: The value 42 is stored in area

2.5.6 Summary

In this Section you have seen how Variables work within the Computer, and been introduced to the idea of Functions. With these concepts you can now work more meaningfully with the data in your programs.

Note

- A Variable has two main aspects:
 - The **Variable** itself, a space in memory. You can think of this like a box, into which a value can be placed.
 - The **value** that is stored within the Variable.
- Variables can be declared in one of three locations:
 - **Local Variables** are declared within Functions and Procedures.
 - **Parameters** are also declared within Functions and Procedures, but are given a value as part of the call to this code.
 - **Global Variables** are declared within the Program, and are accessible within all Functions and Procedures.
- The **Assignment Statement** can be used to store a value in a Variable. The *left hand side* of an assignment is a Variable, the *right hand side* is an Expression.
- You can read the *value* of a Variable in an **Expression**.
- When you call a Function or a Procedure you can pass the Parameter either **by value** or **by reference**. When passed *by reference* the Parameter must be passed a Variable.
- A **Function** is just like a Procedure, except that it returns a result and is therefore called inside Expressions so that the returned value can be used.
- A **Function** is called to *calculate a value*.



2.6 Data Examples

2.6.1 Times Table

This program prints out the times table for a number entered by the user, displaying from 1 x n to 10 x n. The description of the program is in Table 2.7, the pseudocode in Listing 2.32, the C code in Listing 2.33, and the Pascal code in Listing 2.34.

Program Description	
Name	Times Table
Description	Displays the Times Table from 1 x n to 10 x n.

Table 2.7: Description of the Times Table program

Pseudocode

```

-----
Program: Times Table
-----
Variables:
- number (Integer)
Steps:
1: Output 'Times Table' to the Terminal
2: Output 'Enter number:' to the Terminal
3: Read input into number
4: Output '-----' to the Terminal
5: Output ' 1 x ' number ' = ', and 1 * number to the Terminal
6: Output ' 2 x ' number ' = ', and 2 * number to the Terminal
7: Output ' 3 x ' number ' = ', and 3 * number to the Terminal
8: Output ' 4 x ' number ' = ', and 4 * number to the Terminal
9: Output ' 5 x ' number ' = ', and 5 * number to the Terminal
10: Output ' 6 x ' number ' = ', and 6 * number to the Terminal
11: Output ' 7 x ' number ' = ', and 7 * number to the Terminal
12: Output ' 8 x ' number ' = ', and 8 * number to the Terminal
13: Output ' 9 x ' number ' = ', and 9 * number to the Terminal
14: Output '10 x ' number ' = ', and 10 * number to the Terminal
15: Output '-----' to the Terminal

```

Listing 2.32: Pseudocode for Times Table program.



Note

This is an updated version of the Seven Times Table Program. See Section 1.6.1 Seven Times Table.



C++

```
/*
 * Program: times_table.c
 * Displays the Times Table from 1 x n to 10 x n.
 */

#include <stdio.h>

int main()
{
    int number = 0;

    printf("Times Table\n");

    printf("Enter number: ");
    scanf("%d", &number);

    printf("-----\n");
    printf(" 1 x %d = %d\n", number, 1 * number);
    printf(" 2 x %d = %d\n", number, 2 * number);
    printf(" 3 x %d = %d\n", number, 3 * number);
    printf(" 4 x %d = %d\n", number, 4 * number);
    printf(" 5 x %d = %d\n", number, 5 * number);
    printf(" 6 x %d = %d\n", number, 6 * number);
    printf(" 7 x %d = %d\n", number, 7 * number);
    printf(" 8 x %d = %d\n", number, 8 * number);
    printf(" 9 x %d = %d\n", number, 9 * number);
    printf("10 x %d = %d\n", number, 10 * number);
    printf("-----\n");

    return 0;
}
```

Listing 2.33: C Times Table

Pascal

```
//  
// Program: TimesTable.pas  
// Displays the Times Table from 1 x n to 10 x n.  
//  
program TimesTable;  
  
procedure Main();  
var  
    number: Integer;  
begin  
    WriteLn('Times Table');  
  
    Write('Enter number: ');  
    ReadLn(number);  
  
    WriteLn('-----');  
    WriteLn(' 1 x ', number, ' = ', 1 * number);  
    WriteLn(' 2 x ', number, ' = ', 2 * number);  
    WriteLn(' 3 x ', number, ' = ', 3 * number);  
    WriteLn(' 4 x ', number, ' = ', 4 * number);  
    WriteLn(' 5 x ', number, ' = ', 5 * number);  
    WriteLn(' 6 x ', number, ' = ', 6 * number);  
    WriteLn(' 7 x ', number, ' = ', 7 * number);  
    WriteLn(' 8 x ', number, ' = ', 8 * number);  
    WriteLn(' 9 x ', number, ' = ', 9 * number);  
    WriteLn('10 x ', number, ' = ', 10 * number);  
    WriteLn('-----');  
end;  
  
begin  
    Main();  
end.
```

**Listing 2.34:** Pascal Times Table

2.6.2 Circle Area

This program prints out the area of a circle. The description of the program is in Table 2.8, the pseudocode in Listing 2.35, the C code in Listing 2.36, and the Pascal code in Listing 2.37.

Program Description	
Name	Circle Areas
Description	Displays the Circle Areas for circles with radius from 1.0 to 5.0 with increments of 0.5.

Table 2.8: Description of the Circle Areas program

Pseudocode

```
-----
Program: Circle Areas
-----

Constant: PI = 3.1415

Function: Circle Area
-----
Returns: A Double - the area of a circle with the given radius
Parameters:
* radius (Double)
Steps:
1: Return the result PI * radius * radius

Steps:
1: Output 'Circle Areas' to the Terminal
2: Output '-----' to the Terminal
3: Output 'Radius: 1.0 = ', CircleArea(1.0) to the Terminal
4: Output 'Radius: 1.5 = ', CircleArea(1.5) to the Terminal
5: Output 'Radius: 2.0 = ', CircleArea(2.0) to the Terminal
6: Output 'Radius: 2.5 = ', CircleArea(2.5) to the Terminal
7: Output 'Radius: 3.0 = ', CircleArea(3.0) to the Terminal
8: Output 'Radius: 3.5 = ', CircleArea(3.5) to the Terminal
9: Output 'Radius: 4.0 = ', CircleArea(4.0) to the Terminal
10: Output 'Radius: 4.5 = ', CircleArea(4.5) to the Terminal
11: Output 'Radius: 5.0 = ', CircleArea(5.0) to the Terminal
12: Output '-----' to the Terminal
```

Listing 2.35: Pseudocode for Circle Areas program.

Note

This is an updated version of the Circle Areas Program. See Section 1.6.2 Circle Area.

C++

```
/*
 * Program: circle_areas.c
 * Displays the Circle Areas for circles with radius
 * from 1.0 to 5.0 with increments of 0.5.
 */

#include <stdio.h>

#define PI 3.1415

double circle_area(double radius)
{
    return PI * radius * radius;
}

int main()
{
    printf("Circle Areas\n");
    printf("-----\n");
    printf(" Radius: 1.0 = %4.2f\n", circle_area(1.0));
    printf(" Radius: 1.5 = %4.2f\n", circle_area(1.5));
    printf(" Radius: 2.0 = %4.2f\n", circle_area(2.0));
    printf(" Radius: 2.5 = %4.2f\n", circle_area(2.5));
    printf(" Radius: 3.0 = %4.2f\n", circle_area(3.0));
    printf(" Radius: 3.5 = %4.2f\n", circle_area(3.5));
    printf(" Radius: 4.0 = %4.2f\n", circle_area(4.0));
    printf(" Radius: 4.5 = %4.2f\n", circle_area(4.5));
    printf(" Radius: 5.0 = %4.2f\n", circle_area(5.0));
    printf("-----\n");

    return 0;
}
```

Listing 2.36: C Circle Areas

Pascal

```
//  
// Program: circle_areas.c  
// Displays the Circle Areas for circles with radius  
// from 1.0 to 5.0 with increments of 0.5.  
  
program CircleAreas;  
  
const PI = 3.1415;  
  
function CircleArea(radius: Double): Double;  
begin  
    result := PI * radius * radius;  
end;  
  
procedure Main();  
begin  
    WriteLn('Circle Areas');  
    WriteLn('-----');  
    WriteLn(' Radius: 1.0 = ', CircleArea(1.0):4:2);  
    WriteLn(' Radius: 1.5 = ', CircleArea(1.5):4:2);  
    WriteLn(' Radius: 2.0 = ', CircleArea(2.0):4:2);  
    WriteLn(' Radius: 2.5 = ', CircleArea(2.5):4:2);  
    WriteLn(' Radius: 3.0 = ', CircleArea(3.0):4:2);  
    WriteLn(' Radius: 3.5 = ', CircleArea(3.5):4:2);  
    WriteLn(' Radius: 4.0 = ', CircleArea(4.0):4:2);  
    WriteLn(' Radius: 4.5 = ', CircleArea(4.5):4:2);  
    WriteLn(' Radius: 5.0 = ', CircleArea(5.0):4:2);  
    WriteLn('-----');  
end;  
  
begin  
    Main();  
end.
```

**Listing 2.37:** Pascal Circle Areas

Water Tank

The *Water Tank* program draws four water tanks to the terminal. Each water tank is drawn as a cylinder that fills a given area on the screen, and shows its current water level. An example execution is shown in Figure 2.62.

Program Description	
Name	Water Tank
Description	Displays calculates and displays 'Water Tanks'. Each tank has a position on the screen, a width, height, and a percent full.

Table 2.9: Description of the Water Tanks program

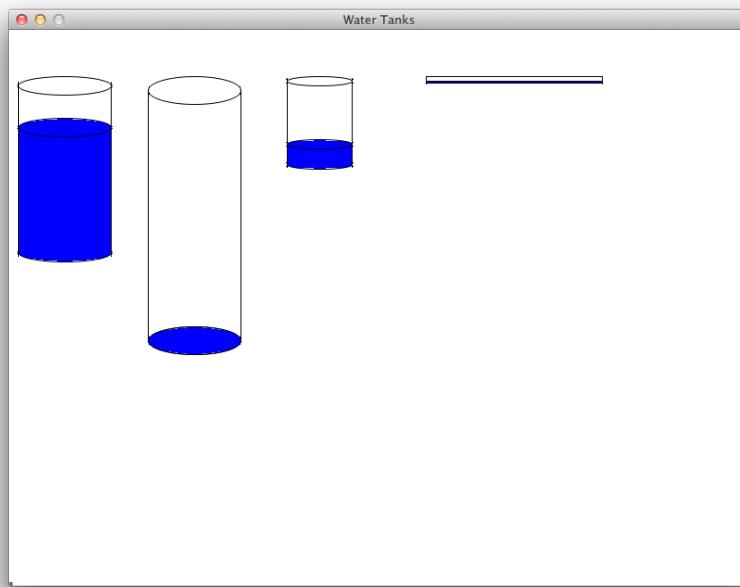


Figure 2.62: Example execution of the Water Tank program

C++

```
#include "splashkit.h"

#define MAX_HEIGHT 400
#define MAX_WIDTH 200

// Draws a water tank, at a given x, y location in a set width and height
// and at a certain percent full
void draw_water_tank(float x, float y, int width, int height, float pct_full)
{
    float ellipse_height;           // the height of the ellipses for top/bottom
    float body_height, body_y;      // the height of the core of the cylinder
    float bottom_ellipse_y, top_ellipse_y; // the y position of the ellipses
    float water_height, water_y;    // the top (y) of the water, and its height

    ellipse_height = height * 0.1;           // 10% of its height
    body_height = height - ellipse_height;
    body_y = y + ellipse_height / 2;
    bottom_ellipse_y = (y + height) - ellipse_height;

    water_height = pct_full * body_height;
    water_y = body_y + (body_height - water_height);
    top_ellipse_y = water_y - ellipse_height / 2;

    // Water...
    // Bottom ellipse
    fill_ellipse(ColorBlue, x, bottom_ellipse_y, width, ellipse_height);
    draw_ellipse(ColorBlack, x, bottom_ellipse_y, width, ellipse_height);
    // Body - center of cylinder
    fill_rectangle(ColorBlue, x, water_y, width, water_height);
    //Top ellipse
    fill_ellipse(ColorBlue, x, top_ellipse_y, width, ellipse_height);
    draw_ellipse(ColorBlack, x, top_ellipse_y, width, ellipse_height);

    // Frame
    draw_ellipse(ColorBlack, x, y, width, ellipse_height);
    draw_line(ColorBlack, x, y + ellipse_height / 2, x,
              bottom_ellipse_y + ellipse_height / 2);
    draw_line(ColorBlack, x + width, y + ellipse_height / 2,
              x + width, bottom_ellipse_y + ellipse_height / 2);
}
```

Listing 2.38: C Water Tank drawing code

C++

```
// Coordinates the drawing of a number of water tanks
int main(int argc, char* argv[])
{
    open_audio();
    open_graphics_window("Water Tanks", 800, 600);
    load_default_colors();

    clear_screen(ColorWhite);
    draw_water_tank(10, 50, 100, 200, 0.75);
    draw_water_tank(150, 50, 100, 300, 0.0);
    draw_water_tank(300, 50, 70, 100, 0.25);
    draw_water_tank(450, 50, rnd() * MAX_HEIGHT, rnd() * MAX_WIDTH, 0.25);
    refresh_screen();

    delay(5000);

    release_all_resources();
    close_audio();
    return 0;
}
```

Listing 2.39: C Water Tank drawing code (continued from Listing 2.38)

Pascal

```

program GameMain;
uses
  sgTypes, sgInput, sgAudio, sgGraphics, sgResources, sgUtils, sgText;

const
  MAX_HEIGHT = 400;
  MAX_WIDTH = 200;

procedure DrawWaterTank(x, y, width, height: Integer; pctFull: Single);
var
  ellipseHeight: Single;           // height of the ellipses for top/bottom
  bodyHeight, bodyY: Single;        // the height of the core of the cylinder
  bottomEllipseY, topEllipseY: Single; // the y position of the ellipses
  waterHeight, waterY: Single;      // the top (y) of the water and its height
begin
  ellipseHeight := height * 0.1;     // ellipse height = 10% total height
  bodyHeight := height - ellipseHeight; // area for the center of the cylinder
  bodyY := y + ellipseHeight / 2;
  bottomEllipseY := (y + height) - ellipseHeight;

  waterHeight := pctFull * bodyHeight;
  waterY := bodyY + (bodyHeight - waterHeight);
  topEllipseY := waterY - ellipseHeight / 2;

  // Water...
  // Bottom ellipse
  FillEllipse(ColorBlue, x, bottomEllipseY, width, Round(ellipseHeight));
  DrawEllipse(ColorBlack, x, bottomEllipseY, width, Round(ellipseHeight));
  // Body - center of cylinder
  FillRectangle(ColorBlue, x, waterY, width, Round(waterHeight));
  //Top ellipse
  FillEllipse(ColorBlue, x, topEllipseY, width, Round(ellipseHeight));
  DrawEllipse(ColorBlack, x, topEllipseY, width, Round(ellipseHeight));

  // Frame
  DrawEllipse(ColorBlack, x, y, width, Round(ellipseHeight));
  DrawLine(ColorBlack, x, y + ellipseHeight / 2, x,
           bottomEllipseY + ellipseHeight / 2);
  DrawLine(ColorBlack, x + width, y + ellipseHeight / 2, x + width,
           bottomEllipseY + ellipseHeight / 2);
end;

```

Listing 2.40: Pascal Water Tank

Pascal

```
procedure Main();
begin
  OpenGraphicsWindow('Water Tanks', 800, 600);

  ClearScreen(ColorWhite);
  DrawWaterTank(10, 50, 100, 200, 0.75);
  DrawWaterTank(150, 50, 100, 300, 0.0);
  DrawWaterTank(300, 50, 70, 100, 0.25);
  DrawWaterTank(450, 50, Round(rnd() * MAX_HEIGHT),
                Round(rnd() * MAX_WIDTH), 0.25);

  RefreshScreen();

  Delay(5000);

  ReleaseAllResources();
end;

begin
  Main();
end.
```



Listing 2.41: Pascal Water Tank (continued from Listing 2.40)

2.6.3 Bicycle Race

The Bicycle Race program will simulate a thirty second sprint race between a number of bicycles. The race has a standing start, and then each racer accelerates as fast as they can for thirty seconds. The winner is the racer who makes it the furthest.

Program Description	
Name	Bike Race
Description	Calculates the position of seven bikes at the end of a timed race, drawing the final positions. Each bike's position is calculated based on a random acceleration over the duration of the race.

Table 2.10: Description of the Bike Race program

- You can calculate distance the racers cover using Equation 2.1.

- s is the distance covered.
- u is the starting speed
- t is time.
- a is acceleration.

$$s = ut + \frac{at^2}{2}$$

2.1

- This race has a standing start, so the initial speed of each racer will be 0.
- The time for the race is 30 seconds, this is constant.
- Each racer will have a randomly determined acceleration, with a maximum acceleration of 10 pixels/second²

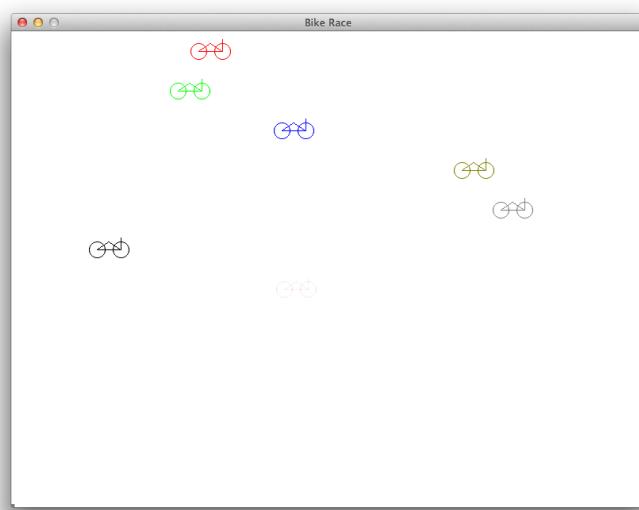


Figure 2.63: Example execution of the Bike Race program

C++

```
/* Program: bike-race.c, splashkit */

#include "splashkit.h"

// =====
// = Define constants =
// =====

#define WHEEL_SIZE 10
#define WHEEL_GAP 10
#define SEAT_GAP 5
#define MAX_ACCELERATION 10
#define RACE_DURATION 30
#define X_SCALE_FACTOR 0.15

// =====
// = Functions =
// =====

// Calculate the distance travelled given acceleration and time
// distance = ut + ((at^2) / 2)
float distance_travelled(float initial_speed, float acceleration, float time)
{
    return (initial_speed * time) + ((acceleration * time * time) / 2);
}

// Calculate the x position of a bike accelerating at the given
// acceleration for the duration of the race
float bike_x_for_accel(float acceleration)
{
    float distance;

    distance = distance_travelled(0, acceleration, RACE_DURATION);

    return distance * X_SCALE_FACTOR;
}

// Come up with a random acceleration value for a bike between 0 and
// MAX_ACCELERATION
float random_accel()
{
    return rnd() * MAX_ACCELERATION;
}
```

Listing 2.42: C Bicycle Race, continued in Listing 2.43

C++

```

// =====
// = Procedures =
// =====

// Draw the bike to the screen in the given color
void draw_bike(color bike_color, float x, float y)
{
    float left_wheel_x, right_wheel_x, wheel_y;
    float seat_x, seat_y;

    left_wheel_x = x + WHEEL_SIZE;
    right_wheel_x = left_wheel_x + WHEEL_SIZE * 2 + WHEEL_GAP;

    wheel_y = y + WHEEL_SIZE + SEAT_GAP;

    seat_x = (right_wheel_x - left_wheel_x) / 2.0f + left_wheel_x;
    seat_y = y + SEAT_GAP;

    draw_circle(bike_color, left_wheel_x, wheel_y, WHEEL_SIZE);
    draw_circle(bike_color, right_wheel_x, wheel_y, WHEEL_SIZE);
    draw_triangle(bike_color, left_wheel_x, wheel_y,
                  right_wheel_x, wheel_y,
                  seat_x, seat_y);
    draw_line(bike_color, right_wheel_x, wheel_y, right_wheel_x, y);
}

// =====
// = Main - Entry Point =
// =====

// Run the bike race...
int main()
{
    open_audio();
    open_graphics_window("Bicycle Race...", 800, 600);
    load_default_colors();

    clear_screen(ColorWhite);

    draw_bike(ColorRed, bike_x_for_accel(random_accel()), 10);
    draw_bike(ColorGreen, bike_x_for_accel(random_accel()), 60);
    draw_bike(ColorBlue, bike_x_for_accel(random_accel()), 110);
    draw_bike(rgbcolor(127, 127, 0), bike_x_for_accel(random_accel()), 160);
    draw_bike(rgbcolor(127, 127, 127), bike_x_for_accel(random_accel()), 210);
    draw_bike(rgbcolor(0, 0, 0), bike_x_for_accel(random_accel()), 260);
    draw_bike(random_color(), bike_x_for_accel(random_accel()), 310);

    refresh_screen();

    delay(5000);

    close_audio();

    release_all_resources();
    return 0;
}

```

Listing 2.43: C Bicycle Race

Pascal

```
program GameMain;
uses
  sgTypes, sgInput, sgAudio, sgGraphics, sgResources, sgUtils, sgText;

// =====
// = Define constants =
// =====
const
  WHEEL_SIZE = 10;
  WHEEL_GAP = 10;
  SEAT_GAP = 5;
  MAX_ACCELERATION = 10;
  RACE_DURATION = 30;
  X_SCALE_FACTOR = 0.15;

// =====
// = Functions =
// =====

// Calculate the distance travelled given acceleration and time
// distance = ut + ((at^2) / 2)
function DistanceTravelled(initialSpeed, acceleration, time: Single): Single;
begin
  result := (initialSpeed * time) + ((acceleration * time * time) / 2);
end;

// Calculate the x position of a bike accelerating at the given
// acceleration for the duration of the race
function BikeXForAccel(acceleration: Single): Single;
var
  distance: Single;
begin
  distance := DistanceTravelled(0, acceleration, RACE_DURATION);
  result := distance * X_SCALE_FACTOR;
end;

// Come up with a random acceleration value for a bike between 0 and
// MAX_ACCELERATION
function RandomAccel(): Single;
begin
  result := rnd() * MAX_ACCELERATION;
end;
```

Listing 2.44: Pascal Bike Race)



Pascal

```

// =====
// = Procedures =
// =====

// Draw the bike to the screen in the given color
procedure DrawBike(bikeColor: Color; x, y: Single);
var
  leftWheelX, rightWheelX, wheely: Single;
  seatX, seatY: Single;
begin
  leftWheelX := x + WHEEL_SIZE;
  rightWheelX := leftWheelX + WHEEL_SIZE * 2 + WHEEL_GAP;

  wheely := y + WHEEL_SIZE + SEAT_GAP;

  seatX := (rightWheelX - leftWheelX) / 2.0 + leftWheelX;
  seatY := y + SEAT_GAP;

  DrawCircle(bikeColor, leftWheelX, wheely, WHEEL_SIZE);
  DrawCircle(bikeColor, rightWheelX, wheely, WHEEL_SIZE);
  DrawTriangle(bikeColor, leftWheelX, wheely,
                rightWheelX, wheely,
                seatX, seatY);
  DrawLine(bikeColor, rightWheelX, wheely, rightWheelX, y);
end;

// =====
// = Main - Entry Point =
// =====

procedure Main();
begin
  OpenGraphicsWindow('Bike Race', 800, 600);

  ClearScreen(ColorWhite);

  DrawBike(ColorRed, BikeXForAccel(RandomAccel()), 10);
  DrawBike(ColorGreen, BikeXForAccel(RandomAccel()), 60);
  DrawBike(ColorBlue, BikeXForAccel(RandomAccel()), 110);
  DrawBike(RGBColor(127, 127, 0), BikeXForAccel(RandomAccel()), 160);
  DrawBike(RGBColor(127, 127, 127), BikeXForAccel(RandomAccel()), 210);
  DrawBike(RGBColor(0, 0, 0), BikeXForAccel(RandomAccel()), 260);
  DrawBike(RandomColor(), BikeXForAccel(RandomAccel()), 310);
  RefreshScreen();

  Delay(5000);

  ReleaseAllResources();
end;

begin
  Main();
end.

```



Listing 2.45: Pascal Bike Race (continued from Listing 2.44)

2.7 Data Exercises

2.7.1 Concept Questions

Read over the concepts in this chapter and answer the following questions:

1. What is a **Variable**?
2. What is the relationship between a variable and a value?
3. What is the relationship between a variable and a type?
4. Where can you use variables? Think both reading the value, and storing a new value.
5. What does it mean if a variable appears on the right hand side of an assignment? What will happen to this variable when the code is run?
6. What does it mean if a variable appears on the left hand side of an assignment? What will happen to this variable when the code is run?
7. What is a **Constant**? How does it differ from a variable?
8. What is a local variable? What code can access the value in a local variable?
9. What is a global variable? What code can access the value in a global variable?
10. Why is it considered good practice to use local variable, but not global variables?
11. How do parameters help make procedures more powerful?
12. What are the two parameter passing mechanisms for passing parameters? How are they different?
13. When would you use each of the parameter passing mechanisms? For what kind of parameters?
14. How does the Terminal input procedure store a value in the variable you pass to it? What kind of parameter passing is involved here?
15. What statement was introduced in this chapter?
16. What does this statement allow you to do?
17. What is a function? How does it differ from a procedure?
18. A procedure call is a statement. What is a function call? Why is this different?
19. What does it mean when you say a function returns a value?
20. What are the values of the following expressions?

Question	Expression	Given
(a)	5	
(b)	a	a is 2.5
(c)	1 + 2 * 3	
(d)	a + b	a is 1 and b is 2
(e)	2 * a	a is 3
(f)	a * 2 + b	a is 1.5 and b is 2
(g)	a + 2 * b	a is 1.5 and b is 2
(h)	(a + b) * c	a is 1, b is 1 and c is 5
(i)	a + b * 2	a is 1.0 and b is 2

21. When creating a program, types allow you to reason about the kind of data the program is using. The three most basic types of data are Double, Integer and String. Use the Double⁷ data type to represent any real numeric value; such as 1, 2.5, -75.201 etc. The Integer data type is used to represent any whole numeric value; such as 1, 0, -27 etc. Use the String data type for any textual data. There are many other data types, but these three are the most frequently used.

When assigning a data type think about the following:

- How will the data be used?
- The range of values expected. Does the type have a sufficient range to cover this?
- Is precision important? Think carefully, especially with fractional values represented as floating point numbers (i.e. Double).

What data type is most appropriate to store the following?

- | | |
|--|--|
| (a) A person's name | (f) The runs scored in a cricket match |
| (b) The number of students in a class | (g) A student's ID number |
| (c) The average age of a group of people | (h) The distance between planets (km) |
| (d) A temperature in Celsius | (i) A person's phone number |
| (e) The name of a subject | (j) The cost of an item |

⁷Computer programming languages often use floating point values to represent real numbers. This format stores an approximation for a large range of values. You need to keep this in mind when thinking about the kind of data you will use.

2.7.2 Code Writing Questions: Applying what you have learnt

Apply what you have learnt to the following tasks:

1. Take the times table program from Section 2.6.1 and re-implement it so that there are two procedures: Print Times Table, and Print Times Table Line. Use these to print the 42, 73, and 126 times tables, as well as printing a times table the user requests.
 - The Print Times Table Line procedure will take two parameters. The first will be the number, the second will be the times. This will output a single line for the table, e.g. ‘ $1 \times 73 = 73$ ’.
 - The Print Times Table procedure will have a single parameter called number. It will output a header for the table, and then call Print Times Table Line ten times. In each call it will pass 1, 2, 3, etc. for the times parameter, and pass across its number value to the number parameter. After printing the last line it will output a footer for the table.
2. Correct and then implement the Trapezoid Area procedure from Section 2.5.5. Adjust the implementation to call a Trapezoid Area function that is passed the two base values and the height, and returns the area. Create a small program to test this procedure.
3. Implement the Change Calculation program, and test it function as you expect.
4. Revisit your Circle Dimensions program from Chapter 1 and adjust its implementation to make use of functions and procedures.
5. Design the structure and then the code for a program that converts temperatures from Celsius to Fahrenheit. This should read the value to convert from the user, and output the results to the Terminal.
6. Take the adjusted Face Shape program from Chapter ??, and re-implement it so that the Draw Face procedure takes in an x and y coordinate for the location where the face will be drawn. Adjust the coordinates of the face’s components in Draw Face, by the amounts in the x and y parameters. Use your new procedure to draw three faces to the screen at different positions.
7. Write a Swap procedure that takes in two integer parameters (passed by reference) and swaps their values. Write a program to test this procedure. This should work so that if you call Swap(a, b); that the values in the a and b variables are swapped over. You can test this by printing the values before and after the call to the Swap procedure.
8. Watch <http://www.youtube.com/watch?v=y2R3FvS4xr4>, which clearly demonstrates the importance of being able to calculate the airspeed velocity of a swallow. This can be calculated using an equation based on the Strouhal Number, see <http://www.style.org/strouhalflight>. Use this information to create a program that can be used to calculate the airspeed velocity of African and European Swallows. Use the following values:
 - Strouhal Number of 0.33
 - African Swallow: frequency 15hz, amplitude 21cm
 - European Swallow: frequency 14hz, amplitude 22cm

2.7.3 Extension Questions

If you want to further your knowledge in this area you can try to answer the following questions. The answers to these questions will require you to think harder, and possibly look at other sources of information.

1. Write a small program to experiment with parameter passing. Create in this program a procedure called `Print It` that takes a integer parameter and prints it to the Terminal. Also create a `Double It` procedure that takes an integer parameter passed by reference⁸ and has its value doubled in the procedure. Try the following (not all will work):
 - (a) Call `Print It`, passing in a literal value like 5.
 - (b) Call `Double It`, passing in a literal value like 5.
 - (c) Call `Print It`, passing in a calculated expression like `a + b`.
 - (d) Call `Double It`, passing in a calculated expression like `a + b`.
 - (e) Call `Print It`, passing in a variable's value.
 - (f) Call `Double It`, passing in a variable's value.
2. Further adjust your Face Drawing program so that the caller can pass in a custom color, width, and height for the face.
3. Adjust the bike race example from Section 2.6.3 so that the racers have a rolling start. Each bike will then have a different initial speed, calculated as a random value. You will need to define a maximum starting speed, and recalculate the x scale factor to ensure that the bikes are all drawn to the screen.

⁸If you are using C, you will need to do this with C++. With C++ your compiler is now `g++`, rather than `gcc`.

3

Control Flow

Time has come for you to learn how to control the flow of the magical energies. This arcane knowledge will unlock great power. Making it possible to do things that until now seemed impossible. Close your eyes and picture the energy travelling through your spell. Take control of that flow by...

The focus so far has been on learning the programming artefacts that you can create within your code. You have seen how to create [Programs](#), [Procedures](#), [Functions](#), and [Variables](#). The actual instructions that you can issue to the computer have been limited to [Procedure Calls](#), [Function Calls](#), and [Assignment Statements](#). This Chapter will introduce you to the other actions that you can command the computer to perform. It will show you how to control the flow of the instructions, unlocking great power, and making it possible to do things that until now seemed impossible.

When you have understood the material in this chapter you will be able to write code that commands the computer to perform a wider range of actions by controlling the sequence in which the basic commands are performed.

Contents

3.1 Control Flow Concepts	201
3.1.1 Boolean Data	202
3.1.2 Branching	205
3.1.3 Looping	208
3.1.4 Jumping	211
3.1.5 Compound Statement	216
3.1.6 Statement (Simple and Structured)	217
3.1.7 Summary	218
3.2 Using these Concepts	219
3.2.1 Designing Guess that Number	219
3.2.2 Understanding Guess that Number	220
3.2.3 Choosing Artefacts for Guess that Number	220
3.2.4 Designing Control Flow for Perform Guess	222
3.2.5 Designing Control Flow for Play Game	236
3.2.6 Designing Control Flow for Print Line	241
3.2.7 Designing the Control Flow for Main	243
3.2.8 Writing the Code for Guess That Number	245
3.2.9 Compiling and Running Guess that Number	245
3.3 Control Flow in C	246
3.3.1 Implementing the Guess that Number in C	246

3.3.2 C Boolean Data	248
3.3.3 C Statement (with loops)	250
3.3.4 C If Statement	251
3.3.5 C Case Statement	252
3.3.6 C Compound Statement	254
3.3.7 C While Loop	255
3.3.8 C Do While Loop	256
3.3.9 C Jump Statements	257
3.4 Control Flow in Pascal	259
3.4.1 Implementing the Guess that Number in Pascal	259
3.4.2 Pascal Boolean Data	261
3.4.3 Pascal Statement (with loops)	263
3.4.4 Pascal If Statement	264
3.4.5 Pascal Case Statement	265
3.4.6 Pascal Compound Statement	267
3.4.7 Pascal While Loop	268
3.4.8 Pascal Repeat Loop	269
3.4.9 Pascal Jump Statements	270
3.5 Understanding Control Flow	271
3.5.1 Understanding Branching in Perform Guess	271
3.5.2 Understanding Looping in Play Game	284
3.5.3 Understanding looping in Print Line	295
3.6 Control Flow Examples	308
3.6.1 Times Table	308
3.6.2 Circle Area	311
3.6.3 Moving Rectangle	314
3.6.4 Button Click in SwinGame	317
3.7 Control Flow Exercises	320
3.7.1 Concept Questions	320
3.7.2 Code Reading Questions	322
3.7.3 Code Writing Questions: Applying what you have learnt	333
3.7.4 Extension Questions	334

3.1 Control Flow Concepts

Programming is about designing code that commands the computer to perform actions. Earlier chapters have introduced the [Program](#), [Procedure](#), and [Function](#) artefacts into which you can enter these instructions, but have not elaborated on the actions that you can perform.

Most of a program's actual work will be carried out in [Assignment Statements](#), and through [Procedure Calls](#) and [Function Calls](#). These are the main commands, allowing you to alter values stored in memory and to execute stored instructions. The remaining commands relate to controlling the order in which the computer performs the instructions; called **control flow statements**.

This chapter introduces the following kinds of instructions. You can use these to get the computer to perform certain **actions** within your program.

- [If Statement](#): Run some code if a condition is true.
- [Case Statement](#): Selectively run a branch of code.
- [Compound Statement](#): Group statements together.
- [Pre-Test Loop](#): Loop after testing a condition.
- [Post-Test Loop](#): Loop then test a condition.

In addition to these actions, you will need have a look at an existing [artefact](#):

- [Boolean Data](#): An existing [Type](#) that has either a *true* or *false* value.

You may need to revise the following programming artefacts:

- [Program](#): The idea of building your own programs.
- [Procedure](#): Creating your own Procedure, as well as calling Procedures from libraries.
- [Function](#): Creating your own Functions, as well as calling Functions from libraries.

The following programming terminology will also be used in this Chapter:

- [Statement](#): An instruction performed in your code.
- [Type](#): A kind of data used in your code.

The example for this chapter is a guessing game, where the user is guessing a number between 1 and 100. An example of this program executing is shown in Figure 3.1.

```

Terminal — bash — 80x24
ocain2-mac:control-flow again$ ./GuessThatNumber
I am thinking of a number between 1 and 100

Guess 1: 73
The number is less than 73
Guess 2: 10
The number is larger than 10
Guess 3: 52
The number is larger than 52
Guess 4: 60
The number is less than 60
Guess 5: 57
The number is less than 57
Guess 6: 54
The number is larger than 54
Guess 7: 56
The number is less than 56
You ran out of guesses... the number was 55

Do you want to play again [Y/n]? n
Bye
ocain2-mac:control-flow again$ 

```

Figure 3.1: Guess that Number run from the Terminal

3.1.1 Boolean Data

The Boolean¹ Data Type is a **Type** used to represent **truth**. A Boolean value will either be **true** or **false**. These values are used extensively in the control flow statements to determine the action to perform.

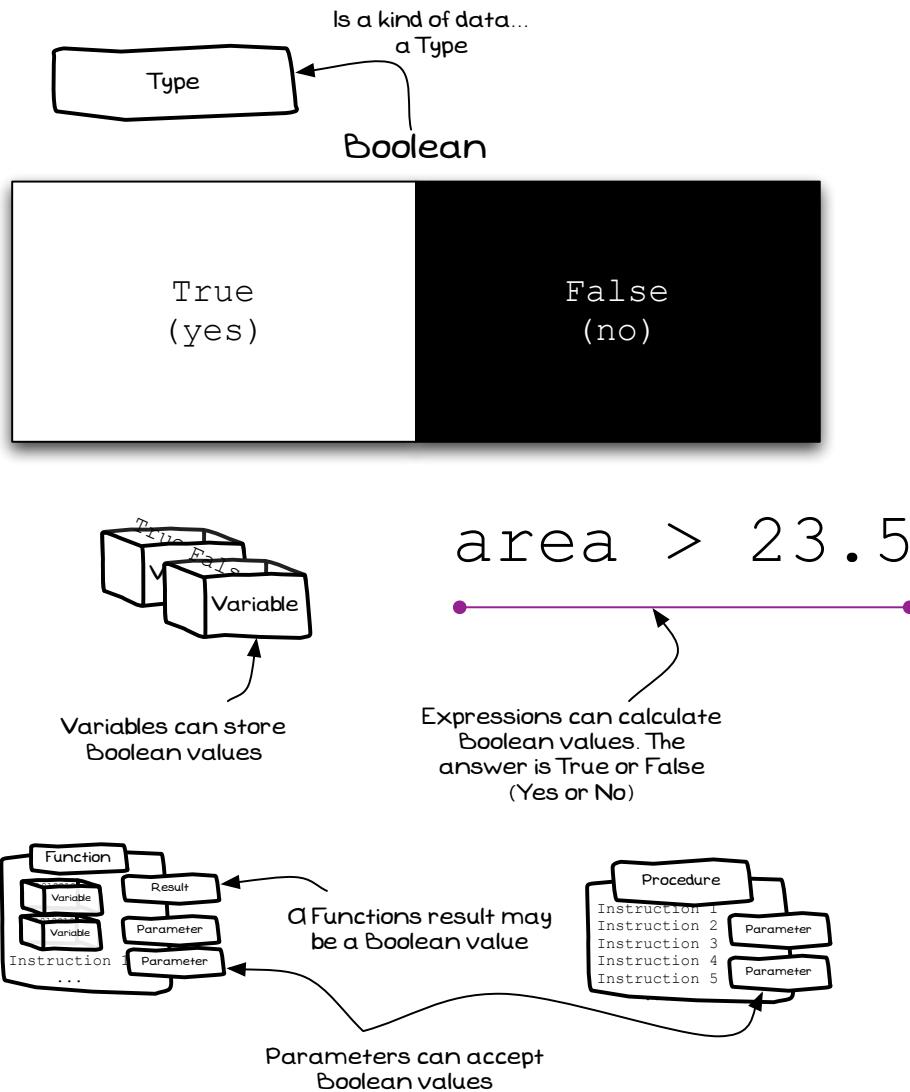


Figure 3.2: Boolean data represents truth

Note

- Boolean is an existing **artefact**, it is a **Type** that has been defined to represent truth values.
- A Boolean value is either **true** or **false**. You can also think of these as *yes* and *no*.
- Boolean values are used in most of the control flow statements.
- The Boolean type can be used in the same way as other types.

¹Named after George Bool's Boolean logic.

Comparisons

Comparisons are a common way of getting Boolean values in your code. These [Expressions](#) allow you to compare two values to check for a given condition. For example, the Expression shown in Figure 3.2 is asking if the *value* in the *area* variable is larger than 23.5. The result of this expression will be either true or false depending on the current value stored in *area*. Table 3.1 lists some example values for this expression, given different values stored in the *area* variable.

Value in area	area > 23.5
73.2	true
-2.5	false
23.5	false

Table 3.1: Example values for the expression `area > 23.5`

Programming languages offer a range of different comparison operators. These typically include comparisons to check if values are the same or different, and to check if one value is larger or smaller than another. The different operators for C and Pascal are listed in Table 3.2.

	Description	C	Pascal
Equal	Are the values the same?	<code>a == b</code>	<code>a = b</code>
Not Equal	Are the values different?	<code>a != b</code>	<code>a <> b</code>
Larger Than	Is the left value larger than the right?	<code>a > b</code>	
Less Than	Is the left value smaller than the right?	<code>a < b</code>	
Larger Or Equal	Is the left value equal or larger than the right?	<code>a >= b</code>	
Less Or Equal	Is the left value smaller or equal to the right?	<code>a <= b</code>	

Table 3.2: Comparison Operators

Note

- Comparisons can only be performed between **two** values.
- The values on either side of the comparison are [Expressions](#), allowing you to calculate the values being compared.

C++

C uses a double equal (==) for comparison as the single equals (=) is used for assignment.



Logical Operators

The comparison operators allow you to compare *two* values. This is very useful, but in itself is incomplete. What, for example, do you do when you want to compare three or more values? While you are limited to two values with the comparison operators, there are other operators that allow you to **combine** Boolean expressions. This will enable you to combine together multiple Boolean values into a single Expression.

There are four main *logical operators*: **and**, **or**, **xor**, and **not**. Each of these operators works on two Boolean values, combining them to give a new Boolean value. For example, the *and* operator allows you to check if *both* of the expressions are true. The expression `area > 0` and `area < 10` will be true only when area is both larger than zero and less than ten.

Logical Operators							
		b					
		True	False				
a	True	True	False	a	b		
	False	False	False				
a and b							
a	True	True	True	a	b		
	False	True	False				
a or b							
a	True	False	True	a	b		
	False	True	False				
a xor b							
a	True	False	True	a	b		
	False	True	False				
not a							

Figure 3.3: Logical Operators combine Boolean values

	Description	C	Pascal
And	Are both values True?	<code>a && b</code>	<code>a and b</code>
Or	Is at least one value True?	<code>a b</code>	<code>a or b</code>
Xor	Is one value True, and the other False?	<code>a ^ b</code>	<code>a xor b</code>
Not	Is the value False?	<code>!a</code>	<code>not a</code>

Table 3.3: Logical Operators

area	<code>area > 0</code>	<code>area < 10</code>	<code>area > 0 ... area < 10</code>		
			and	or	xor
5	True	True	True	True	False
27	True	False	False	True	True
0	False	True	False	True	True

Table 3.4: Example Logical Expressions

Note

- Table 3.4 has some example expressions.
- The tables in Figure 3.3 show the values of the different logical operators. These are known as **Truth Tables**.
- Table 3.3 outlines the different logical operators, and how they are coded in C and Pascal.

3.1.2 Branching

There are two main ways of controlling the sequence of actions in a program. The first of these is called **branching**, or **selection**. Branching allows you to get the computer to take one of a number of paths based on the value of a *condition*.

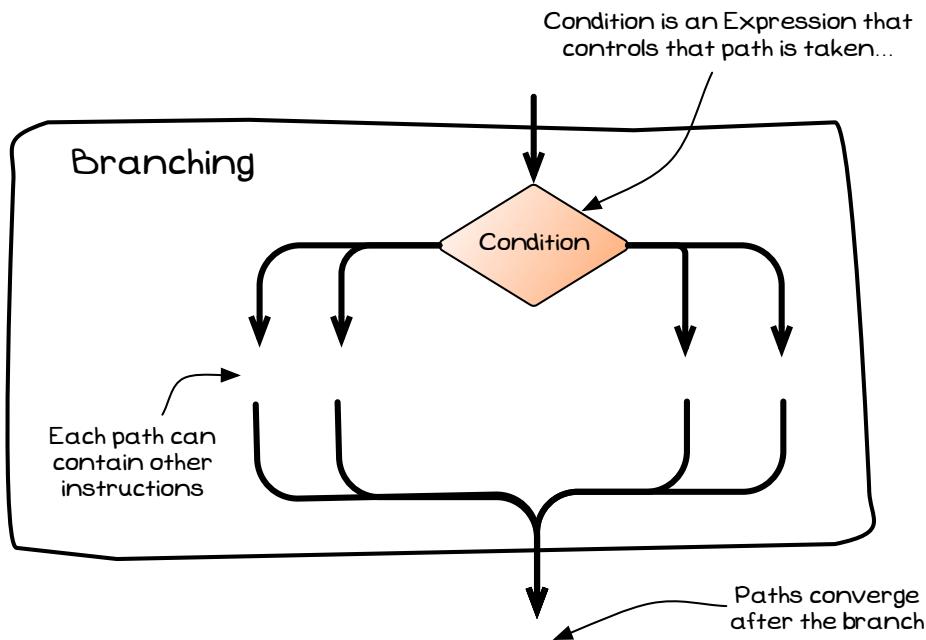


Figure 3.4: Branching commands the computer to take one of a number of possible paths

Note

- Branching is a kind of **action**. You can command the computer to take of a number of paths.
- A branch has a **condition** that is evaluated, and based on the condition the computer takes one path.
- The branch is the act of choosing the path, when its command is performed the computer evaluates the condition and then moves to the instructions in the indicated path.
- Languages usually offer two kinds of branching statements:
 - **If Statement** to select between two paths based on a Boolean expression.
 - **Case Statement** to select a path based on an ordinal^a value.
- The Branch will have one entry point, and one exit point. This feature allows you to combine statements together like building blocks. This idea comes from the principles of **Structured Programming**, where each component in the code should have a single entry and exit point.

^aIntegers and Characters are ordinal values. Ordinal values have a defined sequence, so it is possible to say which value comes next in the sequence. Integers are Ordinal as you can say that the number after 1 is 2. Real numbers are not ordinal as you cannot say which value comes next in the sequence.

If Statement

The if statement is the most frequently used branching statement. It allows you to selectively run code based on the value of a Boolean expression (the condition). The if statement has an optional *else* branch that is executed when the condition is false.

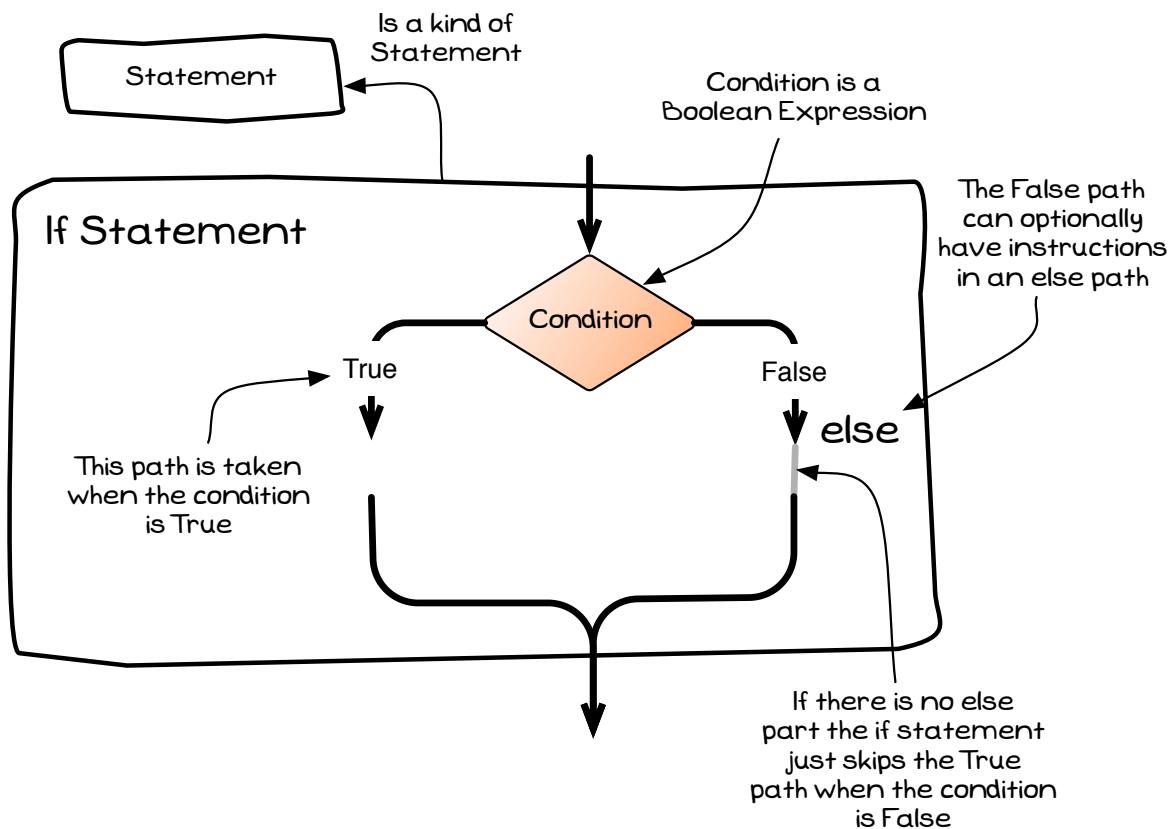


Figure 3.5: If statement lets you selectively run a branch of code

Note

- An if statement is an **action**. It allows you to command the computer to select a path based on a Boolean expression.
- The if statement has two branches, one that is taken when the condition is True, the other when it is False.
- The False branch may *optionally* have instructions that are carried out when the condition is False.
- If there are no instructions you want performed when the condition is False you do not need to include an else branch, and the if statement will just skip the True branch when the condition is False.
- The if statement has one entry point, two paths, and then one exit point.

Case Statement

The case statement is the second kind of branching statement. This allows you to create paths that execute based on matching a value from an expression. This allows one case statement to handle many alternative paths.

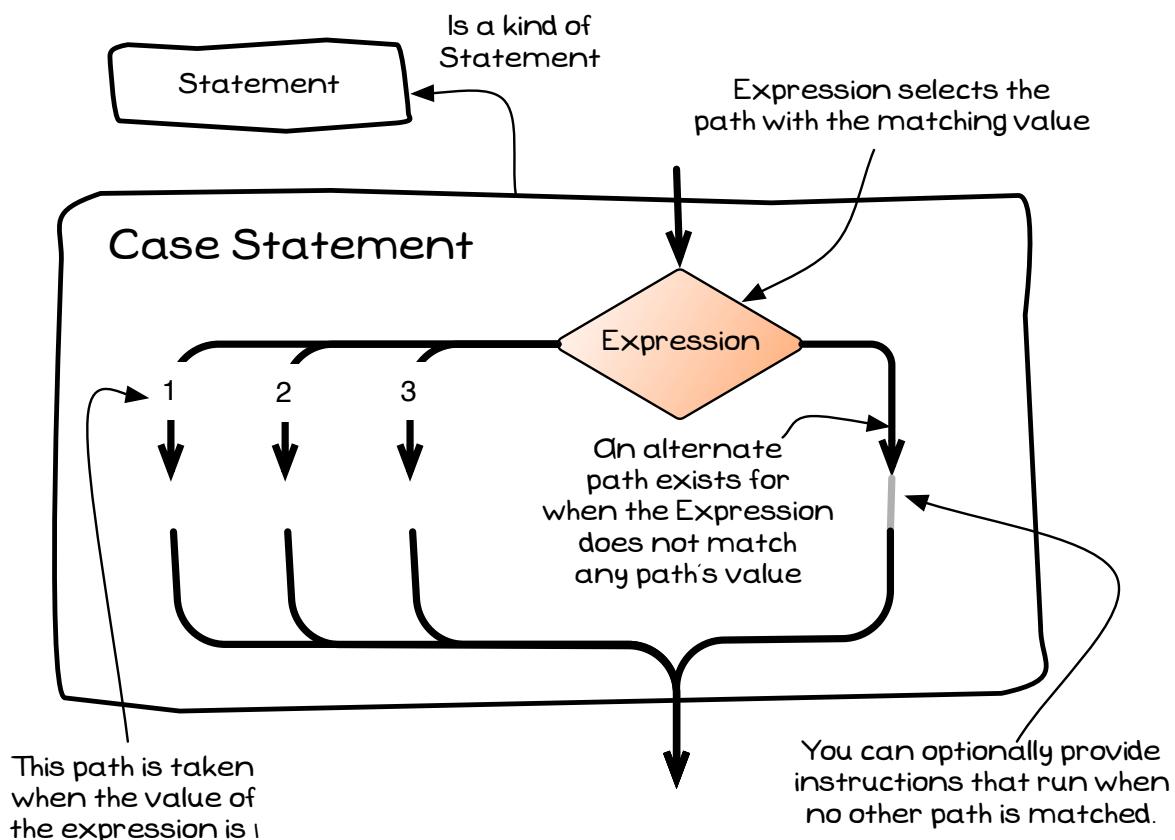


Figure 3.6: Case statement selectively runs multiple branches of code

Note

- The case statement is a kind of **action**. It allows you to command the computer to select a path based upon the value of an expression.
- Each path within the Case Statement has a value. When the computer executes the case statement the path values are used to determine which path will be taken.
- In C and Pascal the Case Statement only works with Ordinal Values. This limits you to using Character or Integer values within the Case Statement's Expression.
- The Case Statement has one entry point, multiple paths, and then one exit point.

3.1.3 Looping

There are two main ways of controlling the sequence of actions in a program. The first was **branching**, the second is called **looping**, or **repetition**. The language's looping statements allow you to have actions repeated.

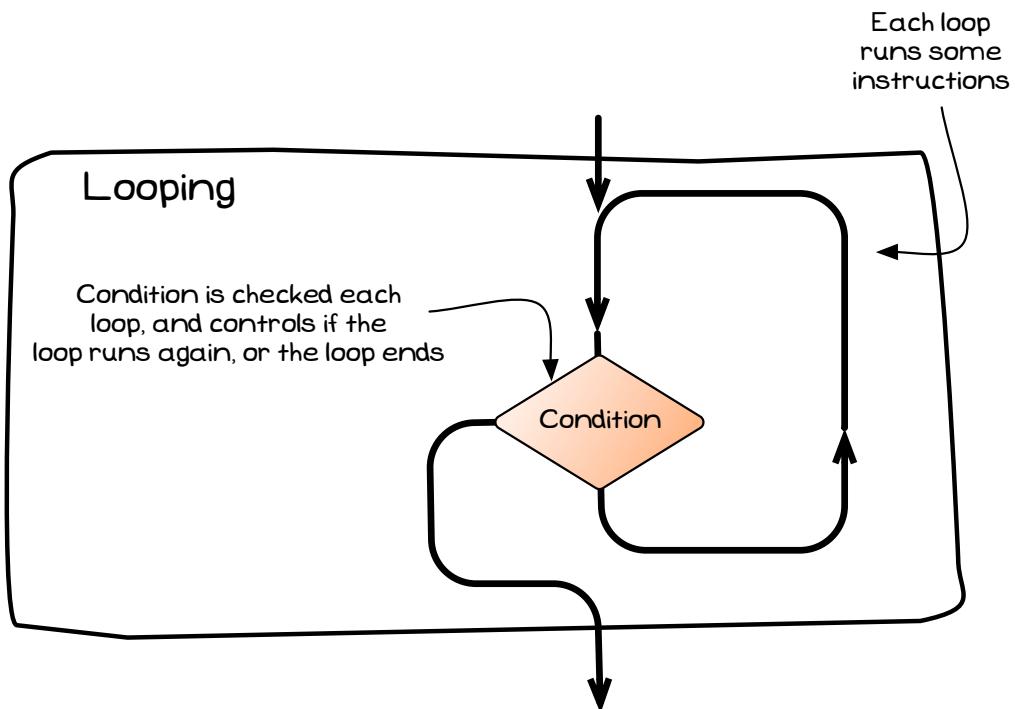


Figure 3.7: Looping commands the computer to repeat a path

Note

- Looping is a kind of **action**. You can command the computer to repeat the steps within a path.
- A number of steps are performed each loop:
 - The instructions within the loop are executed.
 - The *condition* is checked, and the instructions are either run again or the loop ends.
- The *condition* may be checked before or after the instructions are executed, giving two kinds of loops:
 - **Pre-Test Loop:** Repeats instructions 0 or more times.
 - **Post-Test Loop:** Repeats instructions 1 or more times.
- As with Branching, the Looping Statements have a single entry and a single exit in keeping with the principles of **Structured Programming**.

Pre-Test Loop

The Pre-Test Loop is a looping statement that allows code to be run 0 or more times. The loop checks the condition at the start, and if the condition is True the loop's body is executed. At the end of the loop's body the computer jumps back to the condition, checking it again to determine if the loop's body should execute again. If the condition is False when it is checked the loop ends, and control jumps to the next statement in the code.

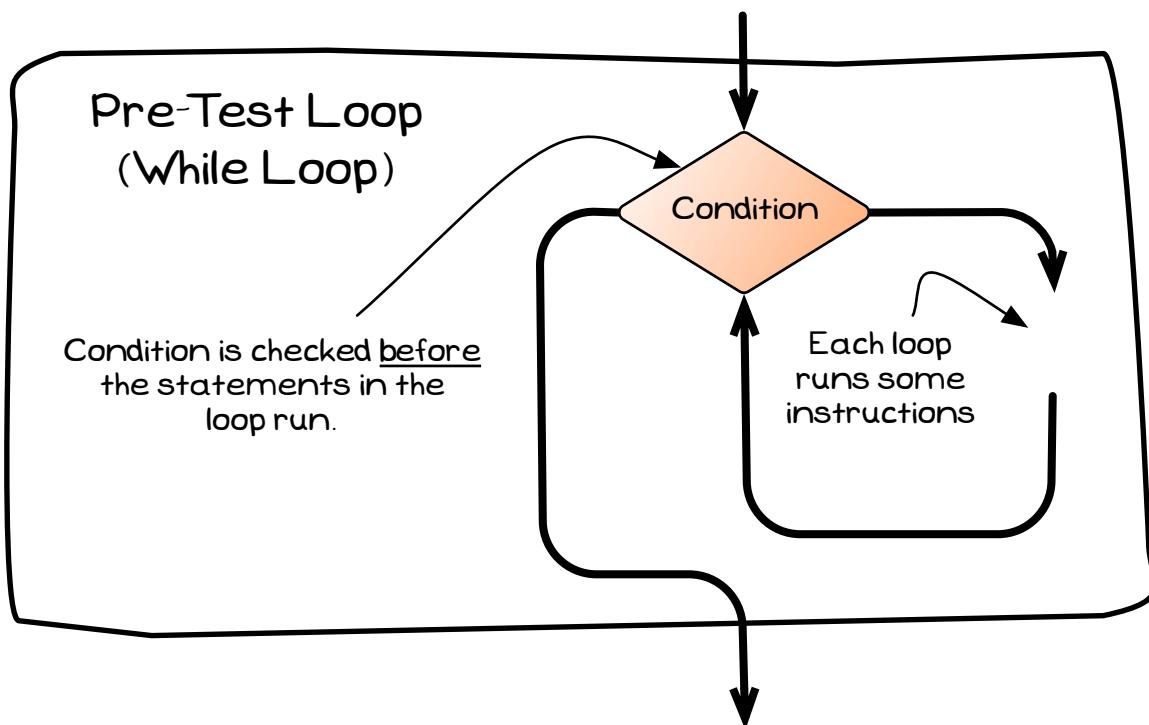


Figure 3.8: The Pre-Test Loop checks the condition, then runs the loop's body

Note

- A pre-test loop is an **action**, creating a loop in the code's sequence of instructions.
- The standard pre-test loop is the **while statement**.
- A pre-test loop allows instructions to be run 0 or more times.
- The condition is checked when the loop's code is entered, and it is checked again at the end of each loop.



Post-Test Loop

The Post-Test Loop is a looping statement that allows code to be run 1 or more times. The post-test loop places the condition after the body of the loop. This means that the first time through the body of the loop must execute before the condition is checked. When it gets to the end of the body, the loop's condition is checked and the computer either jumps back to the start of the loop to repeat the body, or the loop ends and control flows on to the next statement in the code.

There are two common variants for the post-test loop: `do...while` and `repeat...until`. These work in the same way, in that they test the condition after the loop body, but the conditions they use will be different. The `do...while` loop repeats the body of the loop when its condition is **true**, `repeat...until` repeats the body of the loop when its condition is **false**. When implementing a post-test loop you must make sure that the condition you use matches the kind of loop supported by your language.

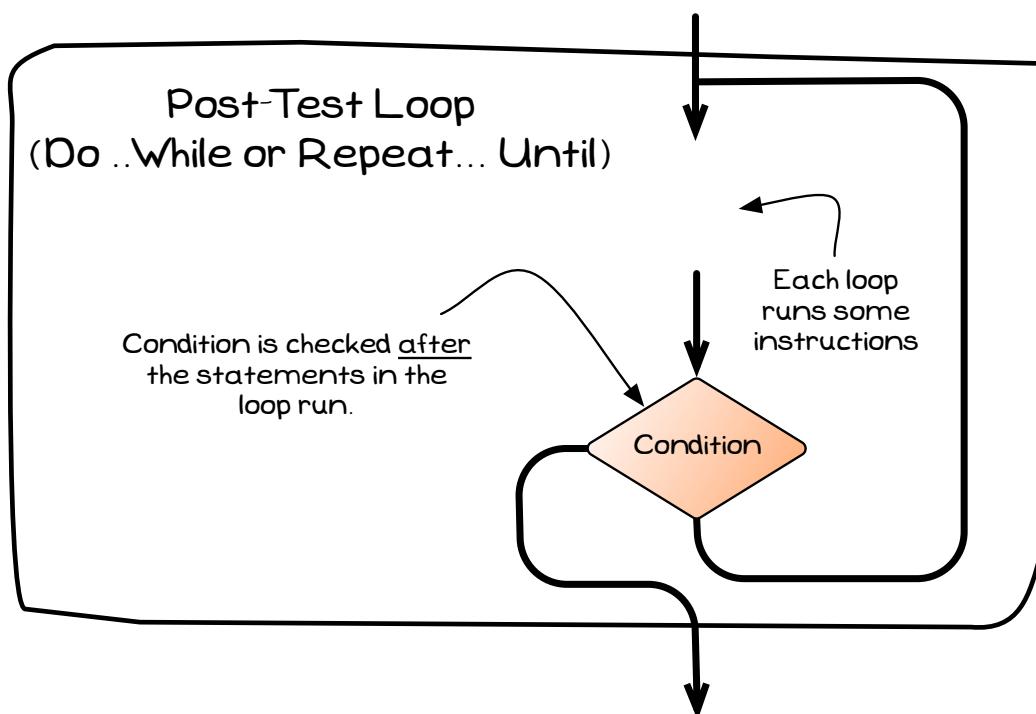


Figure 3.9: The Post-Test Loop runs the loops body, then checks the condition

Note

- A post-test loop is an **action**, creating a loop in the code's sequence of instructions.
- A post-test loop allows instructions to be run 1 or more times.
- The condition is checked after the loop's body is executed, with control jumping back to the start if needed.

C++

C includes support for the `do...while` loop.

Pascal

Pascal includes support for the `repeat...until` loop.

3.1.4 Jumping

The jump statements allow you to alter the sequence of instructions in the code, getting the computer to jump to another instruction.

Jumping

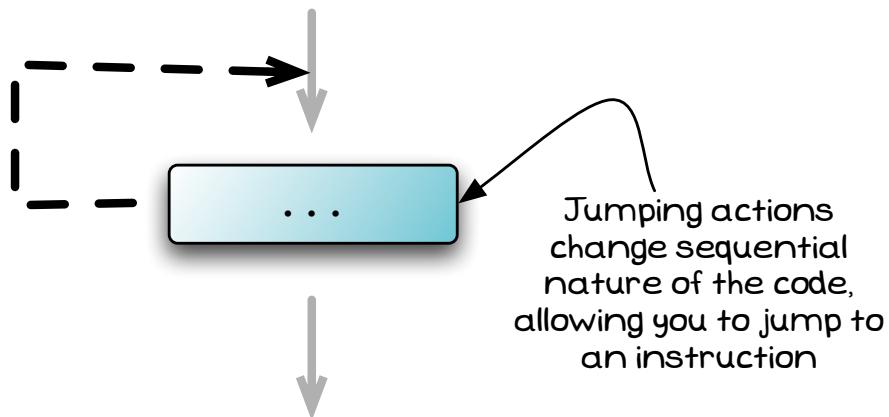


Figure 3.10: Jump Statements cause control to jump to another location in the code

Note

- The jump statements are **actions**, they allow you to alter the standard sequence of the instructions and have the computer jump to a another location in the instructions.
- **Structured Programming** was proposed as a means of providing order and structure to the control flow through the code. These jump statements complicate this sequential flow, but in some cases they are able to simplify code.
- Structured jump statements allow you to control the sequence of actions related to a **Looping** statement, a **Function**, or a **Procedure**. These work the looping and procedural structures used in structured programming.
- Unstructured jump statements allow you to jump to any instruction within the code. You need to be aware that these statements exist, but they should not be used.



Break

The break statement is used to jump out of the current loop, in effect terminating the loop early. This is useful for ending the current loop, skipping all future cycles.

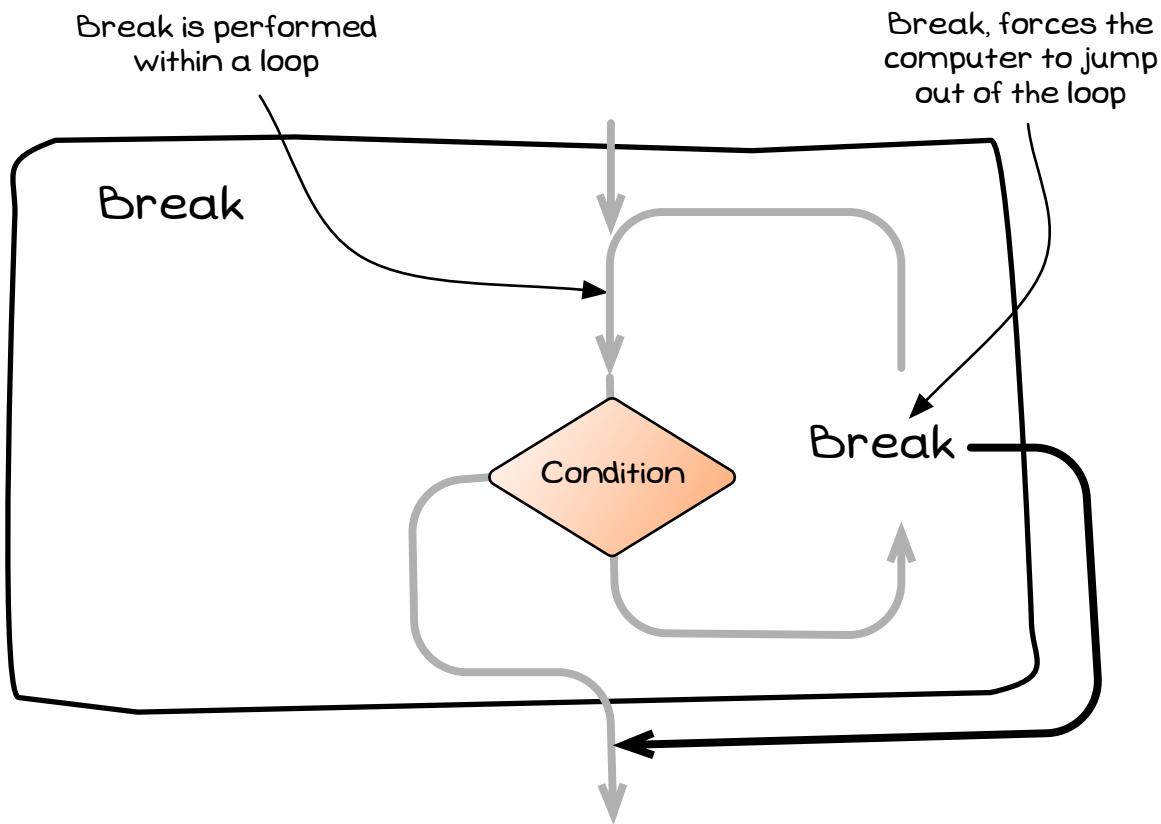


Figure 3.11: The Break Statement allows you to end a loop early

Note

- The break statement is an **action**, allowing you to jump to the end of the current loop.
- The break statement should be coded within an **Branching** statement that checks if the loop should terminate early.

Continue

The continue statement is used to jump to the condition of the current loop. This is useful for skipping the processing of the current loop, but to allow the loop to continue for the next cycle.

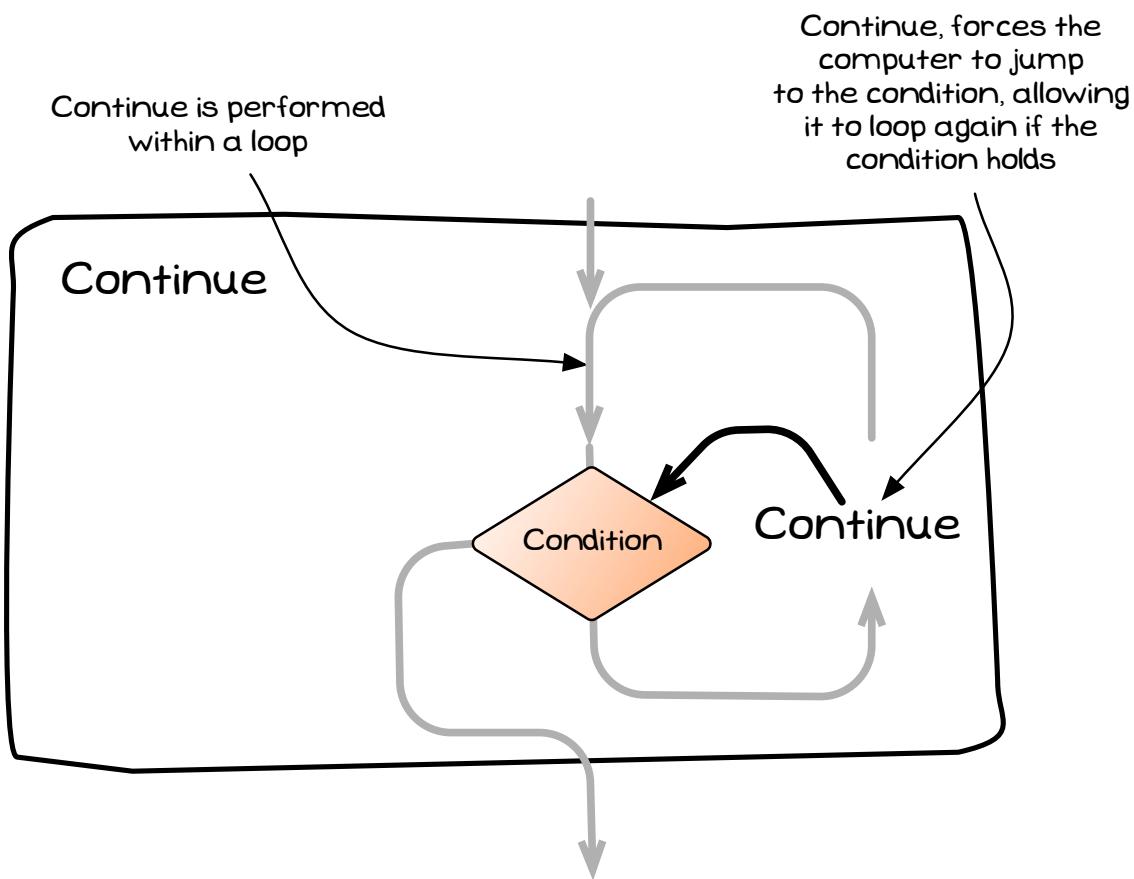


Figure 3.12: The continue Statement allows you to jump to the condition, skipping the remainder of the code in the loop but allowing the loop to continue

Note

- The continue statement is an **action**, allowing you to jump to the condition of the current loop.
- The continue statement should be coded within an **Branching** statement that checks if the loop should skip processing of the current cycle.

Exit

The exit statement, or the return in C, ends the current **Function** or **Procedure**. This is useful for skipping the rest of the processing of the Function or Procedure, exiting it early and returning to the calling code.

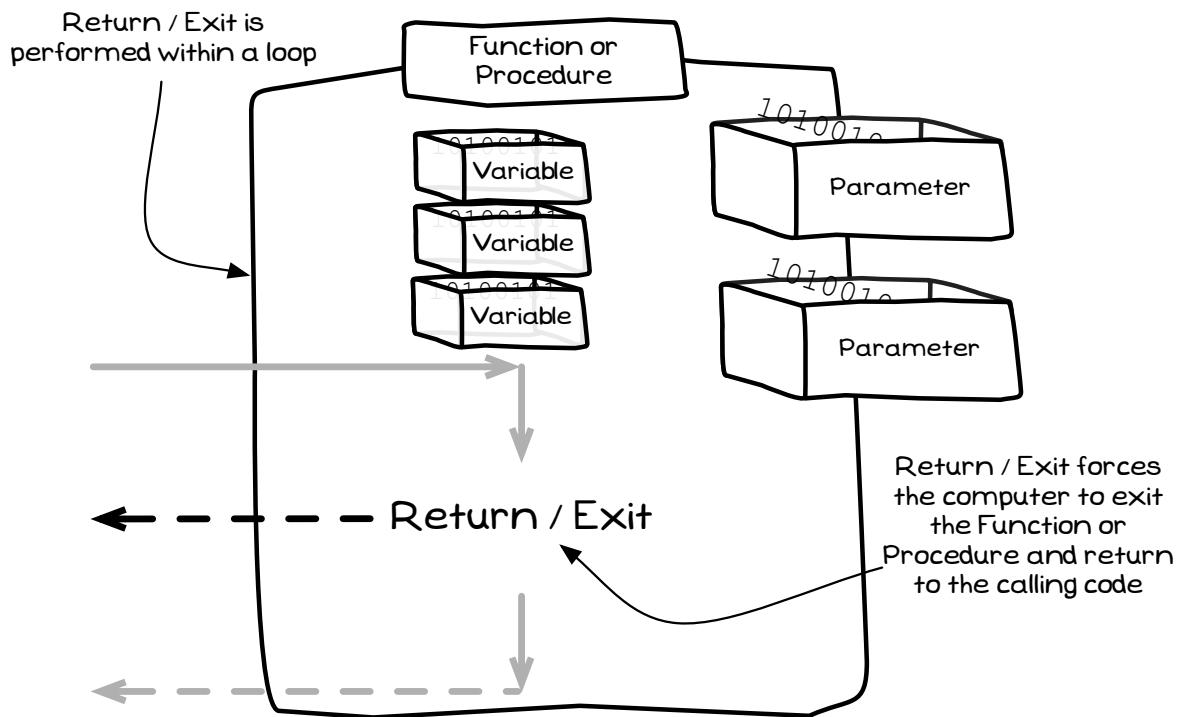
Return / Exit

Figure 3.13: Exit ends the current Function or Procedure

Note

- Exit is an **action**, allowing you to jump out of the current Function or Procedure, and return to the calling code.
- The Exit should be coded within a **Branching** statement that checks if the Function or Procedure should end.

C++

C's version of the exit statement is the **Return Statement**. The return statement also provides the value that will be returned when exiting from a Function. As this sets the value to be returned you must have a return statement as the last action within a Function.

Goto

The last jump statement is the goto statement. This is an unstructured jump, allowing you to jump anywhere in the code. Structured Programming principles called for the abolition of the goto statement. This is a statement you need to be aware of, but not one that should be used.



Figure 3.14: The dangers of using goto, from <http://xkcd.com/292/>

Note

- Goto is an action that allows you to jump to another instruction and continue from there.
- You need to be aware of the goto statement, but you should not use it.



3.1.5 Compound Statement

Branching and [Looping](#) statements need to be able to include a number of instructions within their paths. Often languages will manage this by indicating that only a *single* statement can be included in any of these paths, and then include the ability to code multiple statements in a *single compound statement*.

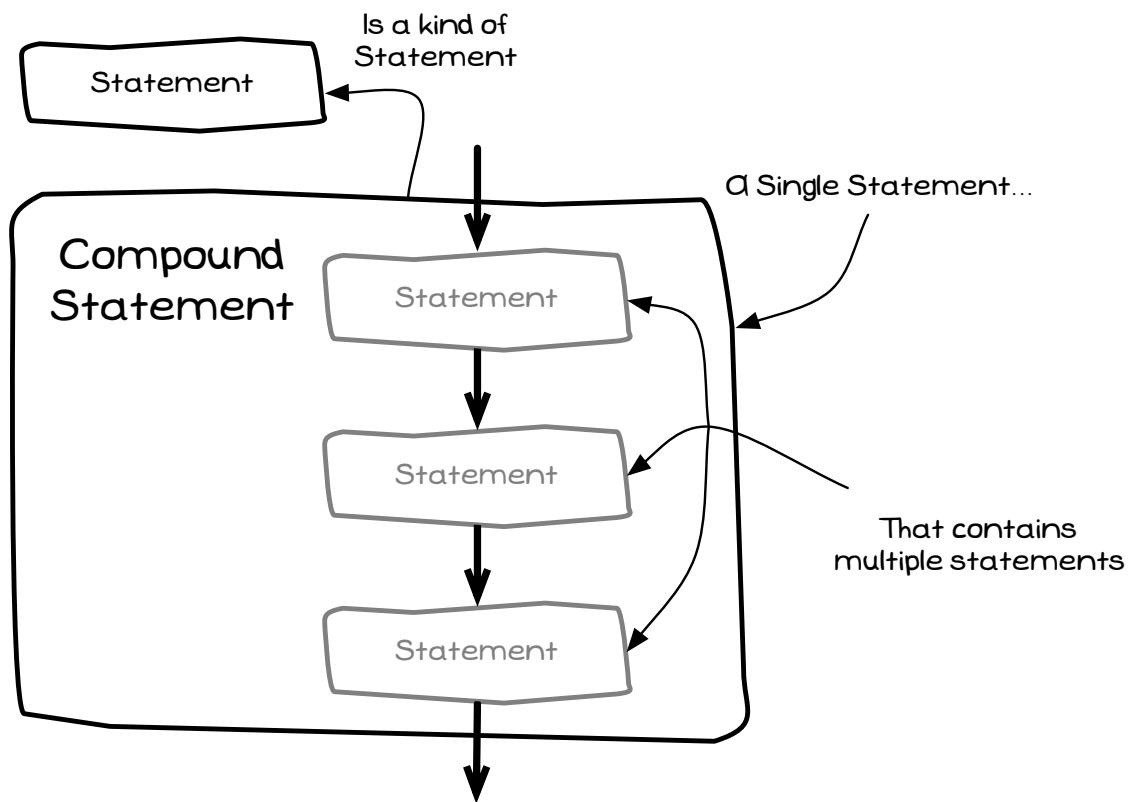


Figure 3.15: A compound statement is a statement that can contain other statements

Note

- *Compound Statement* is a **term** used to describe a way of grouping *actions*, allowing you to create a single statement that contains multiple statements.
- Compound Statements are useful when combined with [Branching](#) and [Looping](#) statements. Allowing you to put multiple statements within a path.

3.1.6 Statement (Simple and Structured)

Statements are the actions that we can get the computer to perform.

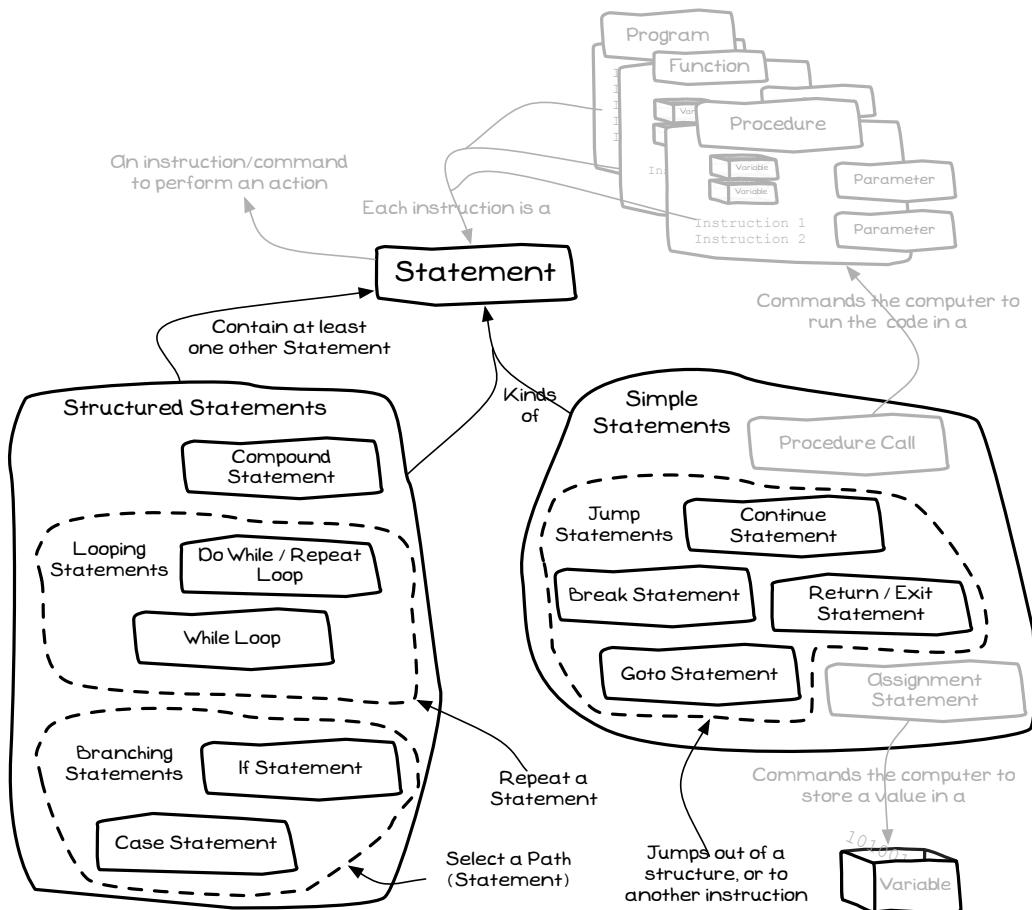


Figure 3.16: A Statement may also be a Looping or Jumping Statement

Note

- Statement is the **term** given to the instructions in code.
- **Simple Statements** that perform an action. The actions you can perform are:
 - **Procedure Call** used to run the code in a Procedure.
 - **Assignment Statement** used to calculate a value and store it in a Variable.
 - Jump Statements allow you to affect which instruction will be performed next. This includes:
 - * **Break**: to jump out of a **Looping Statement**.
 - * **Continue**: jumps to the condition in a Looping Statement.
 - * **Exit**: (return in C) to end the current Function or Procedure.
 - * **Goto**: the unstructured jump^a to an arbitrary location in the code.
- **Structured Statements** contain statements and control the flow of execution:
 - **Looping Statements**: that repeat a statement a number of times.
 - * **Pre-Test Loop**: Test condition before the body, repeating **0 to many** times.
 - * **Post-Test Loop**: Test condition after the body, repeating **1 to many** times.
 - **Branching Statement**: that select from a number of optional statements.
 - * **If Statement**: Branch based on a **Boolean Data Expression** (2 paths).
 - * **Case Statement**: Branch based on an **Ordinal Expression** (n paths).

^aOften resulting in death by Raptor.

3.1.7 Summary

This section has introduced a number of new actions that you can use in your code to create more dynamic programs.

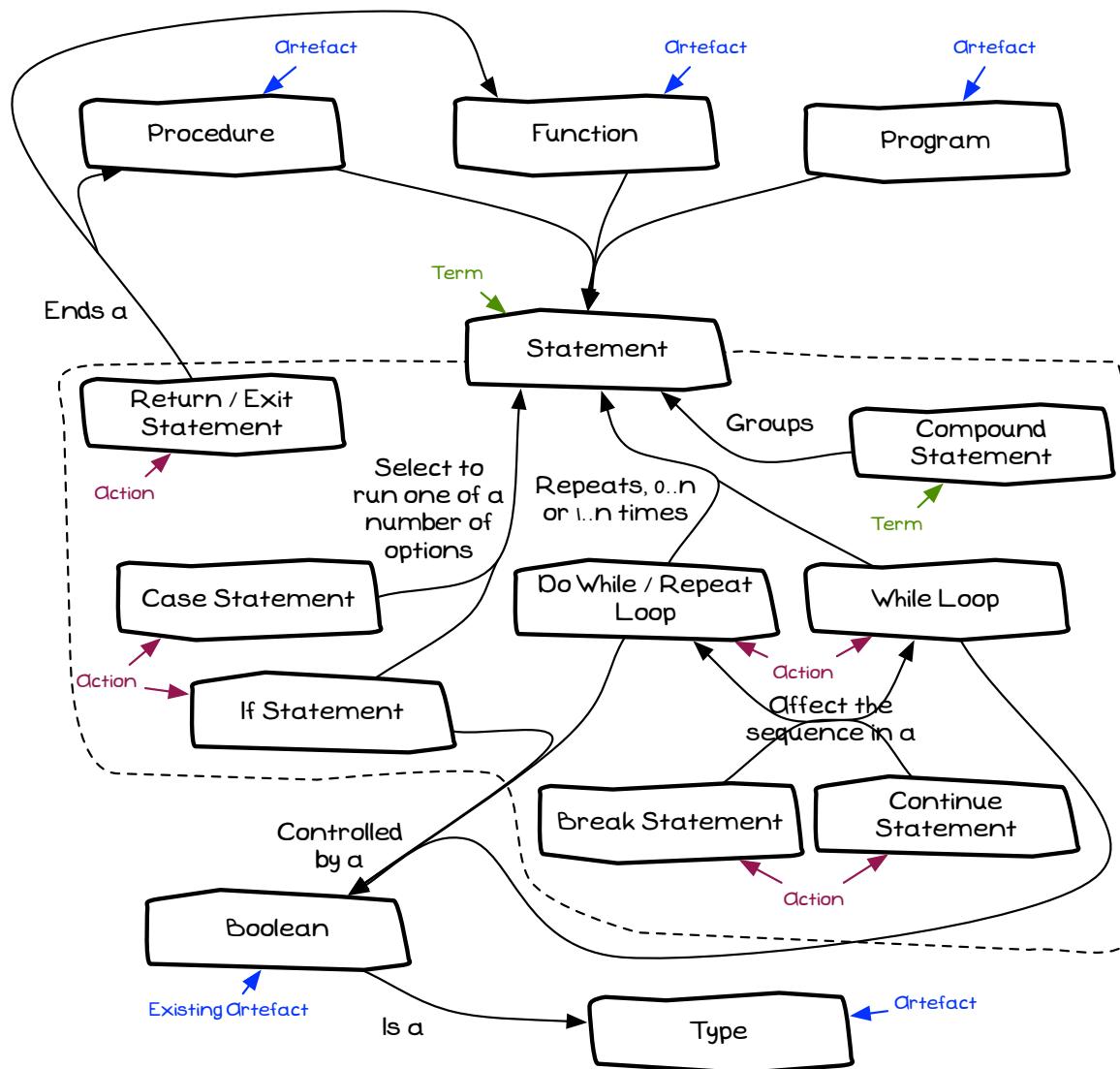


Figure 3.17: Key Concepts introduced in this Chapter

Note

- **Artefacts** are things you can *create* and *use*.
- **Terms** are things you need to *understand*.
- **Actions** are things you can *command* the computer to perform.

3.2 Using these Concepts

The Control Flow Statements enable you to alter the purely sequential order in which instructions are performed. Using these Statements you can have the computer select a path to follow, or jump back and repeat statements a number of times. These capabilities make it possible to create programs that react to the data they receive, giving more interactive results than were possible before.

3.2.1 Designing Guess that Number

Table 3.5 contains a description of the next program we are going to design. This program plays a small guessing game called *Guess that Number*. In this game the computer will think of a target number between 1 and 100. The user then has 7 guesses to determine the target number's value. At each guess the computer outputs a message to indicate if the target number is greater than, less than, or equal to the user's guess.

Program Description	
Name	Guess that Number
Description	A simple guessing game where the computer ‘thinks’ of a number between 1 and 100, and the user has seven guesses to guess it. With each guess the computer indicates if the number it is thinking of is less than, or greater than the user’s guess. If the user runs out of guesses they are shown the answer, and that round ends. After each round the user has the option to play again, or to quit the program.

Table 3.5: Description of the Guess That Number program.

As before, the process to design and implement this program will follow a number of steps:

1. Understand the problem, and get some ideas on the tasks that need to be performed.
2. Choose the artefacts we will create and use
3. Design the control flow for the procedural² artefacts
4. Map these artefacts to code
5. Compile and run the program

The new step in this process will involve designing the flow of the instructions within the program, and the Functions and Procedures that need to be created. Previously each artefact had a sequential flow of instructions, the introduction of the control flow statements has made it possible to add branches and loops to this sequence. The order of these actions needs to be considered for each sequence of instructions in your code.

²The program, and any Functions and Procedures.

3.2.2 Understanding Guess that Number

Guess that Number is a relatively common game involving one player choosing a number, and the other trying to guess it. Each guess is followed by an indication of whether the guess was less than or larger than the target number. The game continues until the player guesses the target number.

The data for this program will be relatively simple. The code will need to keep track of the current **target number** which will be an integer. The player will be able to enter a **guess**, also an integer. These two numbers represent the data model for the game. It is a game that involves a *target number* and *guess*.

The actions within the program involve **playing the game**, and **performing a guess**. Playing the game is the process of performing guesses until the player finally guesses the target number. This version of the game will only allow the player to have seven guesses, so this will need to be a part of this code in the program. The process of performing a guess will involve getting the user's guess, and giving them feedback indicating if they guessed the target number, or if they were larger than, or less than the target.

3.2.3 Choosing Artefacts for Guess that Number

Software design is all about **abstraction**. This is the process of determining the essential features of the problem and modelling that in the artefacts that you create in your code. When designing the artefacts that will make up your code you need to think about problem, and try to create artefacts in your code that represent the actions and data you imagine when thinking about the problem.

Our understanding of the Guess that Number game indicates that there are two processes that need to be performed within the program: **play game** and **perform guess**. These two processes can be coded as either Functions or Procedures in the program's code. The Play Game code can be implemented as a **Procedure**. It will be responsible for running the process of the game, starting with telling the user it has 'thought of a number', through to coordinating the guesses, ending only when the user gets the answer or runs out of guesses.

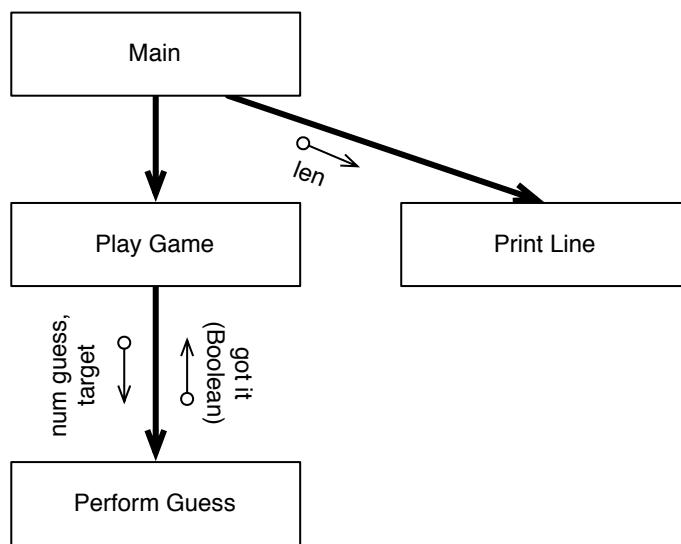
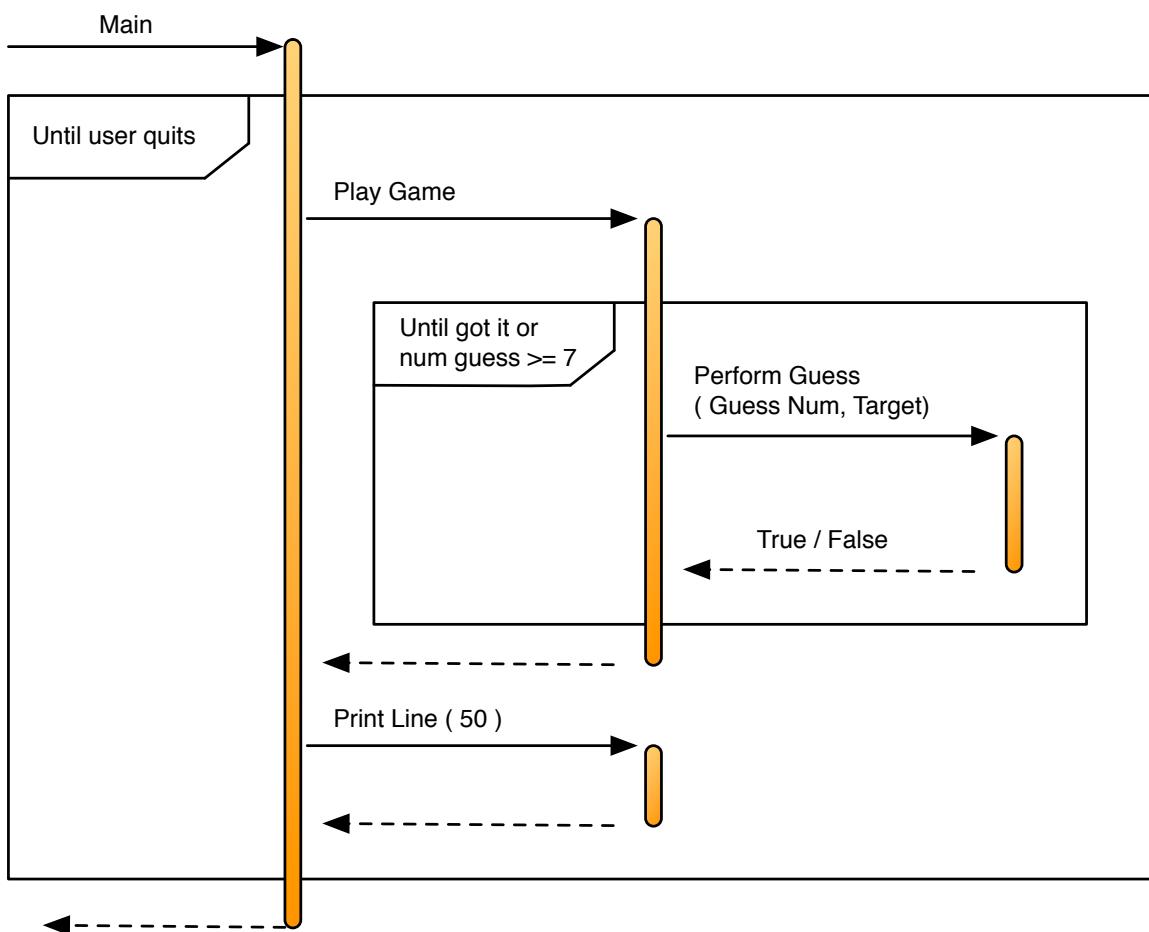
Perform Guess, on the other hand, will need to be a **Function** as it must return back a **Boolean Data** value indicating if the user has guessed the number. The code in Perform Guess will be responsible for asking the user to enter a guess, and then giving them the feedback on their guess. As this has the details of the guess, its result is needed to allow Play Game to determine if the user has guessed the number.

The Perform Guess code will also need to accept parameters to tell it what the current target value is. This data will exist within the Play Game code, so it will need a mechanism to pass that code to Perform Guess. A **Parameter** is needed in Perform Guess to accept target will enable this.

A nicety may be to allow tell the user which guess they are up to. Once again, this information is stored in Play Game, so a second parameter can be added to allow Play Game to pass in the guess number along with the target number.

In addition to these it has been decided to add a Print Line procedure to display a line of '-' characters. This will be displayed at the end of the game before the user is asked if they want to play again. A length parameter will enable the caller to indicate how many of these characters are printed on the line.

The Structure Chart showing these artefacts is shown in Figure 3.18, and the Sequence Diagram is shown in Figure 3.19. Notice that there is some relationship here between the artefacts that we are creating and the steps in the game itself. The earlier it is to see the relationship between these artefacts and the problem, the better job you have done abstracting your solution.

**Figure 3.18:** Structure Chart for the Guess that Number Program**Figure 3.19:** Sequence Diagram for the Guess that Number Program

3.2.4 Designing Control Flow for Perform Guess

Having chosen the artefacts to build, the next step is to design the control flow that will enable these Functions and Procedures, and the program itself, to achieve their goals. Each has a responsibility that it must meet in order for the overall solution to work.

This section will go through designing the logic for the Perform Guess Function. It will cover the following points:

- Drawing control flow using a Flowchart
- Use Structured Programming Principles to guide the design of the flowchart
- The Structured Programming blocks used to build the logic
- Combining blocks for the Perform Guess
- Setting the result using an Expression
- The Pseudocode for Perform Guess

Drawing control flow using a Flowchart

Before examining the control flows within the Functions and Procedures of the Guess that Number program, let us have a look at a means of visually representing flow of actions within code. A common means of doing this is to use a **flowchart**. This is a diagram that depicts a sequence of actions, and can be used to represent the sequence of action actions that occur within your code.

The flowchart has four basic symbols as shown in Figure 3.20.

- **Start/Stop**: This represents the start and end of the process. Typically the start node would have the name of the Function or Procedure within it.
- **Flow**: The arrows between nodes represent control flow, indicating which action is to be performed next in the code.
- **Process**: This node represents a task being performed. In our case this will map to one, or more, simple statements. The text in the Process node should indicate what is being performed at this stage.
- **Decision**: This represents a point where the code needs to make a decision. This is used for the conditions in the **Branching** and **Looping** statements. A decision **must** have more than one flow coming out of it, and each flow should indicate the condition that triggers that path.

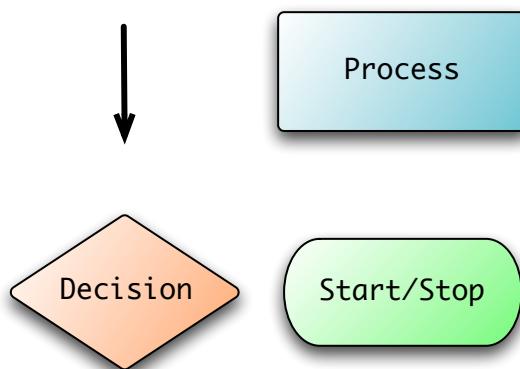


Figure 3.20: Flowchart Symbols

Use Structured Programming Principles to guide the design of the flowchart

Flowcharts can be used to represent any sequence of actions, but not all sequences of actions will be easy to code. Figure 3.21 is an example of a flowchart, but not one that can be coded using the structured statements covered in Chapter 3. Figure 3.22 shows a structured version of this same algorithm. Notice that in this version there are identifiable blocks, each having a single entry and a single exit. This structure could easily be converted to code using structured statements.

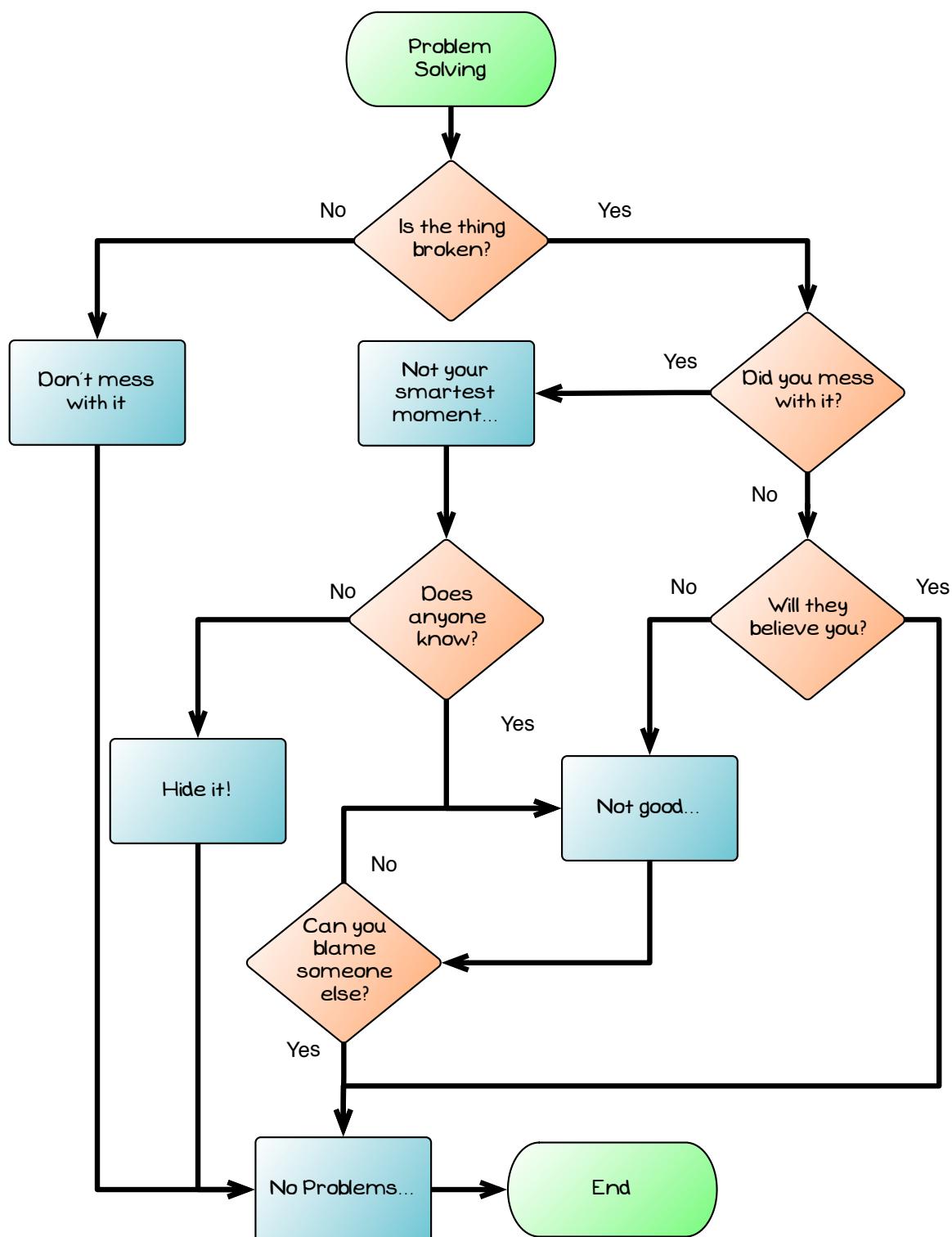
Looking at these two flowcharts, the structured flowchart looks more complicated than the unstructured version. This reflects the fact that in some cases the structured version may be more complicated, but is also an indication that this process may be better broken down into more functions and procedures. For example, the repeated loop looking for someone else to blame could be placed in its own procedure. If you find it difficult to design the control flow using structured blocks try to see if you can identify additional Functions and Procedures to help you break the code down into smaller blocks.

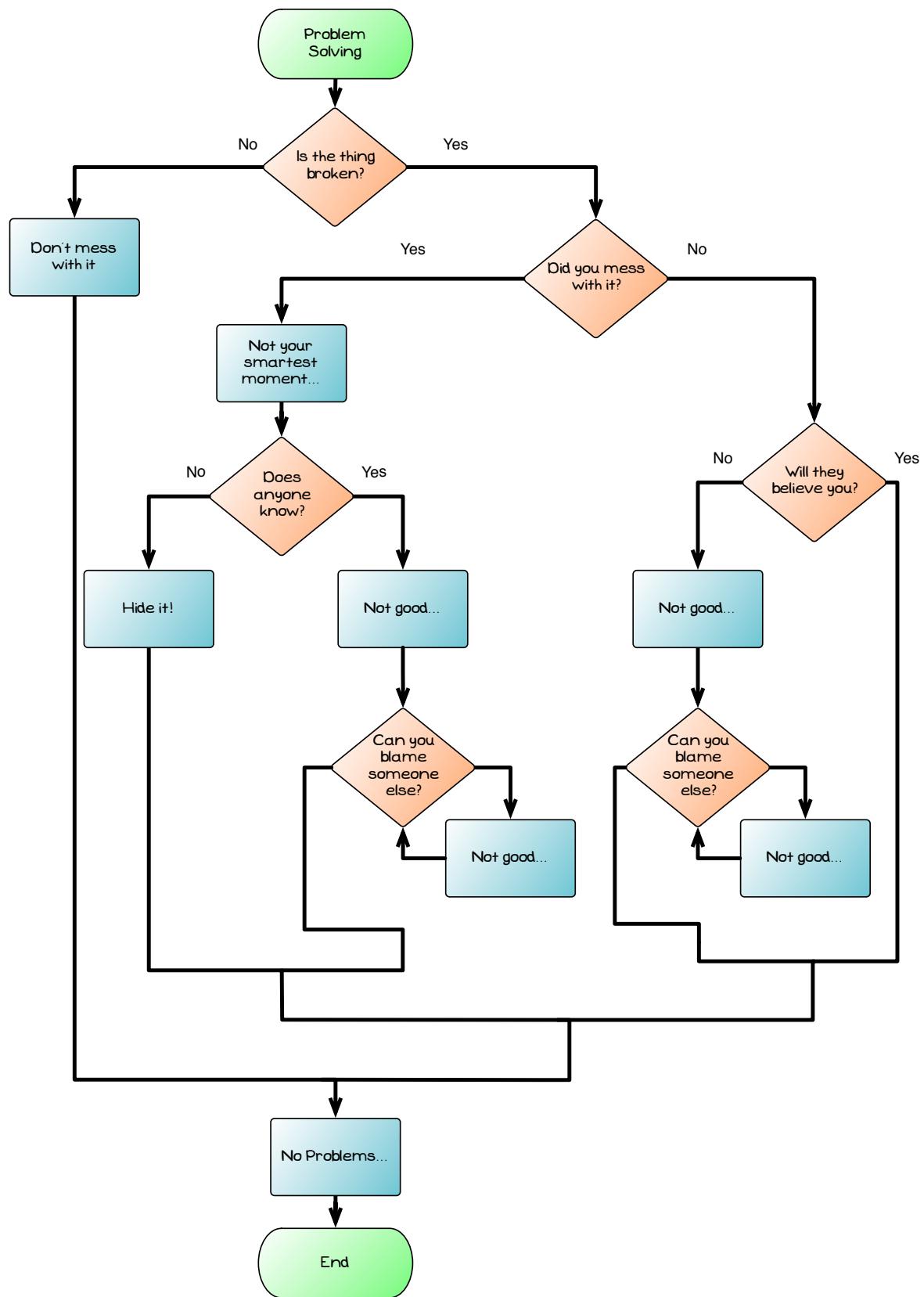
The structured programming principles indicate that code within a Function or Procedure should be organised into **blocks**. These blocks match to the structured statements in modern programming languages: the [Branching](#) and [Looping](#) statements. Each of these blocks has a **single entry point** and a **single exit point**, allowing them to combined together.

The addition of [Jumping](#) statements to the language allows the blocks to have **multiple exit points**; one at the end of the block's code, the other at a [Break](#), [Exit](#), or [Return Statement](#). These statements give you extra flexibility, but still work within the structured programming principles.

Note

Prior to the Structured Programming, programs had two control flow mechanisms: [if](#) and [Goto](#). Control flow was not easy to picture or understand. Programs written in this way are now known as **spaghetti code**, as understanding this code is much like trying to untangle a bowl of spaghetti, though not nearly as tasty.

**Figure 3.21:** Unstructured Flowchart

**Figure 3.22:** Structured version of Figure 3.21

The Structured Programming blocks used to build the logic

In Structured Programming there are three kinds of blocks:

- **Sequence:** one instruction follows the next in a sequence.
- **Selection:** the ability to branch the sequence.
- **Repetition:** repeat a block a number of times.

The flowchart snippets in the section on [Branching](#) and [Looping](#) showed how the flows work for *selection* and *repetition*. Figure 3.24 shows the three alternatives for selection blocks, Figure 3.25 shows the two alternatives for repetition blocks, and Figure 3.23 shows the standard sequence flow. The design of any program's logic is a task in combining these blocks together to perform the desired effect.

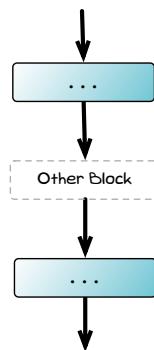


Figure 3.23: Flows for Sequence Blocks

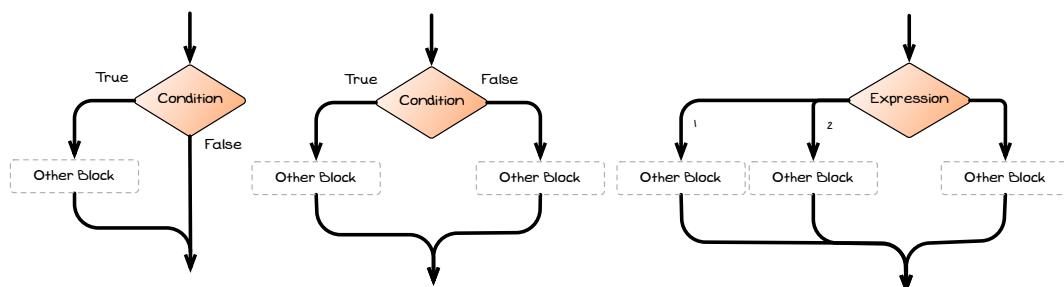


Figure 3.24: Flows for Selection Blocks

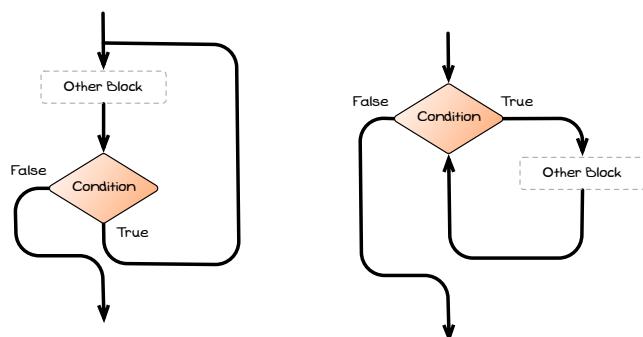


Figure 3.25: Flows for Repetition Blocks

Combining blocks for the Perform Guess

With the basic theory at hand, we can now start to design the control flow for the Guess that Number program. This process will involve, once again, the idea of **abstraction**. When designing the flow for a program you first need to be able to perform the process yourself, even if its just on paper, and then work out the steps that you undertook so that you can code these within the program.

For the Guess that Number program we can start by designing the control flow within the Perform Guess function. The specification of this is shown in Table 3.6. Think about the steps that need to be performed to achieve this. If you had been asked to do this what would you need to do?

Function	
Perform Guess	
Returns	
Boolean	True when the user has guessed the number, False otherwise.
Parameter	Description
Guess Number	The number of the current guess, used in the prompt asking for the user to enter their guess.
Target	The number the user is aiming to guess.
Perform Guess is responsible for coordinating the actions needed to perform a single guess within a game of <i>Guess that Number</i> . The user's guess is read, and the value checked against the target value. A message is then output telling the user if the target value is less than, larger than, or equal to their guess. This function returns True when the user's guess is equal to the target.	

Table 3.6: Specification for the Perform Guess Function.

The first task the Function needs to perform is to get the guess from the user. This can be performed in a **sequence**: display a prompt, read the value from the user. This first sequence is shown in Figure 3.26.

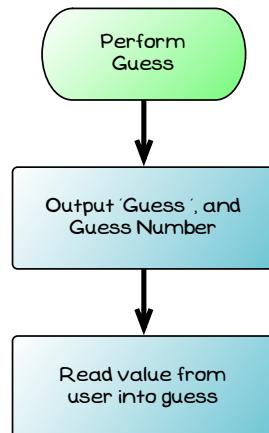


Figure 3.26: Initial Sequence in Perform Guess

The next step in this sequence is to give the user feedback based upon their guess and the target number. This code requires a the ability to *select* a given branch. The computer needs to output different messages based upon the users guess. This can be achieved with a **selection** block. Looking back at Figure 3.24 there are three possible alternatives for implementing this selection. The if with no else is not a valid option as there are three paths we need to take. The case block is also not valid as we are not matching a value, but comparing values to each other. The last option is the if-else block, but this only has two branches. It is not going to be possible to code all three options within one block, but it can be achieved using two if-else blocks.

The first if-else block will check if the target is greater than the user's guess. If this is true then the computer can take the first branch and output the message 'The number is larger than ' and the value from the user's guess. The flow chart for this part is shown in Figure 3.27. This block is the third task in the sequence, this if block has a single entry, causes a branch in the flow, and will have a single exit.

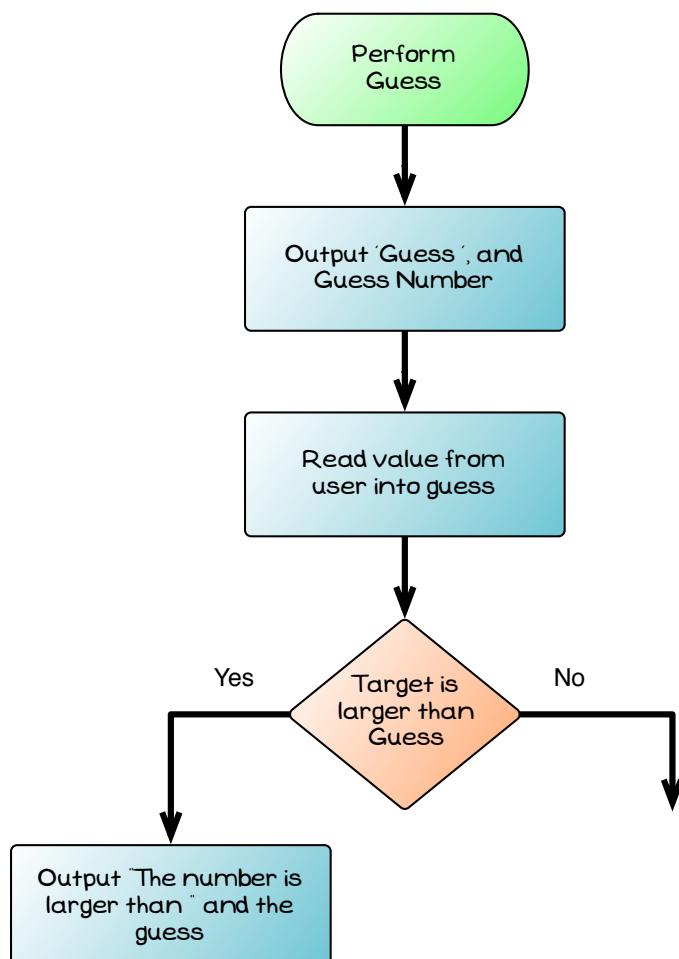


Figure 3.27: First branch in Perform Guess

Note

- The conditions within the **If Statement** are Boolean Expressions.
- This condition is checking if `target > guess`.
- There are now two paths through this code, one when `target is > guess`, and another when it is not.

Following the false path from the first decision, and we have a new location into which to insert a block. At this point we know the target is *not* larger than the user's guess. At this point you can include another if-else block to check if the target is **less than** the user's guess. This is shown in Figure 3.28.

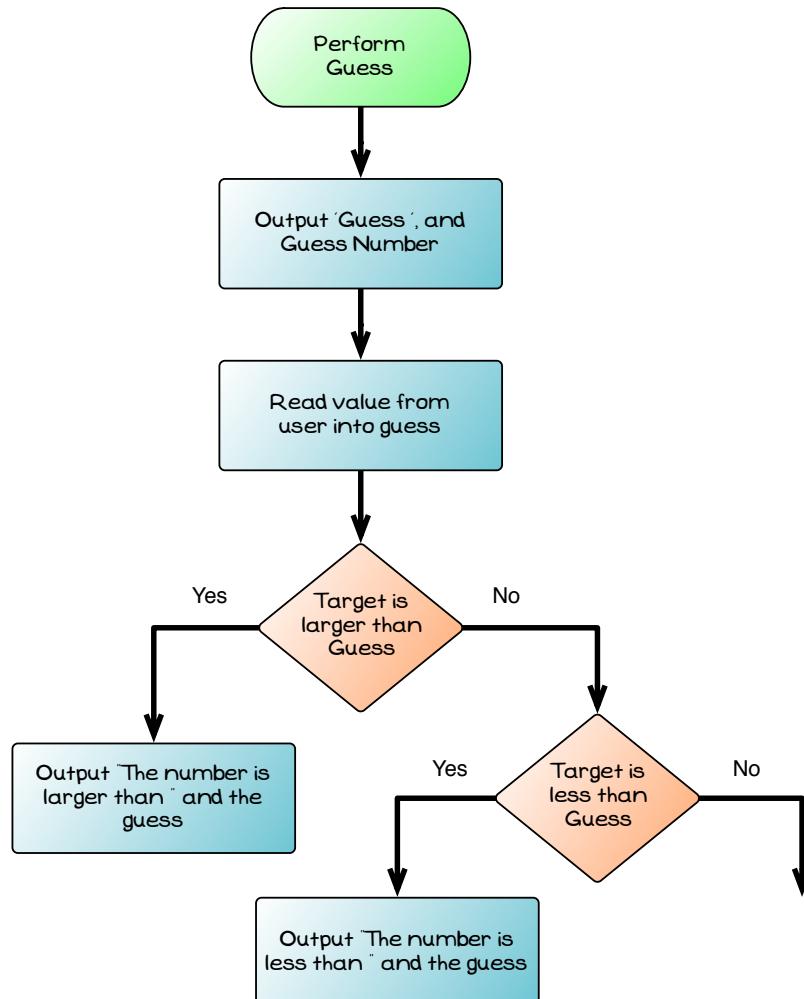


Figure 3.28: Second branch tests if the Target is less than the guess

Note

- The newly added block is another **If Statement**.
- This is nested within the **else** branch of the first If Statement.
- This code checks *if target < guess*.
- There are now three paths through this code.



Taking the false path again, and now we have a location at which the target value *must be* equal to the user's guess. The first condition checked if the target was larger than the guess, which it was not. The second condition checked if the target was less than the guess, which it was not. So the only way this can be the case is if the target and the guess are equal. This path can then be used to output the 'Well done...' message. This is shown in Figure 3.29.

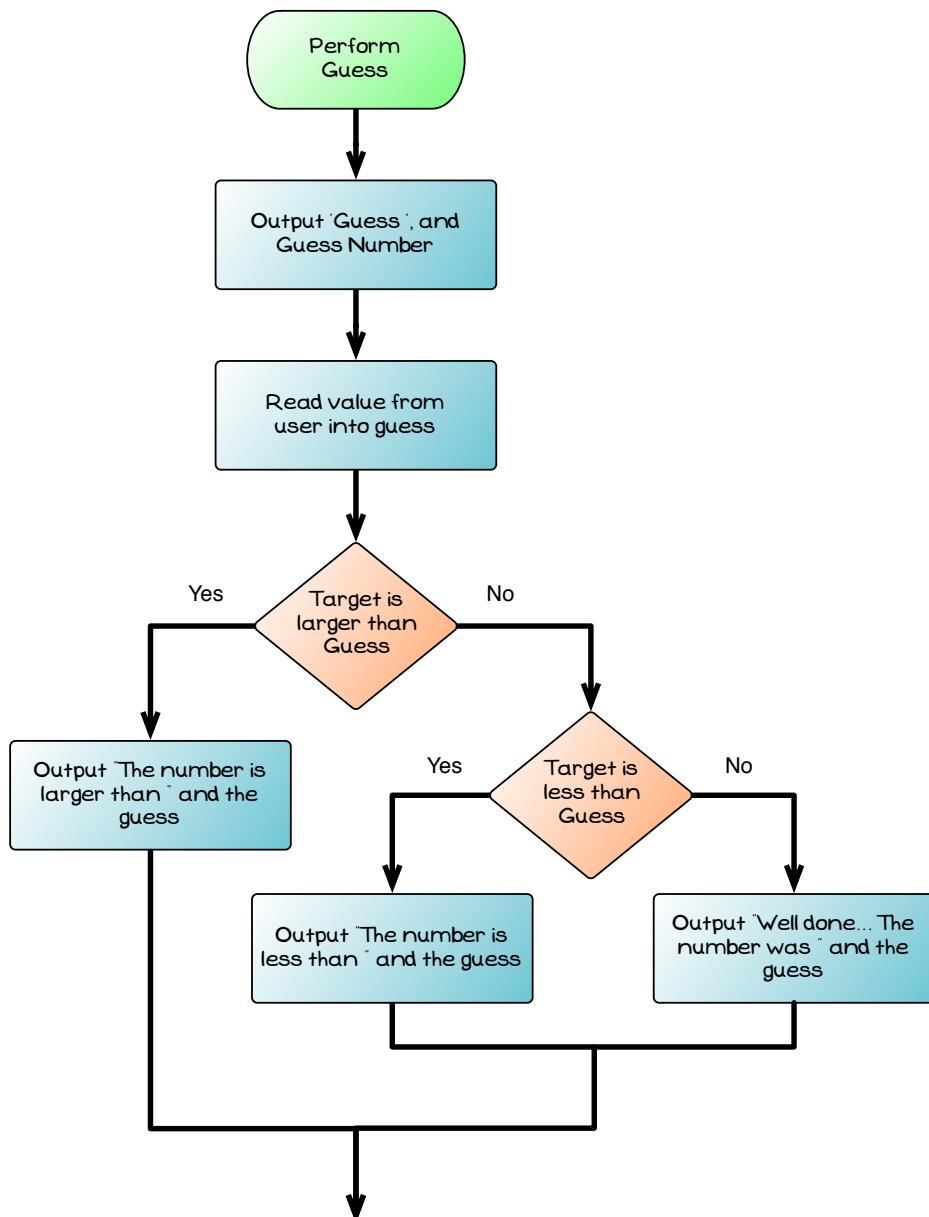


Figure 3.29: The 'Well done...' message can be output on the third path

Note

- The newly added block is a sequence, outputting the 'Well done...' message.
- On this third path the target and guess must be equal.
- Notice the single exit out of the second branch, when then flows to the single exit out of the first branch.

The last action in the code is to return a Boolean result indicating if the user's guess is equal to the target number.

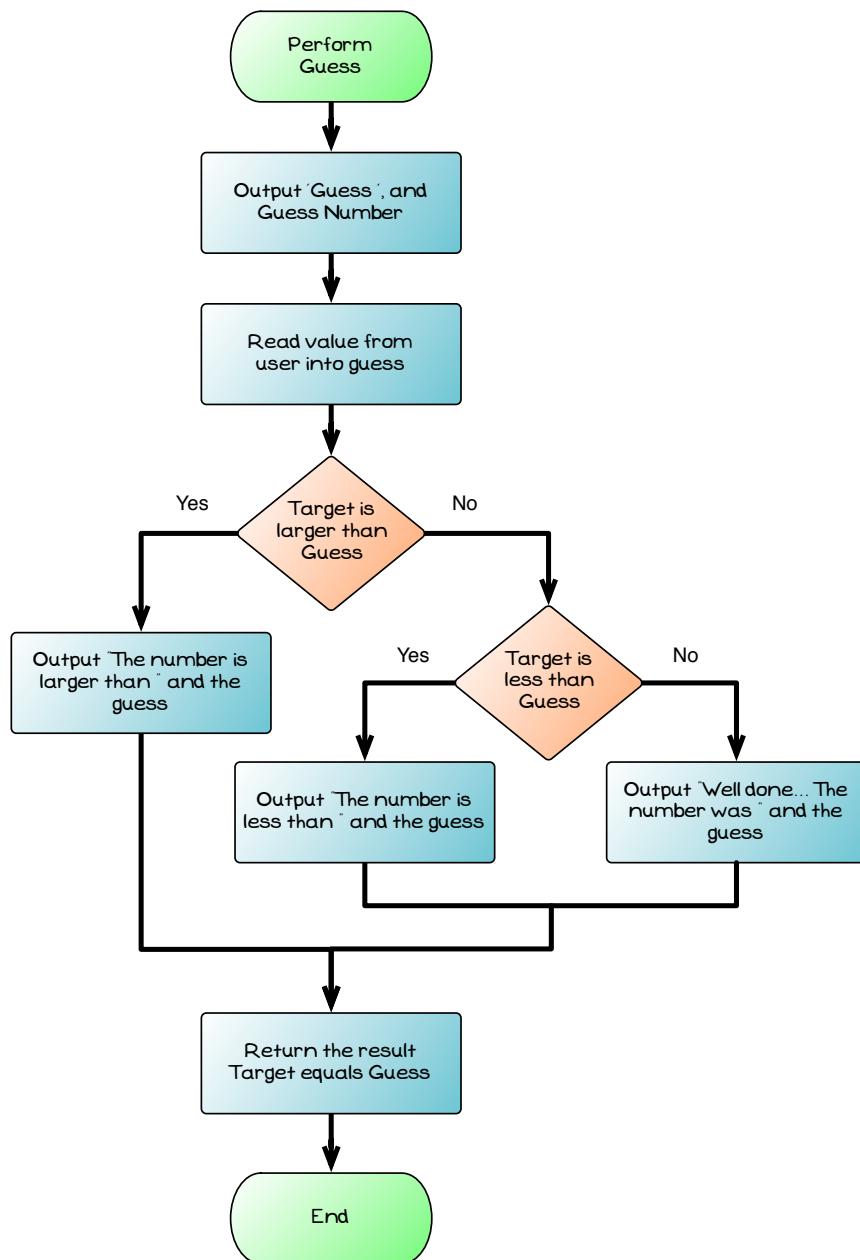


Figure 3.30: A Boolean result is returned from the Function

Note

- The newly added block is a single action, defining the result to be returned.
- This action returns the result True when the target and the guess are equal.



Figure 3.31 shows the flowchart with annotations highlighting the different blocks within the code. The function starts off with a **sequence** that contains all of the code in the Function. Within this there is the **selection**, that internally contains another **selection**.

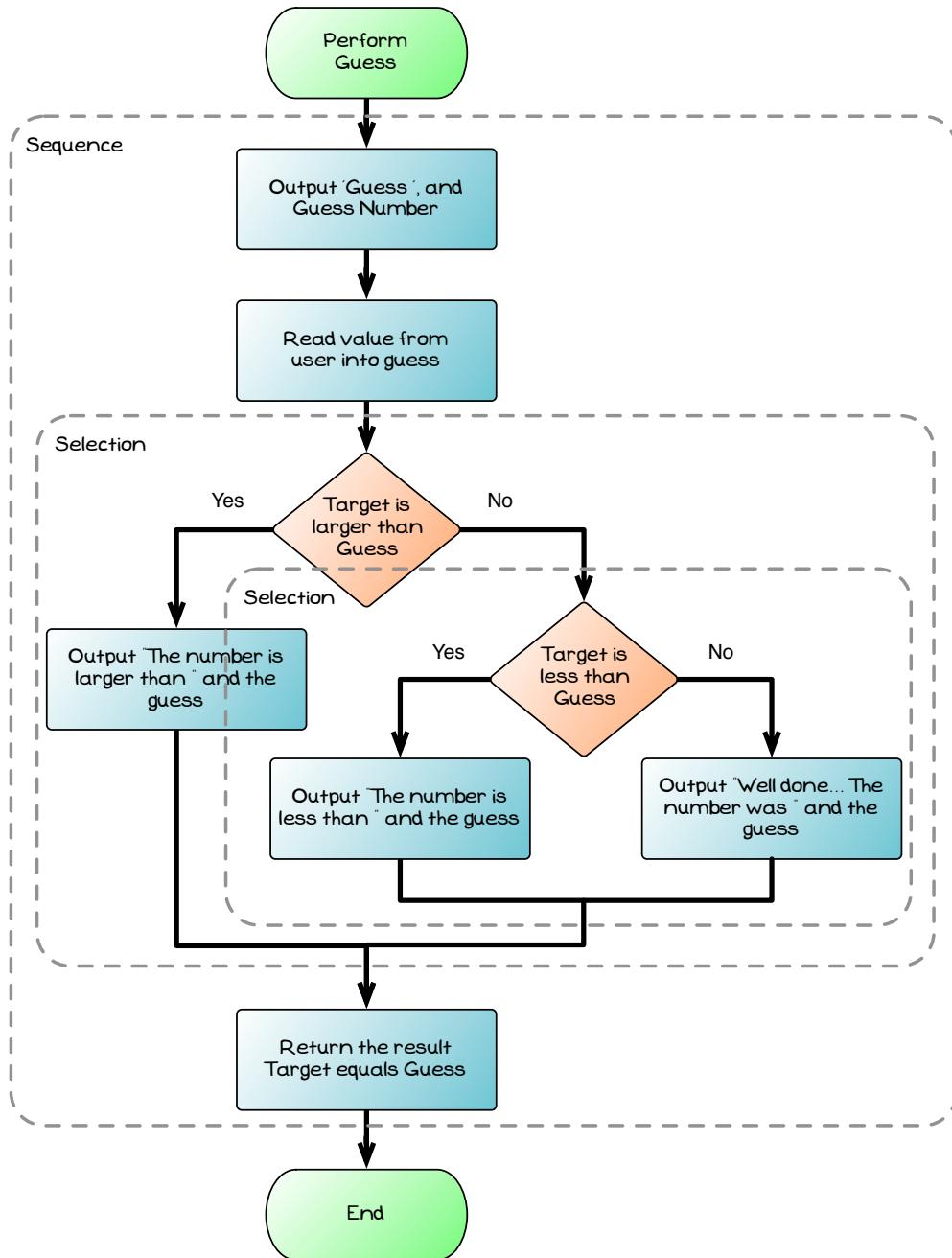


Figure 3.31: Blocks in the Perform Guess code

Note

- Notice that each block has a single path going into it, and a single path coming out.

Setting the result using an Expression

Figure 3.32 shows the *trick* that is being performed at the end of Perform Guess's code. Perform Guess needs to return a result indicating if the user has guessed the number or not. This will be a Boolean value, with True indicating they guessed the number. Initially it may seem that you need a **selection** block to enable this, as shown on the right of Figure 3.32.

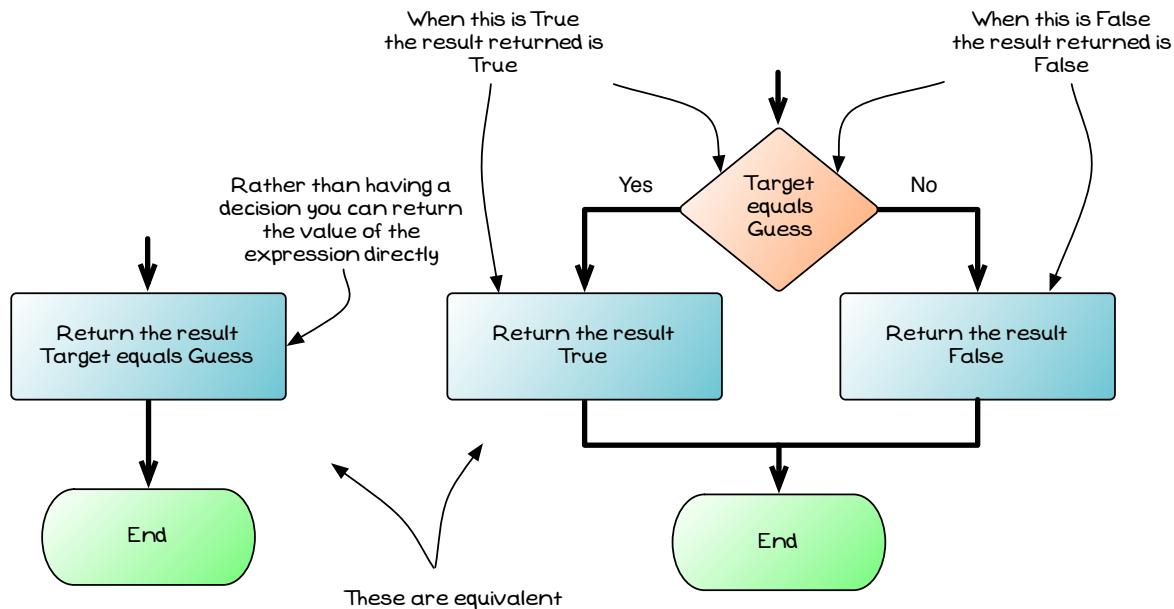


Figure 3.32: Calculating Perform Guess's result

Note

- In *most* cases it is better to have less code if possible, as long as this does not obscure the purpose of the code.
- This is an example of replacing *actions* with *data*. The more *intelligence* you can build into the data in your programs the more flexible they will be.

C++

This is achieved in C using `return target == guess;`

Pascal

This is achieved in Pascal using `result := target = guess;`

The Pseudocode for Perform Guess

Listing 3.1 contains the Pseudocode for the Perform Guess logic from the flowchart in Figure 3.30. Notice how the indentation in this mirrors the block structures in the flowchart. It is good practice to indent your code in this way as it helps you, and any person who reads your code, to see the structure of the logic. You will be able to avoid many errors by making sure that you always indent your code so that it highlights the code's structure.

Pseudocode

```
Function: Perform Guess
-----
Returns: Boolean - True if the user has guessed the Target
Parameters:
1: Num Guess (Integer) - The number of the guess (1..7)
2: Target (Integer) - The target the user is aiming for
Steps:
1: Output 'Guess ', num_guess, and ': '
2: Read input into guess
3:
4: if target is less than guess then
5:     Output 'The number is less than ', guess
6: else
7:     if target is larger than guess then
8:         Output 'The number is larger than ', guess
9:     else
10:        Output 'Well done... the number was ', guess
11: Return the result, target = guess
```

Listing 3.1: Pseudocode for Perform Guess



Note

- Code indentation makes it easier to read, and helps locate many common issues.
- Tab your code in within a **structured statement**.
 - Indent the code in the branches of an **If Statement** and **Case Statement**.
 - Indent the code within the body of the **While Loop** and the **Do While** or **Repeat Until** loops.
- Make this a habit. When you code a **Branching** or **Looping** statement automatically indent the next line of code.
- Always keep your code neat, make it look good.
- The C code for Perform Guess is shown in Listing 3.2.
- The Pascal code for Perform Guess is shown in Listing 3.3.
- Notice how the two code samples are laid out in a similar way. The indentation makes it easy to identify which statements are associated with each of the branches through the Function.



C++

```

bool perform_guess(int num_guess, int target)
{
    int guess;

    printf("Guess %d: ", num_guess);
    scanf("%d", &guess);

    if (target < guess)
    {
        printf("The number is less than %d\n", guess);
    }
    else
    {
        if (target > guess)
            printf("The number is larger than %d\n", guess);
        else
            printf("Well done... the number was %d\n", guess);
    }

    return target == guess;
}

```

Listing 3.2: C code for Perform Guess

Pascal

```

function PerformGuess(num_guess, target: Integer): Boolean
var
    guess: Integer;
begin
    Write('Guess ', num_guess, ': ');
    Readln(guess);

    if target < guess then
    begin
        writeln('The number is less than ', guess);
    end
    else
    begin
        if target > guess then
            writeln('The number is larger than ', guess)
        else
            writeln('Well done... the number was ', guess);
    end;

    result := target = guess;
end;

```

**Listing 3.3:** Pascal code for Perform Guess

3.2.5 Designing Control Flow for Play Game

The Play Game Procedure is another artefact that will require some thought to design its logic. Table 3.7 contains the specification for this Procedure. It will be responsible for coordinating the actions of the game, while Perform Guess coordinates the actions for a *single* guess.

Procedure
Play Game
Play Game is responsible for coordinating the actions involved in playing a single game of <i>Guess that Number</i> . Initially the computer will generate a random target value, and output starting text. Then it will repeatedly ask the user to guess the number, until either the user has guessed the value or they have run out of guesses. If the user does run out of guesses then the computer ends the game, and tell the user the target value.

Table 3.7: Specification for the Play Game Procedure.

The implementation of this Procedure will require us to store some data. The following [Local Variables](#) will be needed to store data within this Procedure:

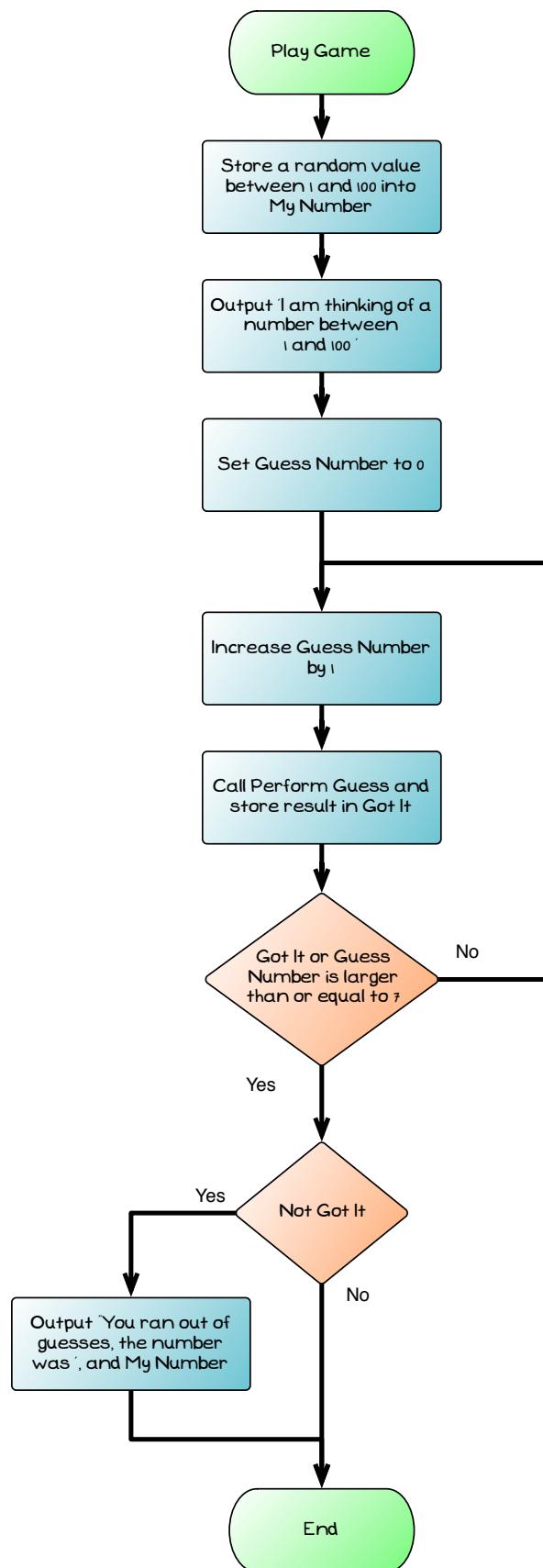
- **My Number:** This will store the computer's randomly chosen number.
- **Guess Number:** This will store the current guess the user is at, allowing the computer to stop looping when the number of guesses is exceeds 7.
- **Got It:** A Boolean value to indicate if the user did guess the number, allowing the computer to stop looping when the user guesses the number.

The flowchart for this is shown in Figure 3.33, and again in Figure 3.34 with its blocks highlighted. This code uses a **repetition** to ask the user to perform up to 7 guesses. The condition on this loop occurs *after* the loop body as the user must have at least one guess.

There is also a **selection** after the loop to output the answer if the user ran out of guesses. This is only done when the user has not guessed it themselves. This does not need to perform any other actions when the user did guess the number, so the False branch has no additional actions. In code this would be implemented with an [If Statement](#) without an **else** branch.

Note

- A loop may have its condition before or after its instructions.
- If the condition is after the loop, as it is with Play Game, then this gives a loop that runs at least once.
- The **else** branch on the If Statement is optional. If you have no processing for this branch you leave the **else** off in the code.
- The condition of the If Statement in Play Game check if the Got It is False using the not logical operator.

**Figure 3.33:** Logic for the Play Game Procedure

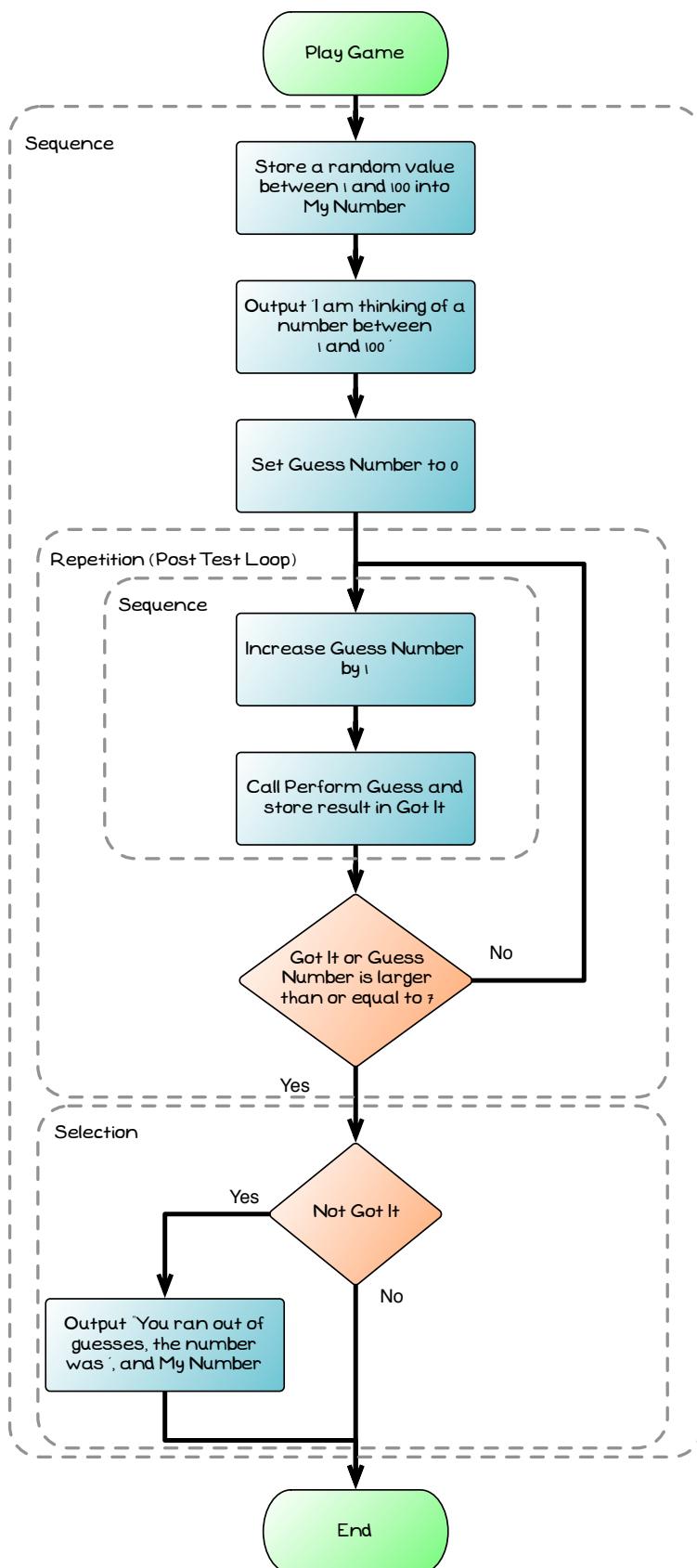


Figure 3.34: Blocks in the Play Game code

Listing 3.4 contains the Pseudocode for the Play Game Procedure. This uses Constants for MAX_NUMBER and a MAX_GUESSES, this will make it easier to change the range of the numbers and the associated number of guesses.

The flowchart for Play Game includes a **Post-Test Loop**, which can be coded as either a do...while or a repeat...until loop. This two variations are shown in step 7 of Listing 3.4, with the while version appearing as a comment on the following line. Both of these versions have the same result, but do require different conditions. The two implementations of this are shown in Listing 3.5 and Listing 3.6. It is important to understand that the basic ideas of a **post-test loop** is the same regardless of whether it is coded using do...while or repeat...until.

Pseudocode

```

Procedure: Play Game
-----
Local Variables:
* My Number, Guess Number (Integer)
* Got It (Boolean)
Steps:
1: Assign My Number, a Random number between 1 and MAX_NUMBER
2: Assign to Guess Number, the value 0
3: Output 'I am thinking of a number between 1 and ', and MAX_NUMBER
4: Repeat
5:     Increase Guess Number by 1
6:     Assign Got It, the result of PerformGuess(Guess Number, My Number)
7: Until Guess Number >= MAX_GUESSES or Got It
(* While Guess Number < MAX_GUESSES and not Got It *)
8: If Not Got It then
9:     Output 'You ran out of guesses... the number was ', and My Number

```

Listing 3.4: Pseudocode for Play Game



Note

- Notice the indentation make it easier to see which instructions are within the loop and branches of this code.
- The C code for this is in Listing 3.5.
- The Pascal code for this is in Listing 3.6.
- The C and Pascal libraries have different Functions for getting random numbers.



C++

In C there is a `random()` function in the `stdlib.h` header file. This returns a random integer value. To get this between 1 and MAX_NUMBER you can use the modulo operator (%) that returns the remainder after division. The expression to use is `random() % MAX_NUMBER + 1`



Pascal

In Pascal there is a `Random` function in the `System` unit. This takes a single parameter representing the number of random values to generate (0..n-1). The expression to use is `Random(MAX_NUMBER) + 1`



C++

```
void play_game()
{
    int my_number, guess_num;
    bool got_it;

    my_number = random() % MAX_NUMBER + 1;
    guess_num = 0; //Keep track of the number of guesses

    printf("I am thinking of a number between 1 and %d\n\n", MAX_NUMBER);

    do
    {
        guess_num++;
        got_it = perform_guess(guess_num, my_number);
    }
    while( guess_num < MAX_GUESSES && !got_it);

    if ( !got_it )
    {
        printf("You ran out of guesses... the number was %d\n", my_number);
    }
}
```

Listing 3.5: C code for Play Game**Pascal**

```
procedure PlayGame();
var
    myNumber, guessNum: Integer;
    gotIt: Boolean;
begin
    myNumber := Random(MAX_NUMBER) + 1;
    guessNum := 0; //Keep track of the number of guesses

    WriteLn('I am thinking of a number between 1 and ', MAX_NUMBER);

    repeat
        guessNum += 1;
        gotIt := PerformGuess(guessNum, myNumber);
    until (guessNum >= MAX_GUESSES) or (gotIt);

    if not gotIt then
    begin
        WriteLn('You ran out of guesses... the number was ', myNumber);
    end;
end;
```

**Listing 3.6:** Pascal code for Play Game

3.2.6 Designing Control Flow for Print Line

Print Line is a short Procedure used to print a line of ‘-’ characters to the Terminal. The flowchart for this Procedure is shown in Figure 3.35, and again with the blocks highlighted in Figure 3.36.

Procedure	
Print Line	
Parameter	Description
Length	The number of characters to print. Represents the length of the line.
Prints a number of ‘-’ characters to the Terminal. The number of characters to print is specified in the Length parameter.	

Table 3.8: Specification for the Print Line Procedure.

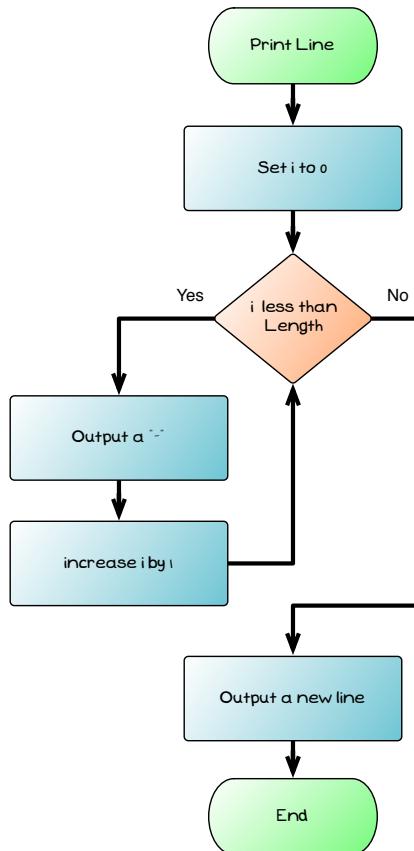
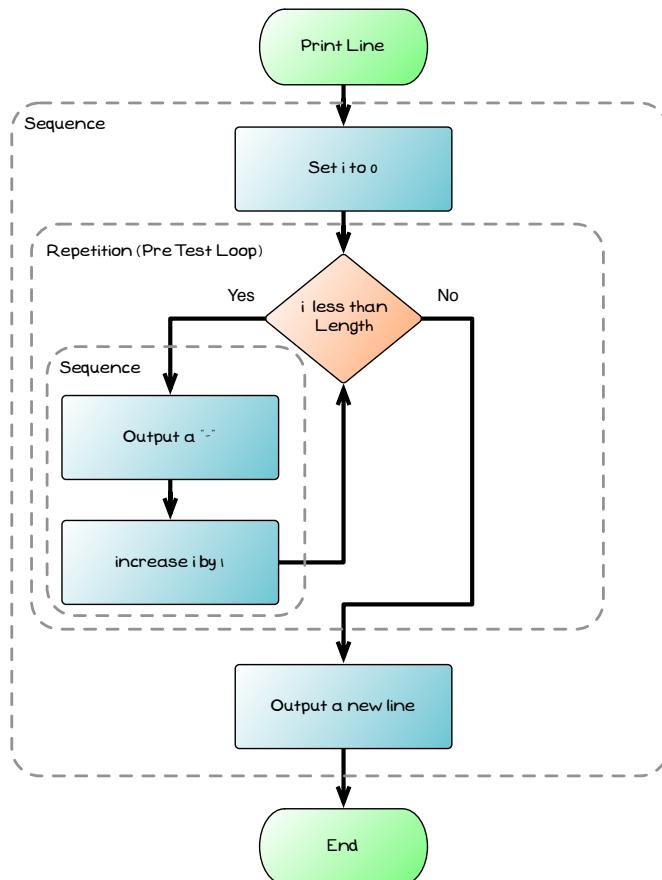


Figure 3.35: Flowchart for the logic in Print Line code

**Figure 3.36:** Blocks in the Print Line code**Pseudocode**

```

Procedure: Print Line
-----
Parameters:
1: Length (Integer)
Local Variables:
* i (Integer)
Steps:
1: Assign i, the value 0
2: While i < Length
3:   Output '-'
4:   Increase i by 1
5: Output a new line
  
```

Listing 3.7: Pseudocode for Print Line**Note**

- This code includes a **Local Variable** *i*.
- *i* is used to count the number of times the loop has executed, allowing it to stop when it has printed enough dashes.
- This uses a **Pre-Test Loop**. This makes sure that no characters are printed if the length is less than or equal to zero.
- Notice that a sequence can be placed within a loop.

3.2.7 Designing the Control Flow for Main

The last Procedure is Main. This is responsible for coordinating the actions of the program. It will call Play Game in a loop that repeatedly plays the game until the user decides to quit. Main will have one local variable called again. This will store a character, and will be used to store the value read from the user's response to the 'play again' prompt.

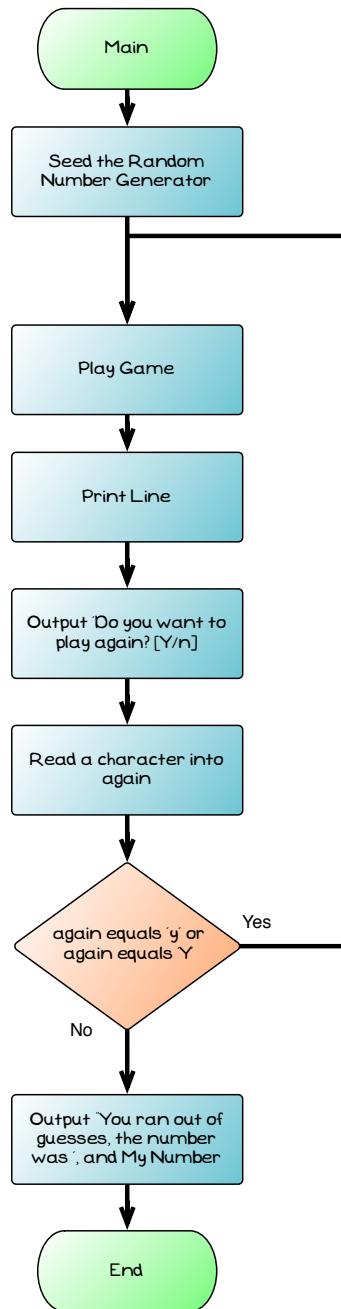
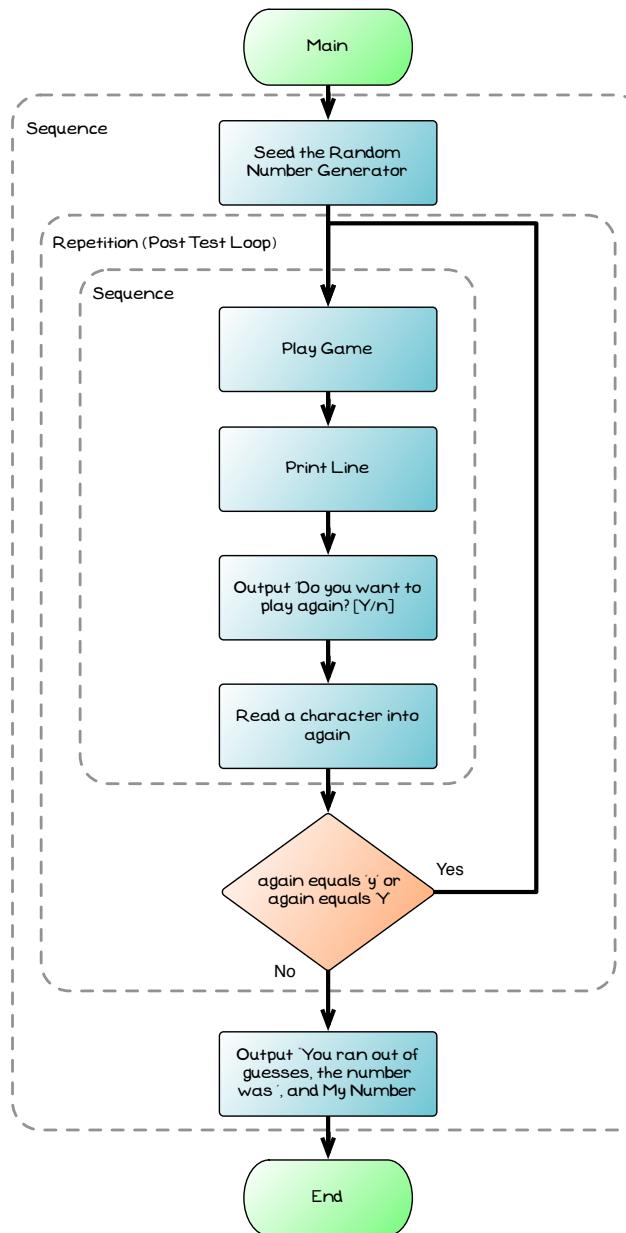


Figure 3.37: Flowchart for the Main Procedure

**Figure 3.38:** Blocks in the Main code**Note**

Computers cannot generate truly random numbers. Instead it uses a numeric sequence that appears to be random. Seeding the generator with the current time ensures that this random sequence starts at a different value each time the program is run.

3.2.8 Writing the Code for Guess That Number

Flowcharts and Pseudocode communicate the same ideas. They describe the actions that need to be performed within your code. The following two sections, Section 3.3 Control Flow in C and Section 3.4 Control Flow in Pascal, contain a description of the syntax needed to code these control flow statements in the C and Pascal programming languages.

Note

Remember the basic process for reading the Syntax Diagrams is to:

1. Find the page with the Syntax rule you are interested in knowing about.
2. Have a quick look at the Syntax Diagram and the rules it contains. Read each rule, and get a basic feel for how it is going to come together for your program.
3. Read the example to see one way of using the Rule. The Syntax Diagram can be used to create any number of variations of the rule, the example gives you at least one way these rules can be coded.
4. Return to the diagram and make sure you can match each part of the example back to the rule that created it.
5. Look up any related rules that are not explained on this rule's page.



3.2.9 Compiling and Running Guess that Number

Once you have completed the code for this program you need to compile and run it. As this uses random numbers you cannot generate standard test data in order to check the execution. Instead you should perform a number of executions and test the different paths through the program. The main condition you want to check are:

- Test failing to get the number in seven guesses.
- Test getting the number correct within seven guesses.
- Check the output messages when your guess is less than the number (you can enter a guess below 0).
- Check the output message when your guess is larger than the number (you can enter a guess larger than 100).
- Check that the random sequence is different each time, if its not make sure you have seeded the random number generator.

3.3 Control Flow in C

3.3.1 Implementing the Guess that Number in C

Section 3.2 of this Chapter introduced the ‘Guess that Number’ program. This program contained a Function to Perform Guess and Procedures to Print Line and Play Game. Each of these involved some control flow in their logic, as shown in the Flowcharts in Section 3.2. The full C implementation of the Guess that Number program is shown in Listing 3.8.

```
/*
* Program: guess-that-number.c
* This program is an implementation of the "guess that number"
* game. The computer randomly chooses a number and the player
* attempts to guess it. (It should never take more than 7 guesses)
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <stdbool.h>

#define MAX_NUMBER 100
#define MAX_GUESSES 7

// Print a line onto the Terminal.
void print_line(int len)
{
    int i = 0;

    while (i < len)
    {
        printf("-");
        i++;
    }

    printf("\n");
}

// Perform the steps for the guess. Reads the value entered by the user,
// outputs a message, and then returns true if the got it otherwise it returns
// false.
bool perform_guess(int num_guess, int target)
{
    int guess;

    printf("Guess %d: ", num_guess);
    scanf("%d", &guess);

    if (target < guess) printf("The number is less than %d\n", guess);
    else if (target > guess) printf("The number is larger than %d\n", guess);
    else printf("Well done... the number was %d\n", guess);

    return target == guess;
}

// Implements a simple guessing game. The program generates
// a random number, and the player tries to guess it.
void play_game()
{
    int my_number, num_guess;
```

```

bool got_it;

my_number = random() % MAX_NUMBER + 1;
num_guess = 0; //Keep track of the number of guesses

printf("I am thinking of a number between 1 and %d\n\n", MAX_NUMBER);

do
{
    num_guess++;
    got_it = perform_guess(num_guess, my_number);
}
while( num_guess < MAX_GUESSES && !got_it);

if ( !got_it )
{
    printf("You ran out of guesses... the number was %d\n", my_number);
}

// Loops the guessing game until the user decided to quite.
int main()
{
    char again;

    srand(time(0));

    do
    {
        play_game();

        printf("\n");
        print_line(50);
        printf("Do you want to play again [y/N]? ");
        scanf(" %c", &again);
    } while (again == 'y' || again == 'Y');

    printf("\nBye\n");
    return 0;
}

```

Listing 3.8: C code for the Guessing Game**Note**

- stdlib.h is needed for the random function and the srand procedure.
- time.h is needed to get the current time used to seed the random number generator.
- stdbool.h gives access to the bool data type in C for Boolean Data.
- In this code perform_guess is laid out differently to that shown previously. This is also an acceptable layout as it shows the different paths clearly. While it does not show the structure as well, it is a clean and neat way of presenting this code.

3.3.2 C Boolean Data

C has very flexible support for Boolean values. In C a `0` value is considered to be false, and any other value is true. Modern C compilers have now added support for an explicit Boolean type, `bool`. This type requires the `stdbool.h` header file, which defines the `bool` type as well as the values `true` and `false`.

Boolean Type		
Name	Size	Values
<code>bool</code>	1 byte/8 bits	true or false

Table 3.9: C Boolean Type

	Description	C
Equal	Are the values the same?	<code>a == b</code>
Not Equal	Are the values different?	<code>a != b</code>
Larger Than	Is the left value larger than the right?	<code>a > b</code>
Less Than	Is the left value smaller than the right?	<code>a < b</code>
Larger Or Equal	Is the left value equal or larger than the right?	<code>a >= b</code>
Less Or Equal	Is the left value smaller or equal to the right?	<code>a <= b</code>

Table 3.10: C Comparison Operators

	Description	C
And	Are both values True?	<code>a && b</code>
Or	Is at least one value True?	<code>a b</code>
Xor	Is one value True, and the other False?	<code>a ^ b</code>
Not	Is the value False?	<code>!a</code>

Table 3.11: Logical Operators

Note

- To use the Boolean type in C you must include the `stdbool.h` header.
- In C `false` is the value `0`, and `true` is any other value.
- The `stdbool.h` header gives you access to two values, `true` and `false`. `true` has the value `1`, `false` has the value `0`.

C++

```
/* Program: test-bools.c */
#include <stdio.h>
#include <stdbool.h>

void print_bool(bool value)
{
    if(value)
        printf("true");
    else
        printf("false");
}

// Is v1 at least double v2
bool at_least_double(int v1, int v2)
{
    return v1 >= 2 * v2;
}

int main()
{
    bool test;
    int num;

    printf("Enter a number: ");
    scanf("%d", &num);

    test = at_least_double(num, 5);
    printf("%d is at least double 5 is ", num);
    print_bool(test);
    printf("\n");

    return 0;
}
```

Listing 3.9: C Boolean Test Code

3.3.3 C Statement (with loops)

In addition to the [Procedure Call](#) and [Assignment Statement](#), C Statements may be [Branching](#), [Looping](#), or [Jumping](#) Statements.

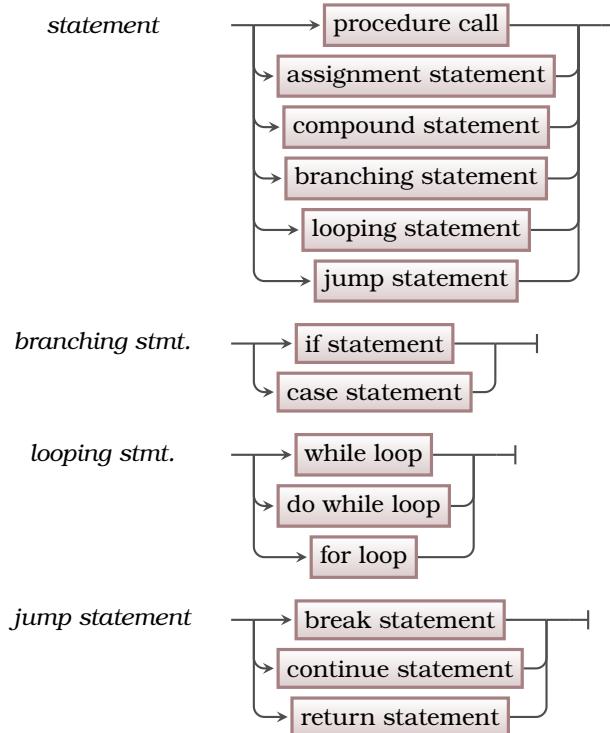


Figure 3.39: C++ Syntax for a Statement (with branches and loops)

Note

- See the following diagrams for details on this syntax:
 - [C If Statement](#): for the syntax of an [If Statement](#).
 - [C Case Statement](#): for the syntax of an [Case Statement](#).
 - [C While Loop](#): for the syntax of an [Pre-Test Loop](#).
 - [C Do While Loop](#): for the syntax of an [Post-Test Loop](#).
 - [C Jump Statements](#): for the [Jumping](#) Statements.
- These statements can be coded within Functions, Procedures, and programs.



3.3.4 C If Statement

The if statement is a [Branching](#) statement. This can be used to optionally run a block of code, providing two alternate paths controlled by a Boolean expression.

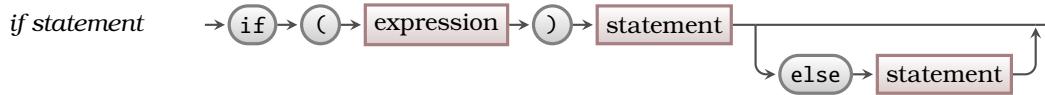


Figure 3.40: C++ Syntax for an If Statement

C++

```

/* Program: test-if.c */

#include <stdio.h>

int main()
{
    int num, num1;

    printf("Enter a number: ");
    scanf("%d", &num);

    if (num != 2)
        printf("Num is not 2!\n");

    printf("Enter another number: ");
    scanf("%d", &num1);

    if (num1 == 2 && num != 2)
        printf("You got the hint... num1 is 2!");

    if (num > num1)
        printf("The first number you entered was the larger.");
    else
        printf("The first number you entered was not larger.");

    return 0;
}
  
```

Listing 3.10: C if test code

Note

- This is the C syntax for the [If Statement](#).
- The parenthesis surround the expression. This enables the compiler to tell where the expression ends.
- Notice that the else branch is optional.
- When the expression is false (0 in C), the else branch is taken.
- For any other value the first path is taken.
- You only need to include stdbool.h if you want to use the bool type and the values true or false.

3.3.5 C Case Statement

The case statement allows you to switch between a number of paths.

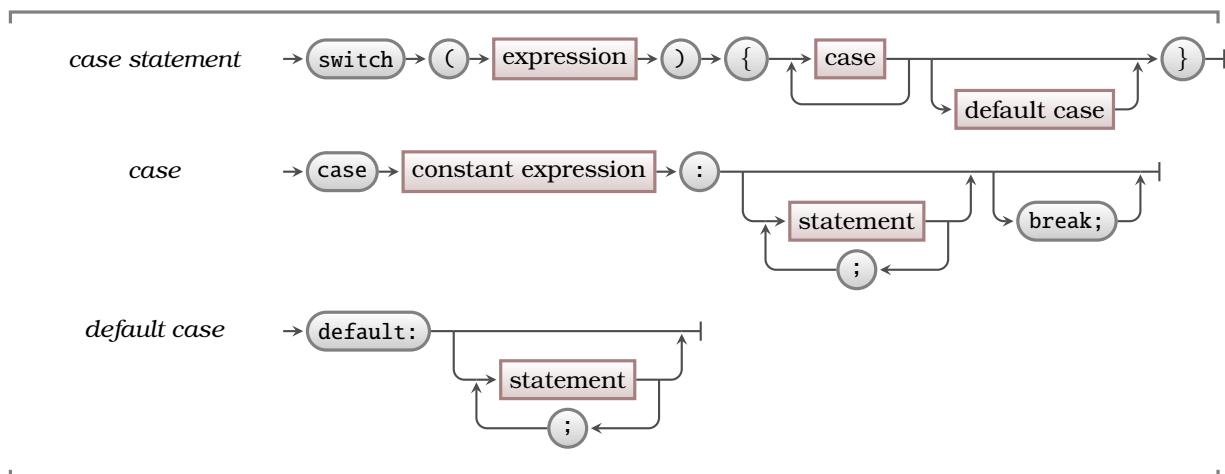


Figure 3.41: C++ Syntax for a Case Statement

Note

- This is the C syntax to declare a [Case Statement](#).
- The *constant expressions* in each *case* must be ordinal values (integers or characters).
- The code in Listing 3.12 shows an example use for a case statement.
- The default path is taken when none of the other paths match the expression.
- If the break is left off the end of a *case* then execution will continue into the next *case*. For example, in Listing 3.11 if the user enters 'c' the output will be 'C' and 'D'
- Each *case* can contain a number of Statements.
- Watch <http://www.youtube.com/watch?v=zIV4poUZAQo> for important details on the legendary Knights of Ni.

C++

```

/* Program: simple-case.c */
#include <stdio.h>

int main()
{
    char ch;
    printf("Enter a character: ");
    scanf("%c", &ch);

    switch(ch)
    {
        case 'a':
        case 'b': printf("A or B\n");
                    break;
        case 'c': printf("C ");
        case 'd': printf("and D\n");
                    default: printf("Something else...\n");
    }
    return 0;
}
  
```

Listing 3.11: C case test code with a character

C++

```
/* Program: test-case.c */

#include<stdio.h>

int read_menu_option()
{
    int result = 0;

    printf("1: Greet the Knights of Ni\n");
    printf("2: Offer Knights a shrubbery\n");
    printf("3: Refuse to cut down tree with Herring\n");
    printf("4: Tell them all about it\n");

    printf("Option: ");
    scanf("%d", &result);
    return result;
}

int main()
{
    int option = read_menu_option();

    switch(option)
    {
        case 1: printf("We say Ni to you!\n");
        break;
        case 2: printf("Cut down the mightiest tree... with a Herring!\n");
        break;
        case 3: printf("Oh please...");
        break;
        case 4: printf("Argh... dont say that word!\n")
        break;
        default:
            printf("Please enter a value between 1 and 4");
    }
    return 0;
}
```

Listing 3.12: C case test code with an integer

3.3.6 C Compound Statement

Most of the C structured statements only allow single statements within each path. For example, the paths in the two branches of an [C If Statement](#) can only contain a single statement. The [Compound Statement](#) allows you to group together multiple statements within a single *compound statement*.

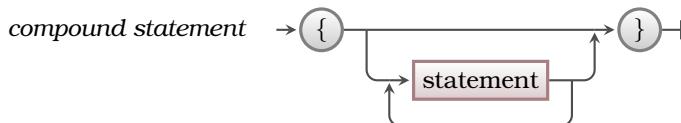


Figure 3.42: C++ Syntax for a Compound Statement

C++

```

/* Program: test-compound.c */

int main()
{
    char ch;

    printf("Enter the forest? [y/N]: ");
    scanf("%c", &ch);

    if (ch == 'y' || ch == 'Y')
    {
        printf("We are the Knights who say 'Ni'!\n");
        printf("We are the keepers of the sacred words: ");
        printf("Ni", 'Peng', and 'Neee-wom'!\n");
        printf("The Knights Who Say 'Ni' demand a sacrifice.\n");
        printf("We want... a shrubbery!\n");
    }
    else
    {
        printf("Greetings Sir Robin.\n");
    }
}
  
```

Listing 3.13: C compound statement test code

Note

- Figure 3.42 shows the syntax for a [Compound Statement](#) in C.
- The code in Listing 3.13 shows an if statement that includes two compound statements within its branches.
- Compound statements in C are marked with curly brackets, '{' for the start, and '}' for the end.
- The compound statement is a standard statement, and can be used anywhere a statement can appear. Its practical use is for grouping statements within other structured statements, and you are unlikely to find it used in any other way.

3.3.7 C While Loop

The while loop is C's [Pre-Test Loop](#), allowing a block to be repeated zero or more times.

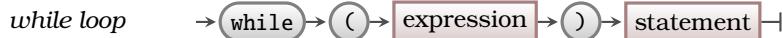


Figure 3.43: C++ Syntax for a While Loop

C++

```

/* Program: test-while.c */
#include <stdio.h>
#include <stdbool.h>

bool beg_for_mercy()
{
    char ch;

    printf("Beg for mercy? [y/N]: ");
    scanf(" %c", &ch);

    return ch == 'y' || ch == 'Y';
}

int main()
{
    bool mercy = false;

    printf("Before you stands a 12 foot tall Knight...\n");
    printf("\We are the Knights who say 'Ni'.\n");
    printf("\I will say Ni to you again if you do not appease us!\n");

    mercy = beg_for_mercy();

    while( !mercy )
    {
        printf("\Ni!\n");
        mercy = beg_for_mercy();
    }

    printf("\Bring us a Shrubbery!\n");

    return 0;
}
  
```

Listing 3.14: C while loop test code

Note

- Figure 3.43 shows the syntax for a [Pre-Test Loop](#) in C.
- This loop runs 0 to many times, checking the condition before executing the body of the loop.
- The parenthesis allow the compiler to determine where the condition ends, and the statement starts.
- The while loop repeats a single statement, you must use a [Compound Statement](#) to have multiple statements repeated. See [C Compound Statement](#).

3.3.8 C Do While Loop

The do while loop is C's [Post-Test Loop](#), allowing a block to be repeated one or more times.

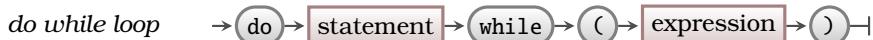


Figure 3.44: C++ Syntax for a Do While Loop

C++

```

/* Program: test-do-while.c */
#include <stdio.h>
#include <stdbool.h>

bool beg_for_mercy()
{
    char ch;

    printf("Beg for mercy? [y/N]: ");
    scanf(" %c", &ch);

    return ch == 'y' || ch == 'Y';
}

int main()
{
    bool mercy = false;

    printf("Before you stands a 12 foot tall Knight...\n");
    printf("\We are the Knights who say 'Ni'.\n");
    printf("\I will say Ni to you again if you do not appease us!\n");

    do
    {
        printf("\Ni!\n");
        mercy = beg_for_mercy();
    } while( !mercy );

    printf("\Bring us a Shrubbery!\n");

    return 0;
}
  
```



Listing 3.15: C do while loop test code

Note

- Figure 3.44 shows the syntax for a [Post-Test Loop](#) in C.
- This loop runs 1 to many times, checking the condition after executing the body of the loop.
- The parenthesis allow the compiler to determine where the condition ends.
- The do while loop repeats a single statement, you must use a [Compound Statement](#) to have multiple statements repeated. See [C Compound Statement](#).



3.3.9 C Jump Statements

The jump statements allow you to exit out of another structure.

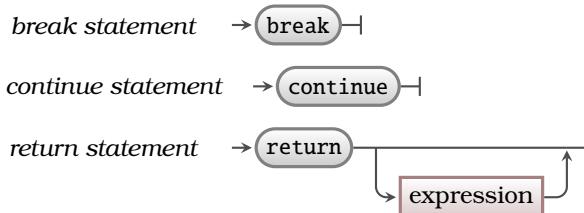


Figure 3.45: C++ Syntax for the Jump Statements

C++

```

/* Program: test-jump.c */
#include <stdio.h>

int main()
{
    int i = 0;
    char ch;

    while(i < 100000000)
    {
        i++;

        // Skip all even numbers
        if (i % 2 == 0) continue;

        printf("At %d\n", i);

        printf("Quit? [y/N]: ");
        scanf("%c", &ch);

        if (ch == 'y' || ch == 'Y')
        {
            printf("Quitting loop...");
            break;
        }
    }

    // Ending function
    return 0;
}

// Code cannot be reached!
printf("This will never be printed!\n");
}
  
```

Listing 3.16: C code demonstrating the jump statements

C++ Reference

Note

- `break` jumps to the end of the current loop.
- `continue` jumps back to the condition in the current loop.
- `return` ends the current Function or Procedure.

3.4 Control Flow in Pascal

3.4.1 Implementing the Guess that Number in Pascal

Section 3.2 of this Chapter introduced the ‘Guess that Number’ program. This program contained a function to Perform Guess and procedures to Print Line and Play Game. Each of these involved some control flow in their logic, as shown in the flowcharts in Section 3.2. The full Pascal implementation of the Guess that Number program is shown in Listing 3.17.

```
// This program is an implementation of the 'guess that number'
// game. The computer randomly chooses a number and the player
// attempts to guess it. (It should never take more than 7 guesses)
program GuessThatNumber;

const
  MAX_NUMBER = 100;
  MAX_GUESSES = 7;

// Print a line onto the Terminal.
procedure PrintLine(len: Integer);
var
  i: Integer = 0;
begin
  while ( i < len ) do
  begin
    Write('-');
    i += 1;
  end;
  WriteLn();
end;

// Perform the steps for the guess. Reads the value entered by the user,
// outputs a message, and then returns true if the got it otherwise it returns
// false.
function PerformGuess(numGuess, target: Integer): Boolean;
var
  guess: Integer;
begin
  Write('Guess ', numGuess, ': ');
  ReadLn(guess);

  if target < guess then WriteLn('The number is less than ', guess)
  else if target > guess then WriteLn('The number is larger than ', guess)
  else WriteLn('Well done... the number was ', guess);

  result := target = guess; // return true when "target equals guess"
end;

// Implements a simple guessing game. The program generates
// a random number, and the player tries to guess it.
procedure PlayGame();
var
  myNumber, numGuess: Integer;
  gotIt: Boolean = False;
begin
  myNumber := Random(MAX_NUMBER) + 1;
  numGuess := 0; //Keep track of the number of guesses

  WriteLn('I am thinking of a number between 1 and ', MAX_NUMBER);
  WriteLn();
end;
```

```
repeat
    numGuess += 1;
    gotIt := PerformGuess(numGuess, myNumber);
until (numGuess > MAX_GUESSES) or gotIt;

if not gotIt then
begin
    WriteLn('You ran out of guesses... the number was ', myNumber);
end;
end;

// Loops the guessing game until the user decided to quite.
procedure Main();
var
    again: Char;
begin
    Randomize();

repeat
    PlayGame();
    WriteLn();
    PrintLine(50);
    WriteLn('Do you want to play again [Y/n]? ');
    ReadLn(again);
until (again = 'n') or (again = 'N');

    WriteLn('Bye');
end;

begin
    Main();
end.
```

Listing 3.17: Pascal code for the Guessing Game

3.4.2 Pascal Boolean Data

Pascal includes a Boolean type, as well as True and False values.

Boolean Type		
Name	Size	Values
Boolean	1 byte/8 bits	True or False

Table 3.12: Pascal Boolean Type

	Description	C
Equal	Are the values the same?	<code>a = b</code>
Not Equal	Are the values different?	<code>a <> b</code>
Larger Than	Is the left value larger than the right?	<code>a > b</code>
Less Than	Is the left value smaller than the right?	<code>a < b</code>
Larger Or Equal	Is the left value equal or larger than the right?	<code>a >= b</code>
Less Or Equal	Is the left value smaller or equal to the right?	<code>a <= b</code>

Table 3.13: Pascal Comparison Operators

	Description	Pascal
And	Are both values True?	<code>a and b</code>
Or	Is at least one value True?	<code>a or b</code>
Xor	Is one value True, and the other False?	<code>a xor b</code>
Not	Is the value False?	<code>not a</code>

Table 3.14: Logical Operators

Note

The Pascal logical operators have precedence over the comparison operators, this means you will need to use parenthesis to group comparisons when combining them using logical operators. For example, `(a > b) or (c < d)`.



Pascal

```
program TestBools;

procedure PrintBoolean(value: Boolean);
begin
    if value then // if value is true then...
        Write('true')
    else
        Write('false');
end;

// Is v1 at least double v2
function AtLeastDouble(v1, v2: Integer): Boolean;
begin
    result := v1 >= 2 * v2;
end;

procedure Main();
var
    test: Boolean;
    num: Integer;
begin
    Write('Enter a number: ');
    ReadLn(num);

    test := AtLeastDouble(num, 5);
    Write(num, ' is at least double 5 is ');
    PrintBoolean(test);
    WriteLn();
end;

begin
    Main();
end.
```



Listing 3.18: Pascal Boolean Test Code

3.4.3 Pascal Statement (with loops)

In addition to the [Procedure Call](#) and [Assignment Statement](#), Pascal statements may be [Branching](#), [Looping](#), or [Jumping](#) statements.

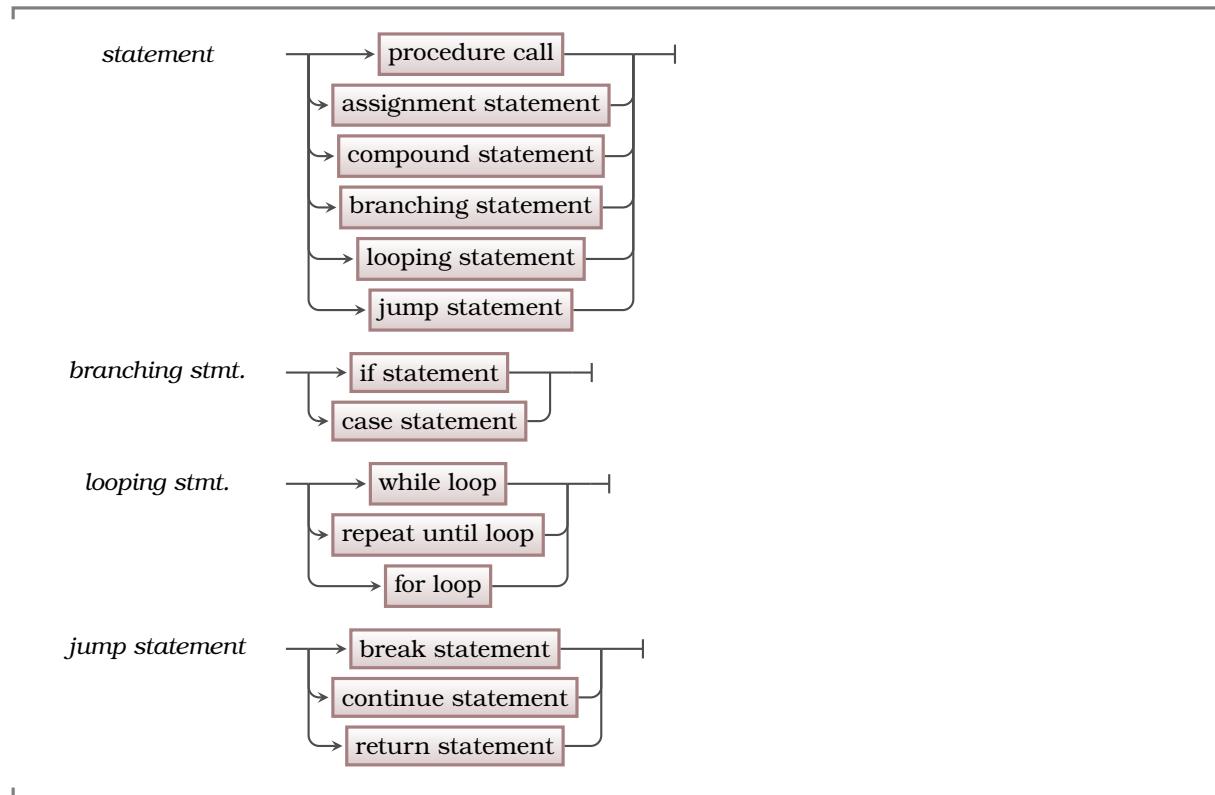


Figure 3.46: Pascal Syntax for a statement (with branches and loops)

Note

- See the following diagrams for details on this syntax:
 - [Pascal If Statement](#): for the syntax of an [If Statement](#).
 - [Pascal Case Statement](#): for the syntax of an [Case Statement](#).
 - [Pascal While Loop](#): for the syntax of an [Pre-Test Loop](#).
 - [Pascal Repeat Loop](#): for the syntax of an [Post-Test Loop](#).
 - [Pascal Jump Statements](#): for the [Jumping](#) statements.
- These statements can be coded within functions, procedures, and programs.



3.4.4 Pascal If Statement

The if statement is a [Branching](#) statement. This can be used to optionally run a block of code, providing two alternate paths controlled by a Boolean expression.

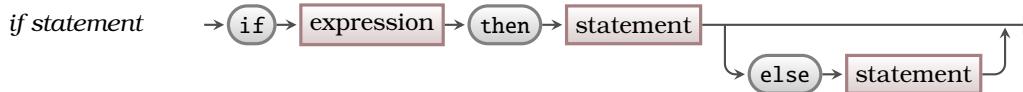


Figure 3.47: Pascal Syntax for an if statement

Pascal

```

program TestIf;

procedure Main();
var
  num, num1: Integer;
begin
  Write('Enter a number: ');
  ReadLn(num);

  if num <> 2 then
    WriteLn('Num is not 2!');

  Write('Enter another number: ');
  ReadLn(num1);

  if (num1 = 2) and (num <> 2) then
    WriteLn('You got the hint... num1 is 2!');

  if num > num1 then
    WriteLn('The first number you entered was the larger.')
  else
    WriteLn('The first number you entered was not larger.');
end;

begin
  Main();
end.
  
```

Listing 3.19: Pascal if test code

Note

- This is the Pascal syntax for the [If Statement](#).
- The then keyword tells the compiler where the if's condition ends.
- Notice that the else branch is optional.
- When the expression is True the first path is taken.
- When the expression is False the else branch is taken.
- Notice that there is **no** semicolon (;) after the first statement before the else branch.

3.4.5 Pascal Case Statement

The case statement allows you to switch between a number of paths.

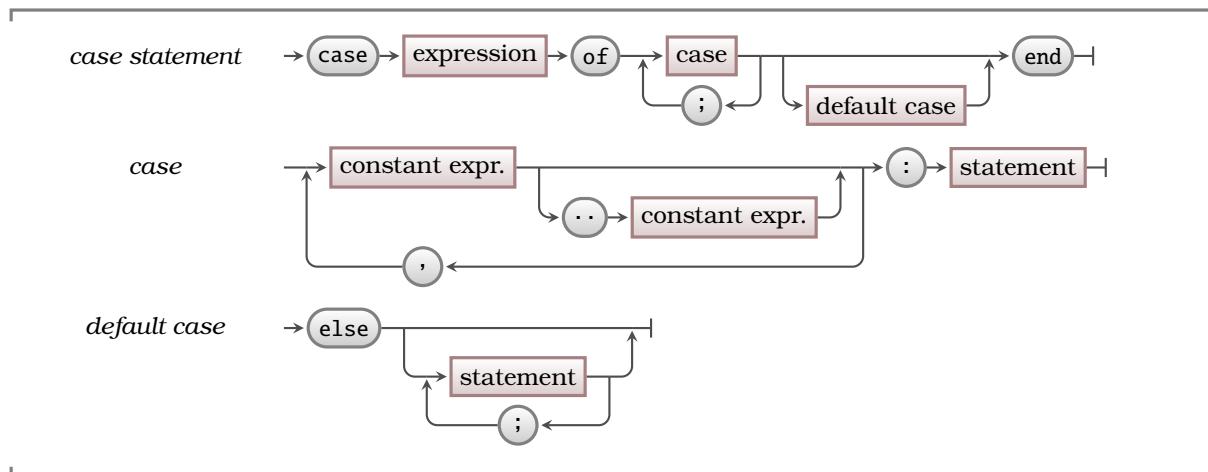


Figure 3.48: Pascal Syntax for a case statement

Note

- This is the Pascal syntax to declare a [Case Statement](#).
 - The *constant expressions* in each *case* must be ordinal values (integers or characters).
 - By using *constant..constant* the case will match any value in this range, e.g. `0..9`.
 - The code in Listing 3.21 shows an example use for a case statement.
 - The default path is taken when none of the other paths match the expression.
 - Each *case* contain a single statement.
 - Watch <http://www.youtube.com/watch?v=zIV4poUZAQo> for important details on the legendary Knights of Ni.



Pascal

```
program SimpleCase;

procedure Main();
var
    ch: Char;
begin
    Write('Enter a character: ');
    ReadLn(ch);

    case ch of
        'a', 'b':           WriteLn('a or b');
        'c', 'e':           WriteLn('c or e');
        'd':                WriteLn('d');
        'f'..'z', 'F'..'Z': WriteLn('f to z or F to Z');
        else                WriteLn('Something else...');

    end;
end;

begin
    Main();
end.
```



Listing 3.20: Pascal case test code with a character

Pascal

```
program TestCase;

function ReadMenuOption(): Integer;
begin
    WriteLn('1: Greet the Knights of Ni');
    WriteLn('2: Offer Knights a shrubbery');
    WriteLn('3: Refuse to cut down tree with Herring');
    WriteLn('4: Tell them all about it');

    Write('Option: ');
    ReadLn(result);
end;

procedure Main();
var
    option: Integer;
begin
    option := ReadMenuOption();

    case option of
        1: WriteLn('We say Ni to you!');
        2: WriteLn('Cut down the mightiest tree... with a Herring!');
        3: WriteLn('Oh please...');
        4: WriteLn('Argh... dont say that word!')
        else WriteLn('Please enter a value between 1 and 4');
    end;
end;

begin
    Main();
end.
```



Listing 3.21: Pascal case test code with an integer

3.4.6 Pascal Compound Statement

Most of the Pascal structured statements only allow a single statement within each path. For example, the paths in the two branches of an [Pascal If Statement](#) can only contain a single statement. The [Compound Statement](#) allows you to group together multiple statements within a single *compound statement*.



Figure 3.49: Pascal Syntax for a compound statement

Pascal

```

program TestCompound;

procedure Main();
var
  ch: Char;
begin
  Write('Enter the forest? [y/N]: ');
  ReadLn(ch);

  if (ch = 'y') or (ch = 'Y') then
  begin
    WriteLn('We are the Knights who say "Ni"!');
    WriteLn('We are the keepers of the sacred words: ');
    WriteLn('"Ni", "Peng", and "Neee-wom"!');
    WriteLn('The Knights Who Say "Ni" demand a sacrifice.');
    WriteLn('We want... a shrubbery!');
  end
  else
  begin
    WriteLn('Greetings Sir Robin.');
  end;
end;
  
```



Listing 3.22: Pascal compound statement test code

Note

- Figure 3.49 shows the syntax for a [Compound Statement](#) in Pascal.
- The code in Listing 3.22 shows an if statement that includes two compound statements within its branches.
- The compound statement is a standard statement, and can be used anywhere a statement can appear. Its practical use is for grouping statements within other structured statements, and you are unlikely to find it used in any other way.



3.4.7 Pascal While Loop

The while loop is Pascal's [Pre-Test Loop](#), allowing a block to be repeated zero or more times.



Figure 3.50: Pascal Syntax for a while loop

Pascal

```

program TestWhile;

function BegForMercy(): Boolean;
var
  ch: Char;
begin
  Write('Beg for mercy? [y/N]: ');
  ReadLn(ch);

  result := (ch = 'y') or (ch = 'Y');
end;

procedure Main();
var
  mercy: Boolean = False;
begin
  WriteLn('Before you stands a 12 foot tall Knight... ');
  WriteLn('"We are the Knights who say ''Ni'''');
  WriteLn('"I will say Ni to you again if you do not appease us!"');

  mercy := BegForMercy();

  while not mercy do
  begin
    WriteLn('"Ni!"');
    mercy := BegForMercy();
  end;

  WriteLn('"Bring us a Shrubbery!"');
end;

begin
  Main();
end.
  
```



Listing 3.23: Pascal while loop test code

Note

- Figure 3.50 shows the syntax for a [Pre-Test Loop](#) in Pascal.
- This loop runs 0 to many times, checking the condition before executing the body of the loop each time.
- The do keyword marks the end of the condition and the start of the statement.
- The while loop repeats a single statement, you must use a [Compound Statement](#) to have multiple statements repeated. See [Pascal Compound Statement](#).



3.4.8 Pascal Repeat Loop

The repeat loop is Pascal's [Post-Test Loop](#), allowing a block to be repeated one or more times.

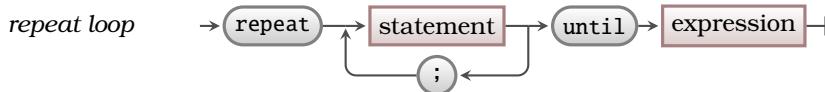


Figure 3.51: Pascal Syntax for a repeat loop

Pascal

```

program TestRepeat;

function BegForMercy(): Boolean;
var
  ch: Char;
begin
  Write('Beg for mercy? [y/N]: ');
  ReadLn(ch);

  result := (ch = 'y') or (ch = 'Y');
end;

procedure Main();
var
  mercy: Boolean = False;
begin
  WriteLn('Before you stands a 12 foot tall Knight... ');
  WriteLn('We are the Knights who say ''Ni''.');
  WriteLn('"I will say Ni to you again if you do not appease us!"');

  repeat
    WriteLn('"Ni!"');
    mercy := BegForMercy();
  until mercy;

  WriteLn('"Bring us a Shrubbery!"');
end;

begin
  Main();
end.

```

Listing 3.24: Pascal repeat loop test code

Note

- Figure 3.51 shows the syntax for a [Post-Test Loop](#) in Pascal.
- This loop runs 1 to many times, checking the condition after executing the body of the loop.
- The repeat loop repeats multiple statements, so there is no need for a compound statement.

3.4.9 Pascal Jump Statements

The jump statements allow you to exit out of another structure.

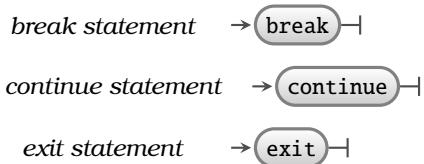


Figure 3.52: Pascal Syntax for the jump statements

Pascal

```

program TestJump;

procedure Main();
var
  i: Integer = 0;
  ch: Char;
begin
  while i < 10000000 do
  begin
    i := i + 1;

    if i mod 2 = 0 then continue; // Skip all even numbers

    WriteLn('At ', i);

    Write('Quit? [y/N]: ');
    ReadLn(ch);

    if (ch = 'y') or (ch = 'Y') then
    begin
      WriteLn('Quitting loop...');
      break; // End the loop
    end;
  end;
  WriteLn('Bye...');

  exit; // Exit function

  // Code cannot be reached!
  WriteLn('This will never be printed!');
end;

begin
  Main();
end.

```

Listing 3.25: Pascal code demonstrating the jump statements

Note

- `break` jumps to the end of the current loop.
- `continue` jumps back to the condition in the current loop.
- `exit` ends the current function or procedure.

3.5 Understanding Control Flow

This Chapter has introduced new statements that can be used to control the sequence of actions the computer performs. These statements allow you to add **Branching** and **Looping** paths to your code. The flowcharts presented in Section 3.2 are a great way of visualising the order in which the computer will execute the instructions. To help you fully understand these concepts this section will look at how these statements work within the computer.

3.5.1 Understanding Branching in Perform Guess

Figure 3.53 shows the flowchart for the Perform Guess that was developed in Section 3.2.4 on **Designing Control Flow for Perform Guess**. The following sections show how the computer executes these actions. These illustrations will start at the call into Perform Guess, skipping the illustration of the steps that lead up to this call.

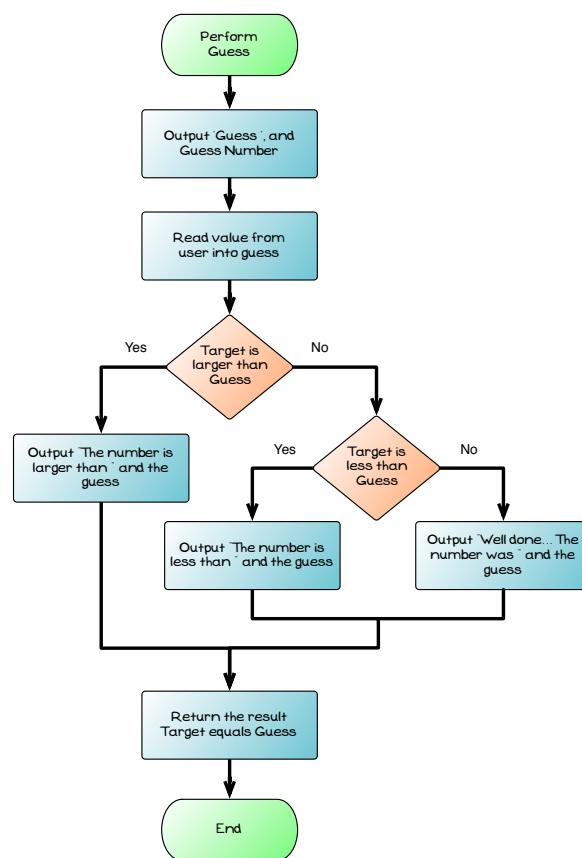


Figure 3.53: Logic for the Perform Guess Procedure from Figure 3.58

In the following illustrations Perform Guess will be called three times with the target number being 37 in each case. The following three guesses will be performed, ensuring that all paths through the flowchart are covered.

1. On the first guess the user enters a guess of 50, allowing for the left most branch of this flowchart to be followed.
2. The second guess will be 25 to test the middle branch, taking the *else* branch of the first decision and the *true* branch of the second decision.
3. Finally the third guess will be 37, testing the right most path through the code.

Perform Guess is called for guess 1

In the Guess that Number program, the Perform Guess function is responsible for reading in the user's guess and giving them feedback. Figure 3.54 shows the Perform Guess code being called for the first time, it is passed 1 to its num_guess parameter and 37 to its target parameter.

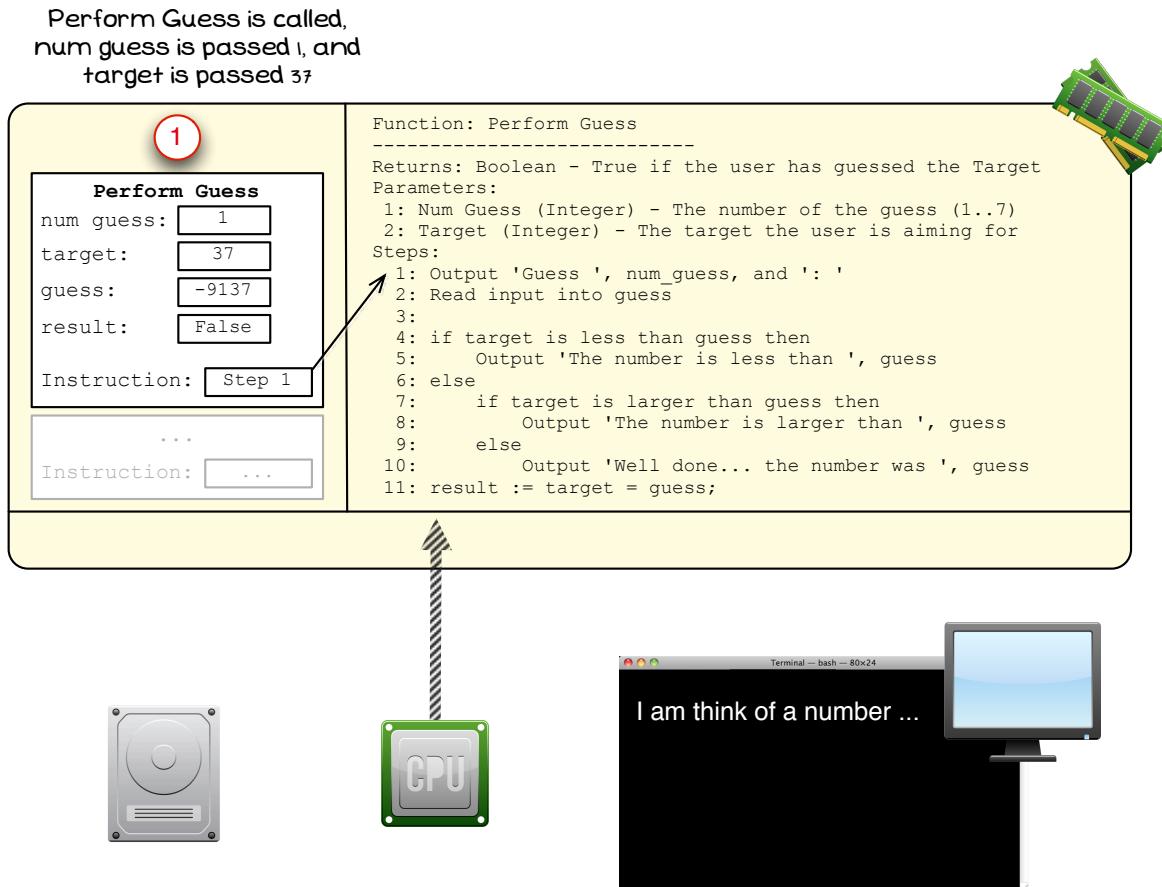


Figure 3.54: Perform Guess is called for the first time

Note

- In Figure 3.54 the indicated areas show the following:
 1. Perform Guess is called, with 1 being passed to num guess and 37 passed to target.
- At this point the previous code would have output 'I am thinking of a number ...' to the Terminal.
- The values in guess and result have not been initialised, so they have whatever value was in that memory location previously.

Execution reaches the if branch for guess 1

Execution of `Perform Guess` occurs as normal, each instruction is run one after the other.

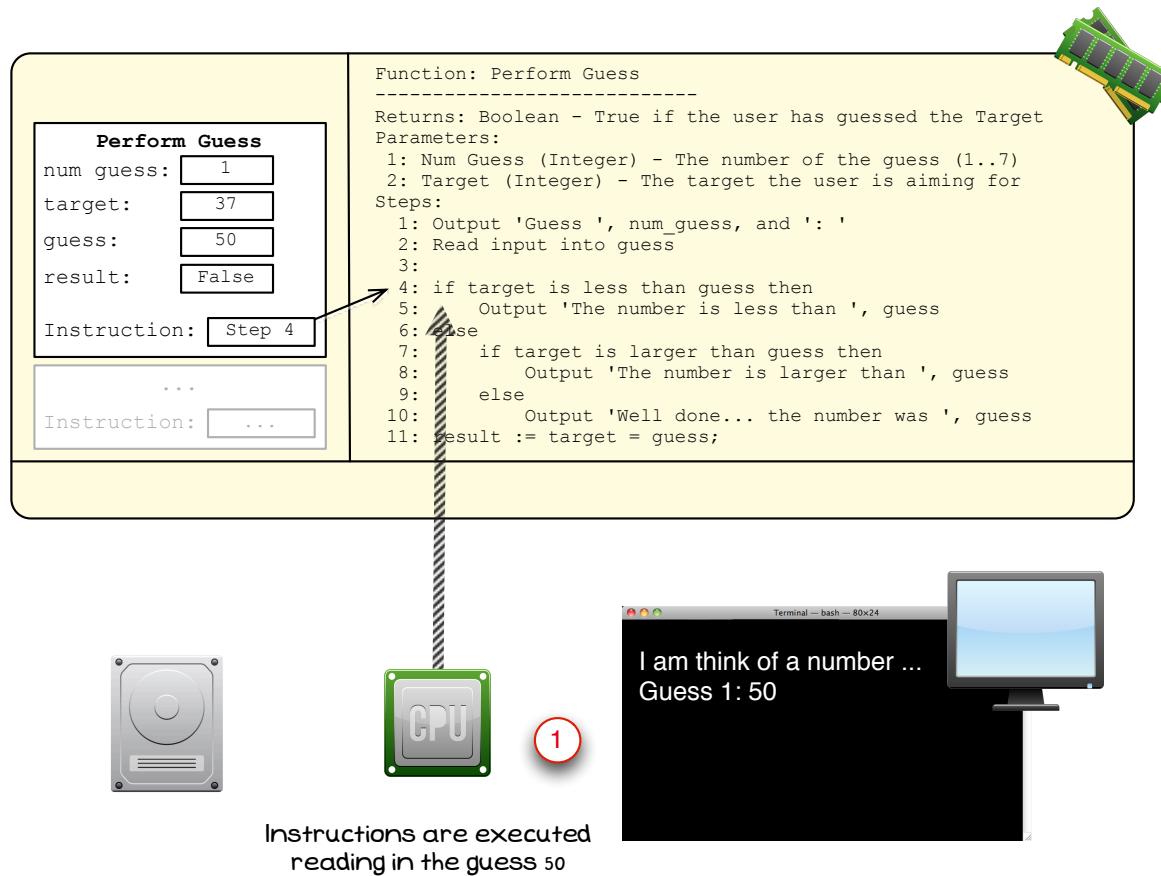


Figure 3.55: Program's steps are executed up to step 4

Note

- In Figure 3.55 the indicated areas show the following:
 - Execution proceeds one instruction at a time up to the **If Statement** at step 4.
- These steps output a prompt (step 1), and read in a response (step 2).
- Step 2 reads the user's response into the `guess` variable.
- When the computer executes step 4 it evaluates the **If Statement**'s expression and then selectively runs one of the two available paths.

If takes the True branch for guess 1

With the first guess the expression in the **If Statement** is true, target **is** less than **guess**. The computer jumps into the *true* branch of the if statement.

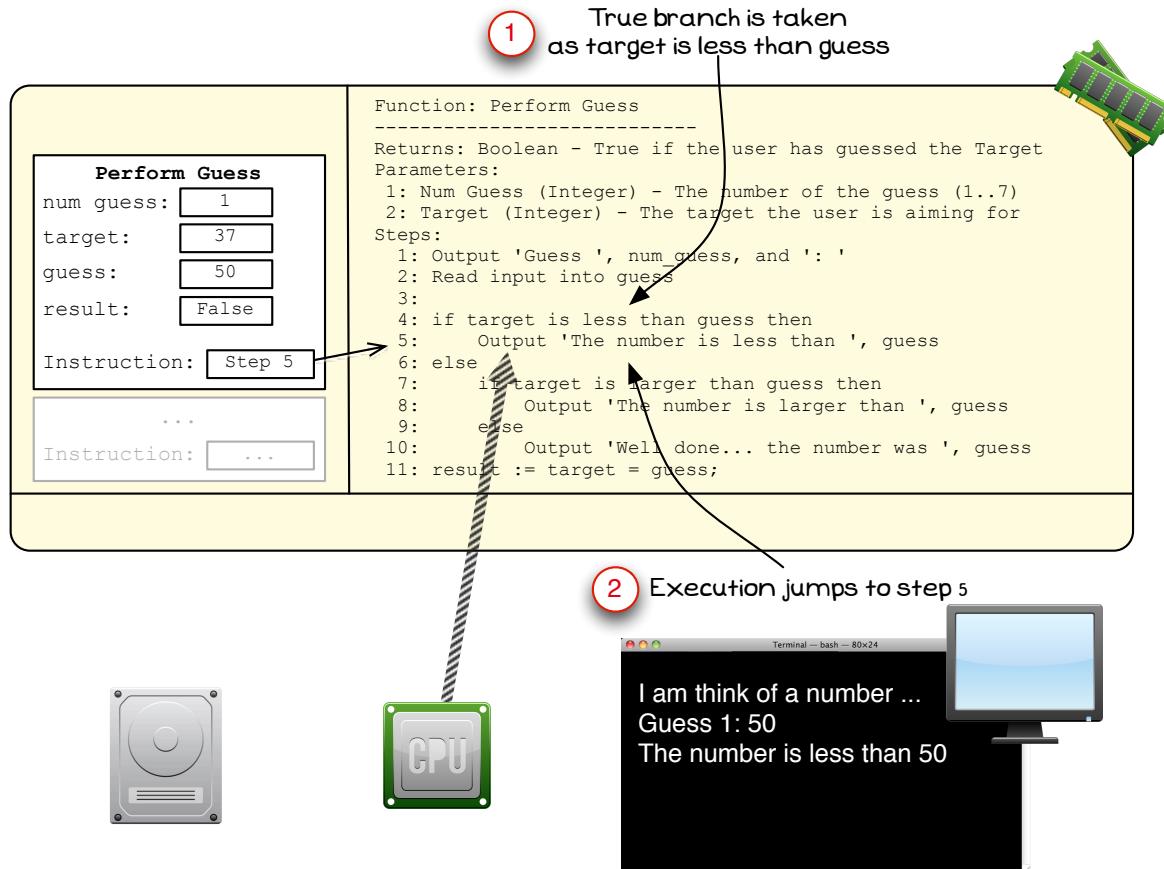


Figure 3.56: The *true* branch is taken as target is less than guess

- In Figure 3.56 the indicated areas show the following:
 1. target is less than guess so the *true* branch is executed.
 2. This means the code jumps to step 5.
 - Step 5 outputs 'The number is less than 50' to the Terminal.

Control jumps to the end of Perform Guess for guess 1

Once step 5 finishes, control jumps to step 11, skipping the *else* branch of the if statement from step 4.

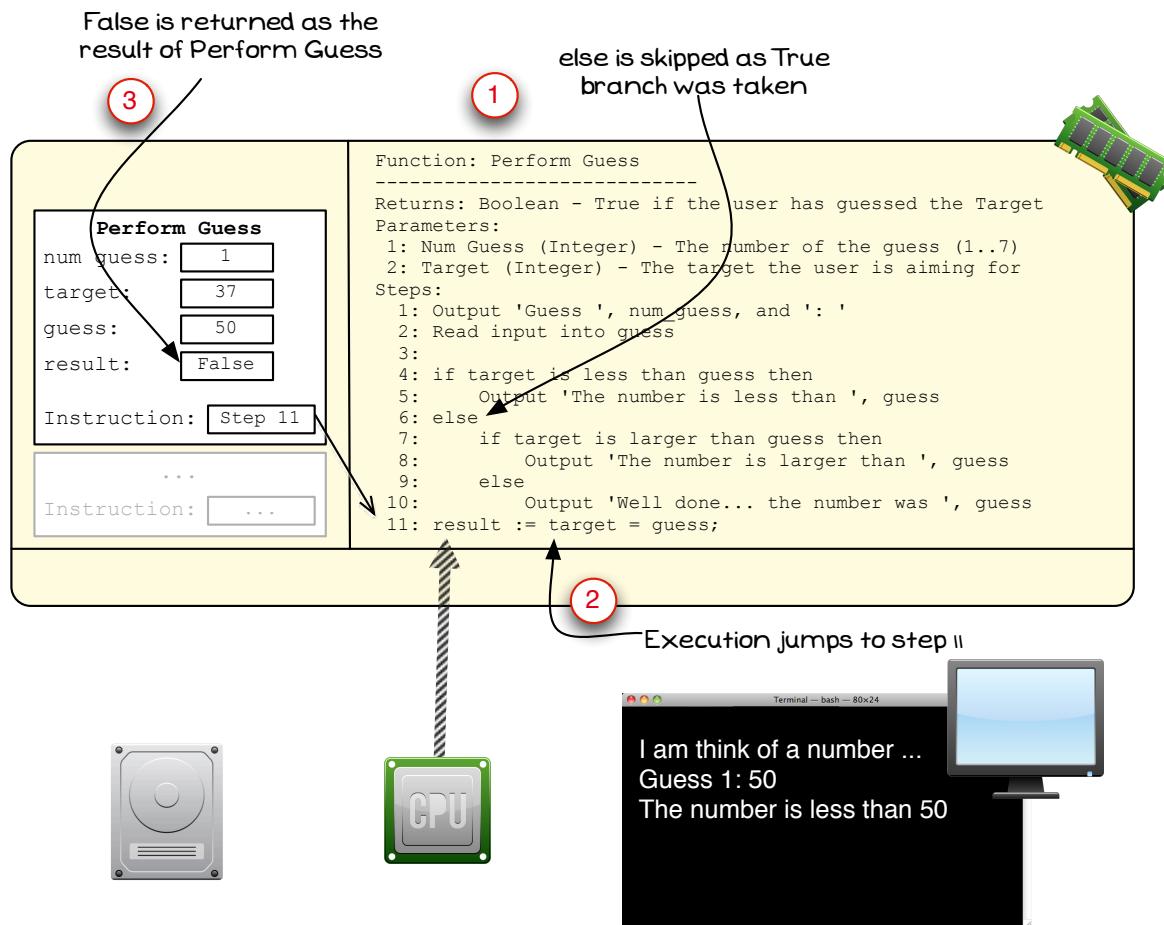


Figure 3.57: Perform Guess finishes for guess 1, returning the result false

Note

- In Figure 3.57 the indicated areas show the following:
 - Steps 6 to 10 are skipped as the *true* branch was taken.
 - The next instruction is therefore step 11.
 - This evaluates the expression **target equals guess**, which is false.

Perform Guess is called again for guess 2

Perform Guess is called again, this time it is passed 2 for guess num and 37 for target. This executes the code in Perform Guess, one instruction at a time, eventually reaching step 4.

The next call to Perform Guess
passed 2 to num guess, and
37 to target

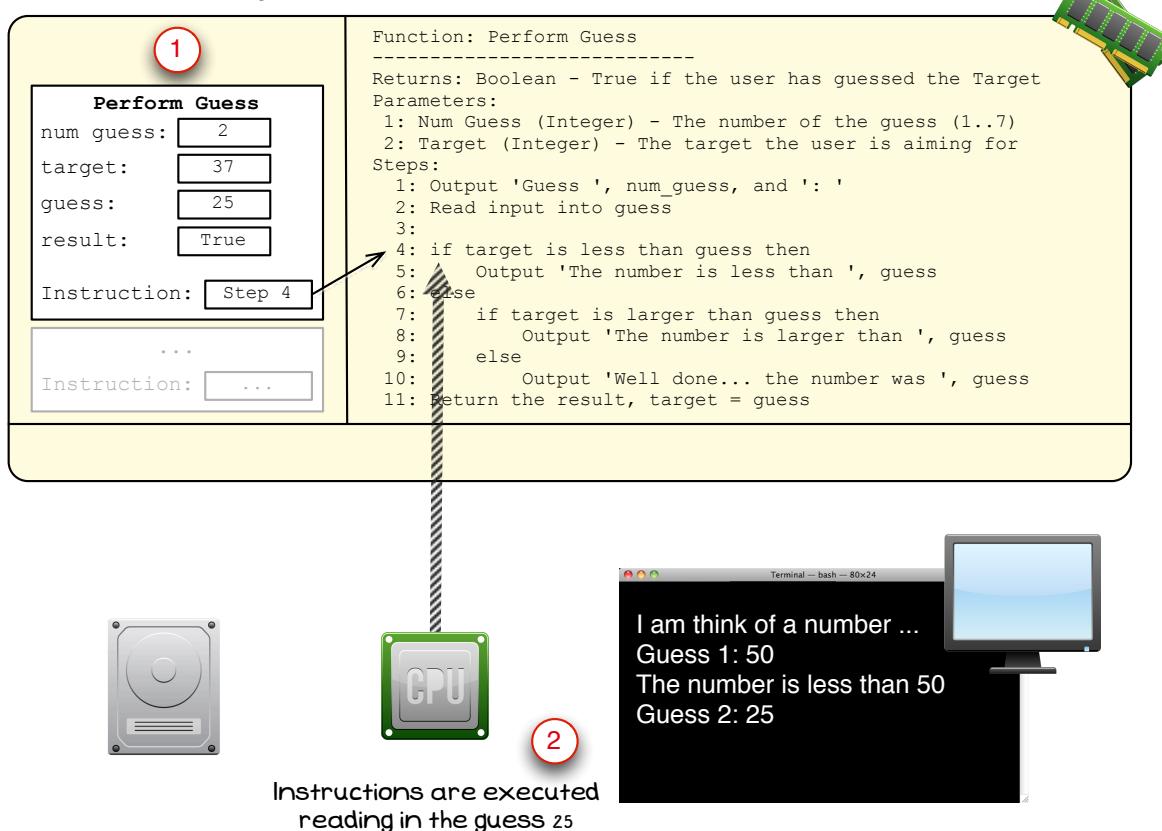


Figure 3.58: Perform Guess is run for guess 2 and advanced to step 4

Note

- In Figure 3.58 the indicated areas show the following:
 - The call to Perform Guess is passed 2 for the guess num parameter, and 37 for the target parameter.
 - The user enters 25 as their guess, this is read into the guess variable.
- The previous call ended and control returned to the caller. This is now a new call to Perform Guess.
- Step 1 outputs a prompt for the user to enter 'Guess 2: '.
- Step 2 reads the user's response into the guess variable.
- When the computer executes step 4 it evaluates the If Statement's expression and then selectively runs one of the two available paths.

If takes the else branch for guess 2

With the second guess the expression in the **If Statement** is false, target **is not** less than guess. The computer jumps into the *else* branch of the if statement.

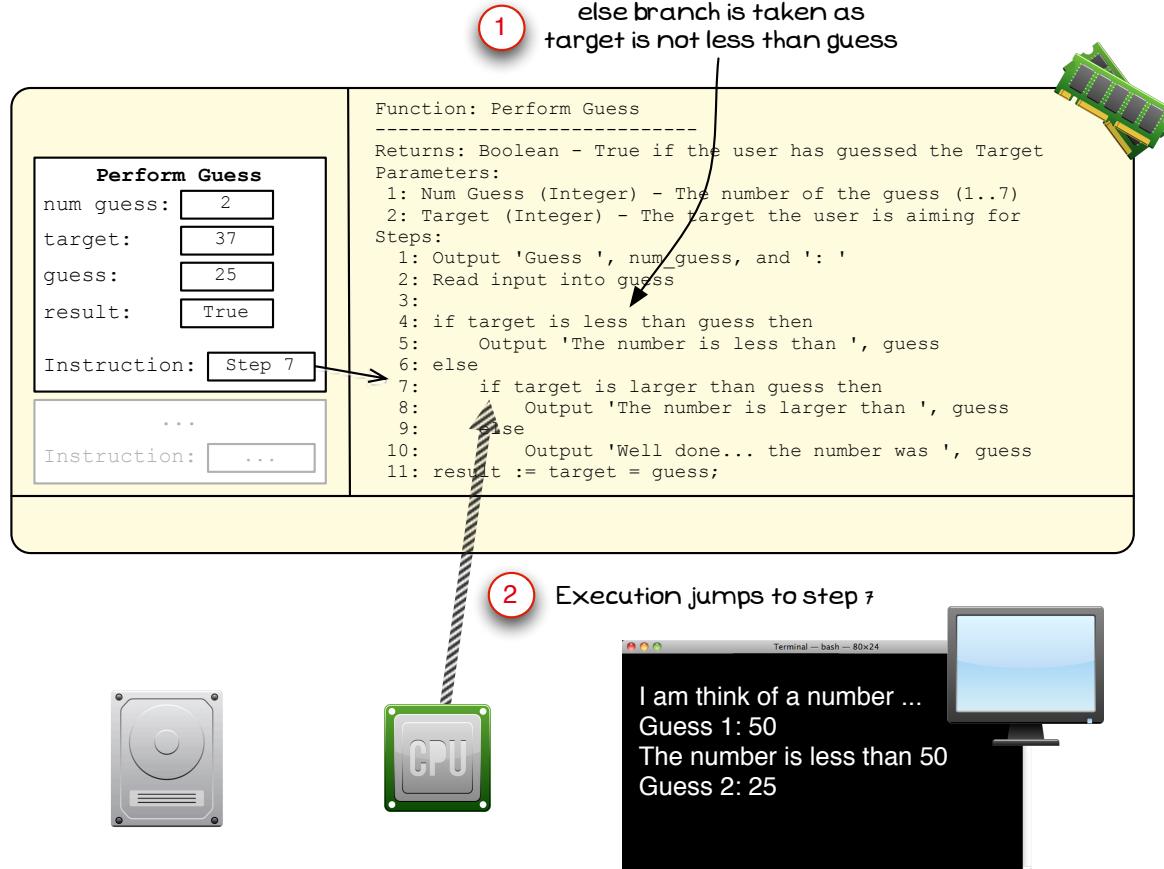


Figure 3.59: Control jumps to step 7 as the target is not less than the guess

Note

- In Figure 3.59 the indicated areas show the following:
 - target is not less than guess so the *else* branch is executed.
 - This means the code jumps to step 7, skipping the *true* branch.
- Step 7 is another **If Statement**, it checks if target is *larger* than guess. This expression is true, so it will take the *true* branch next.

The inner if's true branch is taken in guess 2

The expression in step 7 is true, so the if statement directs the computer into the *true* branch at step 8.

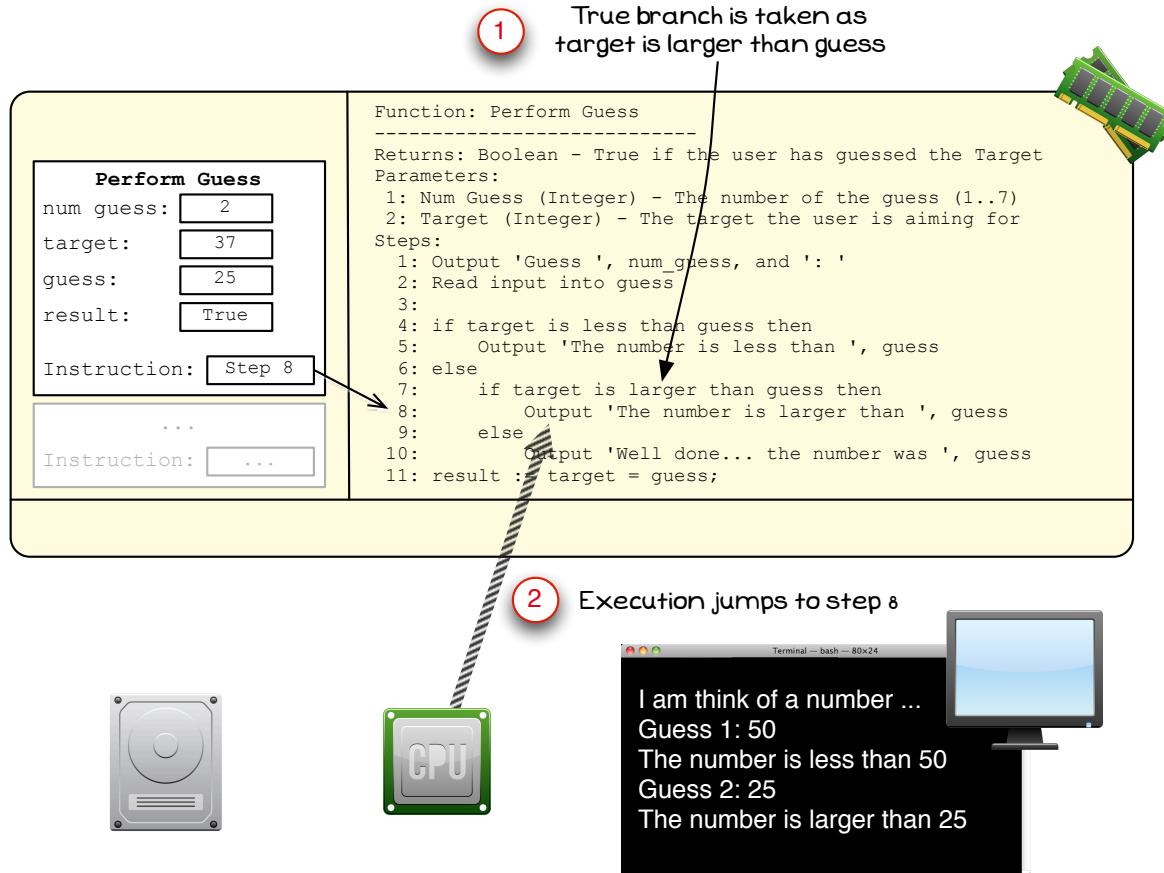


Figure 3.60: Control continues into the *true* branch of the inner if statement

Note

- In Figure 3.60 the indicated areas show the following:
 1. target is larger than guess so the *true* branch is taken.
 2. This means the code jumps to step 8.
- Step 8 outputs the message 'The number if larger than 25' to the Terminal.

Control jumps to the end of Perform Guess for guess 2

Once step 8 finishes, control jumps to step 11, skipping the *else* branch of the if statement from step 7 and also ending the if statement started at step 4.

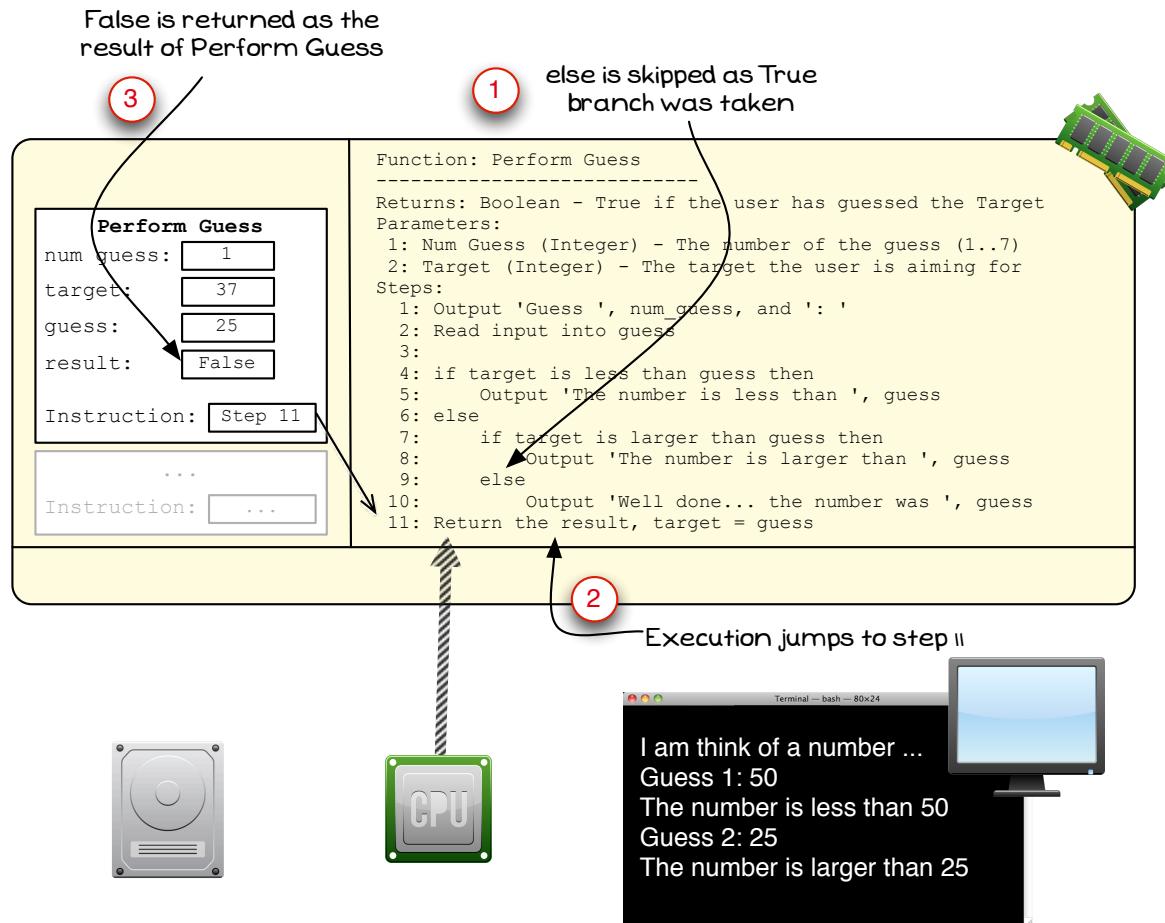


Figure 3.61: Guess 2 ends with Perform Guess returning False

Note

- In Figure 3.61 the indicated areas show the following:
 - The else branch starting at step 9 is skipped as the *true* branch was taken.
 - The next instruction is therefore step 11, ending the if statements starts at steps 7 and 4.
 - Perform Guess returns the result false as the expression target equals guess is false.

Perform Guess is called again for guess 3

Perform Guess is called again, this time it is passed 3 for guess num and 37 for target. This executes the code in Perform Guess, one instruction at a time, eventually reaching step 4.

The next call to Perform Guess
passed 3 to num guess, and
37 to target

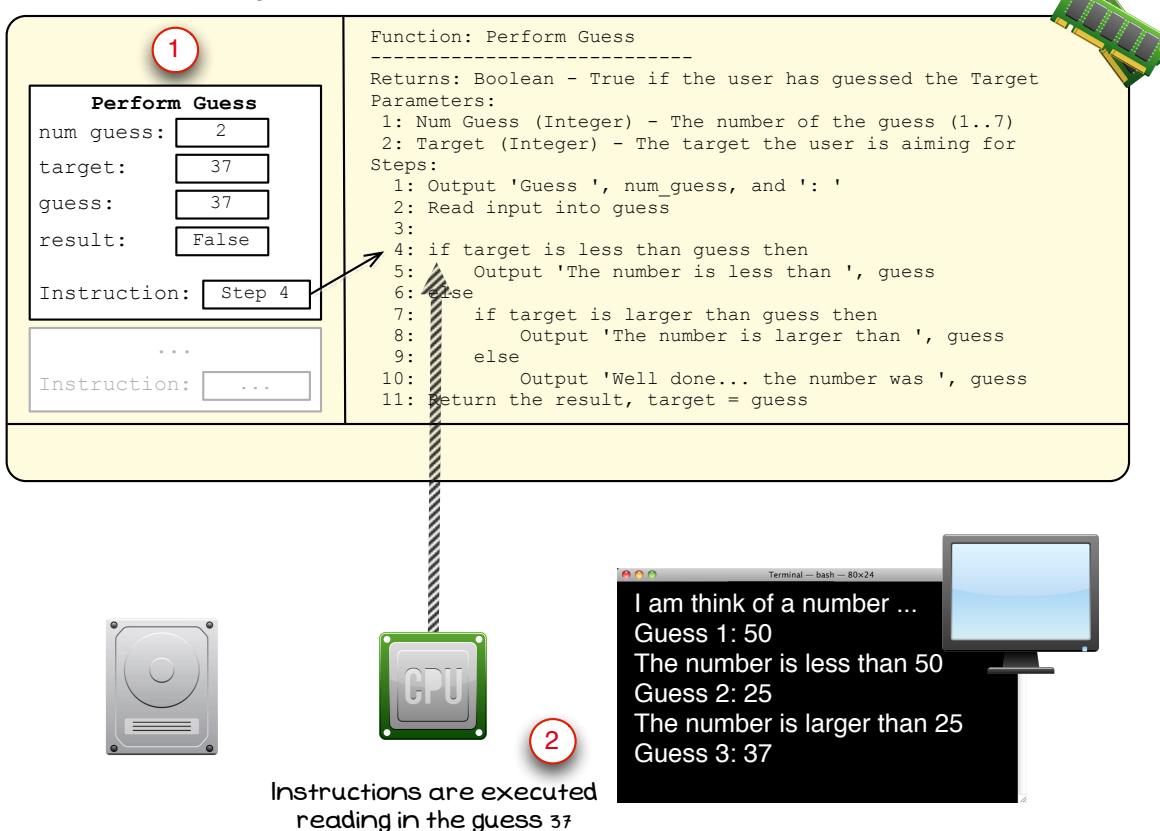


Figure 3.62: Perform Guess is run for guess 3 and advanced to step 4

Note

- In Figure 3.62 the indicated areas show the following:
 - The call to `Perform Guess` is passed 3 for the `guess num` parameter, and 37 for the `target` parameter.
 - The user enters 37 as their `guess`, this is read into the `guess` variable.
- The previous call ended and control returned to the caller. This is now a new call to `Perform Guess`.
- Step 1 outputs a prompt for the user to enter 'Guess 3: '.
- Step 2 reads the user's response into the `guess` variable.
- When the computer executes step 4 it evaluates the `If Statement`'s expression and then selectively runs one of the two available paths. In this case the expression is `false` as `target` is not less than `guess`.

If takes the else branch for guess 3

The target is **not** less than the user's guess, so the *else* branch of the if statement at step 4 is taken, with the computer jumping to step 7.

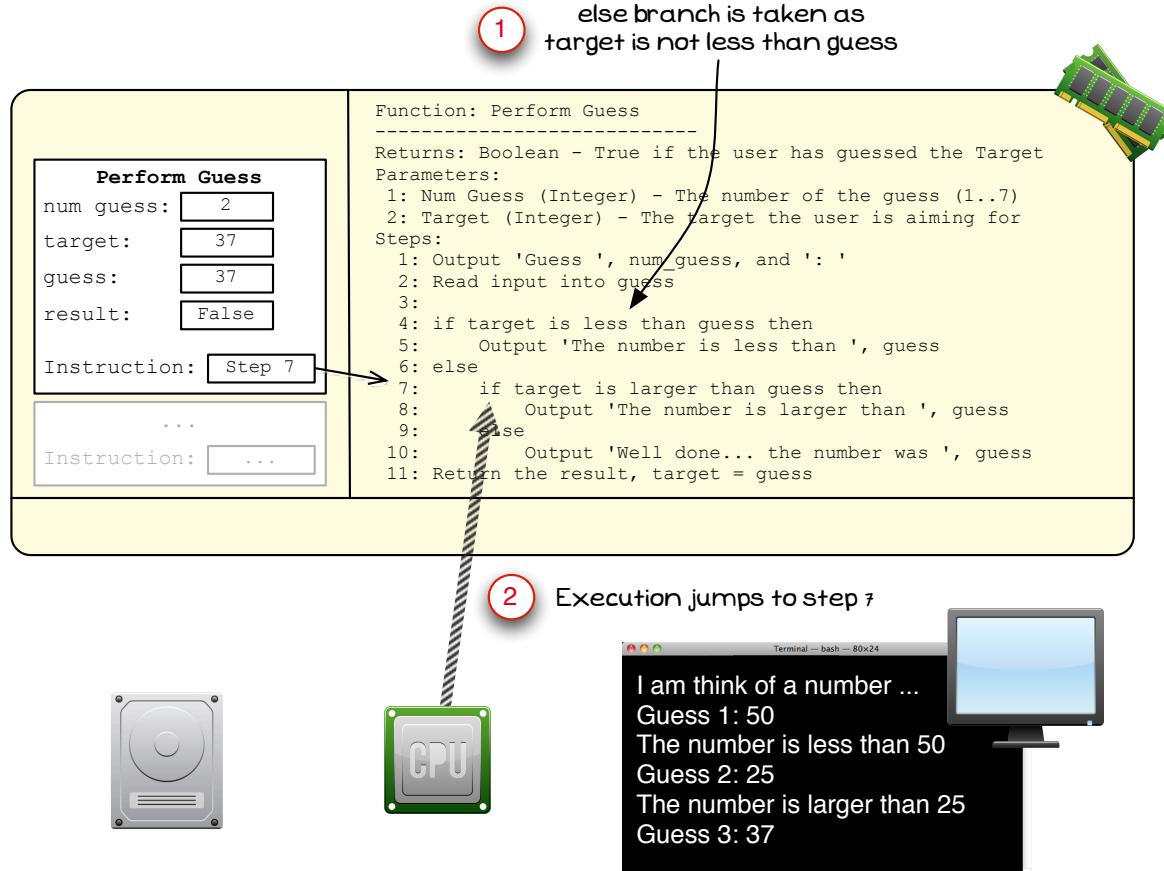


Figure 3.63: The *else* branch is taken as target is not less than guess for guess 3

Note

- In Figure 3.63 the indicated areas show the following:
 - target is not less than guess so the *else* branch is executed.
 - This means the code jumps to step 7, skipping the *true* branch.
- Step 7 is another **If Statement**, it checks if target is *larger* than guess. This expression is also false, target is *not* larger than guess.

The inner if's else branch is taken in guess 3

The target is **not** larger than the user's guess, so the *else* branch of the if statement at step 7 is taken, with the computer jumping to step 10.

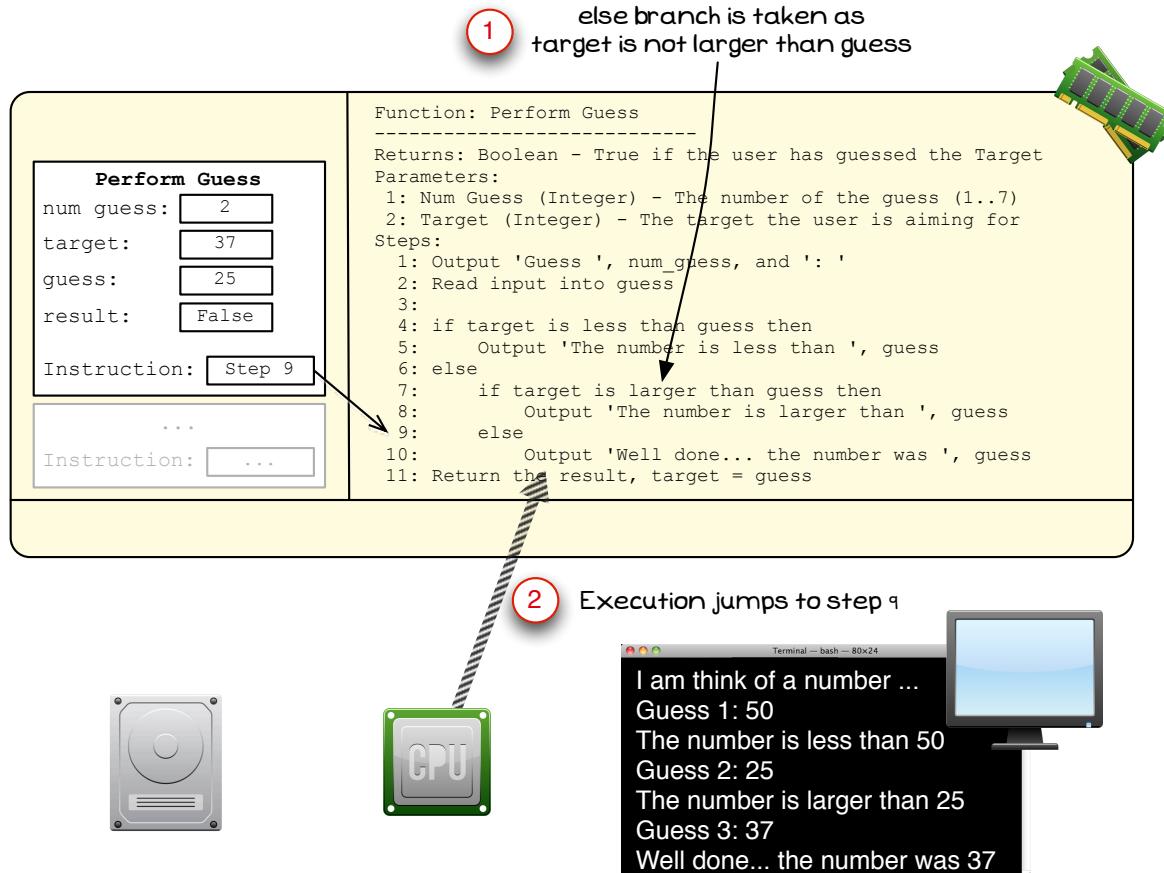


Figure 3.64: The target is not larger than guess, so the *else* branch of the if statement at step 7 is taken.

Note

- In Figure 3.64 the indicated areas show the following:
 - target is not larger than guess so the *else* branch is executed.
 - This means the code jumps to step 10, skipping the *true* branch.
- Step 10 outputs 'Well done... the number was 37' to the Terminal.

Control jumps to the end of Perform Guess for guess 3

Once step 10 finishes, control jumps to step 11, ending the if statements from step 7 and step 4.

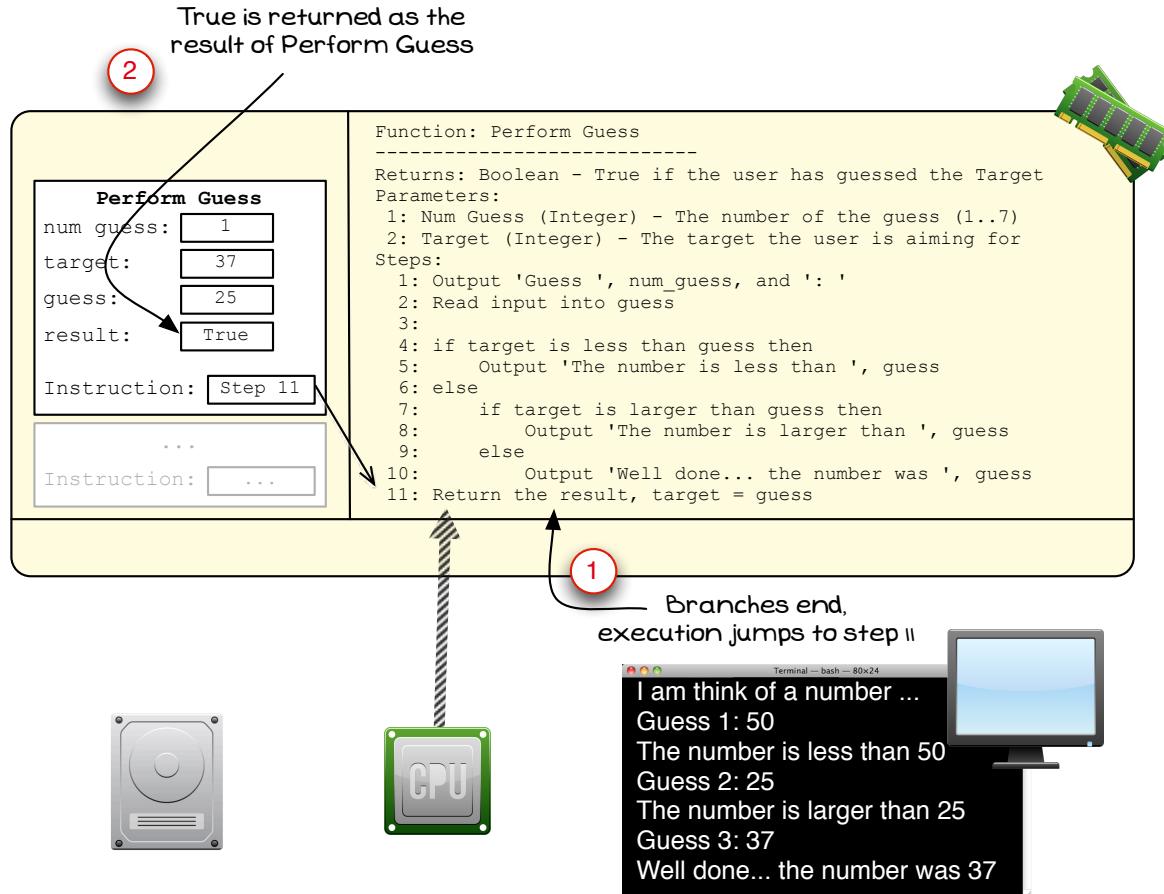


Figure 3.65: Program's entry point is loaded onto the Stack

Note

- In Figure 3.65 the indicated areas show the following:
 - The if statements at steps 7 and 4 end, with control jumping to step 11.
 - The result returned will be true, target does equal guess.

3.5.2 Understanding Looping in Play Game

Figure 3.66 shows the flowchart for the Play Game procedure that was developed in Section 3.2.5 on [Designing Control Flow for Play Game](#). The following sections outline how these actions are executed within the computer.

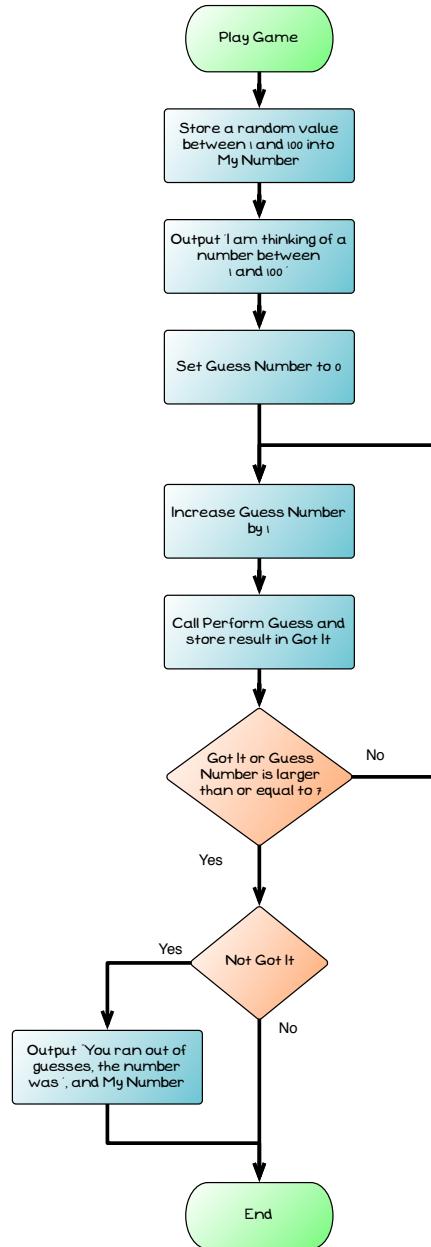


Figure 3.66: Logic for the Play Game Procedure from Figure 3.33

In the following illustrations Play Game will be called once, and will perform three guesses. These three guesses match the calls illustrated in Section 3.5.1 [Understanding Branching in Perform Guess](#). In each case the details of the call into Perform Guess will not be recovered, but you can refer back to the previous section if needed.

The illustrations will show how the loop in the flowchart is handled by the computer. As this includes a do...while/repeat...until loop, the explanations will present both boolean expressions. Please ensure that you check the logic based on the implementation you will use.

Play Game is called

At some point the Play Game procedure is called. This will be responsible for coordinating the actions needed to play one game of Guess that Number. It will call Perform Guess to do the work needed to perform each individual guess.

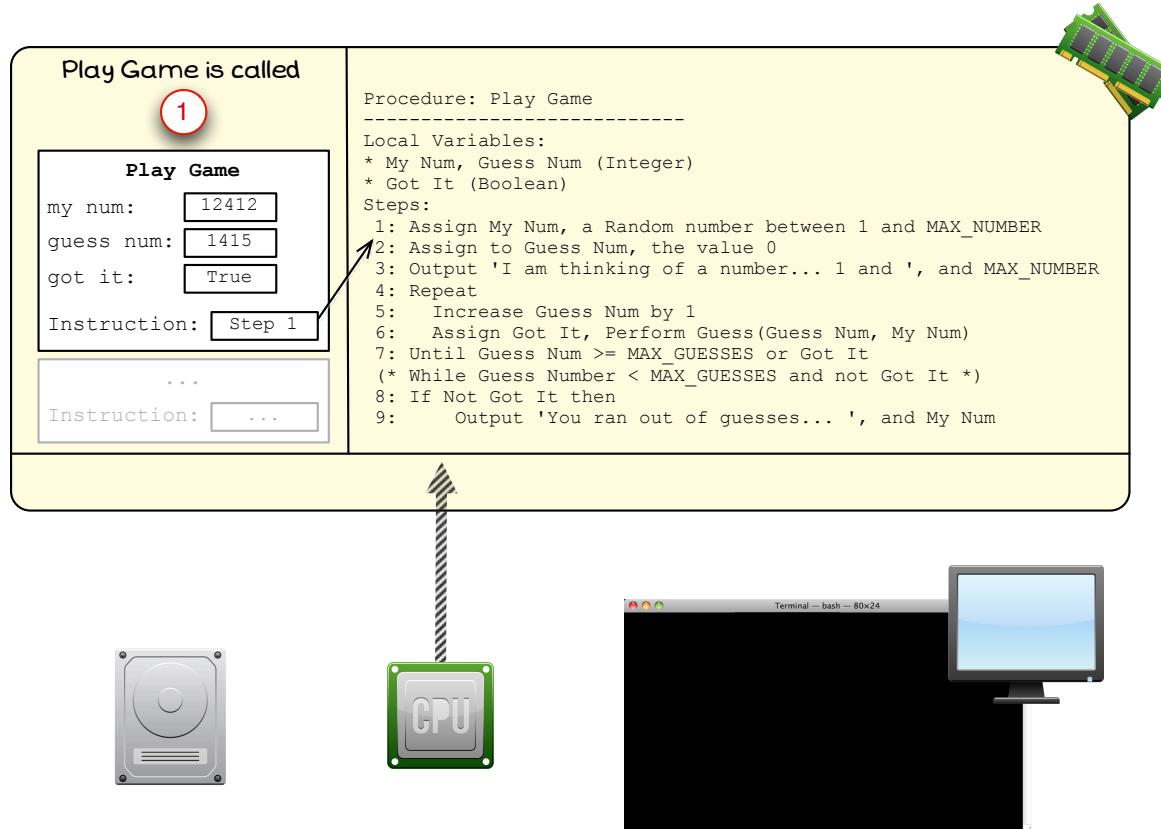


Figure 3.67: Play Game is called, it is allocated space on the Stack for its local variables

Note

- In Figure 3.67 the indicated areas show the following:
 1. Play Game is allocated spaces on the stack for its local variables.
- Local variables are not initialised, so their values are whatever happened to be at that location in memory before.
- Step 1 generates a random number and stores this in `my num`.
- Step 2 initialises the value of `guess num` with the value 0.
- Step 3 will output a message indicating the start of the game.

The loop is entered

Steps 1 to 3 execute and initialise the `my num` and `guess num` local variables.

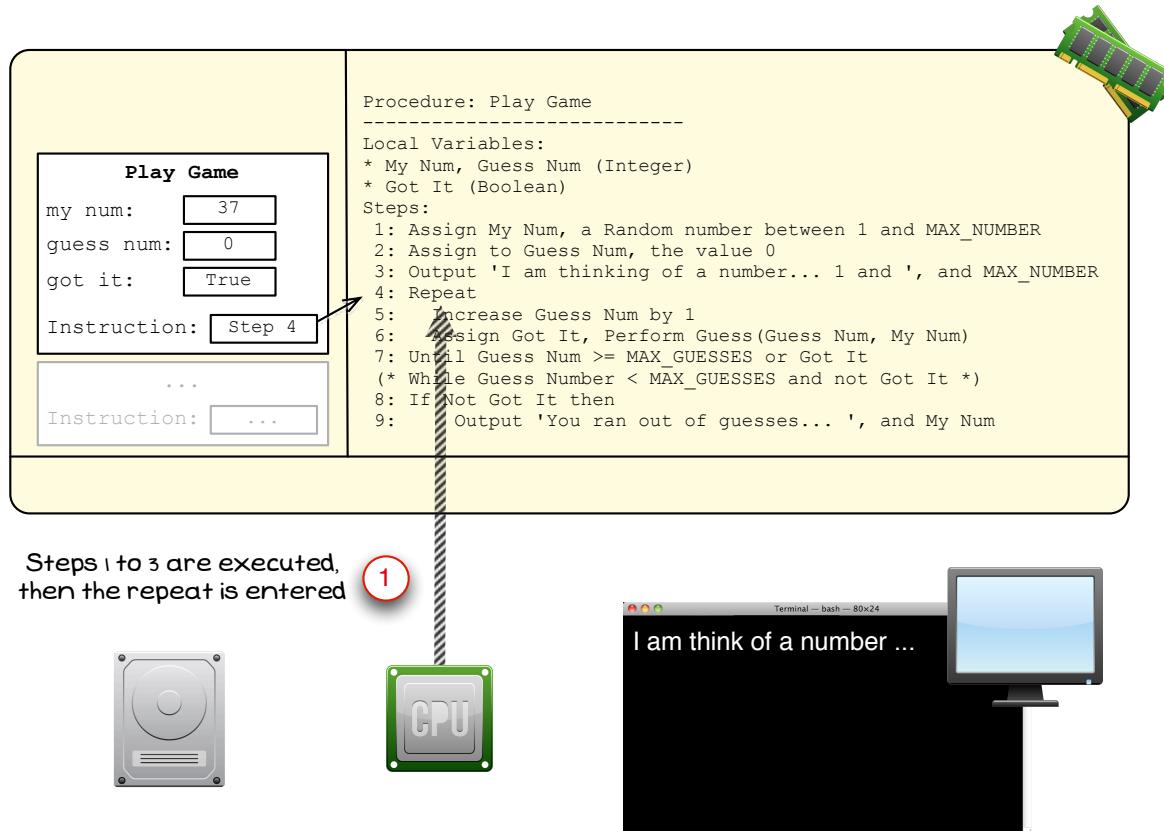


Figure 3.68

Note

- In Figure 3.68 the indicated areas show the following:
 - Steps 1 to 3 are executed, and then control enters the repeated block.
- The repeated instructions will continue as normal.
- If this were a **Pre-Test Loop**, the condition would be checked *before* the loop body was executed. Other than that the two kinds of loops are the same.

C++

Remember that in C the **Post-Test Loop** is the **C Do While Loop**. This is similar to `repeat ...until`, differing only in the way the boolean condition is expressed.

Pascal

Remember that in Pascal the **Post-Test Loop** is `repeat...until`. This is similar to `do ...while`, differing only in the way the boolean condition is expressed.

Perform Guess is called, and returns false for guess 1

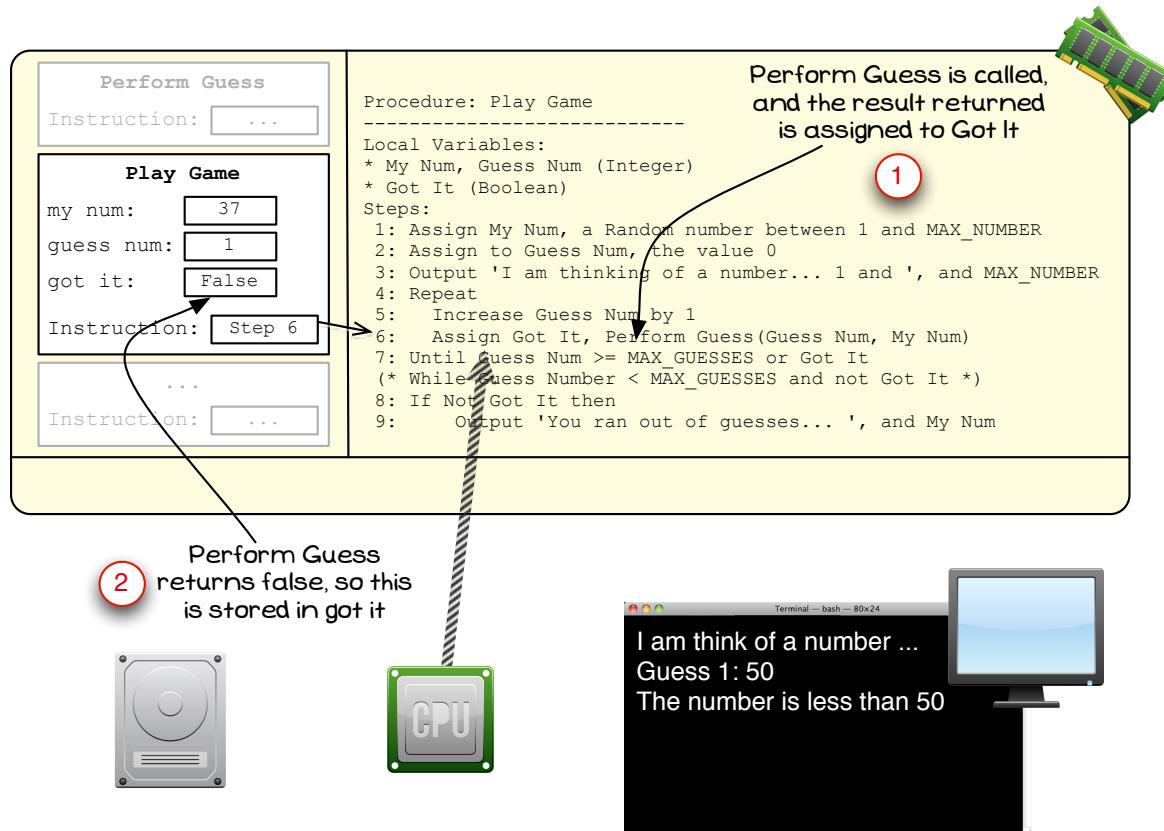


Figure 3.69

Note

- In Figure 3.69 the indicated areas show the following:
 1. This calls `Perform Guess` passing 1 to its `guess num` parameter, and 37 to its `target` parameter.
 2. `Perform Guess` returns false, indicating that the user did not guess the number.
- The following sections outline what occurred in `Perform Guess`:
 - `Perform Guess` is called for `guess 1`
 - Execution reaches the if branch for `guess 1`
 - If takes the True branch for `guess 1`
 - Control jumps to the end of `Perform Guess` for `guess 1`

Loop condition is checked at the end of guess 1, with the loop being repeated

At the end of the loop the condition is checked, in this case the loop will run again.

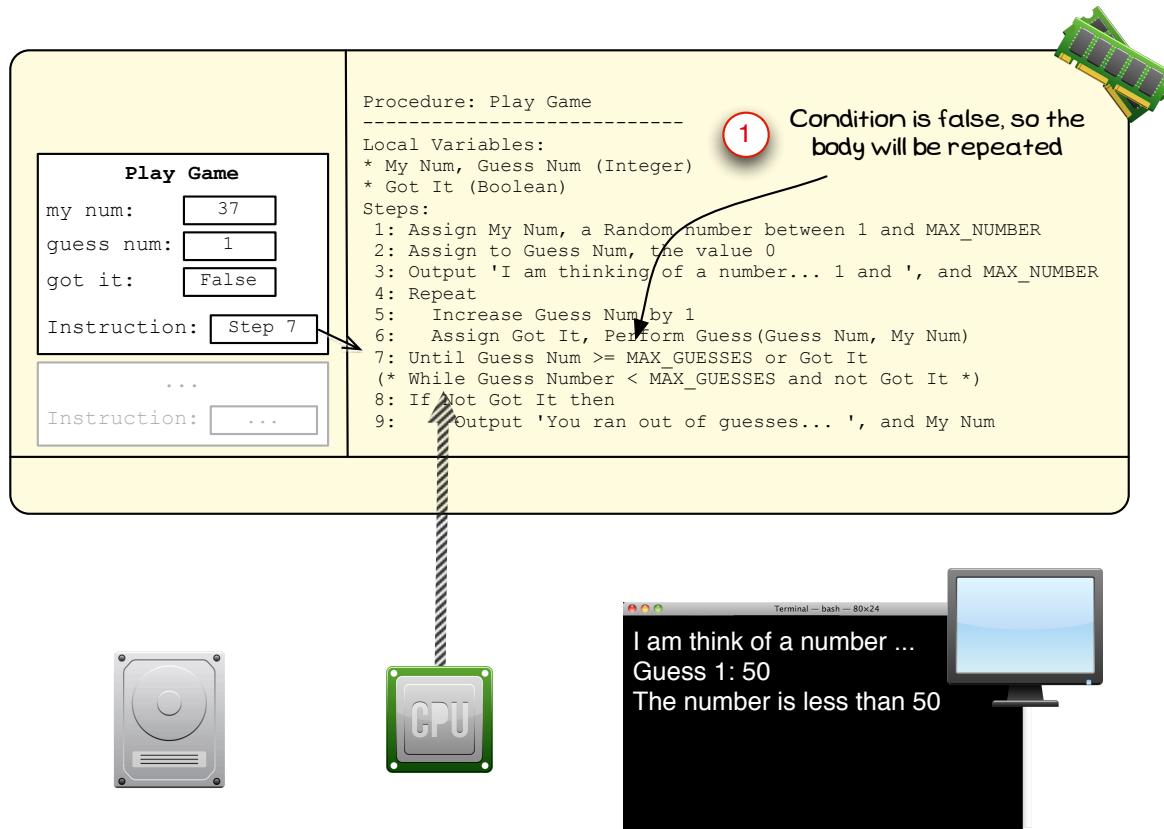


Figure 3.70: Condition indicates that the loop's body should be executed again

Note

- In Figure 3.70 the indicated areas show the following:
 - The condition is checked, and the expression is **false**.
- With **repeat...until** you can evaluate the expression by:
 - Guess Num \geq MAX_GUESSES is $1 \geq 7$, this is **false**
 - Got it, this is a variable, its value is **false**
 - Or the above together, **false or false**, this is **false**, repeating the loop.
- With **do...while** you can evaluate the expression by:
 - Guess Num $<$ MAX_GUESSES is $1 < 7$, this is **true**
 - Got it, this is a variable, its value is **false**, so **!Got it**, is not **false**, is **true**
 - And together these results, **true and true** is **true**, repeating the loop.

C++

For C you will need to code this as a **C Do While Loop**. The code for this will be
`do...while(guess_num < MAX_GUESSES && !got_it);`

Pascal

For Pascal you will need to code this as a Pascal Repeat Until Loop. The code for this will be
`repeat...until (guess_num >= MAX_GUESSES) or (got_it);`

Control returns to the top of the loop to perform guess 2

The loop repeats to perform guess 2.

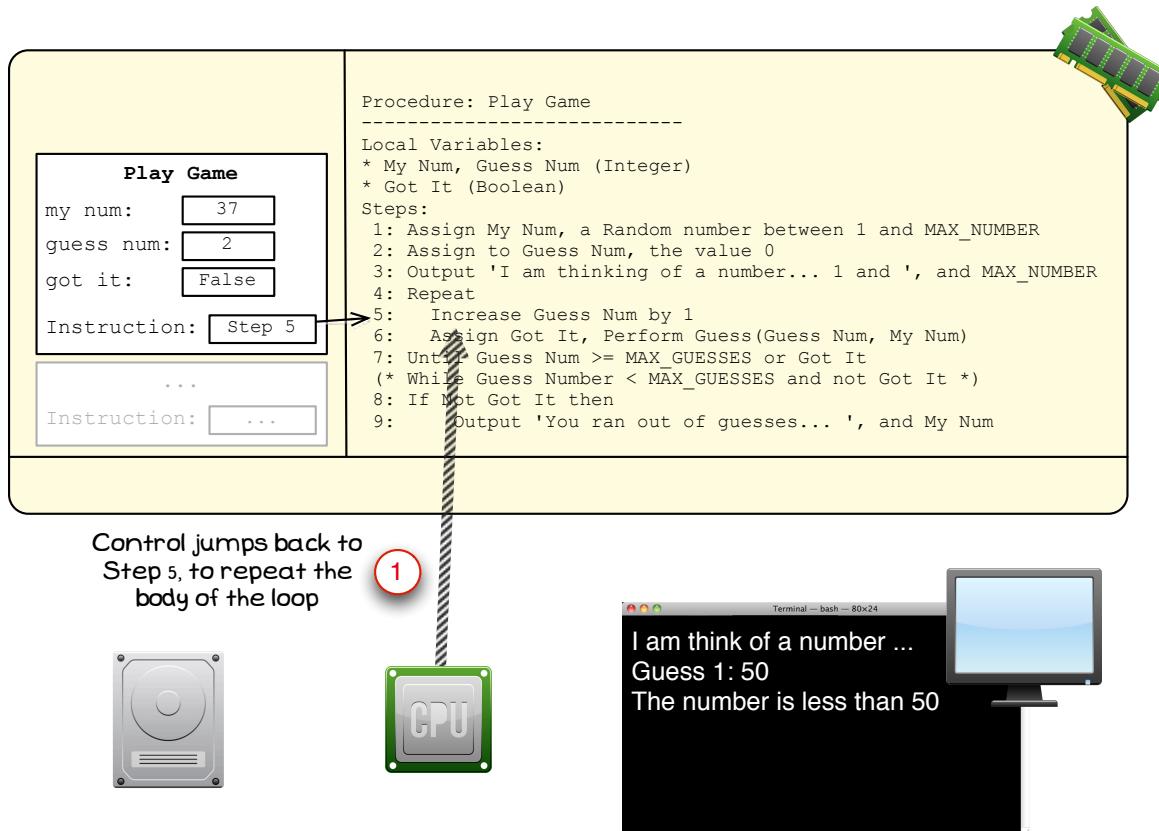


Figure 3.71: Control returns to step 5 to repeat the body of the loop

Note

- In Figure 3.71 the indicated areas show the following:
 1. The loop's body starts at step 5, so control jumps back to this point to repeat the instructions in the loop.
- Step 5 increases the value of the guess num local variable.

Perform Guess is called again, and returns false for guess 2

In the body of the loop, step 5 increases the value in `guess num` to 2 then control continues to step 6. This step calls `Perform Guess`, to allow the user to perform the second guess. This time around it is passed 2 for the `guess num`, and 37 for the target. When `Perform Guess` ends the result is false again, which is stored in `got it`.

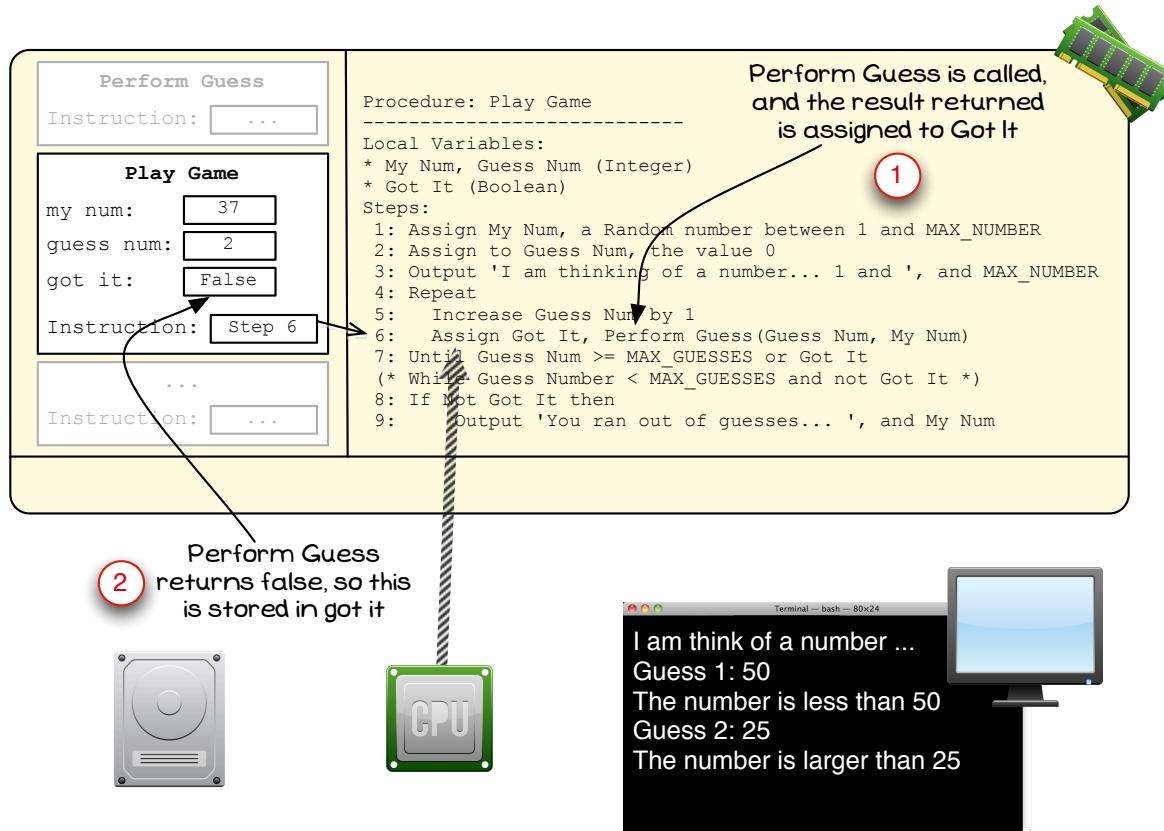


Figure 3.72: `Perform Guess` is called again, it is passed 2 for its `guess num` and 37 for its target parameter

Note

- In Figure 3.72 the indicated areas show the following:
 1. This calls `Perform Guess` passing 2 to its `guess num` parameter, and 37 to its target parameter.
 2. `Perform Guess` returns false, indicating that the user did not guess the number.
- The following sections outline what occurred in this call to `Perform Guess`:
 1. `Perform Guess` is called again for `guess 2`
 2. If takes the `else` branch for `guess 2`
 3. The inner if's `true` branch is taken in `guess 2`
 4. Control jumps to the end of `Perform Guess` for `guess 2`

Loop condition is checked at the end of guess 2, with the loop being repeated

The loop condition is checked again, and it indicates that the body needs to be repeated again.

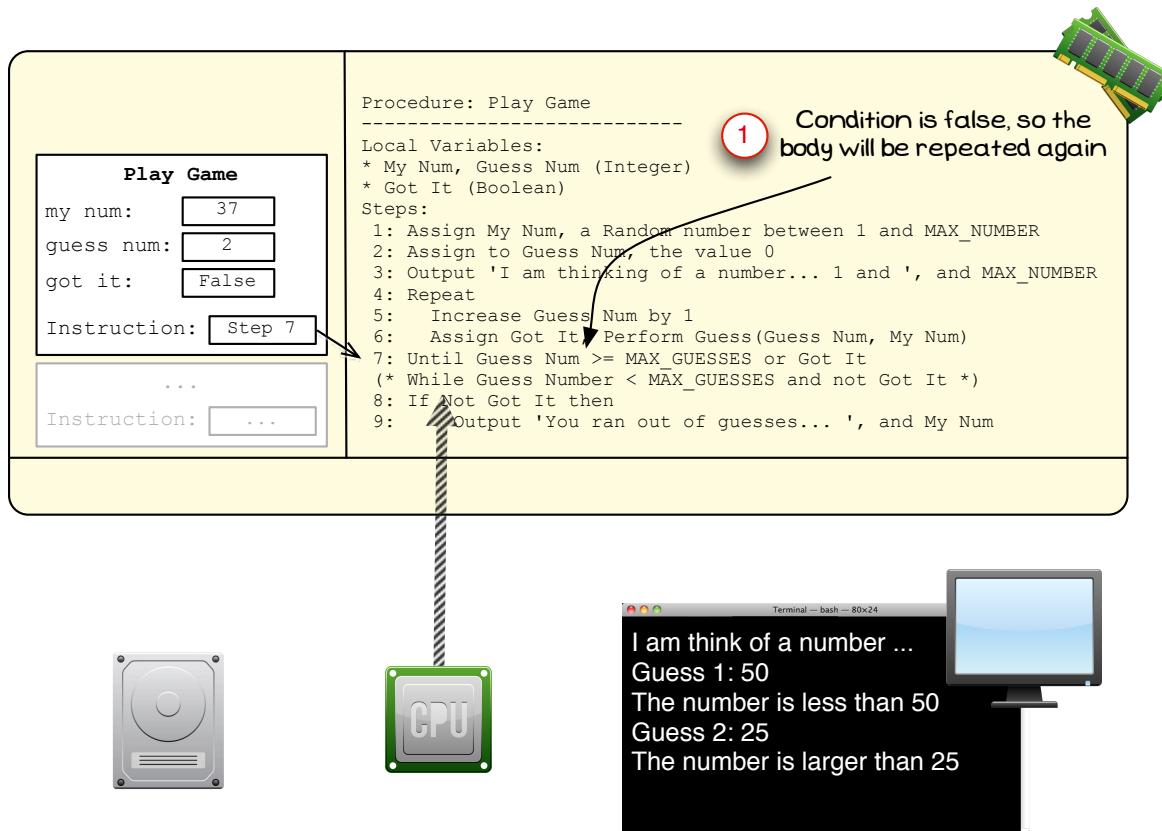


Figure 3.73

Note

- In Figure 3.73 the indicated areas show the following:
 1. The condition indicates that the loop's body needs to be repeated, allowing the user to enter guess 3.
 - The same logic can be applied as shown in [Loop condition is checked at the end of guess 1, with the loop being repeated](#).
 - With **repeat...until** you can evaluate the expression by:
 1. Guess Num >= MAX_GUESSES is $2 \geq 7$, this is **false**
 2. Got it, this is a variable, its value is **false**
 3. Or the above together, **false or false**, this is **false**, repeating the loop.
 - With **do...while** you can evaluate the expression by:
 1. Guess Num < MAX_GUESSES is $2 < 7$, this is **true**
 2. Got it, this is a variable, its value is **false**, so !Got it, is not **false**, is **true**
 3. And together these results, **true and true** is **true**, repeating the loop.

Perform Guess is called a third time, and returns true for guess 3

Just as with guess 2, the body of the loop is repeated. Step 5 increases the value in guess num to 3 then control continues to step 6. This step calls Perform Guess, to allow the user to perform the third guess. This time around it is passed 3 for the guess num, and 37 for the target. When Perform Guess ends the result is now true, which is stored in got it.

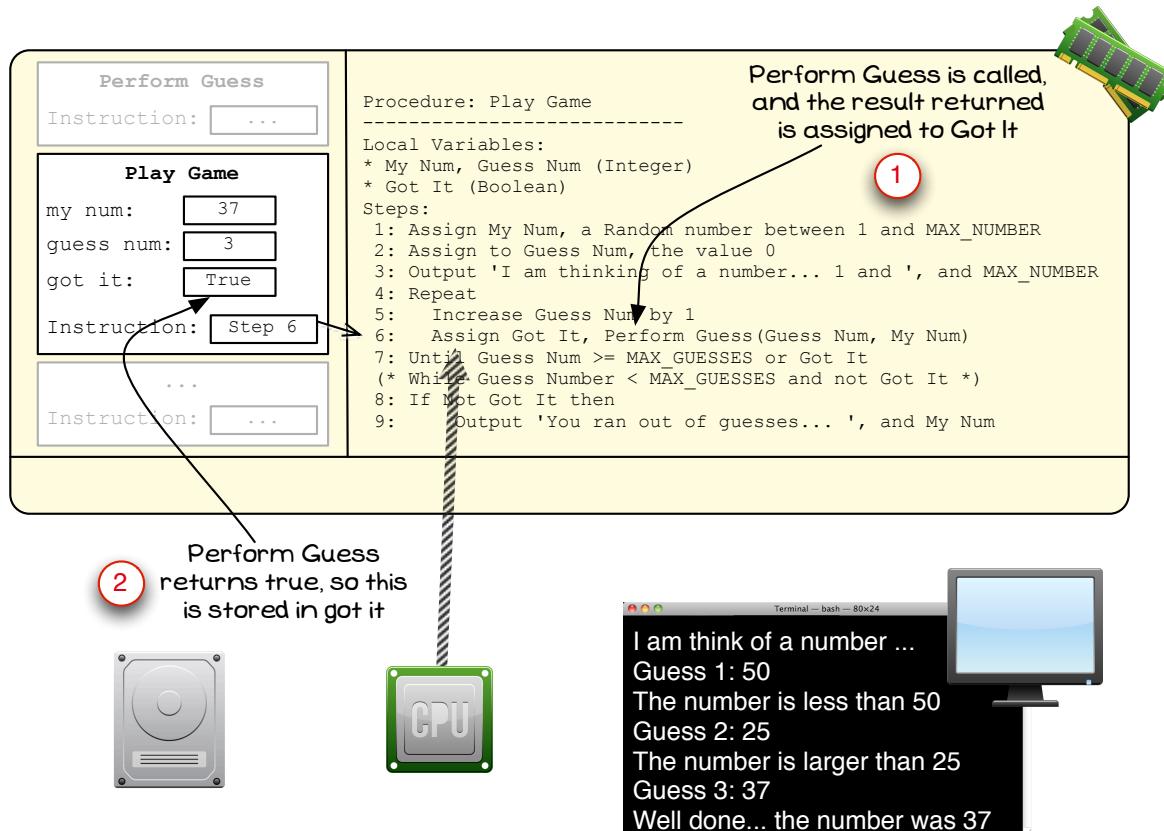


Figure 3.74: Perform Guess is called for the third time, this time the user guesses the number

Note

- In Figure 3.74 the indicated areas show the following:
 1. This calls Perform Guess passing 3 to its guess num parameter, and 37 to its target parameter.
 2. Perform Guess returns true, indicating that the user has guessed the number. We now want the loop to end, as the number has been guessed.
- The following sections outline what occurred in this call to Perform Guess:
 1. Perform Guess is called again for guess 3.
 2. If takes the else branch for guess 3.
 3. The inner if's else branch is taken in guess 3.
 4. Control jumps to the end of Perform Guess for guess 3.

Loop condition is checked at the end of guess 3, ending the loop

The loop condition is checked again, and this time it indicates that the loop should end.

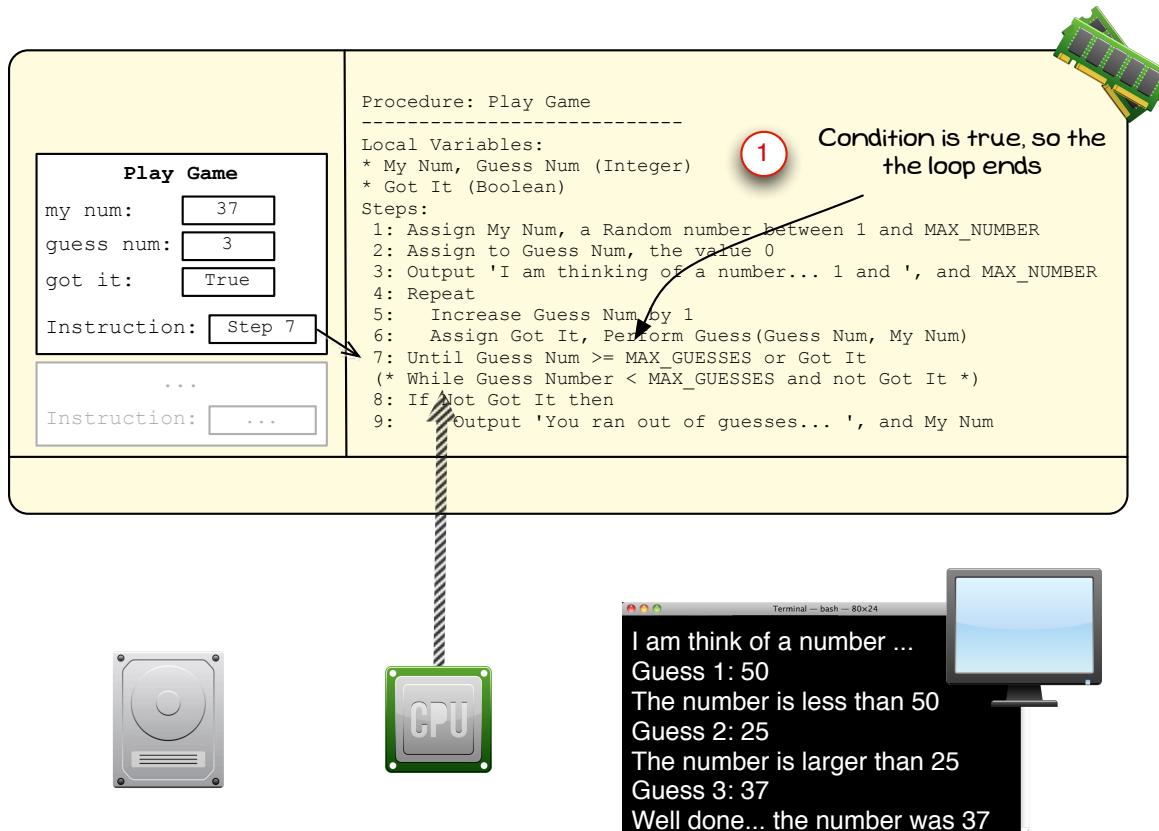


Figure 3.75: Condition is checked, this time it indicates that the loop should end

Note

- In Figure 3.75 the indicated areas show the following:
 - The condition indicates that the loop should end.
- With **repeat...until** you can evaluate the expression by:
 - Guess Num >= MAX_GUESSES is $3 \geq 7$, this is **false**
 - Got it, this is a variable, its value is **true**
 - Or the above together, **false or true**, is **true**, ending the loop.
- With **do...while** you can evaluate the expression by:
 - Guess Num < MAX_GUESSES is $3 < 7$, this is **true**
 - Got it, this is a variable, its value is **true**, so !Got it, is not true, is **false**
 - And together these results, **true and false** is **false**, ending the loop.
- Notice how the got_it boolean is used to stop the loop when the user gets the answer. This is the purpose of this variable.

Branch is checked after the loop

The branch after the loop's body outputs the number if the user did not guess it.

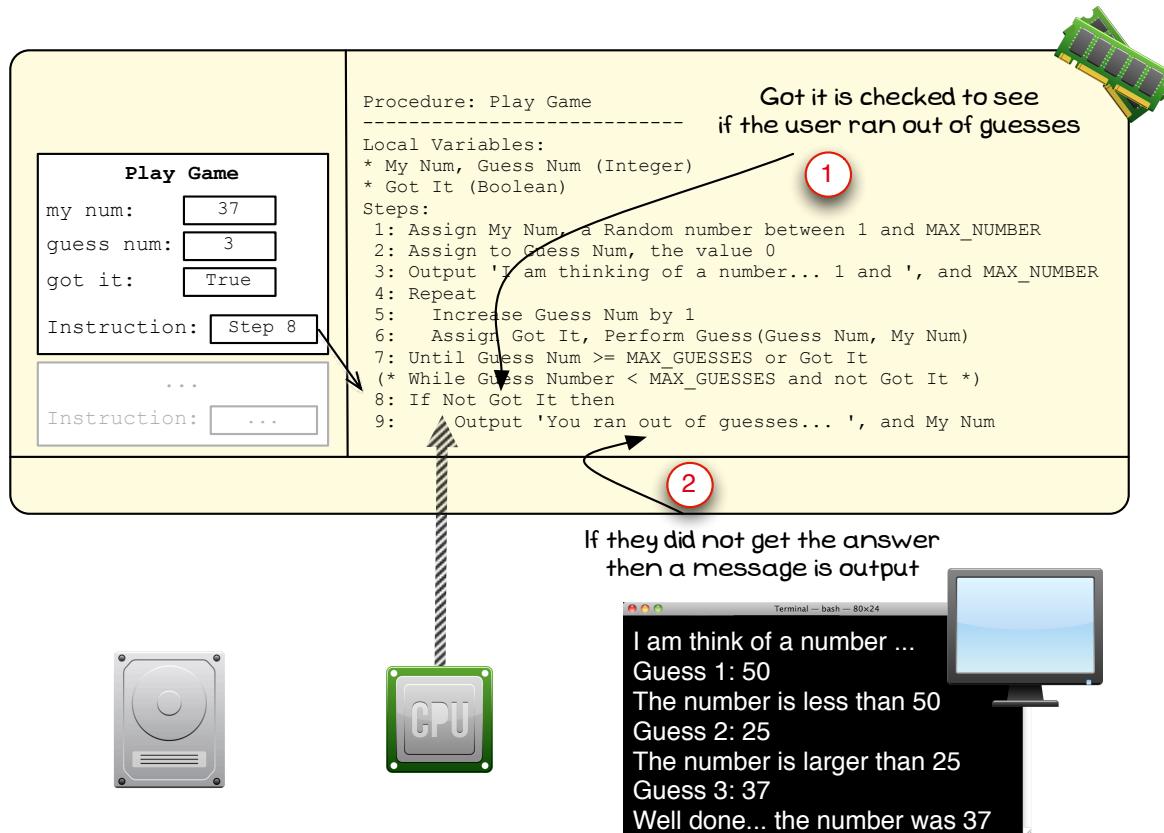


Figure 3.76: Branch after the loop outputs the answer if the users did not guess the number

Note

- In Figure 3.76 the indicated areas show the following:
 - The **If Statement** checks if the user guessed the number (`got it`).
 - If they did not guess the number then the number is output.
- In this case the condition is false, so the *true* branch is skipped. This ends the if statement, and the procedure allowing control to return to the calling code.
- This needed to use `got it`, as the user may guess the number on the last guess.

3.5.3 Understanding looping in Print Line

Figure 3.77 shows the flowchart for the Print Line procedure that was developed in Section 3.2.6 on [Designing Control Flow for Print Line](#). The following sections outline how these actions are executed within the computer.

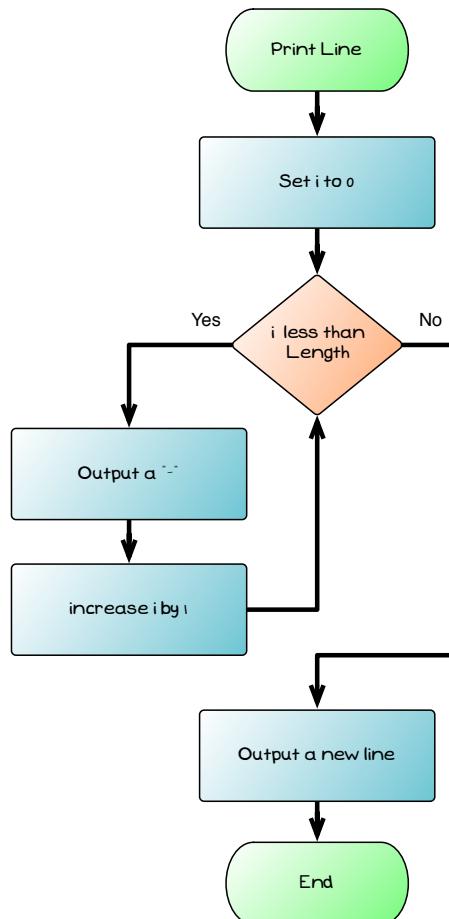


Figure 3.77: Logic for the Print Line Procedure from Figure 3.35

The illustrations will show a single execution of the Print Line procedure, with 3 being passed to its length parameter. This call will output a three line of dash characters, demonstrating how the [Pre-Test Loop](#) differs from the [Post-Test Loop](#).

Print Line is called to print a line of three characters

The illustration starts with the call to Print Line. It is called to print a line of three dash (-) characters to the Terminal.

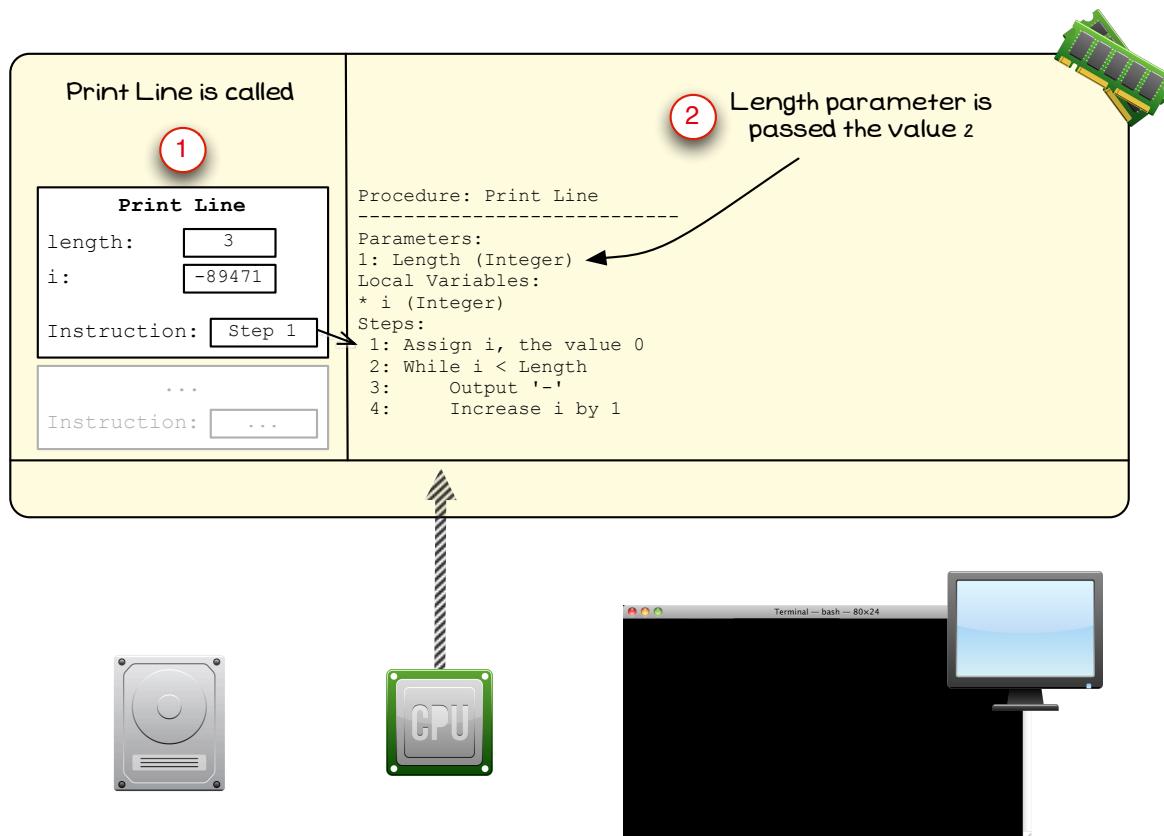


Figure 3.78: Print Line starts, and is passed the value 3

Note

- In Figure 3.78 the indicated areas show the following:
 - At some stage in the program Print Line is called, and its loaded onto the stack.
 - The length parameter is passed the number of dash characters to be printed in the line. In this case that is 3.
- The i local variable is uninitialized, and has the value that was last at that location in memory.

Execution proceeds to the while loop

The first instructions are executed as normal, initialising the value of the *i* variable. At the while loop the computer checks the condition and determines that the loop should execute. This means that the next instruction will be taken from Step 3 of the code.

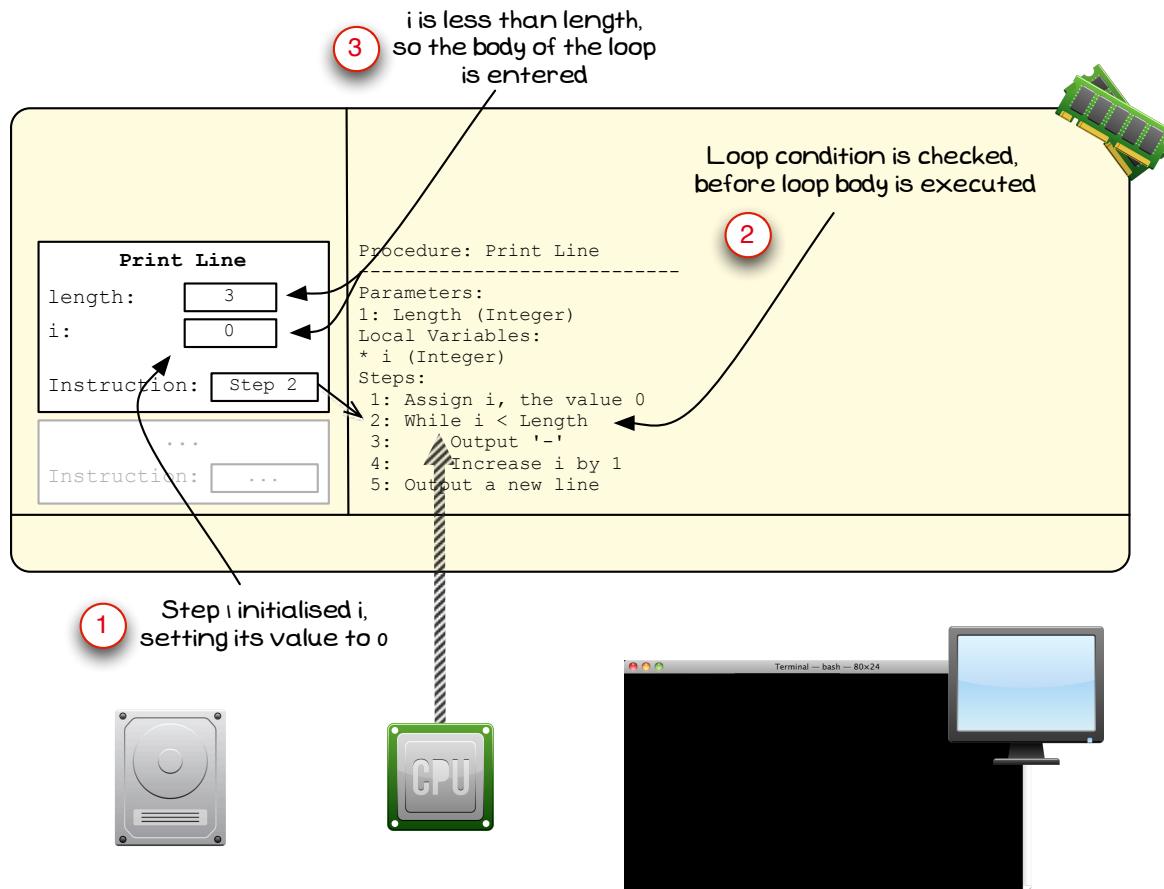


Figure 3.79: *i* is initialised, and the loop checks its condition

Note

- In Figure 3.79 the indicated areas show the following:
 1. Step 1 assigns the value 0 to *i*.
 2. Step 2 checks the condition to determine which path to take.
 3. As *i* is less than length control will flow to Step 3.
- The while loop only checks the condition at the start of the loop. It is at this point that it decides if the code should enter the loop, or if it should end the loop by skipping the loop's body.
- In this case the condition indicate the loop should enter, so control enters the body of the loop.

First dash is output to the Terminal

The body of the loop is entered, and the first instruction (Step 3) outputs a dash to the Terminal.

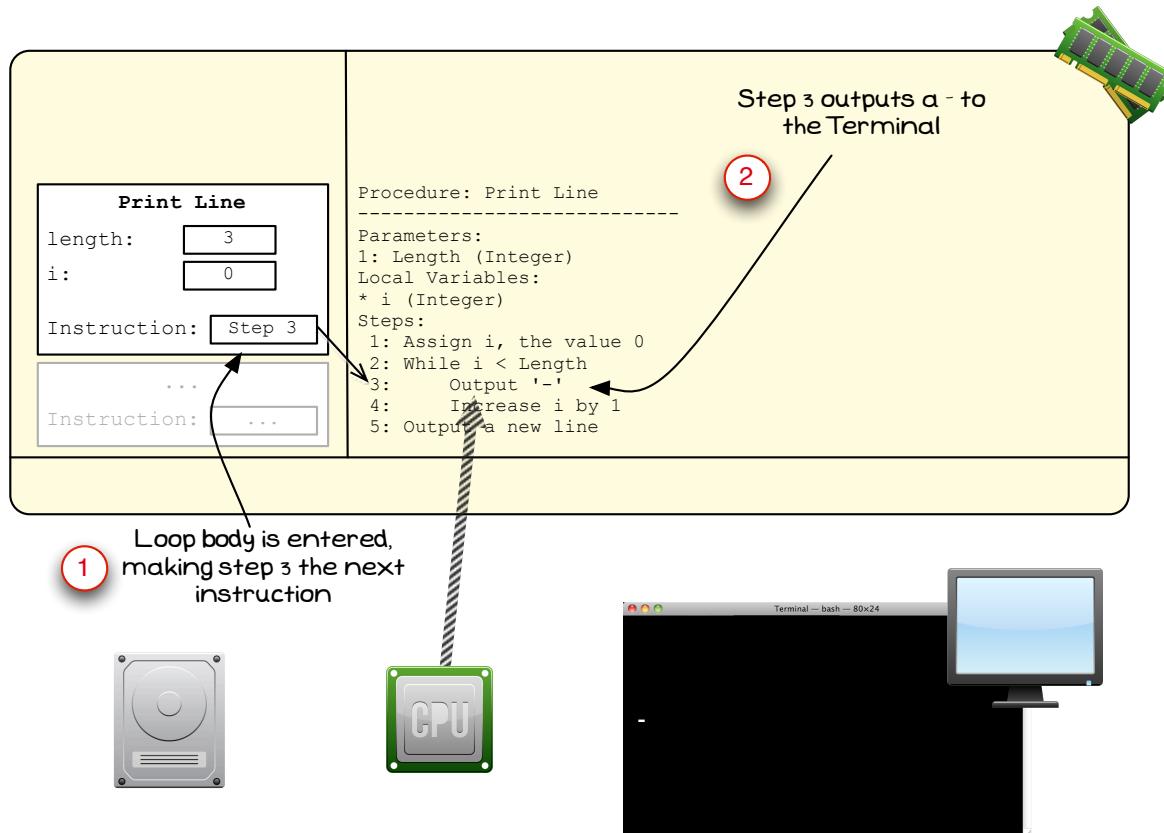


Figure 3.80: The first dash is output to the Terminal

Note

- In Figure 3.80 the indicated areas show the following:
 1. The loop body starts at Step 3.
 2. This step outputs a dash to the Terminal.
- You could use this kind of loop to perform any action a number of times. The body of the loop is then performed once each time the loop executes.

i is incremented, counting the number of times the loop has run

After outputting the dash, the next instruction (Step 4) increments the value of i. In this code i is counting the number of times the loop has been performed. This is the end of the first loop, so now i has the value 1.

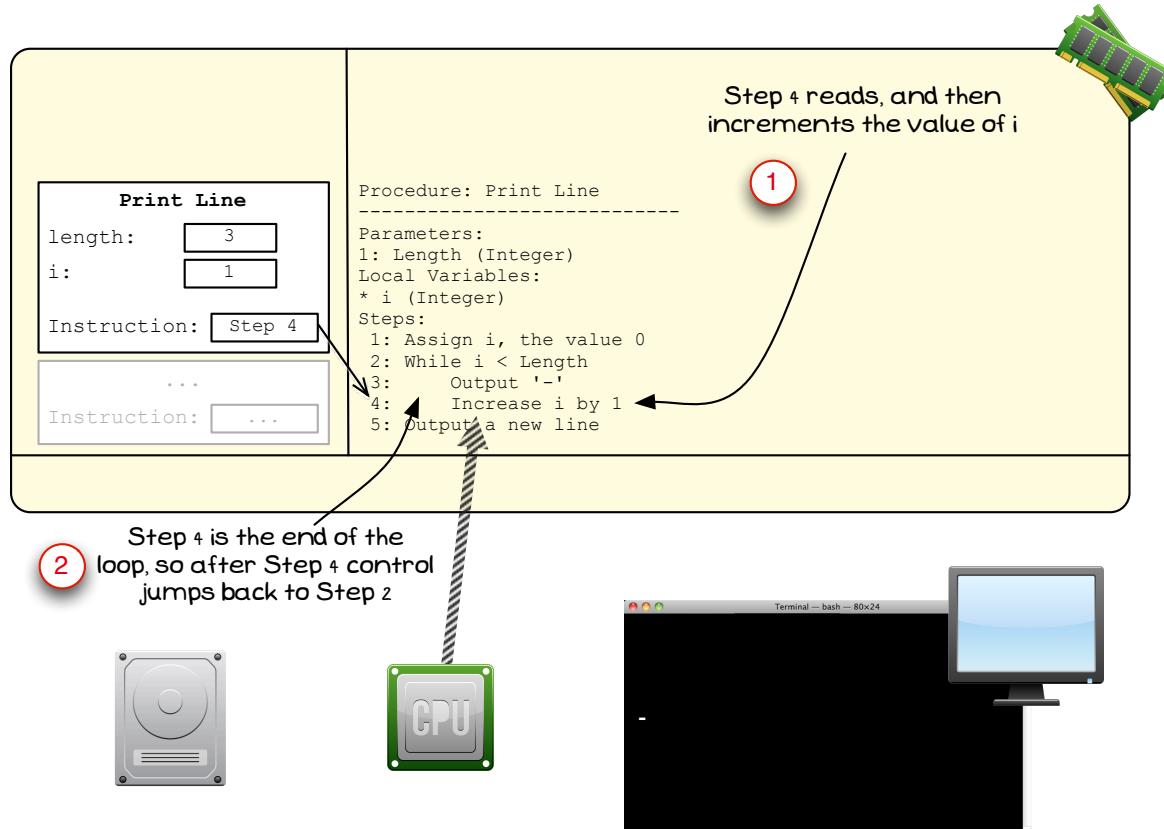


Figure 3.81: i is keeping track of the number of times the loop has been performed

Note

- In Figure 3.81 the indicated areas show the following:
 - Step 4 increments the value of i, allowing it to keep track of the number of times the loop has executed.
 - Control has reached the end of the loop, but will now return back to the condition.
- The compiler adds jump code to the end of each [Pre-Test Loop](#). This returns control to the loop's condition, thereby making the loop.
-

Condition is checked again, and the body of the loop reentered

The while loop checks the condition to determine if the body should be executed again, or if the loop should end. In this case *i* is still less than *length* so the body of the loop is executed again.

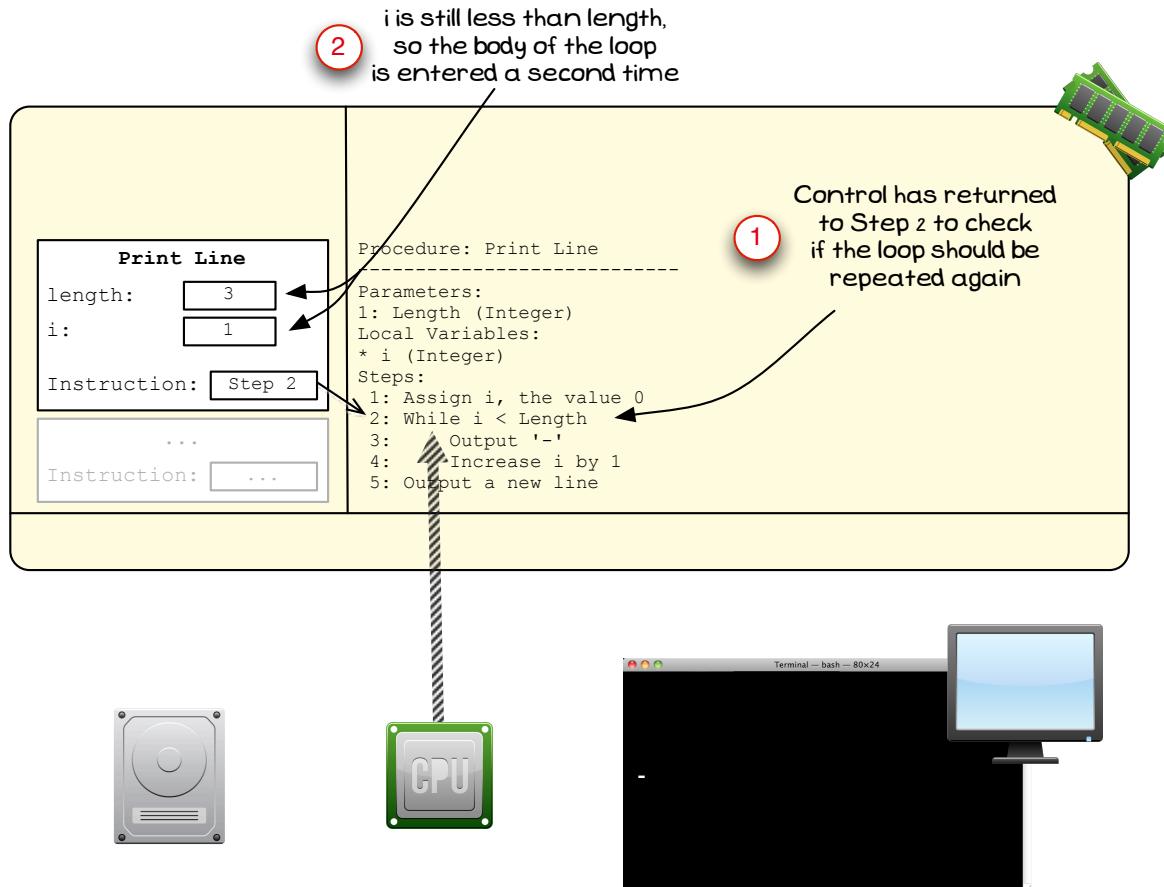


Figure 3.82: Loop condition is checked to determine if the loop should run again

Note

- In Figure 3.82 the indicated areas show the following:
 1. The condition is checked again, to determine if the body is run or skipped.
 2. In this case the loop body is run again as *i* is less than length.
- The **Pre-Test Loop** is controlled by the condition at the start of the loop.

Loop 2 starts, outputting the second dash

The loop body is run again with its first instruction (Step 3) outputting another dash to the Terminal.

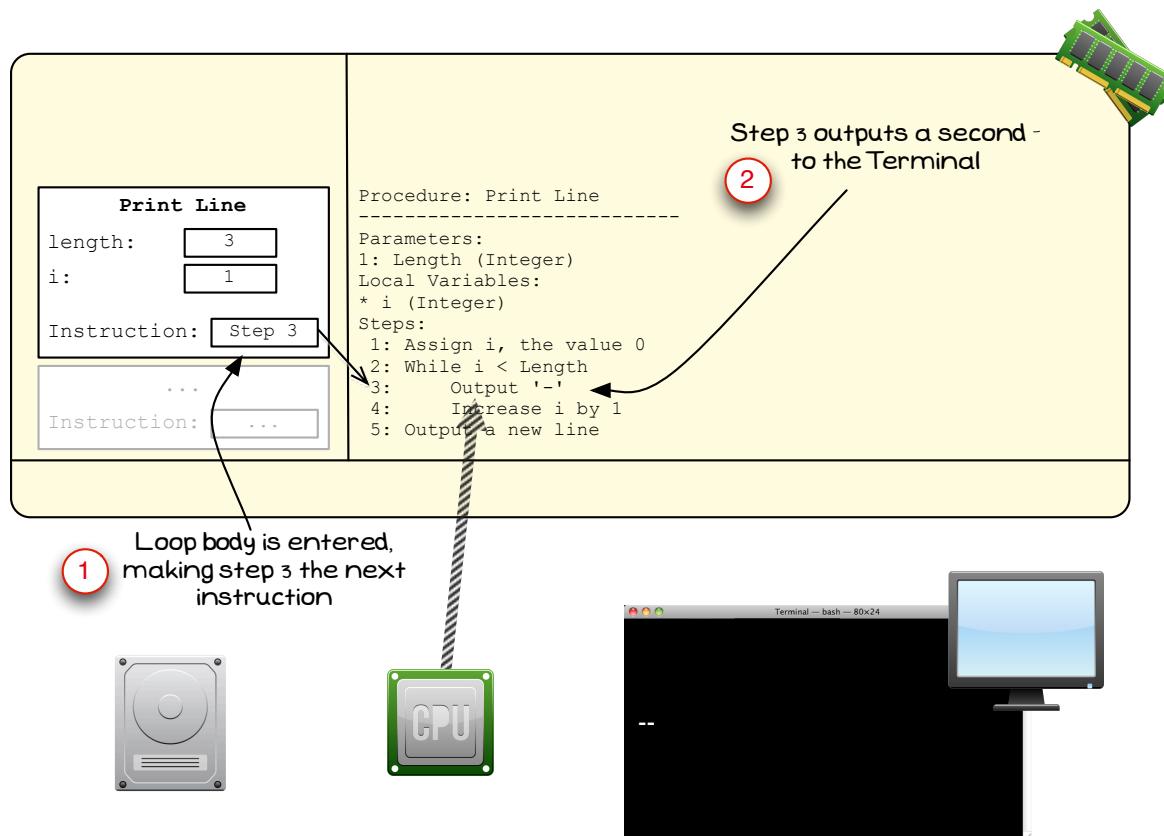


Figure 3.83: The loop body is run again, and a second dash is output

Note

- In Figure 3.83 the indicated areas show the following:
 1. As before, the loop body starts at Step 3.
 2. This step outputs a second dash to the Terminal.
- All the instructions in this sequence will be executed each time the loop is repeated.

i is incremented again, indicating that the loop has run twice

The last instruction in this sequence increments the value of i, indicating that the loop has run twice. Once again, control returns jumps back to the condition. It is the condition that will determine when the loop ends, the end of the loop just indicates that control will return back to the condition.

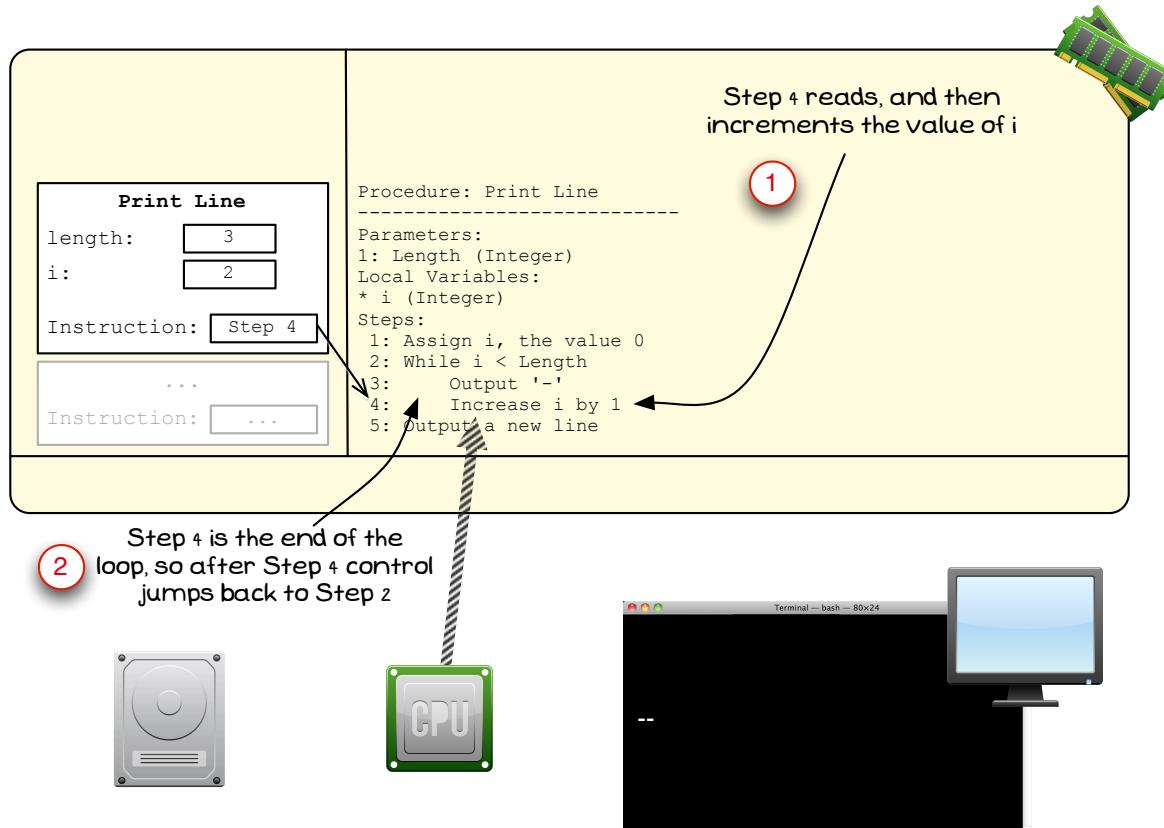


Figure 3.84: i is incremented, and then control returns back to the loop's condition

Note

- In Figure 3.84 the indicated areas show the following:
 1. Step 4 increments the value of i. This is keeping track of the number of times through the loop, indicating that the loop has run twice.
 2. Control has reached the end of the loop and now jumps back to the condition.
- The **Post-Test Loop** will always be performed in this way. The body of the loop runs, and then control jumps back to the condition.

Condition is checked again, and the loop runs a third time

When the condition is checked it determines if the loop body runs, or is skipped. In this case *i* is still less than *length* so the body is run a third time.

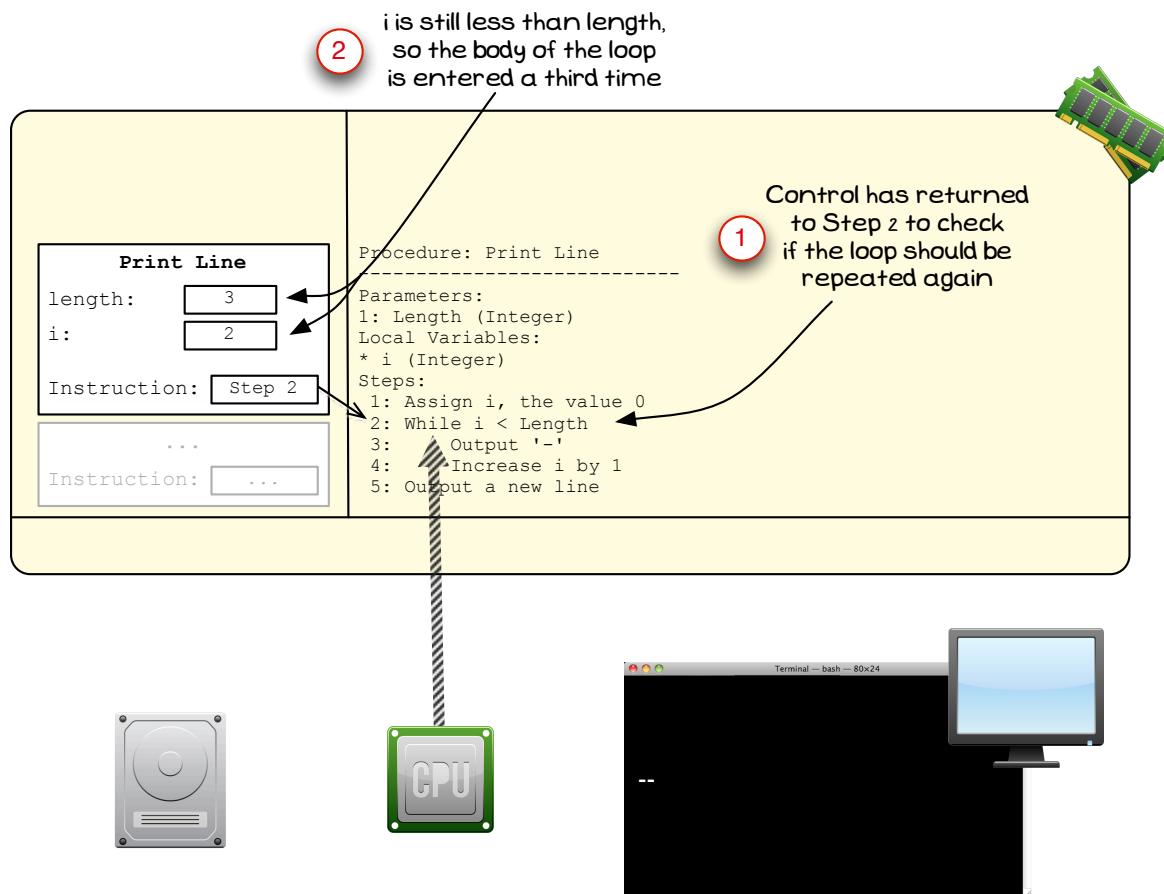


Figure 3.85: While determines that the loop should run again

Note

- In Figure 3.85 the indicated areas show the following:
 - The condition is checked again, to determine if the body is run or skipped.
 - In this case the loop body is run again, as ***i* is less than *length***.
- It is important to notice that this is only checked once each time the loop runs. It is checked when the condition is checked upon entry, and then again each time the loop jumps back to this step.

Another dash is output as the loop body runs a third time

The loop body is entered a third time, and its sequence of instructions is executed. This outputs a third dash to the Terminal.

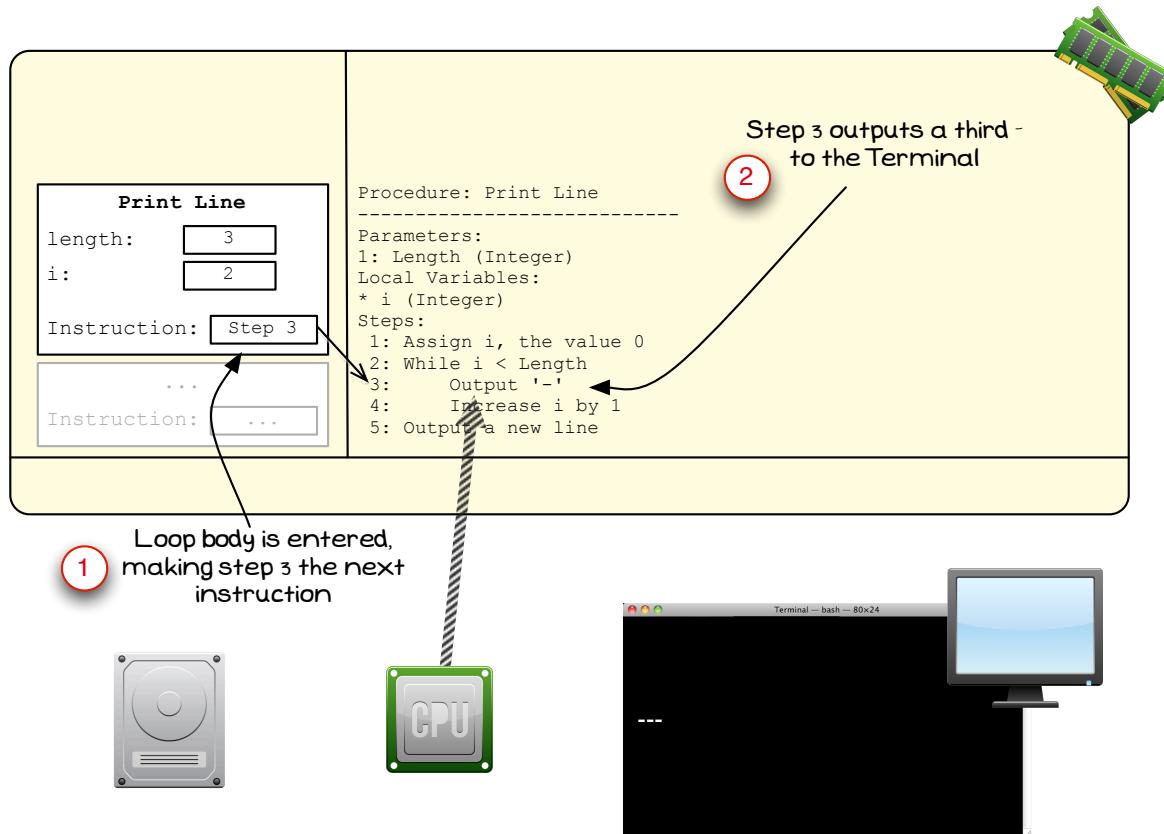


Figure 3.86: The third dash is output to the Terminal

Note

- In Figure 3.86 the indicated areas show the following:
 1. As before, the loop body starts at Step 3.
 2. This step outputs a second dash to the Terminal.
- As before each instruction within the loop's body will be executed.

i is incremented again, indicating the loop has been performed three times

When Step 4 is executed i is incremented again, and now has the value 3. As this is the end of the loop, control will jump back to condition.

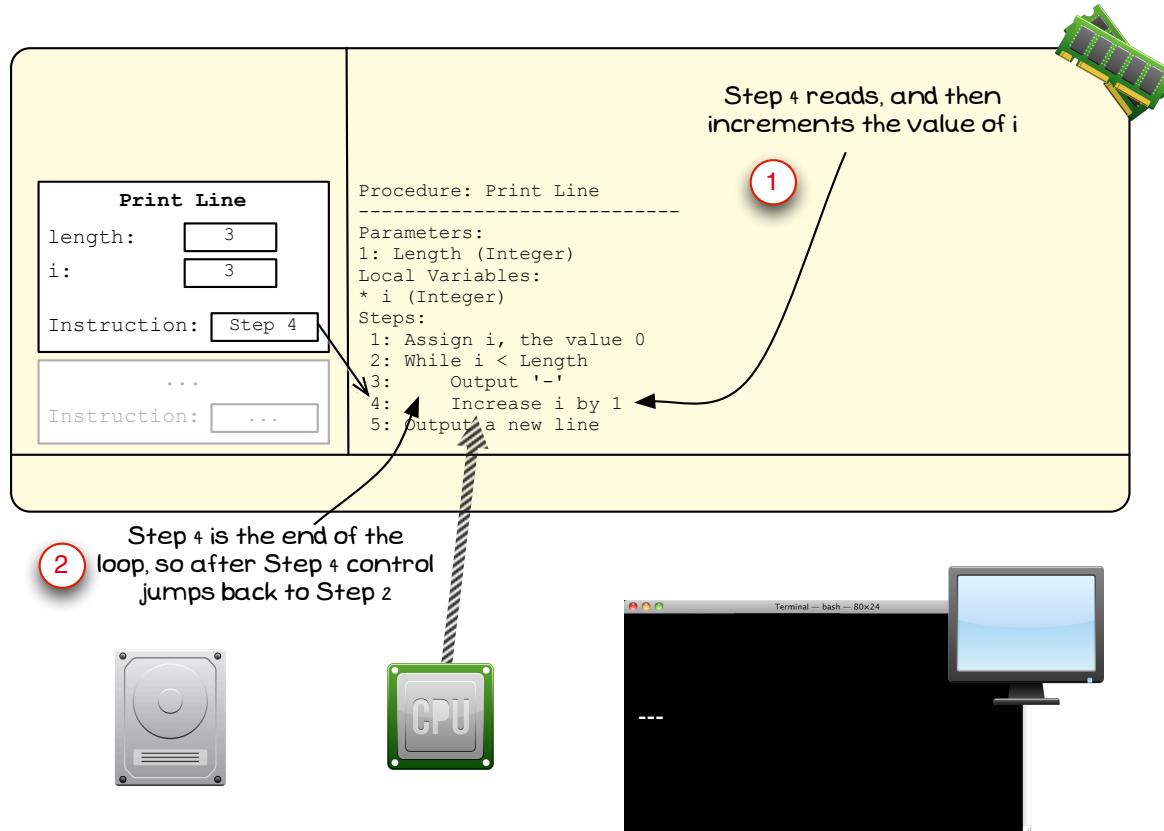


Figure 3.87: i is incremented again, and then control jumps back to the condition

Note

- In Figure 3.87 the indicated areas show the following:
 1. Step 4 increments the value of i. This is keeping track of the number of times through the loop, indicating that the loop has run twice.
 2. Control has reached the end of the loop and now jumps back to the condition.
- The loop is controlled by the condition at the start, so it does not end at this point but it will end when the condition is next checked.

Condition is checked a fourth time, and the loop body is skipped

The condition is checked again, and this time **i is not less than length** and so the loop body is not run again. Instead the computer jumps to the first instruction after the loop, Step 5.

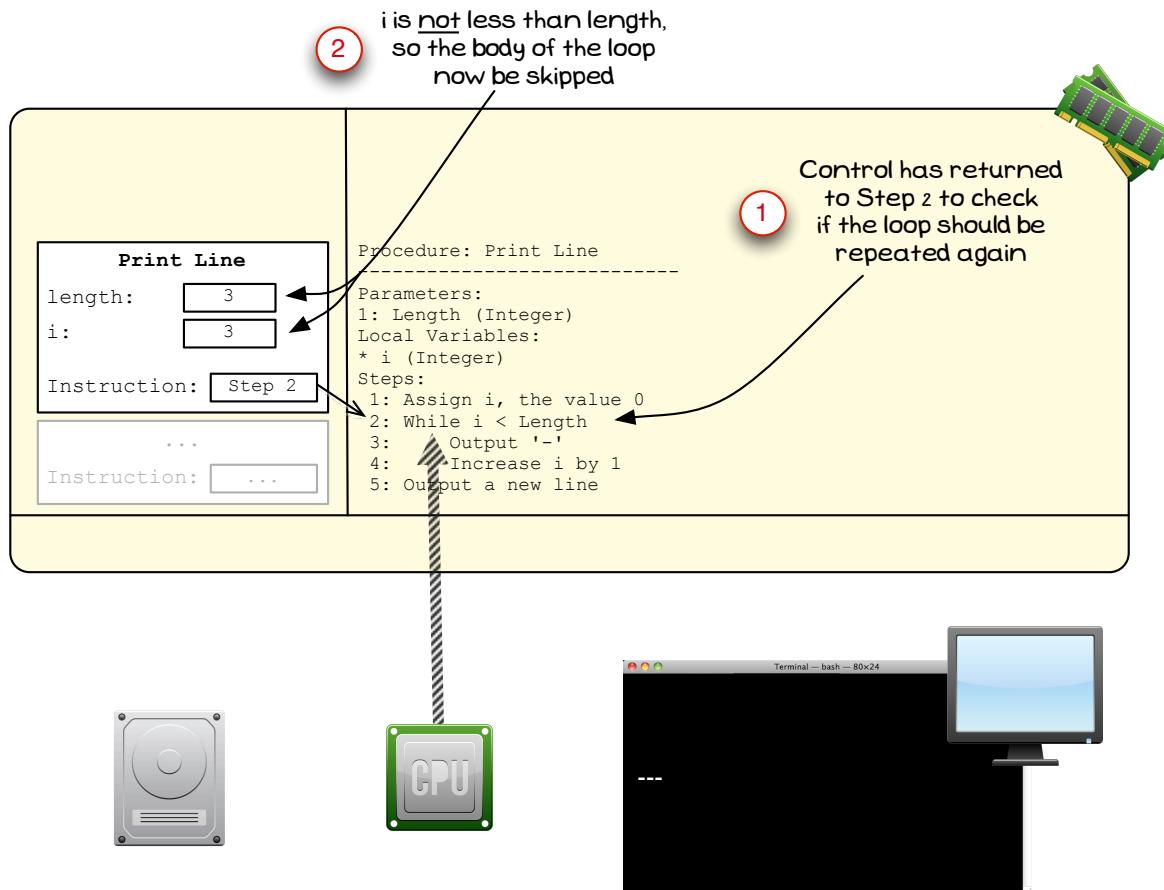


Figure 3.88: The condition is now false, and so the computer jumps past the loop body to Step 5

Note

- In Figure 3.88 the indicated areas show the following:
 1. After the loop body completes, control **always** returns back to the start of the loop. It is this check that is controlling the loop, and it is only performed at this point.
 2. At this point **i is not less than length**, and therefore the body of the loop must be skipped.
- The next instruction will be Step 5, the first instruction past the end of the loop. This is the **single exit** point from the loop.

Having completed the loop, the next instruction outputs a new line

Now that the loop has finished, control continues running the sequence from the Procedure.

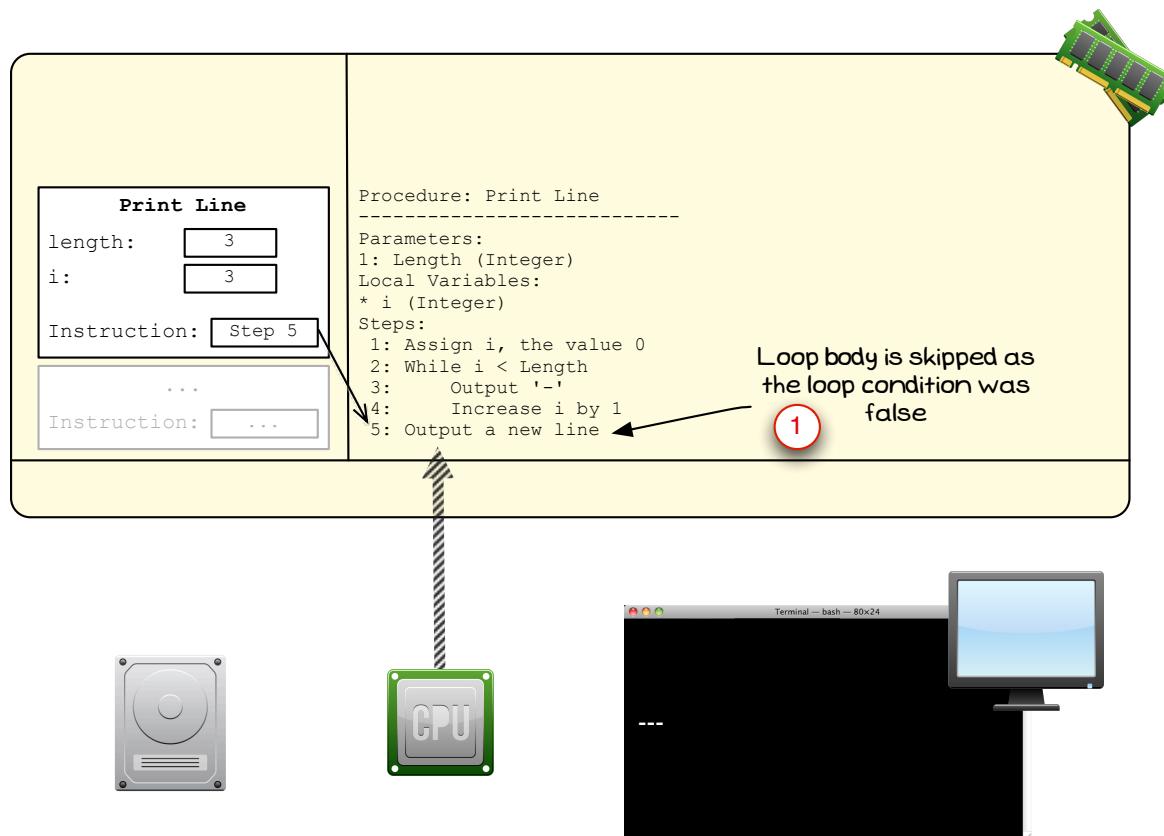


Figure 3.89: Processing continues after the loop

Note

- In Figure 3.89 the indicated areas show the following:
 - The condition indicates that the loop has ended, and so the next instruction is Step 5. This step outputs a new line, ending the line being printed to the Terminal.
- Notice how the condition controlled the behaviour of the loop. Each time it is checked it determines if the body is run again, or if it is skipped.
- The condition is only checked once each loop: once at the start, and then once after each time through the loop body.



3.6 Control Flow Examples

3.6.1 Times Table

This program prints out the times table for a number entered by the user, displaying from 1 x n to 10 x n. The description of the program is in Table 3.15, the pseudocode in Listing 3.26, the C code in Listing 3.27, and the Pascal code in Listing 3.28.

Program Description	
Name	Times Table
Description	Displays the Times Table from 1 x n to 10 x n.

Table 3.15: Description of the Times Table program

Pseudocode

```
-----
Program: Times Table
-----
Variables:
- number (Integer)
- i (Integer)
Steps:
1: Output 'Times Table' to the Terminal
2: Output 'Enter number: ' to the Terminal
3: Read input into number
4: Output '-----' to the Terminal
5: Assign i, the value 1
6: While i is less than 10
7:   Output i ' x ' number ' = ', and i * number to the Terminal
8:   Assign i, the value i + 1
9: Output '-----' to the Terminal
```

Listing 3.26: Pseudocode for Times Table program.



Note

This is an updated version of the Seven Times Table Program. See Section 2.6.1 Times Table.



C++

```
/*
 * Program: times_table.c
 * Displays the Times Table from 1 x n to 10 x n.
 */

#include <stdio.h>

int main()
{
    int number = 0;
    int i;

    printf("Times Table\n");

    printf("Enter number: ");
    scanf("%d", &number);

    printf("-----\n");

    i = 1;
    while(i < 10)
    {
        printf(" %d x %d = %d\n", i, number, i * number);
        i++;
    }
    printf("-----\n");

    return 0;
}
```

Listing 3.27: C Times Table

Pascal

```
//  
// Program: TimesTable.pas  
// Displays the Times Table from 1 x n to 10 x n.  
//  
program TimesTable;  
  
procedure Main();  
var  
    number: Integer = 0;  
    i: Integer = 0;  
begin  
    WriteLn('Times Table');  
  
    Write('Enter number: ');  
    ReadLn(number);  
  
    WriteLn('-----');  
  
    i := 1;  
    while i < 10 do  
    begin  
        WriteLn(' ', i, ' x ', number, ' = ', i * number);  
        i += 1;  
    end;  
  
    WriteLn('-----');  
end;  
  
begin  
    Main();  
end.
```

**Listing 3.28:** Pascal Times Table

3.6.2 Circle Area

This program prints out the area of a circle. The description of the program is in Table 3.16, the pseudocode in Listing 3.29, the C code in Listing 3.30, and the Pascal code in Listing 3.31.

Program Description	
Name	Circle Areas
Description	Displays the Circle Areas for circles with radius from 1.0 to 5.0 with increments of 0.1.

Table 3.16: Description of the Circle Areas program

Pseudocode

```

-----
Program: Circle Areas
-----

Constant:
- PI = 3.1415
- START_RADIUS = 1.0
- END_RADIUS = 5.0
- RADIUS_INC = 0.1

Function: Circle Area
-----
Returns: A Double - the area of a circle with the given radius
Parameters:
- radius (Double)
Steps:
 1: Return the result PI * radius * radius

Variables:
- radius (Double)
Steps:
 1: Output 'Circle Areas' to the Terminal
 2: Output '-----' to the Terminal
 3: Assign to radius the value START_RADIUS
 4: While radius <= END_RADIUS
 5:   Output ' Radius: ', radius, ' = ', CircleArea(radius) to the Terminal
 6:   Assign to radius, the value radius + RADIUS_INC
 7: Output '-----' to the Terminal

```



Listing 3.29: Pseudocode for Circle Areas program.

Note

This is an updated version of the Circle Areas Program. See Section 2.6.2 Circle Area.



C++

```
/*
 * Program: circle_areas.c
 * Displays the Circle Areas for circles with radius
 * from 1.0 to 5.0 with increments of 0.1.
 */

#include <stdio.h>

#define PI 3.1415

#define START_RADIUS    1.0
#define END_RADIUS      5.0
#define RADIUS_INC     0.1

double circle_area(double radius)
{
    return PI * radius * radius;
}

int main()
{
    double radius;

    printf("Circle Areas\n");
    printf("-----\n");

    radius = START_RADIUS;

    while (radius <= END_RADIUS)
    {
        printf(" Radius: %4.2f = %4.2f\n", radius, circle_area(radius));
        radius = radius + RADIUS_INC;
    }

    printf("-----\n");

    return 0;
}
```

Listing 3.30: C Circle Areas

Pascal

```

//  

// Program: circle_areas.c  

// Displays the Circle Areas for circles with radius  

// from 1.0 to 5.0 with increments of 0.1.  

//  

program CircleAreas;  

const PI = 3.1415;  

const START_RADIUS = 1.0;  

const END_RADIUS = 5.0;  

const RADIUS_INC = 0.1;  

function CircleArea(radius: Double): Double;  

begin
    result := PI * radius * radius;
end;  

procedure Main();
var
    radius: Double;
begin
    WriteLn('Circle Areas');
    WriteLn('-----');
    radius := START_RADIUS;  

    while radius <= END_RADIUS do
    begin
        WriteLn(' Radius: ', radius:4:2, ' = ', CircleArea(radius):4:2);
        radius := radius + RADIUS_INC;
    end;
    WriteLn('-----');
end;  

begin
    Main();
end.

```

**Listing 3.31:** Pascal Circle Areas

3.6.3 Moving Rectangle

This example SwinGame code will move a rectangle back and forth across the screen.

Program Description	
Name	<i>Moving Rectangle</i>
Description	Displays a rectangle that is moved back and forth across the screen.

Table 3.17: Description of the Moving Rectangle program

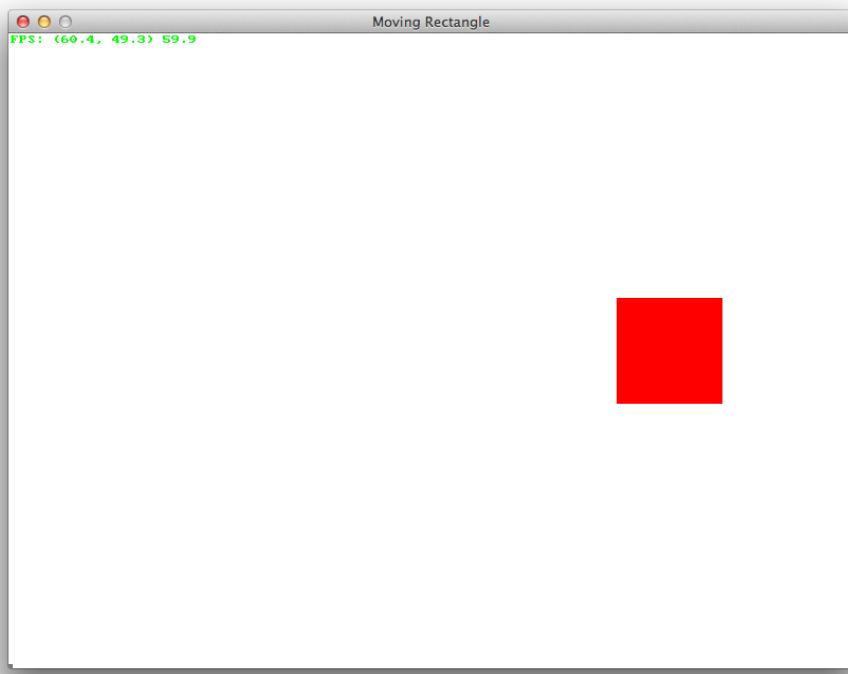


Figure 3.90: Example execution of the Moving Rectangle program

C++

```
#include "SwinGame.h"

#define RECT_WIDTH 100
#define RECT_HEIGHT 100
#define MOVE_X 5

// Update the x position of the rectangle, by the specified amount
void update_rect_position(int &x, int &dx)
{
    // Move x (passed in by reference)
    x += dx;

    // Check if it went off the screen
    if (x < 0)
    {
        // off the left of the screen
        dx = -dx; // change movement direction
        x = 0; // put it back on the screen
    }
    else if ((x + RECT_WIDTH) > screen_width())
    {
        // off the screen to the right
        dx = -dx; // change movement direction
        x = (screen_width() - RECT_WIDTH); // put it back on the screen
    }
}

// Draw a rectangle moving across the screen
int main()
{
    int rect_x = 0;
    int rect_y = 250;
    int rect_x_move = MOVE_X;

    open_graphics_window("Moving Rectangle", 800, 600);
    load_default_colors();

    do
    {
        process_events();

        // Update the location of the rectangle
        update_rect_position(rect_x, rect_x_move);

        // Clear the screen, then draw the rectangle
        clear_screen(ColorWhite);
        fill_rectangle(ColorRed, rect_x, rect_y, RECT_WIDTH, RECT_HEIGHT);
        draw_framerate(0,0);

        // Refresh the screen, keep it at 60fps
        refresh_screen(60);
    } while ( ! window_close_requested() );

    return 0;
}
```

Listing 3.32: C++ Moving Rect SwinGame code

Pascal

```

program MovingRect;
uses sgGraphics, sgUtils, sgInput, sgText;

const
  RECT_WIDTH = 100;
  RECT_HEIGHT = 100;
  MOVE_X = 5;

// Update the x position of the rectangle, by the specified amount
procedure UpdateRectPosition(var x, dx: Integer);
begin
  // Move x (passed in by reference)
  x += dx;

  // Check if it went off the screen
  if x < 0 then
    begin
      // off the left of the screen
      dx := -dx; // change movement direction
      x := 0; // put it back on the screen
    end
  else if (x + RECT_WIDTH) > ScreenWidth() then
    begin
      // off the screen to the right
      dx := -dx; // change movement direction
      x := (ScreenWidth() - RECT_WIDTH); // put it back on the screen
    end;
end;

// Draw a rectangle moving across the screen
procedure Main();
var
  rectX: Integer = 0;
  rectY: Integer = 250;
  rectXMove: Integer = MOVE_X;
begin
  OpenGraphicsWindow('Moving Rectangle', 800, 600);

  repeat
    ProcessEvents();

    // Update the location of the rectangle
    UpdateRectPosition(rectX, rectXMove);

    // Clear the screen, then draw the rectangle
    ClearScreen(ColorWhite);
    FillRectangle(ColorRed, rectX, rectY, RECT_WIDTH, RECT_HEIGHT);
    DrawFramerate(0,0);

    // Refresh the screen, keep it at 60fps
    RefreshScreen(60);
  until WindowCloseRequested();
end;

begin
  Main();
end.

```



Listing 3.33: Pascal Moving Rect SwinGame code

3.6.4 Button Click in SwinGame

This example SwinGame code draws a rectangle that the user can ‘click’.

Program Description	
Name	<i>Button Click</i>
Description	Displays a rectangle the user can ‘click’. Having the mouse held down over the rectangle changes it to a filled rectangle. Clicking the rectangle shows the text ‘Clicked’ in the top left corner.

Table 3.18: Description of the Button Click program

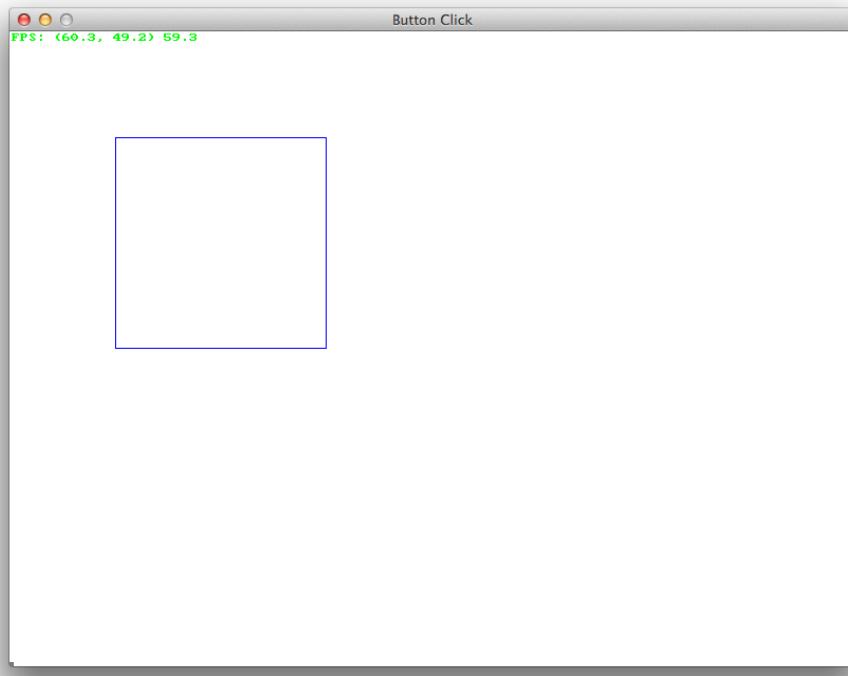


Figure 3.91: Example execution of the Button Click program

C++

```
#include <stdbool.h>
#include "SwinGame.h"

#define BTN_X 100
#define BTN_Y 100
#define BTN_W 200
#define BTN_H 200

bool mouse_over(int x, int y, int width, int height)
{
    float mx, my;

    mx = mouse_x();
    my = mouse_y();

    return mx >= x && mx <= x + width && my >= y && my <= y + height;
}

bool button_clicked(int x, int y, int width, int height)
{
    return mouse_clicked(LEFT_BUTTON) && mouse_over(x, y, width, height);
}

// Draw a rectangle moving across the screen
int main()
{
    open_graphics_window("Moving Rectangle", 800, 600);
    load_default_colors();

    do
    {
        process_events();

        // Clear the screen, then draw the "button"
        clear_screen();

        if (mouse_down(LEFT_BUTTON) && mouse_over(BTN_X, BTN_Y, BTN_W, BTN_H))
            fill_rectangle(ColorBlue, BTN_X, BTN_Y, BTN_W, BTN_H);
        else
            draw_rectangle(ColorBlue, BTN_X, BTN_Y, BTN_W, BTN_H);

        if (button_clicked(BTN_X, BTN_Y, BTN_W, BTN_H))
        {
            draw_text("CLICKED", ColorBlue, 0, 20);
        }

        draw_framerate(0,0);

        // Refresh the screen, keep it at 60fps
        refresh_screen(60);
    } while ( ! window_close_requested() );

    release_all_resources();
    return 0;
}
```

Listing 3.34: C++ Simple Button using SwinGame code

Pascal

```

program ButtonClick;
uses sgGraphics, sgInput, sgText, sgTypes;

const
  BTN_X = 100;      BTN_Y = 100;
  BTN_W = 200;      BTN_H = 200;

function MouseOver(x, y, width, height: Integer): Boolean;
var
  mx, my: Single;
begin
  mx := MouseX();
  my := MouseY();

  result := (mx >= x) and (mx <= x + width) and
            (my >= y) and (my <= y + height);
end;

function ButtonClicked(x, y, width, height: Integer): Boolean;
begin
  result := MouseClicked(LeftButton) and MouseOver(x, y, width, height);
end;

// Draw a rectangle moving across the screen
procedure Main();
begin
  OpenGraphicsWindow('Button Click', 800, 600);

  repeat
    ProcessEvents();

    // Clear the screen, then draw the "button"
    ClearScreen(ColorWhite);

    if MouseDown(LeftButton) and MouseOver(BTN_X, BTN_Y, BTN_W, BTN_H) then
      FillRectangle(ColorBlue, BTN_X, BTN_Y, BTN_W, BTN_H)
    else
      DrawRectangle(ColorBlue, BTN_X, BTN_Y, BTN_W, BTN_H);

    if ButtonClicked(BTN_X, BTN_Y, BTN_W, BTN_H) then
    begin
      DrawText('CLICKED', ColorBlue, 0, 20);
    end;

    DrawFramerate(0,0);

    // Refresh the screen, keep it at 60fps
    RefreshScreen(60);
  until WindowCloseRequested();
end;

begin
  Main();
end.

```



Listing 3.35: Pascal Button Click code

3.7 Control Flow Exercises

3.7.1 Concept Questions

Read over the concepts in this chapter and answer the following questions:

1. What values can a Boolean expression have?
2. What are the values of the following expressions?

Question	Expression	Given
(a)	$1 > 5$	
(b)	$1 < 5$	
(c)	$a > b$	a is 1 and b is 2
(d)	$(a > b) \text{ or } (b > a)$	a is 1 and b is 2
(e)	$(a > b) \text{ and } (b > a)$	a is 2 and b is 2
(f)	$a \text{ or } b \text{ or } c$	a is False, b is False, and c is True
(g)	$(a \text{ or } b) \text{ and } (c \text{ or } d)$	a is False, b is True, c is True, and d is False
(h)	$(a \text{ or } b) \text{ and } (c \text{ or } d)$	a is False, b is True, c is False, and d is False
(i)	$(a \text{ and } b) \text{ or } (c \text{ and } d)$	a is False, b is True, c is True, and d is True
(j)	$a \text{ xor } b$	a is True and b is True
(k)	$(a \text{ or } b) \text{ and } (\text{not } (a \text{ and } b))$	a is True and b is True
(l)	$\text{not } \text{True}$	
(m)	$a \text{ and } (\text{not } b)$	a is True and b is False

C++

The following table shows the C syntax for these boolean expressions.

Question	Expression
(a)	$1 > 5$
(b)	$1 < 5$
(c)	$a > b$
(d)	$(a > b) \text{ } (b > a)$
(e)	$(a > b) \text{ && } (b > a)$
(f)	$a \text{ } b \text{ } c$
(g)	$(a \text{ } b) \text{ && } (c \text{ } d)$
(h)	$(a \text{ } b) \text{ && } (c \text{ } d)$
(i)	$(a \text{ && } b) \text{ } (c \text{ && } d)$
(j)	$a \text{ ^ } b$
(k)	$(a \text{ } b) \text{ && } (\text{!}(a \text{ && } b))$
(l)	!false
(m)	$a \text{ && } (\text{!}b)$

3. What are the two kinds of branching statements?
4. What are the differences between these statements?
5. When would you use each of these kinds of branching statements?
6. What are the two different kinds of looping statements?
7. What are the differences between these statements?
8. When would you use each of these kinds of looping statements?
9. How do Boolean expressions relate to branching and looping statements?
10. What are the four jumping statements?
11. What are the differences between these statements?

12. Why do you need compound statements? Where would these be used?
13. In structured programming, what are the three different kinds of blocks?
14. How many entry/exits are there from these blocks?
15. What are the principles of structured programming?
16. How does this influence the way you design programs?
17. Open a new SwinGame project and examine the startup code. How does this program keep the window open until the user closes it?

3.7.2 Code Reading Questions

Use what you have learnt to read and understand the following code samples, and answer the associated questions.

1. Read the C code in Listing 3.36, or the Pascal code in Listing 3.37, and then answer the following questions:
 - (a) What are X_SPEED, width, height, x, and y? How are they similar/different? What purpose does each of these play? What values are they assigned?
 - (b) What does the loop in this code do?
 - (c) Briefly explain what this code does, and suggest a name for the ???? procedure.
 - (d) Provide an example of how you could call this procedure.

C++

```
#define X_SPEED 5

void ????(int width, int height)
{
    float x = 0, y;

    y = (screen_height() - height) / 2;

    while ( (x + width) < screen_width() )
    {
        clear_screen();

        fill_rectangle(ColorRed, x, y, width, height);
        x = x + X_SPEED;

        refresh_screen();
    }
}
```



Listing 3.36: What does this C code do?

2. Read the C code in Listing 3.38, or the Pascal code in Listing 3.39, and then answer the following questions:
 - (a) What will be drawn to the screen when the user is not holding down any key?
 - (b) When the user is holding down the G key what will be drawn?
 - (c) When the user is holding down the R and B keys what will be drawn?
 - (d) What condition causes the loop to end? How can a user end this loop?
 - (e) Draw a flow chart to illustrate the possible paths through this code.

Pascal

```

const X_SPEED = 5;

procedure ????(width, height: Integer);
var
  y: Single;
  x: Single = 0;
begin
  y := (ScreenHeight() - height) / 2;

  while (x + width) < ScreenWidth() do
  begin
    ClearScreen();

    FillRectangle(ColorRed, x, y, width, height);
    x := x + X_SPEED;

    RefreshScreen();
  end;
end;

```



Listing 3.37: What does this Pascal code do?

C++

```

void draw_colored_rects()
{
  do
  {
    process_events();
    clear_screen();

    if (key_down(VK_R)) fill_rectangle(ColorRed, 10, 10, 780, 580);
    if (key_down(VK_G)) fill_rectangle(ColorGreen, 20, 20, 760, 560);
    if (key_down(VK_B)) fill_rectangle(ColorBlue, 30, 30, 740, 540);

    refresh_screen();
  } while ( ! key_typed(VK_ESCAPE) );
}

```



Listing 3.38: What does this C code do?

Pascal

```
procedure DrawColoredRects();
begin
  repeat
    ProcessEvents();
    ClearScreen();

    if KeyDown(vk_R) then FillRectangle(ColorRed, 10, 10, 780, 580);
    if KeyDown(vk_G) then FillRectangle(ColorGreen, 20, 20, 760, 560);
    if KeyDown(vk_B) then FillRectangle(ColorBlue, 30, 30, 740, 540);

    RefreshScreen();
  until KeyTyped(vk_ESCAPE);
end;
```



Listing 3.39: What does this Pascal code do?

3. Read the C code in Listing 3.40, or the Pascal code in Listing 3.41, and then answer the following questions:

- (a) What do the local variables and parameter keep track of?
- (b) How many times will the loop execute?
- (c) What is the purpose of the if statement within the loop?
- (d) What will be drawn to the screen when this procedure is called?

C++

```
void draw_bar_pattern(int num_bars)
{
    int x, i, bar_width;

    i = 0;
    bar_width = screen_width() / num_bars;

    while( i < num_bars )
    {
        x = i * bar_width;
        if ( (i % 2) == 0 )
            fill_rectangle(ColorWhite, x, 0, bar_width, screen_height());
        else
            fill_rectangle(ColorBlack, x, 0, bar_width, screen_height());

        i++;
    }
    refresh_screen();
}
```

Listing 3.40: What does this C code do?



Pascal

```
procedure DrawBarPattern(numBars: Integer);
var
    x, i, barWidth: Integer;
begin
    i := 0;
    barWidth := ScreenWidth() / numBars;

    while i < numBars do
    begin
        x := i * barWidth;
        if (i mod 2) = 0 then
            FillRectangle(ColorWhite, x, 0, barWidth, ScreenHeight())
        else
            FillRectangle(ColorBlack, x, 0, barWidth, ScreenHeight());

        i += 1;
    end;

    RefreshScreen();
end
```



Listing 3.41: What does this Pascal code do?

4. Read the C code in Listing 3.42, or the Pascal code in Listing 3.43, and then answer the following questions:
- What will be drawn to the screen when the user is not holding down any key?
 - When the user is holding down the G key what will be drawn?
 - When the user is holding down the R and B keys what will be drawn?
 - What condition causes the loop to end? How can a user end this loop?
 - Draw a flow chart to illustrate the possible paths through this code.

C++

```
void draw_colored_rects_v2()
{
    do
    {
        process_events();
        clear_screen();

        if (key_down(VK_R)) fill_rectangle(ColorRed, 10, 10, 780, 580);
        else if (key_down(VK_G)) fill_rectangle(ColorGreen, 20, 20, 760, 560);
        else if (key_down(VK_B)) fill_rectangle(ColorBlue, 30, 30, 740, 540);
        else fill_rectangle(ColorGrey, 40, 40, 720, 520);

        refresh_screen();
    } while ( ! ( key_typed(VK_ESCAPE) || window_close_requested() ) );
}
```

Listing 3.42: What does this C code do?

**Pascal**

```
procedure DrawColoredRectsV2();
begin
    repeat
        ProcessEvents();
        ClearScreen();

        if KeyDown(vk_R) then FillRectangle(ColorRed, 10, 10, 780, 580)
        else if KeyDown(vk_G) then FillRectangle(ColorGreen, 20, 20, 760, 560)
        else if KeyDown(vk_B) then FillRectangle(ColorBlue, 30, 30, 740, 540)
        else FillRectangle(ColorGrey, 40, 40, 720, 520);

        RefreshScreen();
    until KeyTyped(vk_ESCAPE) or WindowCloseRequested();
end;
```

Listing 3.43: What does this Pascal code do?

5. Read the C code in Listing 3.44, or the Pascal code in Listing 3.45, and then answer the following questions:

- What value is stored in the `dot_x` variable each time through the loop?
- Explain what will be drawn on the screen when this code is executed. How does this code respond to user actions?
- Do the circles appear on top of, or below the frame rate? Explain.
- How frequently are the actions in the loop performed?
- How would putting a call to `delay` inside this loop effect the running of this program?

C++

```
#define LARGE_RADIUS 6
#define SMALL_RADIUS 3

void draw_cursor()
{
    float dot_x, dot_y;
    hide_mouse();

    while( ! window_close_requested() )
    {
        process_events();

        dot_x = mouse_x();
        dot_y = mouse_y();

        clear_screen(ColorWhite);
        draw_framerate(0,0);
        draw_circle(ColorBlack, dot_x, dot_y, LARGE_RADIUS);

        if ( mouse_down(LEFT_BUTTON) )
            fill_circle(ColorBlack, dot_x, dot_y, SMALL_RADIUS);
        else
            draw_circle(ColorBlack, dot_x, dot_y, SMALL_RADIUS);

        refresh_screen();
    }

    show_mouse();
}
```

Listing 3.44: What does this C code do?

Pascal

```
const LARGE_RADIUS = 6;
const SMALL_RADIUS = 3;

procedure DrawCursor();
var
    dotX, dotY: Single;
begin
    HideMouse();

    while not WindowCloseRequested() do
    begin
        ProcessEvents();

        dotX := MouseX();
        dotY := MouseY();

        ClearScreen(ColorWhite);
        DrawFramerate(0, 0);
        DrawCircle(ColorBlack, dotX, dotY, LARGE_RADIUS);

        if MouseDown(LEFT_BUTTON) then
            FillCircle(ColorBlack, dotX, dotY, SMALL_RADIUS)
        else
            DrawCircle(ColorBlack, dotX, dotY, SMALL_RADIUS);

        RefreshScreen();
    end;

    ShowMouse();
end;
```

Listing 3.45: What does this Pascal code do?



6. Read the C code in Listing 3.46, or the Pascal code in Listing 3.47, and then answer the following questions:
- What is the purpose of the if statement in this code? What is it testing?
 - When will the color of the rectangle change? What are the conditions that must be met?
 - If the user clicks the left button on their mouse will the color of the rectangle change? discuss
 - What would need to be added to keep a score of the number of times the user changed the color? explain

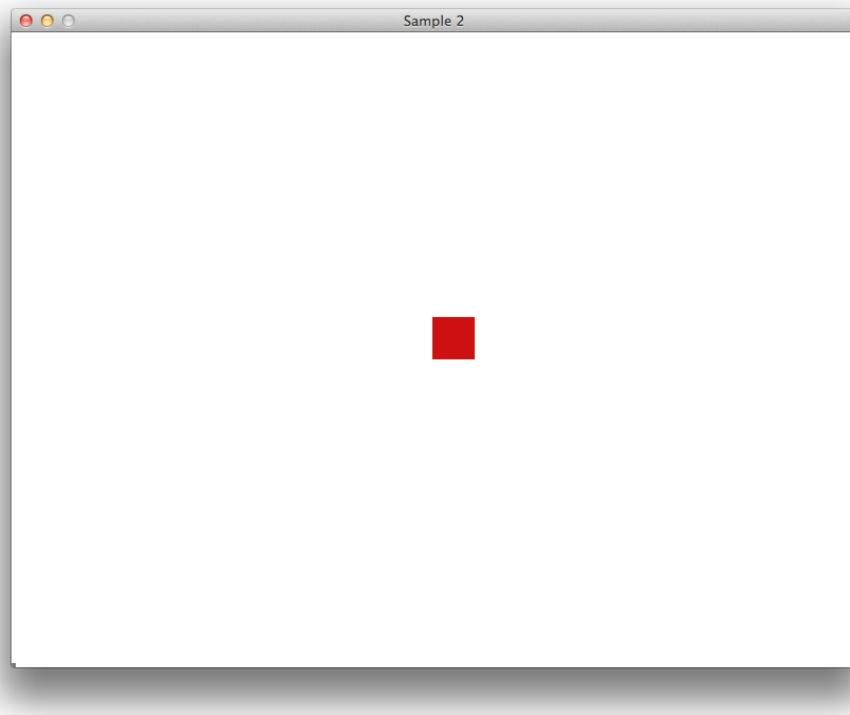


Figure 3.92: The game in play

C++

```
void play_game()
{
    int x = 380, y = 280;
    color clr = random_rgbcOLOR(255); // Opaque color

    do
    {
        process_events();
        if (mouse_clicked(LEFT_BUTTON) &&
            point_in_rect(mouse_x(), mouse_y(), x, y, 40, 40))
        {
            clr = random_rgbcOLOR(255);
        }

        clear_screen(ColorWhite);
        fill_rectangle(clr, x, y, 40, 40);
        refresh_screen();

        x = x + (rnd() * 6.0) - 3;
        y = y + (rnd() * 6.0) - 3;
    } while ( ! (key_typed(VK_ESCAPE) || window_close_requested()) );
}
```

Listing 3.46: What does this C code do?

**Pascal**

```
procedure PlayGame();
var
    x: Single = 380.0;
    y: Single = 280.0;
    clr: Color;
begin
    clr := RandomRGBColor(255); // Opaque color

    repeat
        process_events();
        if MouseClicked(LEFT_BUTTON) and
            PointInRect(MouseX(), MouseY(), x, y, 40, 40) then
        begin
            clr = RandomRGBColor(255);
        end;

        ClearScreen(ColorWhite);
        FillRectangle(clr, x, y, 40, 40);
        RefreshScreen();

        x = x + (rnd() * 6.0) - 3;
        y = y + (rnd() * 6.0) - 3;
    until KeyTyped(vk_ESCAPE) or WindowCloseRequested();
end;
```



Listing 3.47: What does this Pascal code do?

7. Read the C code in Listing 3.48, or the Pascal code in Listing 3.49, and then answer the following questions:

- What would be better names for the `i` and `j` variables?
- How many rectangles are drawn to the screen when this procedure is called?
- What does this code draw to the screen when it is called?
- How would you need to change the code to ensure that the rectangles were always square?

C++

```
void draw_grid_pattern(int num_rows, int num_cols)
{
    int x, y, i, j, bar_width, bar_height;

    i = 0;
    bar_width = screen_width() / num_cols;
    bar_height = screen_height() / num_rows;

    while (i < num_rows)
    {
        y = i * bar_height;
        j = 0;

        while (j < num_cols)
        {
            x = j * bar_width;
            if ((i + j) % 2 == 0)
                fill_rectangle(ColorWhite, x, y, bar_width, bar_height);
            else
                fill_rectangle(ColorBlack, x, y, bar_width, bar_height);

            j++;
        }
        i++;
    }
    refresh_screen();
}
```

Listing 3.48: What does this C code do?

Pascal

```
procedure DrawGridPattern(numRows, numCols: Integer);
var
  x, y, i, j, barWidth, barHeight: Integer;
begin
  i = 0;
  barWidth = screen_width() / numCols;
  barHeight = screen_height() / numRows;

  while i < numRows do
  begin
    y := i * barHeight;
    j := 0;

    while j < numCols do
    begin
      x := j * barWidth;
      if ((i + j) mod 2) = 0 then
        fill_rectangle(ColorWhite, x, y, barWidth, barHeight)
      else
        fill_rectangle(ColorBlack, x, y, barWidth, barHeight);

      j += 1;
    end;
    i += 1;
  end;
  RefreshScreen();
end;
```

Listing 3.49: What does this Pascal code do?



3.7.3 Code Writing Questions: Applying what you have learnt

Apply what you have learnt to the following tasks:

1. Write a small program that will print the message 'Hello World' to the Terminal '1 MILLION TIMES'... mmmmmwwwahahahah³.
2. Revisit your Circle Dimensions program from Chapter 2 and adjust its implementation to make use of looping statements. The new output should display the radius, circle area, diameter, and circumference of circles with radius from 1.0cm to 1.0m at 0.25cm increments.
3. Take your Times Table program from Chapter 2, and change it so that you can print a variable number of times. For example, have it print the 5 times table from 1 x 5 to 10 x 5, the 73 times table from 1 x 73 to 73 * 73, and the 42 times table from 1 x 42 to 7 * 42.
4. Implement the Guess that Number program, and test that your implementation works successfully.
5. Take the adjusted Face Shape program from Chapter 2, and re-implement it so that it draws a new randomly places face each time through the loop in Main. SwinGame include a rnd function that can be used to generate a random number between 0 and 1, you can then multiply this by the Screen Width or Screen Height to get a random position.
6. Try implementing some of the code reading questions. See if you can combine a few of them together to create a small game.
7. Write a function called Read Integer that will read a number from the user. If the user does not enter a number the function will output the message 'Please enter a number.', and will repeatedly ask the user to enter a number until they do so. Write a small program that tests if your function works correctly.

C++

You can use `scanf(" %d", &result) != 1` to read in the value entered into a `result` variable, and to check if it was a number^a, and you can use `scanf("%*[^\n]");` to flush the input when you get something other than a number.

^aThe `scanf` function returns the number of values successfully read. See [C Terminal Input](#).

8. Use your Read Integer function to read two values from the user, and then output the larger of the two values.
9. Use your Read Integer function to read in a number, and then print out a custom message for different values. For example, the value 42 could have the message 'The meaning of life...', the value 73 could have the message 'The best number, according to Dr. Sheldon Cooper', etc. Have at least five custom messages, and a default message when the user does not enter one of the selected values.
10. Update your *custom message* program, and have it ask the user if they want to quit at the end, allowing them to print multiple message values.

³Dr Evil, Austin Powers, 1997

3.7.4 Extension Questions

If you want to further your knowledge in this area you can try to answer the following questions. The answers to these questions will require you to think harder, and possibly look at other sources of information.

1. Implement a Number Guesser program that will guess numbers between 1 and 100. See the program description in Table 3.19

Program Description	
Name	Number Guesser
Description	<p>This program will work to <i>guess</i> a number that the player is thinking of between 1 and 100. This will prompt the user to think of a number, and it will then guess using the following algorithm. The first guess will be 50 (the point half way between 1 and 100). The user will then enter a character to tell the computer how that value relates to the target value: 'E' for equal, 'L' for your guess is less than the target, and 'G' for your guess is greater than the target.</p> <p>When the guess is equal, a success message is output.</p> <p>When the guess is less than the target the computer will take another guess that is half way between the guess and the previous upper value (e.g. 1-100 = 50, 50-100 = 75, 75-100 = 87, 25-50 = 37, etc).</p> <p>When the guess is larger than the target the computer will take another guess that is half way between the guess and the previous lower value (e.g. 1-100 = 50, 1-50 = 25, 1-25 = 12, 50-75 = 62, etc).</p> <p>Once the number is guessed (or the computer determines the player is cheating...) the program asks if the user wants to play again, and the process is repeated or the program quits.</p>

Table 3.19: Description of the Number Guesser program.

2. Create a SwinGame project called *Key Test*. See the description in Table 3.20.

Program Description	
Name	Key Test
Description	<p>This SwinGame will draw a filled red rectangle in the center of the screen when the user holds down the r key.</p>

Table 3.20: Description of the Key Test program.

3. Extend the *Key Test* program to draw other shapes when different keys are held down. For example, draw a filled circle when the user holds down the **c** key.

C++

You can call `key_down(VK_R)` to test if the 'r' key is held down, then only draw the rectangle *if* this key is held down.

4. Color can be represented as consisting of three components: hue, saturation, and brightness. The hue component represents the color, the saturation is how much of this color is added to the white point, and the white point is controlled by the brightness and moves from black through grey to white. An illustration of this is shown in Figure 3.93.

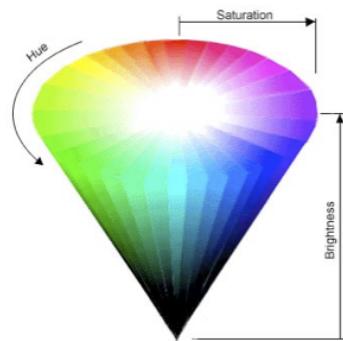


Figure 3.93: Color wheel showing HSB color

SwinGame includes a Draw Pixel procedure that allows you to draw a single pixel on the screen. We can use this to draw a color banding across the screen by calculating the color for each pixel using SwinGame's HSB Color function.

The structure of this program can be broken into three blocks: a Color At function, a My Clear Screen procedure and a Main procedure. This structure is shown in Figure 3.94.

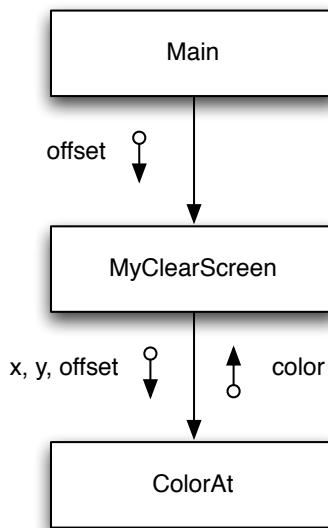


Figure 3.94: Structure Chart for My Clear Screen

The `Color At` function will be used to calculate the color for each pixel on the screen. This will be passed the `x` and `y` coordinates of the pixel (Integers) and an `offset`⁴ (a Double value). This function will call SwinGame's HSB Color function, passing in a hue calculated using the equation shown below, with saturation and brightness set to 0.9 and 0.8 respectively, and it will return the resulting color.

$$hue = \frac{(Offset + \frac{x + y}{ScreenWidth + ScreenHeight})}{2}$$

3.1

The equation will cycle the hue value between 0 and 1, covering the full spectrum of color. The `x + y` component will alter the color of the pixel based on its position on the screen, while the offset allows the program to change the color of the first pixel as time passes.

The `MyClearScreen` procedure will loop through all of the pixels across the screen (with `x` going from 0 up to (but not including) `ScreenWidth()`). This loop will select a single column of pixels on the screen, the pixels at the given `x` position. As this column has `ScreenHeight()` pixels you need an inner loop that loops from 0 to `ScreenHeight()`. Inside the inner loop the `x,y` values give you the coordinates of an individual pixel. You can now use the `Draw Pixel` procedure to draw this pixel, using the `Color At` function you wrote to determine its color.

Lastly the `Main` procedure will contain the standard game loop provided in the SwinGame project template. Replace the code within the loop so that it calls `My Clear Screen` then `Refresh Screen` and lastly `Process Events`. Within `Main` you need to maintain an offset value that you increment by a fixed value each loop⁵, always ensuring the value is between 0 and 1. This offset is then passed to `My Clear Screen` when it is called.

See if you can use this information, and program design to create a small screensaver program. Experiment with different values and loops to see what effects you can create.

⁴Offset must have a value between 0 and 1 as the value for Hue must be between 0 and 1.

⁵Trial this with different value ; 1

4

Managing Multiple Values

True magic has the power to affect not just one, but many targets. The secrets you will learn today will make it possible for your magic to be applied to many targets. The key is to...

Previous chapters have introduced a number programming artefacts for you to create within your code. However, when it comes to working with data in your programs you have been limited in the way you deal with multiple values. This chapter introduces the concepts and practices that make it easier to work with multiple values in your code.

When you have understood the material in this chapter you will be able to work with multiple values more easily, allowing your programs to work with many data values.

Contents

4.1 Concepts Related to Managing Multiple Values	339
4.1.1 Array	340
4.1.2 Assignment Statement (with Arrays)	342
4.1.3 Expressions (with Arrays)	344
4.1.4 Pass by Reference	345
4.1.5 For Loop	346
4.1.6 String	347
4.1.7 Summary	348
4.2 Using these Concepts	349
4.2.1 Designing Statistics Calculator	349
4.2.2 Understanding the Statistics Calculator	349
4.2.3 Choosing Artefacts for Statistics Calculator	355
4.2.4 Writing the Code for Statistics Calculator	360
4.2.5 Compiling and Running Statistics Calculator	361
4.3 Managing Multiple Values in C	362
4.3.1 Implementing Statistics Calculator in C	362
4.3.2 C Array Declaration	364
4.3.3 C Array Copying	365
4.3.4 C For Loop	366
4.3.5 C String	367
4.3.6 C Function (with Array Parameters)	371
4.4 Managing Multiple Values in Pascal	373
4.4.1 Implementing Statistics Calculator in Pascal	373
4.4.2 Pascal Array Declaration	375
4.4.3 Pascal For Loop	376
4.4.4 Pascal Array Functions	377

4.4.5 Pascal Array Parameters	378
4.4.6 Pascal String	380
4.5 Understanding Arrays	381
4.5.1 Understanding Populate Array	381
4.5.2 Understanding Sum	395
4.6 Array Examples	407
4.6.1 Bubble Game (start)	407
4.7 Array Exercises	412
4.7.1 Concept Questions	412
4.7.2 Code Reading Questions	413
4.7.3 Code Writing Questions: Applying what you have learnt	417
4.7.4 Extension Questions	420

4.1 Concepts Related to Managing Multiple Values

Data is an important part of any program. Earlier chapters have shown how to store and work with data. These chapters covered both the [Types](#) of data you can work with, and means of storing and exchanging data using [Variables](#). So far each Variable has stored only a single value, making it difficult to work with multiple values. This chapter introduces the concepts needed start working more effectively with multiple values.

The chapter introduces the following **artefacts**:

- [Array](#): A kind of variable that is used to store multiple values. The individual values in the array are called *elements*.
- [String](#): An existing [Type](#) that stores textual data.

The following **actions** are then needed to work with Arrays:

- [For Loop](#): A [Pre-Test Loop](#) that can be used to easily repeat a block of code for each element of an array.
- [Expressions \(with Arrays\)](#): Expressions can read the value from the element of an array.
- [Assignment Statement \(with Arrays\)](#): The assignment statement can be used to assign a value to an element in an array.

You may need to revise the following programming artefacts:

- [Variable](#): The idea of storing data within your code.
- [Local Variable](#): Storing data in a [Function](#) or [Procedure](#).
- [Parameter](#): Passing data to a Function or Procedure.

The following programming terminology will also be used in this Chapter:

- [Expression](#): A value used in a statement.
- [Type](#): A kind of data used in your code.

The example for this chapter is a statistics calculator, where the user enters 10 values, and the program calculates some statistics. An example of this program executing is shown in Figure 4.1.

```
acain-mbp:array acain$ ./StatsCalc
Enter value 1: Hello World
Please enter a number.
Enter value 1: 1
Enter value 2: 2
Enter value 3: 3
Enter value 4: 4
Enter value 5: 5
Enter value 6: 6.0
Enter value 7: 7.0
Enter value 8: 8
Enter value 9: 9
Enter value 10: 10

Calculating statistics...

Sum:      55.00
Mean:     5.50
Variance: 9.17
Max:      10.00
acain-mbp:array acain$
```

Figure 4.1: Statistics Calculator run from the Terminal

4.1.1 Array

An array is a special kind of **Variable**, one that stores multiple values instead of a single value. The values within an array, called elements, are all of the same **Type**.

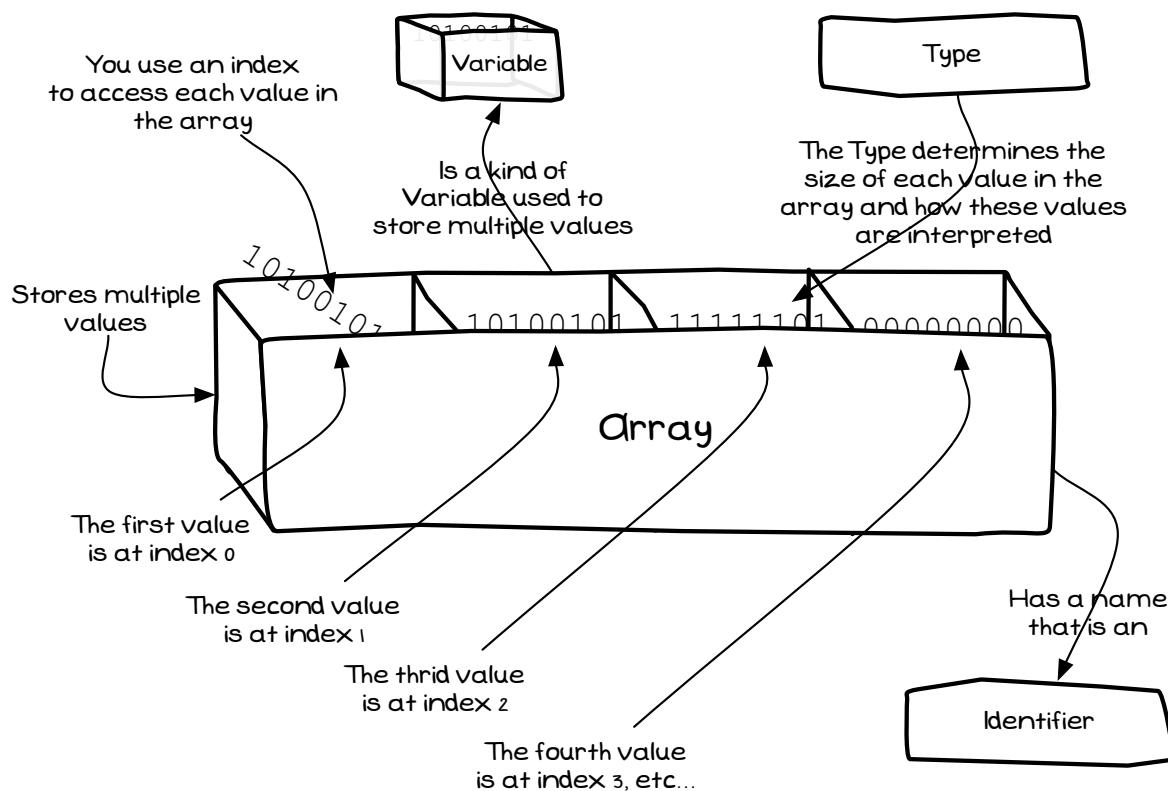


Figure 4.2: Arrays allow you to store multiple values in a variable

Note

- An array is an **artefact**, a kind of variable that you can create in your code.
- Each array has a number of elements.
- The elements of an array are accessed via an index. The first element has the index **0**, the second has the index **1**, etc. See [Arrays in Memory](#).
- Individual elements of an array are much like a standard variable.
- You can store a value in an element of an array using an [Assignment Statement \(with Arrays\)](#).
- You can use a the value from an element of an array using [Expressions \(with Arrays\)](#).
- The [For Loop](#) can be used to perform an action for each element of an array.

Arrays in Memory

Arrays store multiple values, with an index that provides access to the individual elements within the array. Conceptually this can be viewed as a [Variable](#) that contains multiple slots (the elements) into which the values are stored.

Many languages have **0** as the index of the first element. This reflects how the values are actually stored in memory. Figure 4.3 shows how an array (named arr in this Figure) is stored in memory. The array is a **contiguous** area in memory, with the elements being next to each other. You can think of the array as starting at the first element, so you need to skip **0** elements to access the first element. The second element is accessed by skipping **1** element, so it has index **1**. The third element is accessed by skipping **2** elements, so it has the index **2**, etc.

The size of each of the elements of the array can then be used to quickly locate each element, given its index. If you have an array of Integer values then these are each 32 bits (4 bytes), so the element at index 3 is $3 \times 32\text{bits}$ (96 bits) past the start of the array.

The great thing is that you do not need to think about these details, but knowing this should you remember that the first¹ element of an array is at index **0**.

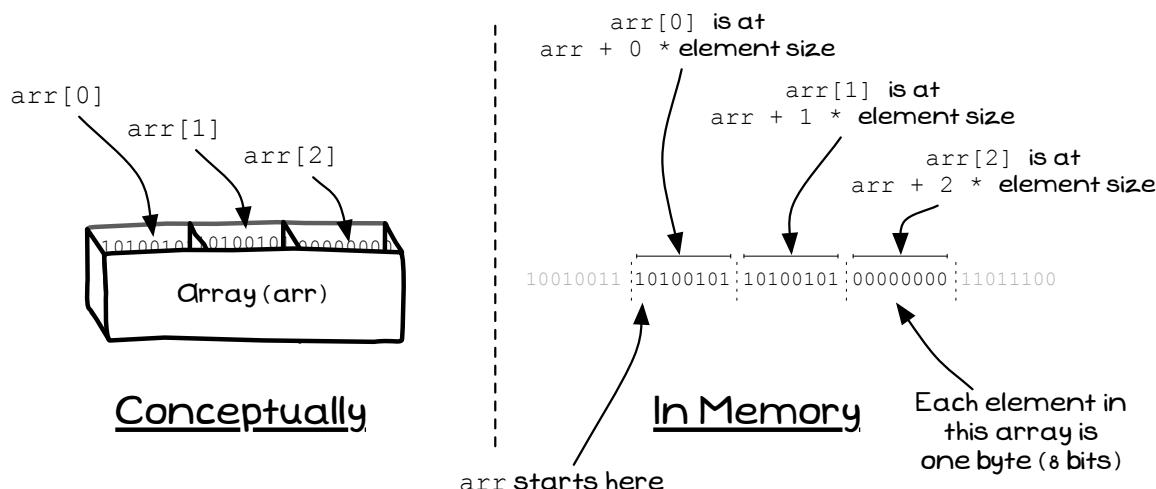


Figure 4.3: Arrays occupy a contiguous area of memory, with elements next to each other in memory

Note

- Arrays are allocated a contiguous area of memory, with the elements next to each other.
- The array index is used to determine how many elements (values) are skipped to access the element you require.
- The index of the first element of an array is **0**.
- The last index of the array is therefore **n - 1**, where **n** is the number of elements in the array.



¹Some languages allow you to start at other values, but many use 0 based arrays which map more closely to how the data is stored in memory.

4.1.2 Assignment Statement (with Arrays)

The assignment statement allows you to store a value in a variable. This can now be extended to allow you to store a value in an element of an array. To achieve this you indicate the array you want to store the value in, as well as the index at which the value is to be stored.

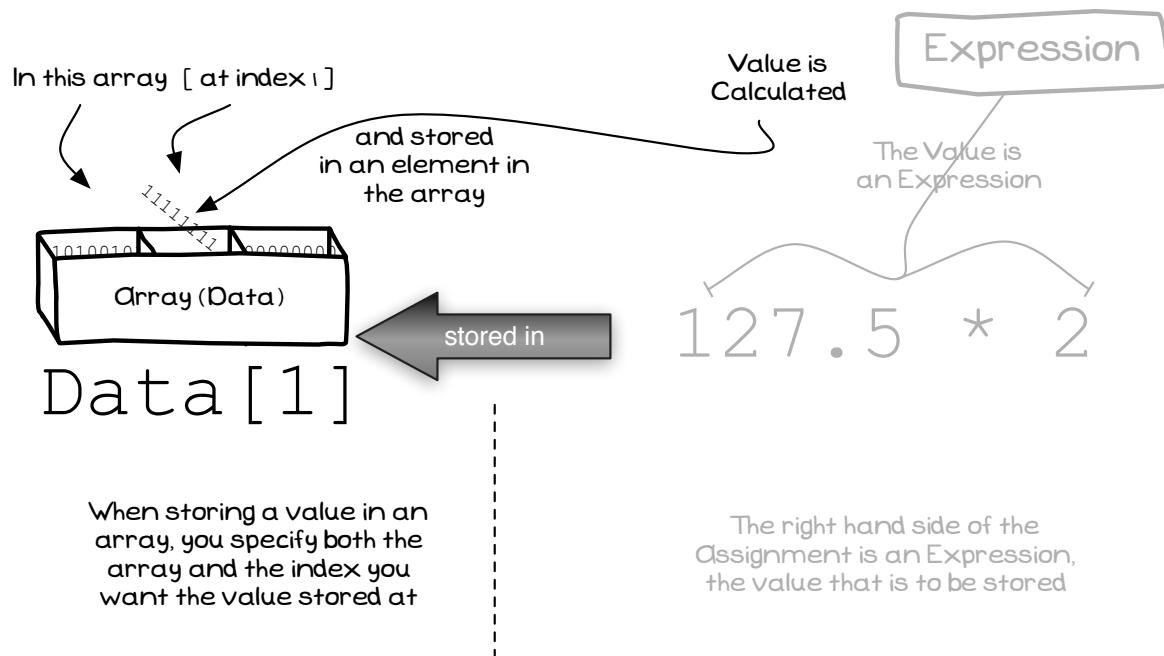


Figure 4.4: Arrays allow you to store multiple values in a variable

Note

- The assignment statement is an **action**, storing a value in a variable or array.
- When working with arrays you specify the array, and index at which to store the value.
- This stores a value into an element of an array.
- The two snippets below indicate how this is coded in C and Pascal. In each case data is the name of the array variable, and 1 is the index of the second element. The code in the snippet stores a 255 in the second element of the array.

C++

In C you can achieve this using: `data[1] = 127.5 * 2; .`

Pascal

In Pascal you can achieve this using: `data[1] := 127.5 * 2; .`

Assigning all elements of an array

Many languages also allow you to copy the entire contents of an array into another array. In these cases each of the elements of one array are copied into the elements of the destination array (left hand side of the assignment).

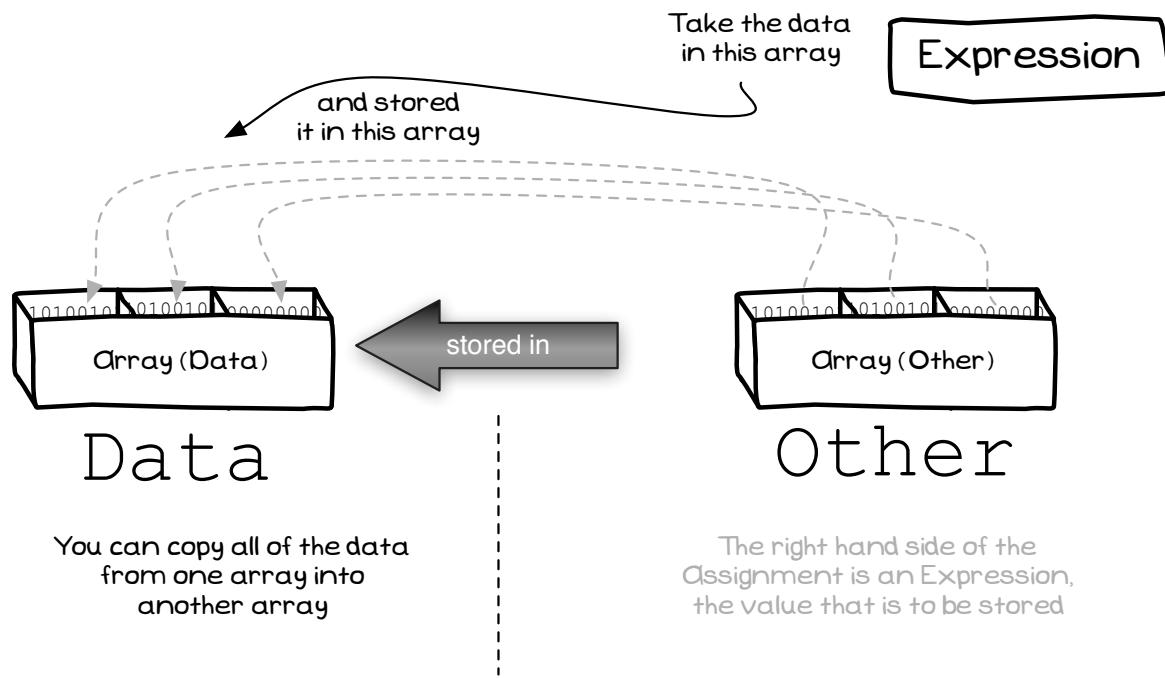


Figure 4.5: All of the elements of an array can be copied across in the assignment statement

Note

- The size of the two arrays should match.
- The value from each of the elements of the array on the right hand side will be copied into the matching element in the array on the left hand side. ♠

C++

This **cannot** be done in C with an assignment statement, rather it is achieved using the `memcpy` function. The code for this is as follows, with `sz` being the number of elements to copy. `memcpy(data, source, size_of(double) * sz)`

Pascal

In Pascal you can achieve this using: `data := other;`

4.1.3 Expressions (with Arrays)

Expressions allow you to read values from the elements of an array. To get an element's value you must supply the name of the array, and the index of the element you want to read.

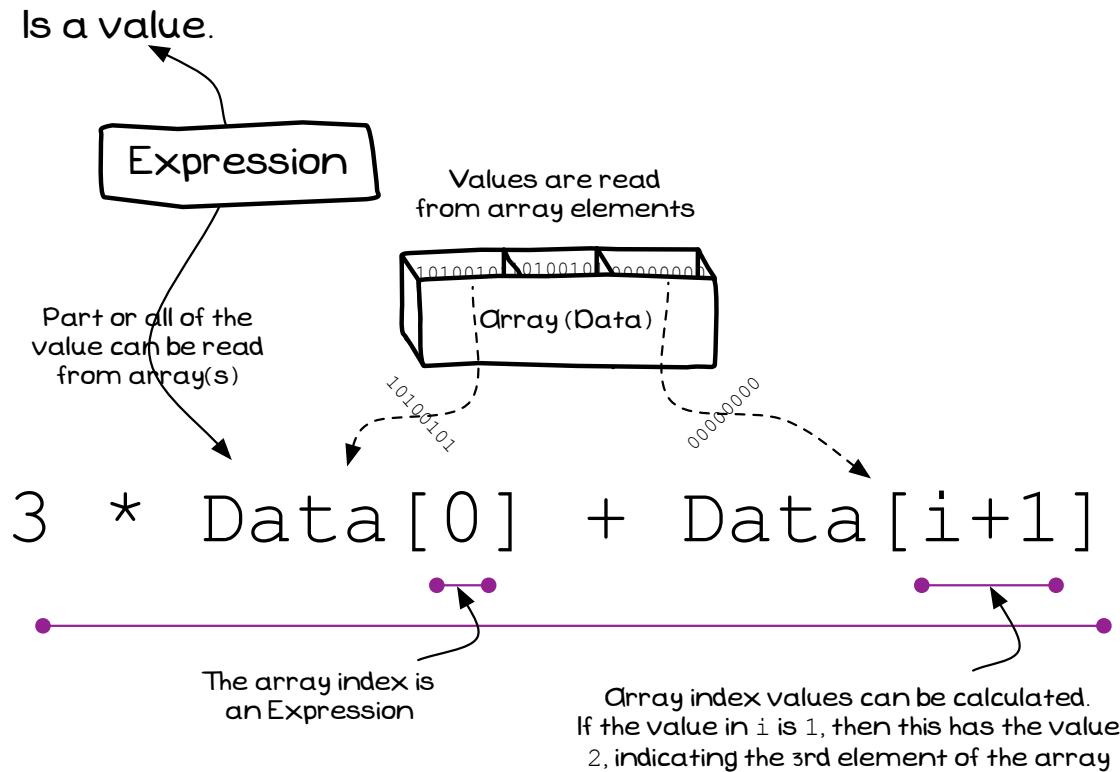


Figure 4.6: You can read the values back from an array

Note

- Expression is the **term** given to the code that calculates values within your statements.
- You can read the values of elements of an array in an expression.
- The index values used to access the individual elements of an array are expressions themselves.
- Arrays are similar to variables in expressions, the expression reads the value from the element of the array.
- The index you supply determine which value is read.

4.1.4 Pass by Reference

As with other Variables, you can pass an array to Functions and Procedures. The only real difference is that the array can potentially store significantly more data than other variables. When the array is passed by value each of its elements must be passed to the Parameter. Passing the parameter in this way means that there will be two copies of the data in memory, which takes more time and more memory.

You should avoid passing arrays by value, and instead pass them by reference. When passed by reference the array itself is passed across. This gives the called Function or Procedure access to the data, but does not require that the values be copied across.

One issue with always using pass by reference is that it allows the called code to change the data in array you are passing across. This can be useful, but at other times you want to pass the data across without allowing it to be changed by the called code. Both C and Pascal allow you to indicate that the data you are passing should be passed by reference, but that it cannot be changed in the called code.

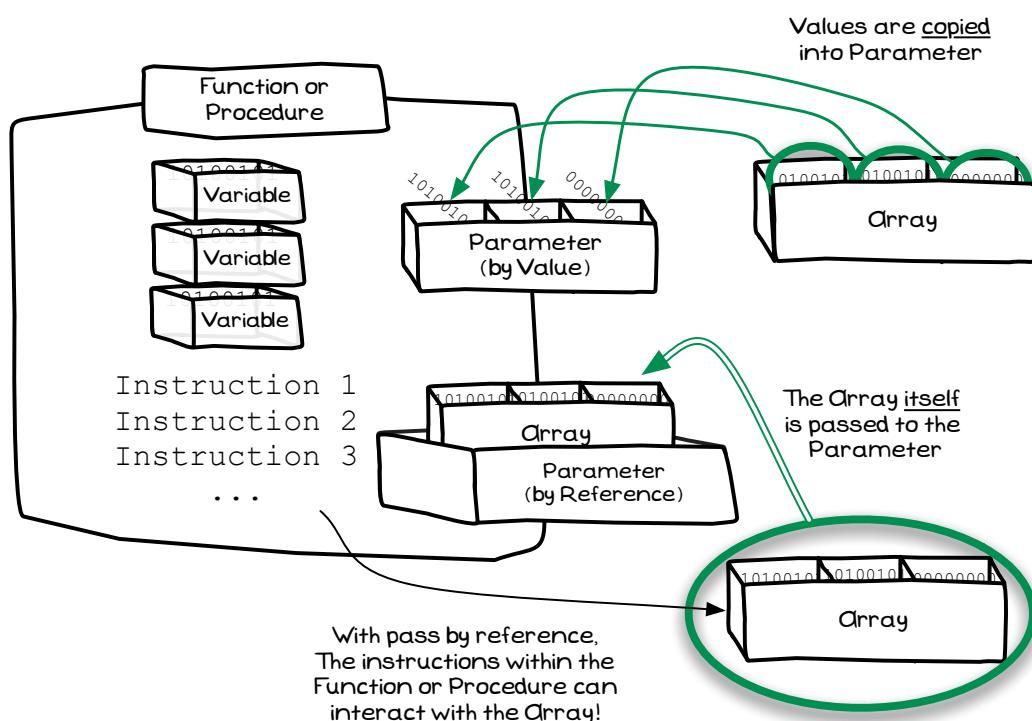


Figure 4.7: Arrays should always be passed by Reference

Note

- Pass by Reference and Pass by Value are **terms** that explain how data is passed to a Parameter.
- With arrays you should always use *Pass by Reference*. This will be faster and take less memory.
- The **const** keyword can be used to indicate that the parameter should not be able to be changed by the called code.

C++

In C you cannot pass an array by value, instead all arrays are passed by reference automatically by the language.

4.1.5 For Loop

As has been shown in previous chapters, Computers can only perform simple actions. They cannot perform an action on all of the elements in our arrays. For example, a computer cannot sum **all** of the values in an array. What you need to do is think of these tasks so that they can be performed **for each** value in the array. So the sum would become, for each of the numbers in the array, add the number to a running total. When this has been performed for each element in the array you have the sum of all of the elements.

The for loop is a **Pre-Test Loop** that repeats a block of code a number of times. You can think of it like a counting loop, with it counting from a start value to an end value. The for loop has a **control variable** that has the number of the current loop as its value.

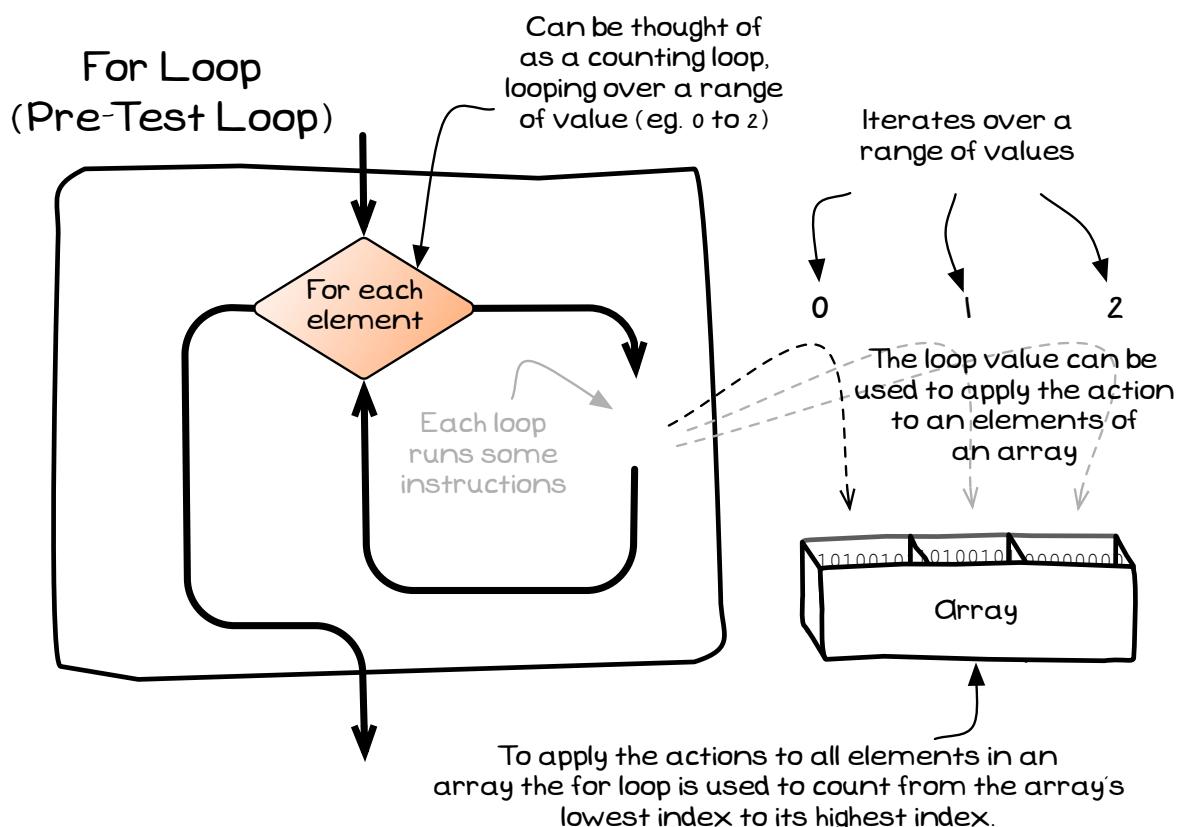


Figure 4.8: The for loop can be used to loop through the elements of an array

Note

- A for loop is an **action**, a kind of statement you can use to command the computer to perform an action.
- The key is to think about processing **each** element in an array, rather than thinking about *all* elements of an array.
- The for loop can then provide the infrastructure to repeat this code *for each* element in the array.
- The for loop is designed to work well with the array. The values in the *control variable* can be used to access the individual elements of the array.
- When processing the elements of an array you have it loop from the lowest index value (0) to the highest index value ($n - 1$).

4.1.6 String

Textual data is stored as an array of characters.

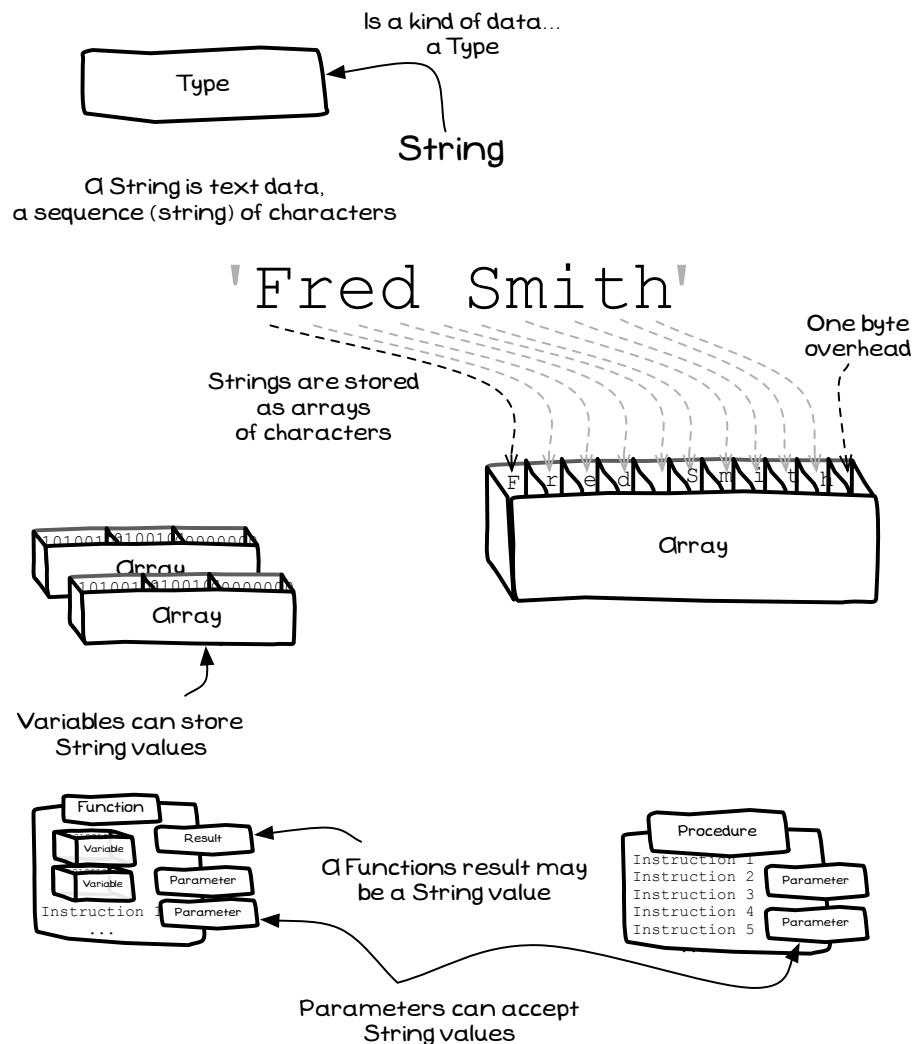


Figure 4.9: Strings are textual data, stores as an array of characters

Note

- Boolean is an existing **artefact**, it is a **Type** that has been defined to represent text values.
- A string is an array of characters.
- Both C and Pascal have additional overhead in the string data. In C the overhead is used to store a single terminating character that indicates the end of the string, in Pascal this stores the number of characters.

C++

C does not have a native String type, instead you use an array of characters. As a result, C has very limited support for string data. For details see [C String](#).

4.1.7 Summary

This Chapter introduced a number of concepts related to working with multiple values in code.

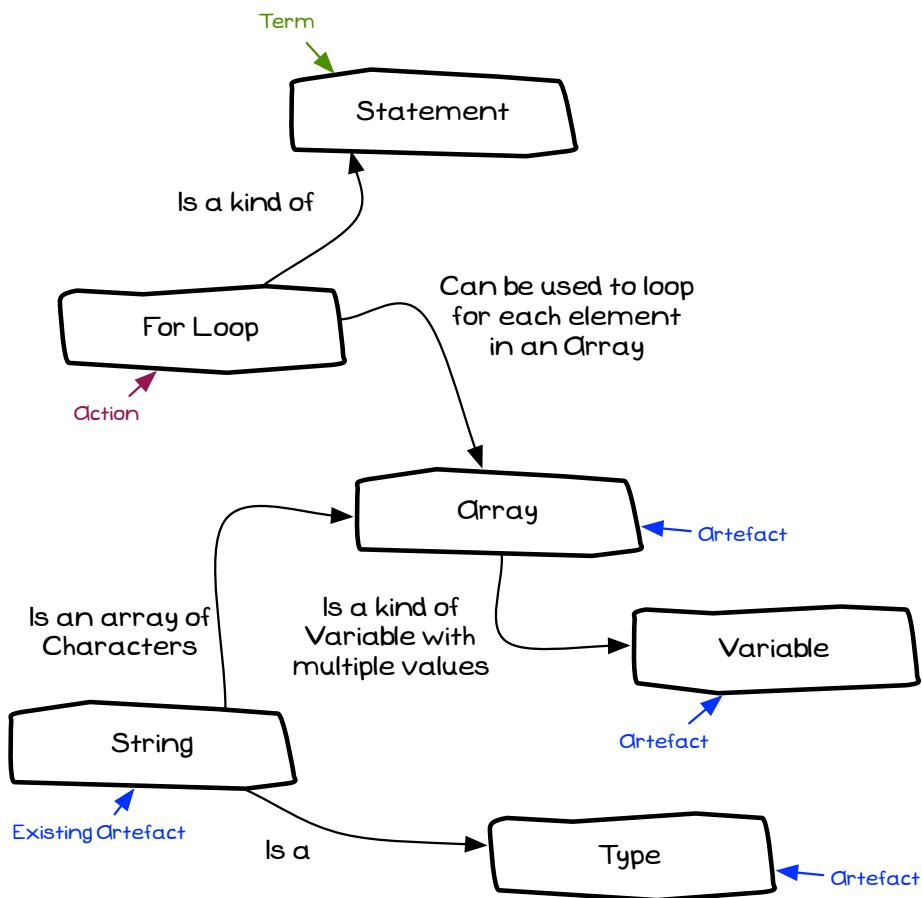


Figure 4.10: Concepts covered in this Chapter

Note

- The central concept of this chapter is the **Array**. An array is a variable that can store multiple values.
- When you work with arrays make any logic you want to apply to *all elements* will be coded using **for each** element and the **For Loop**.
- **Strings** are arrays of characters, and can be used to store textual data in your code.

4.2 Using these Concepts

Arrays make it easier to work with multiple values, allowing you to have a single variable that stores multiple values. With arrays you can start to create programs that process larger quantities of data.

4.2.1 Designing Statistics Calculator

Table 4.1 contains a description of a small statistics programs. This program reads a number of values from the user, and then outputs some statistics calculated from these values. This program will make use of arrays to store the values entered, and then calculate the required statistics.

Program Description	
Name	Statistics Calculator
Description	Reads values from the user and calculates a range of statistics that are output to the Terminal. Statistics output include mean , maximum , sum , and variance .

Table 4.1: Description of the Statistics Calculator program.

As before, the process to design and implement this program will follow a number of steps:

1. Understand the problem, and get some ideas on the tasks that need to be performed.
2. Choose the artefacts we will create and use
3. Design the control flow for the procedural² artefacts
4. Map these artefacts to code
5. Compile and run the program

4.2.2 Understanding the Statistics Calculator

Most of the ideas around the Statistics Calculator should be fairly straight forward. The main thing to be checked if the equation needed to calculate the different statistics, and then to convert these into steps for the computer.

Calculating Sum

The **sum** is the simplest of the Statistics to calculate. This involves adding together all of the numbers in the array. The main issue here is that the computer cannot add all of these values together, and we must rethink our logic to express it in terms of processing *for each* element.

Think about the way you would sum a list of numbers, this is now the task you need to code for the Computer. What is it that you do with each number? To think this through write a list of random numbers down, and then calculate the sum. Do it slowly, and think about the tasks that you are performing *for each number*.

You should have noticed that you are keeping a running total, and that you add the value of each number from the list to that. When you have done this for each number in the list you have the total. The Pseudocode for this is shown in Listing 4.1.

²The program, and any Functions and Procedures.

Pseudocode

```

Function: Sum
-----
Returns: Double - The sum of the numbers from the data array
Parameters:
1: data (by const ref, array of Double) - the list of values to sum
2: size (Integer) - the number of elements in data (C only)
Local Variables:
*: i (Integer) - index of the current element in the array
*: total (Double) - running total
Steps:
1: total is assigned 0
2: For i, starts at 0 and loops to the highest index of data
3: total is assigned total + the value of the ith element of data
4: Return the result, total

```

Listing 4.1: Pseudocode for Sum**C++**

```

// Calculate the sum of the values in the array
double sum(const double data[], int size)
{
    int i;
    double total = 0;

    for(i = 0; i < size; i++)
    {
        total += data[i];
    }

    return total;
}

```

**Listing 4.2:** C code for Sum Function. You can read the for loop as 'i starts at 0; while i is less than size; increment i at the end of the loop'**Pascal**

```

function Sum(const data: array of Double): Double;
var
    i: Integer;
    total: Double;
begin
    total := 0;

    for i := Low(data) to High(data) do
    begin
        total += data[i];
    end;

    result := total;
end;

```

**Listing 4.3:** Pascal code for Sum Function

There are three key things to notice about the Pseudocode in Listing 4.1. Firstly the **For Loop**

is used to repeat the loop once for each element in the array. Second, the *i* variable moves through the valid indexes for the array. Finally, the total is used to keep the running total throughout the code.

The for loop in [For Loop](#) will repeat its body once for each value in the array. The *i* variable will be updated to have the *current* index value each time the loop is repeated. Within the loop the *ith* value from the array is accessed. This is how the for loop processes each of the values from the array.

The total value keeps track of the current running total. Before the loop its value is set to 0, ensuring that it is appropriately initialised. In the body of the loop the current (*ith*) value of the array is added to the total, and the result stored back into total. This means that by the end of the loop the total variable is now storing the sum of all of the elements of the array.

C and Pascal differ in the amount of support they have for working with arrays. C has very limited support, meaning that you need to do some extra work. Pascal has more build in support for arrays, making some common tasks easier to achieve. The main difference is that C does not keep track of the length of an array. This means that you need to pass the number of elements in the array along with the array to functions and procedures that will work with this data. Pascal, on the other hand, does keep track of the length of arrays and gives you three functions you can use to manage this: Low returns the first index of the array, High returns the last index of the array, Length returns the number of elements in the array.

C++

The C code for the Sum function is in Listing 4.2. Notice how the size parameter is being used to represent the number of elements in the array. This loop is the standard pattern used to code [For Loops](#) that loop over elements in an array in C. 

Pascal

The Pascal code for the Sum function is in Listing 4.3. Notice how it is using the Low and High functions to get the range of valid array indexes. This loop is the standard pattern used to code [For Loops](#) that loop over elements in an array in Pascal. 

Calculating Mean

The mean of a list of values is the sum of those values divided by the number of values. In the case of the Statistics Calculator program there is already a `Sum` function, so the `Mean` function does not need to recalculate the sum, it can just call the `Sum` and use the result returned.

The length of the array can be calculated in Pascal using its `Length` function, whereas C can use the `size` parameter to determine the number of elements in the array. In both cases the basic logic is the same, you use the `Sum` function to calculate the sum and then divide this by the number of elements in the array.

Pseudocode

```
Function: Mean
-----
Returns: Double - The average (mean) of the numbers from the data array
Parameters:
1: data (array of Double) - the list of values
2: size (Integer) - the number of elements in data (C only)
Steps:
1: Return the result, Sum(data, size) / size
```

Listing 4.4: Pseudocode for Mean



C++

```
double mean(double data[], int size)
{
    return sum(data, size) / size;
}
```

Listing 4.5: C code for Mean Function



Pascal

```
function Mean(const data: array of Double): Double;
begin
    result := Sum(data) / Length(data);
end;
```

Listing 4.6: Pascal code for Mean Function



Calculating Maximum

Calculating the largest value in the array, the maximum, will require the logic be adjusted to use the *for each* style. How do you calculate the largest value in a list of numbers? With a small list you are likely to just quickly scan it and see the largest value. Think about the tasks you need to perform, and maybe think about how you would do it for a very long list of numbers, one that spans across many pages.

The algorithm needed to find the maximum value in an array needs to perform an action *for each* element of the array. It needs to process each value in isolation, ignoring the other values from the list.

The key is similar to the logic from the `Sum` function. You need to keep a *running* maximum. This will store the *current* maximum from the array as you loop through *each element* of the array. Like the sum this value can be updated within the loop.

The Pseudocode for this is in Listing 4.7. Notice that its basic layout is the same as the `Sum` function in Listing 4.1. It initialises the `max` value and then loops through the array performing an action for each value. In this case the action is to check if the i^{th} value of the array is larger than the running maximum in the `max` variable. When this is the case a new maximum has been found and is stored in the `max` variable.

One of the important differences between `Maximum` and `Sum` is the initialisation of the `max` value. In `Maximum` this cannot be initialised to 0 as this would fail to find the maximum if all values were negative. The maximum must be a value from the array, so it is initialised to the first value in the array. The `for` loop will then start looping from the 2^{nd} element, as the 1^{st} has already been processed.

Pseudocode

```

Function: Maximum
-----
Returns: Double - The sum of the value of the largest number from the data array
Parameters:
1: data (array of Double) - the list of values to sum
2: size (Integer) - the number of elements in data (C only)
Local Variables:
*: i (Integer) - index of the current element in the array
*: max (Double) - current largest number
Steps:
1: max is assigned the value of the 0th element of the array
2: For i, starts at 1 and loops to the highest index of data
3:   if the ith element is larger than max then
4:     max is assigned the value of the ith element of the array
4: Return the result, max

```

Listing 4.7: Pseudocode for Maximum



Calculating Variance

The last statistic to calculate is the Variance. The processing for this will be very similar to the Sum and Maximum functions, though the actual calculation is a little more complex. Equation 4.1 shows how the Variance of a sample is calculated.

$$var(x) = \frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n - 1}$$

4.1

In Equation 4.1 x is the array being processed, \bar{x} is the mean of x , x_i is the value of the i^{th} element of x , and n is the number of elements in the array. The Sigma indicates that $x_i - \bar{x}$ needs to be summed for each element of x .

The steps to calculate the Variance are therefore:

1. Determine the value of the mean (\bar{x}).
2. Calculate $(x_i - \bar{x})^2$ for each element, and store these in a running sum (called `temp`).
3. Divide the value (from `temp`) by the number of elements in the array minus one.

The matching Pseudocode for this is shown in Listing 4.8. In this case x is the data array. In Step 1 the mean (\bar{x}) is calculated once and stored in `avg`. The loop starts at Step 3, and runs *for each* element of the array. The value $(x_i - \bar{x})^2$ is calculated for each element in Step 4, and added to the running total stored in `temp`. The final result is then calculated and the result returned in Step 5.

Pseudocode

```

Function: Variance
-----
Returns: Double - The Variance of the numbers from the data array
Parameters:
1: data (array of Double) - the list of values
2: size (Integer) - the number of elements in data (C only)
Local Variables:
- i (Integer) - index of the current element in the array
- avg (Double) - used to store the mean
- temp (Double) - stores a temporary values used in the calculation
Steps:
1: avg is assigned Mean(data, size)
2: temp is assigned 0
3: For i, starts at 0 and loops to the highest index of data
4:   temp is assigned temp + the square of (the ith element of data - avg)
5: Return the result, temp / (size - 1)
```

Listing 4.8: Pseudocode for Variance

4.2.3 Choosing Artefacts for Statistics Calculator

In understanding these concepts we have uncovered some Functions that will be included in the program's design.

With the calculations thought through the design seems to be coming together. So far we have thought through the steps needed to calculate the output, but we have not thought about how these values will be read into the program.

Programs can be thought of as transforming data, taking inputs and generating outputs, as shown in Figure 4.11. So far we have examined the processing needed to create the outputs, but we still need to consider how the data gets into the program, the inputs.

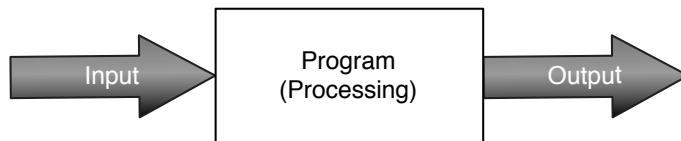


Figure 4.11: Programs convert Inputs to Outputs

At the start of the program the user will need to enter the values that will be stored in the array. This task can be coded in a `Populate Array` procedure. This will get the user to enter all of the values into the array. In other words it will allow the user to enter *each value* in the array.

The logic for populating the array can be split into a `Populate Array` procedure that calls a `Read Double` function. The `Read Double` function will be very useful across a number of different programs, so this may be able to be used elsewhere.

Reading double values from the user

Figure 4.12 shows the flowchart for the process of reading a double value from the user. This includes a [Pre-Test Loop](#) that repeatedly asks the user to enter a number if they value they enter is not a number. This demonstrates a standard *validation loop*, in which you read a value, and check that it is valid in a loop.

The C and Pascal code for this are both slightly different to the flowchart due to different way they handle input and the features they offer to for converting the value read to a number. Details of these are shown alongside Listing 4.9 and Listing 4.10.

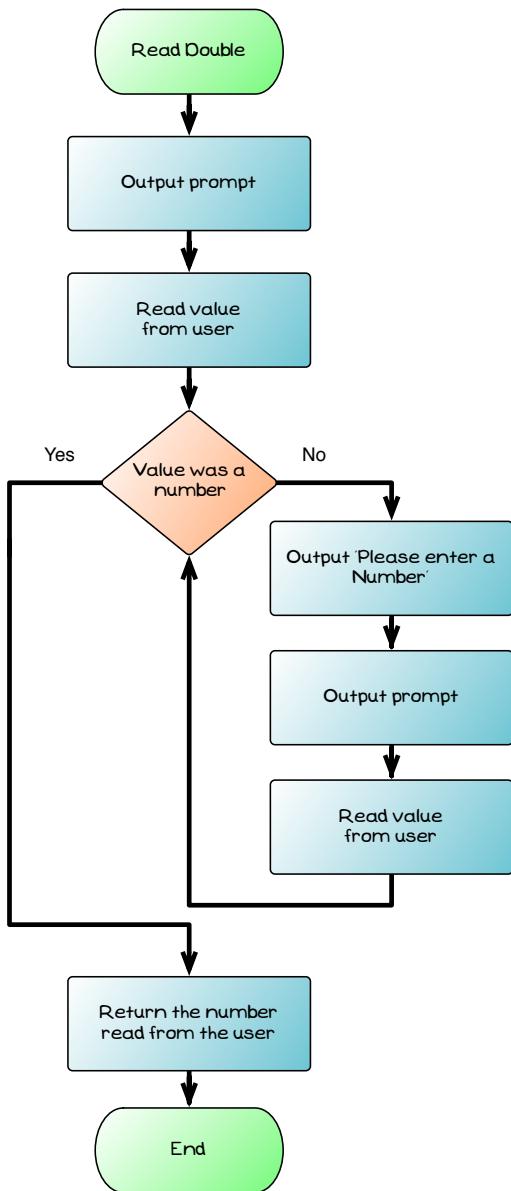


Figure 4.12: Flowchart showing the process for reading a double

C++

The C version of Read Double, shown in Listing 4.9, uses `scanf` (see [C Terminal Input](#)) to read and format the number in the one action. The `scanf` Function returns a value indicating the number of conversions that were successful. This can be checked in the loop, and the loop is repeated *while* `scanf` did not convert one value.

In the loop itself the `scanf` is used to read past the value that is not a number, as `scanf` does not proceed when it finds an error. The format string in `scanf` indicates that it should read everything up to the end of the line.

```
double read_double(const char *prompt)
{
    double result = 0;

    printf("%s", prompt);
    while (scanf(" %lf", &result) != 1) // Read value, and try to convert double
    {
        // Convert failed, as input was not a number
        scanf("%*[^\n]"); // Read past the end of the current line
        printf("Please enter a number.\n");
        printf("%s", prompt);
    }

    return result;
}
```

Listing 4.9: C code for Read Double

Pascal

The Pascal version of Read Double, shown in Listing 4.10, reads the input as a string and then try to convert it to a double. The `TryStrToFloat` Function attempts to convert the text read into a number, storing the value in the `result` variable.

```
function ReadDouble(const prompt: String): Double;
var
    temp: String;
begin
    Write(prompt);
    ReadLn(temp); // Read the input as a string

    while not TryStrToFloat(temp, result) do
    begin
        // Convert failed, as input was not a number
        WriteLn('Please enter a number.');

        Write(prompt);
        ReadLn(temp);
    end;
end;
```

Listing 4.10: Pascal code for Read Double

Populating the Array

With the logic for Read Double in place the next step is to determine the steps needed in the Populate Array procedure. This procedure will loop and read a value from the user for each element of the array. This can use the Read Double function to get the value from the user, and then store this in the array's elements.

A flowchart illustrating the steps in Populate Array is shown in Figure 4.13. The decision node is being used to show the control mechanism of the for loop, counting from the lowest index of the array to the highest index. Within the body of the loop the two instructions build a prompt string, and then use this in the call to Read Double. The result returned from Read Double is stored in the current (i^{th}) element of the array.

Once again the C and Pascal code differ in how this is implemented, centred on how the *prompt* is built within the loop. Pascal has built in support for Strings, so its code is much simpler. The C code for this requires you to coordinate the steps needed to build the text for the prompt. The details for these are shown in the text accompanying Listing 4.11 and Listing 4.12.

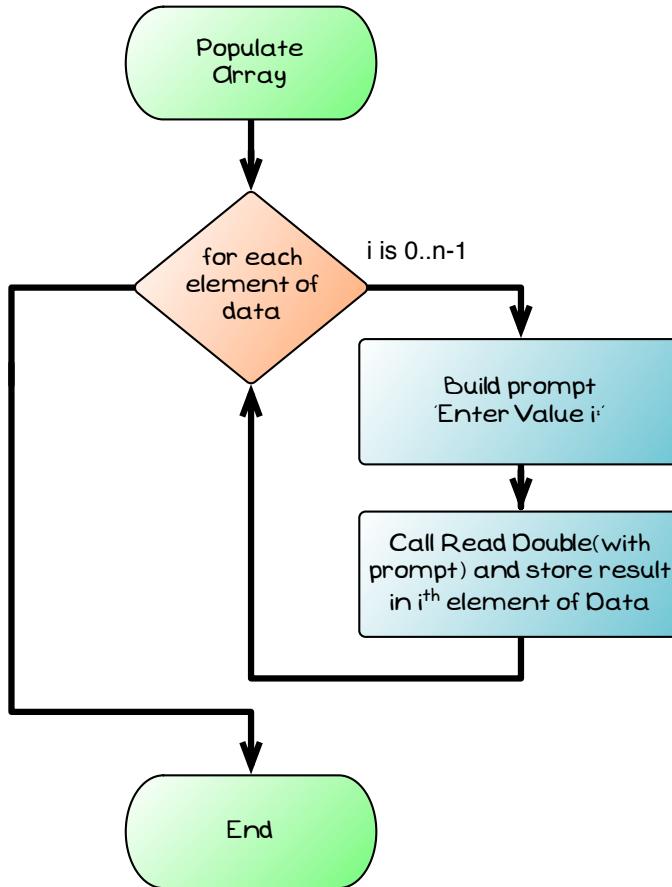


Figure 4.13: Flowchart showing the process for Populate Array

Where is the data stored

The last question to remain is where will the data be stored. The array is a kind of variable, and therefore the array could be a **Local Variable** or a **Global Variable**. As Global Variables should be avoided where possible, this will be coded as a **Local Variable** within the program's Main procedure. It can then be passed from there to the other Functions and Procedures in the code.

C++

The C version of Populate Array is shown in Listing 4.11. This uses the following functions from strings.h. As C does not know the length of the string each of these functions takes a number (n) that indicates the maximum number of characters to copy.

- **strncpy**: String Copy (n characters). The first parameter is the destination, the second the source, the third is the number of characters.
- **sprintf**: Same as printf (see [C++ SplashKit Terminal Output](#)), except that the output is stored in a c-string. In this case the "% 100" ensures that only 2 characters (plus the terminator) are written into the string. See Section [4.3.5 C String](#).
- **strncat**: String Concatenate (n characters). Adds the text in parameter 2 to the end of the string in parameter 1.

```
void populate_array(double data[], int size)
{
    int i;
    char prompt[17] = ""; // enough space for "Enter value 99: " + terminator
    char buffer[3] = ""; // enough space for "99" + terminator
    for(i = 0; i < size; i++)
    {
        // Ensure that the terminator is included in the copy
        // so that the later calls to strncat know where to
        // append their details.
        strncpy(prompt, "Enter value ", 13); // 12 + terminator
        sprintf(buffer, "%d", (i + 1) % 100); // % 100 ensure only 2 chars(+ null)
        strncat(prompt, buffer, 2); // takes 2 spaces, terminator moves
        strncat(prompt, ":", 2); // takes 2 spaces, terminator moves

        data[i] = read_double(prompt);
    }
}
```

Listing 4.11: C code for Populate Array

Pascal

The Pascal version of Populate Array is shown in Listing 4.12. This uses Pascal's **IntToStr** function to convert the value $i + 1$ from Integer to String.

```
procedure PopulateArray(var data: array of Double);
var
    i: Integer;
    prompt: String;
begin
    for i := Low(data) to High(data) do
    begin
        prompt := 'Enter value ' + IntToStr(i + 1) + ': ';
        data[i] := ReadDouble(prompt);
    end;
end;
```

Listing 4.12: Pascal code for Populate Array

Overview of Statistics Calculator's design

That completes the logic needed to implement the Statistics Calculator Program. The final structure is shown in Figure 4.14 as a Structure Chart. Notice the double headed arrow on data in the call from Main to Populate Array. This indicates that the data parameter is passing the values into, and getting values out of the Populate Array procedure. Also see how the data value is passed out of Main to the functions that calculate the statistics.

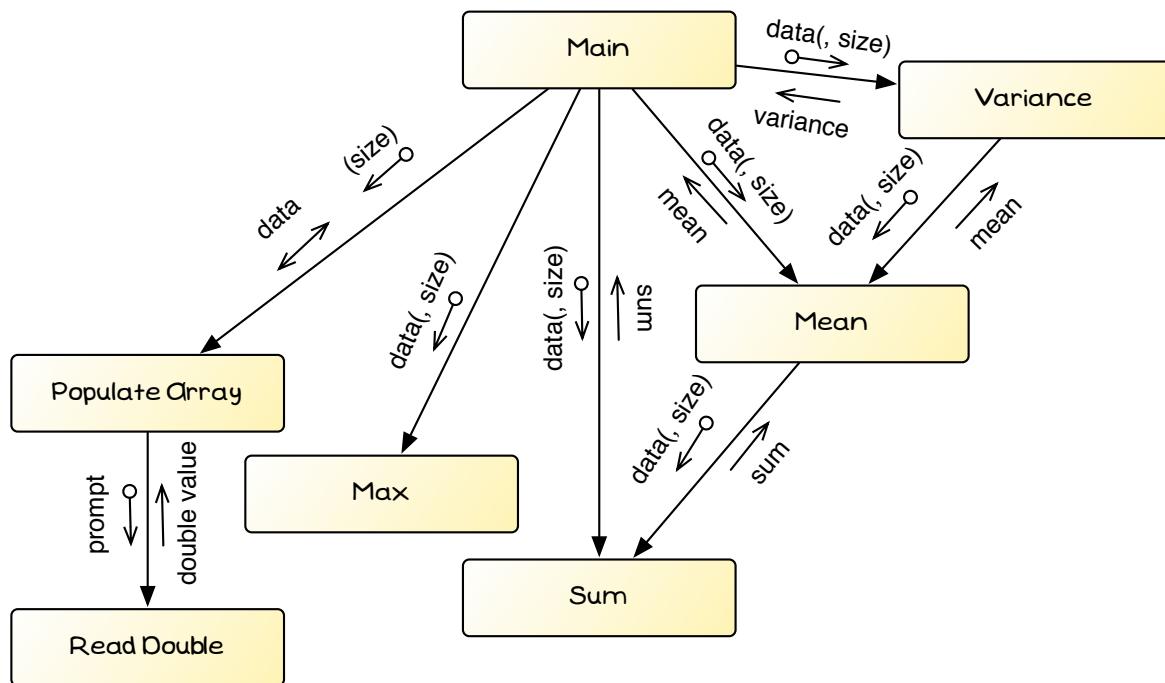


Figure 4.14: Structure Chart showing the structure of the Statistics Calculator program

Note

- Remember anytime you need to do something with all of the elements in an array you need to work out how you can achieve this **for each** element in the array.
- Notice that the processing of individual value from the arrays are always similar.
- Make sure you can see how the for loop is able to perform these values for each element of the array.



4.2.4 Writing the Code for Statistics Calculator

The flowcharts and Pseudocode shown communicate the logic that needs to be coded into the Functions and Procedures of this program. The following two sections, Section 4.3 Managing Multiple Values in C and Section 4.4 Managing Multiple Values in Pascal, contain a description of the syntax needed to code arrays in the C and Pascal programming languages. This information can be used to write the code for the Statistics Calculator, and other programs.

4.2.5 Compiling and Running Statistics Calculator

When the code is finished you can compile and run the program. It is a good idea to implement the solution a little bit at a time, compiling and running it frequently as you progress. Try implementing the solution using the following smaller steps, and the tests shown for each.

1. Start by getting `Read Double` to work. In `Main` just read a single value and output it to the Terminal. **Tests:**
 - Check that a number can be read correctly.
 - Try entering text, and check the error message is shown and that you can enter a number the next time.
 - Try entering multiple text values on a single line.
 - Try entering multiple text values, one after the other.
2. Implement `Populate Array`. Include an array in `Main`, and have its values read in by `Populate Array`. Print the values back to the Terminal so that you can check this code is working. **Tests:**
 - Enter each of the values and check they are printed out correctly.
 - Try entering text, this should be handled by `Read Double` but check it is working correctly with `Populate Array`.
3. Implement the `Sum` Function. **Tests:**
 - Test that it works with some basic values.
 - Try all negative values.
 - Try a mix of positive and negative values.
4. Implement the `Mean` Function. Same tests as `Sum`.
5. Implement the `Variance` Function. Same tests as `Sum`.
6. Finish by implementing the `Maximum` Function. Same tests as `Sum`.

By building the code a little at a time, and running tests as you go, you will have less code to search when you do find an issue. This makes it easier to fix those little errors that are likely to slip into the code from time to time.

When this iterative process is complete you should have a solution for the Statistics Calculator. You should be able to easily change this so that it can read in ten, a hundred, or even a thousand values from the user. This is something that would not have been possible without using arrays.

Note

- Get your code running as soon as you can.
- Build a little and test a little as you go.
- When you find a bug start looking for it in the code you just added (or changed).
- You can perform this as a **design, implement, compile, test** cycle. This enables you to build the program a small piece at a time. Before you start this it is likely to be a good idea to have at least a sketched out plan for the design.
- As you develop your design skills you will be able to create larger designs before you code. While you are learning to program it is ok to design and code at the same time as this lets you to quickly test if your design will work.

4.3 Managing Multiple Values in C

4.3.1 Implementing Statistics Calculator in C

Section 4.2 of this Chapter introduced the Statistics Calculator. A partial implementation of this program is shown in Listing 4.13, with the logic in the `max` and `variance` functions still to be implemented. This program reads a number of values from the user into an array, and then calculates and outputs the **sum**, **mean**, **variance**, and **maximum** value from this data.

```
/* stats-calc.c */

#include <stdio.h>
#include <math.h>
#include <string.h>

#define DATA_SIZE 10

// Calculate the sum of the values in the array
double sum(const double data[], int size)
{
    int i;
    double result = 0;

    for(i = 0; i < size; i++)
    {
        result += data[i];
    }

    return result;
}

// Calculate the mean of the values in the array
double mean(const double data[], int size)
{
    return sum(data, size) / size;
}

// Find the largest value in the array
double max(const double data[], int size)
{
    //todo: add logic here...
    return 0;
}

// Find the standard deviation of the values in the array
double variance(const double data[], int size)
{
    //todo: add logic here...
    return 0;
}

double read_double(const char *prompt)
{
    double result;

    printf("%s", prompt);
    while (scanf(" %lf", &result) != 1)
    {
        scanf("%*[^\n]");
        printf("Please enter a number.\n");
    }
}
```

```

        printf("%s", prompt);
    }

    return result;
}

void populate_array(double data[], int size)
{
    int i;
    char prompt[17] = ""; // enough space for "Enter value 99: " + terminator
    char buffer[3] = ""; // enough space for "99" + terminator

    for(i = 0; i < size; i++)
    {
        // Ensure that the terminator is included in the copy
        // so that the later calls to strncat know where to
        // append their details.
        strncpy(prompt, "Enter value ", 13); // 12 + terminator
        sprintf(buffer, "%d", (i + 1) % 100); // needs space for 3 (2 + terminator)
        strncat(prompt, buffer, 2); // takes 3 spaces, 2 + terminator
        strncat(prompt, ":", 2); // takes 3 spaces, 2 + terminator

        data[i] = read_double(prompt);
    }
}

// Implements a statistics calculator. The program reads in values entered by the user
// and then calculates the sum, mean, variance, and max
int main()
{
    double data[DATA_SIZE];

    populate_array(data, DATA_SIZE);

    printf("\nCalculating statistics...\n\n");

    printf("Sum:      %4.2f\n", sum(data, DATA_SIZE));
    printf("Mean:     %4.2f\n", mean(data, DATA_SIZE));
    printf("Variance: %4.2f\n", variance(data, DATA_SIZE));
    printf("Max:      %4.2f\n", max(data, DATA_SIZE));

    return 0;
}

```

Listing 4.13: C code for the Statistics Calculator**Note**

- `strings.h` is included to give access to the various functions needed to manipulate string values. See the comments associated with Listing 4.11.
- `math.h` is included to give access to the `pow` function that will be needed in the implementation of the `variance` function.
- Arrays in C are always passed by reference.
- C does not keep track of the size of an array, the `size` parameter in each function call carries this data along with the array.
- The `DATA_SIZE` constant stores the number of values that will be stored in the array. This can easily be changed to allow the program to read a different number of values.

4.3.2 C Array Declaration

C allows you to declare variables that are arrays. This is done using the [] to denote the number of elements in the array (n). Indexes will then be 0 to $n-1$.

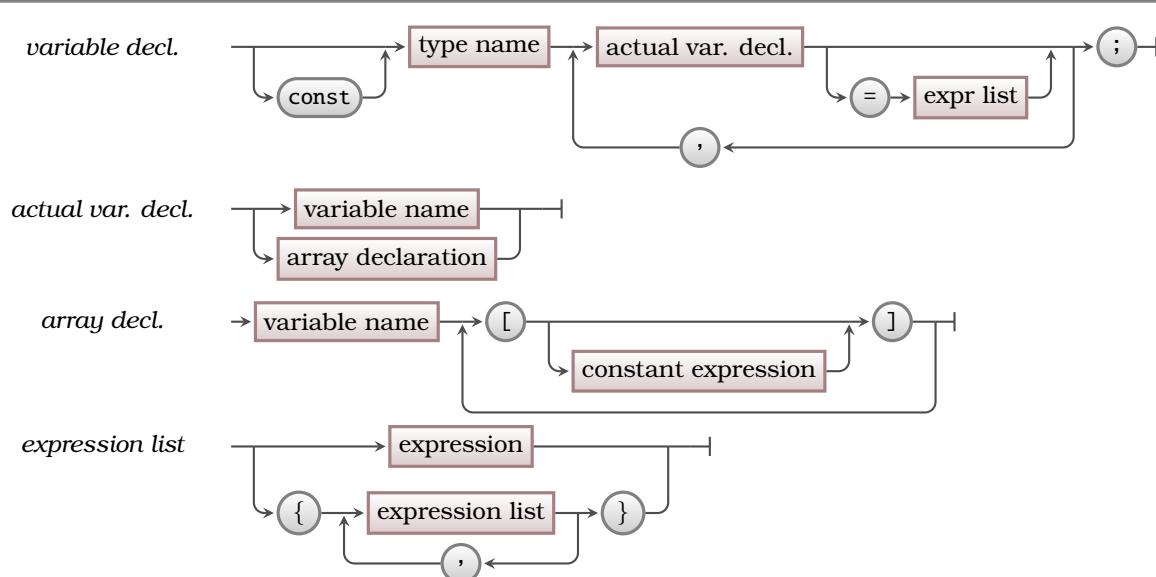


Figure 4.15: C++ Syntax for Array Variable and Type Declarations

C++

```

/* Program: test-array.c */
#include <stdio.h>

int main()
{
    int      data[5] = { 0, -1, 2, -3, 4 };
    double   my_data[5];
    float    other[] = {1.2f, 2.5f, 0, -1, 6};
    int i;

    for(i = 0; i < 5; i++)
    {
        printf("data[%d]: %d\n", i, data[i]);
        printf("name[%d]: %f\n", i, other[i]);
    }
    return 0;
}
  
```

Listing 4.14: C code demonstrating array declaration

Note

- This is the C syntax to declare a **Array**.
- Arrays in C do not remember their length, you must keep track of this yourself.
- You can initialise an array when it is declared using a list of values in braces ({...}). This can only be done to initialise arrays, and is not valid elsewhere.
- The size of the array must be able to be determined at compile time.

4.3.3 C Array Copying

In C you cannot use simple assignment to copy all of the elements of an array into another array. Instead you can use the `memcpy` (memory copy) function to perform this task for you. It copies a chunk of memory from one location to another.

Function Prototype	
<code>void *memcpy(void *destination, const void *source, size_t num)</code>	
Returns	
<code>void *</code>	Destination is returned, can be ignored.
Parameter	Description
<code>destination</code>	The location where the data is copied to.
<code>source</code>	The data to copy.
<code>num</code>	The number of bytes to copy.

Table 4.2: Details of the `memcpy` function

The `memcpy` function needs to be told the number of bytes to copy. The `sizeof` operator can be used to get this information for a type or variable. The details of this operator are shown in Figure 4.16.



```
sizeof → ( sizeof → ( → expression → ) ) →
```

Figure 4.16: C++ Syntax for The `sizeof` operator lets you get the size, in bytes, of a type or variable.

C++

```
/* program: array-copy.c */

#include <string.h>

int main()
{
    int data[3] = {1, 2, 3};
    int other_data[3];
    int more_data[3];

    memcpy(other_data, data, 3 * sizeof(int));
    memcpy(more_data, data, sizeof(data));

    return 0;
}
```

Listing 4.15: C code demonstrating array copying

Note

- The `size_t` type is a whole number used to store the size of types.
- To get the size of a variable or type using the `sizeof` operator.
- You can think of the `sizeof` operator as being a function that returns the size of a type or variable.
- The expression in `sizeof` must be a variable or type name.

4.3.4 C For Loop

The **For Loop** in C can do much more than just counting, but that is its primary purpose. You can use this to implement the logic to process each element of an array.

The for loop itself is controlled by three aspects. The first is an initialiser, it sets the first value for the control variable (usually *i* if you are using it to index an array). The second part is the condition, the body will run *while* this is true just like a **C While Loop**. The third part is a post loop increment, you use this to move the index to the next value.

The standard for loop is: `for(i = 0; i < size; i++){...}`. This can be read as ‘for *i* starts at 0, while *i* is less than *size*, do the following then increment *i*’. If *size* is three then this counts 0, 1, 2. Repeating the body of the loop three times.

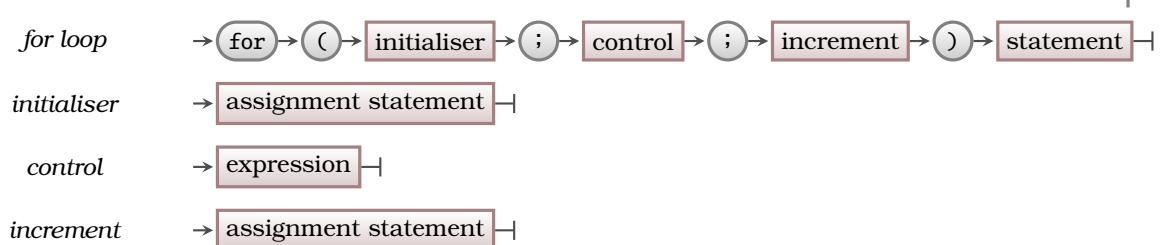


Figure 4.17: C++ Syntax for a For Loop

C++

```

/* Program: test-for.c */
#include <stdio.h>

void print_characters(const char *text, int size)
{
    int i;
    for (i = 0; i < size; i++)
    {
        printf("%c (ASCII %hd) at index %d\n", text[i], text[i], i);
    }
}

int main()
{
    print_characters("Hello World", 12);
    print_characters("Fred", 5);
    return 0;
}
  
```

Listing 4.16: Code illustrating the for loop in C

Note

- This is the C syntax for implementing a **For Loop**.
- The *initialiser* of the for loop is used to set the initial values before the loop body is first executed.
- The *control* is a *condition*, the loop executes *while*^a this is true.
- The *increment* is a post loop action, used to move to the next index.

^aIn the same way a while loop does, checking the condition to determine if the body should execute or be skipped when the condition is checked.

4.3.5 C String

C was designed to build for use with the Unix operating system. When the language was designed string manipulation was not a high priority, and therefore C does not have built in capabilities to perform tasks like concatenating strings, and copying strings (i.e. assigning a string a value after it has been declared).

Working with c-strings requires that you think about how the text is represented in the computer. Table 4.3 shows the characters used to store the text value ‘Fred’.

As C does not keep the length of the array there needs to be a means of determining how long the string is. The method that C choose was to place a **sentinel** value at the end of the string. This marks the position in the array where the string ends. The sentinel is the null character, the one with value the `\0`.

Characters:	F	r	e	d	<code>\0</code>
Bytes Values ^a :	70	114	101	100	0

Table 4.3: The characters and byte values for the c-string containing the text ‘Fred’

^aByte values are shown as decimal.

Space characters are distinct from the null character. Table 4.4 shows the characters involved in storing the text ‘Fred Smith’. The space character is the value 32, and the sentinel value only appears at the end of the c-string. To store ‘Fred Smith’ you need an array that can store at least 11 characters. Ten for the characters in the name, and one for the sentinel.

Characters:	F	r	e	d		S	m	i	t	h	<code>\0</code>
Bytes Values ^a :	70	114	101	100	32	83	109	105	116	104	0

Table 4.4: Characters for ‘Fred Smith’, the space has the character value 32.

^aByte values are shown as decimal.

It is possible for an array to have more characters than are needed. Table 4.5 shows an array with 11 characters that is storing the c-string ‘Fred’. The null character at index 4 (the 5th character) ends the c-string and the remainder of the data in the array will be ignored by the c-string functions.

Characters:	F	r	e	d	<code>\0</code>	S	m	i	t	h	<code>\0</code>
Bytes Values ^a :	70	114	101	100	0	83	109	105	116	104	0

Table 4.5: This would only print ‘Fred’, as the 0 character indicates the end of the c-string

^aByte values are shown as decimal.

The code in Listing 4.17 shows some examples of the main operations you may want to perform on strings. This includes the following actions:

- **Initialisation:** Creating and initialising a string.
- **Input:** Reading words, and lines, from the Terminal.
- **Comparison:** Checking if two strings are equal. Notice that you also need to check the null value.

Other common string operations are found in Listing 4.11. These included:

- **Copy:** Assigning one string to another, as you cannot use the assignment statement to achieve this in C.
- **Concatenate:** Adding one string to the end of another.

C++

```

/* Program: test-string.c */
#include <stdio.h>
#include <strings.h>

int main()
{
    const char *sample_text = "Hello World";
    char name[256] = ""; // initialise all 256 characters to null (255 + 1)
    char word[10] = ""; // space for 9 + sentinel

    printf("%s has %ld characters\n", sample_text, strlen(sample_text));

    printf("Enter a word (upto 9 characters long): ");
    scanf("%9s", word); // no & as this is an array, c will pass it by reference
    printf("You entered %s\n", word);

    printf("Enter your full name: ");
    scanf(" %255[^\\n]", name); // again no & as name is an array.
    printf("Welcome %s\n", name);

    if (strncmp(name, "Fred Smith", 11) == 0)
        printf("Wow, you have the same name as used in the text!\n");

    return 0;
}

```

Listing 4.17: Code illustrating working with Strings in C

Note

- In C a String is an array of characters. There is little built in support beyond this in the C language itself. The Standard C libraries include functions that can be used to work with String data, in `strings.h`.
- **Remember** to ask for enough space to store the text and the sentinel value when declaring a c-string. If you want to store 4 characters then you need to ask for an array with space for 5, the 4 characters + and 1 sentinel value.
- The c-string functions will look for the null character. If the null character is missing from the end of the c-string then these functions will not work as you want. The problem is that they may appear to be working, though in reality they are interacting with memory that is not associated with the c-string you are working on.
- **Take care when working with c-strings!** Many security issues in software relate to incorrect handling of c-strings.

Print and Scanning in Strings

The stdio.h header also provides version of printf and scanf that are used to write values to, and read values from strings. The sprintf function writes data into a destination string, whereas the sscanf function reads data out of a source string. Table 4.6 shows the details for the sprintf function, Table 4.7 shows the details for sscanf.

Function Prototype	
<code>int sprintf(char *destination, const char *format, ...)</code>	
Returns	
Parameter	Description
destination	The number of characters written to the destination by sprintf.
format	The string to write the output into. Warning: You are responsible for ensuring there is enough space.
...	The text that is to be written to the string. This text may contain format tags to include other values. This is the same as printf, see Figure ?? for the syntax of the format tag.
...	Optional values, must have at least as many values as format tags.

Table 4.6: Parameters that must be passed to sprintf

Function Prototype	
<code>int sscanf(const char *source, const char *format, ...)</code>	
Returns	
Parameter	Description
int	The number of values read by sscanf.
source	The string from which the input is read.
format	The format specifier describing what is to be read from the Terminal. This is the same as with scanf, see Table 2.4.
...	The variables into which the values will be read. There must be at least as many variables as format tags in the format specifier.

Table 4.7: Parameters that must be passed to sscanf

Note

- These functions are useful for converting data to and from string values.
- You can use sprintf to convert numeric values and store them in a string. Though care must be taken to ensure that there is sufficient space for these values in the destination string.
- sscanf can be used to read values out of a string. For example, you could read a numeric value out of a string value entered by the user.

Example use of C-string functions

The Statistics Calculator requires some string manipulation to generate the prompt that will be shown to the user. The prompt is created from the text 'Enter value', the value of $i + 1$, and the text ': '. The four function calls needed to achieve this are shown in Figure 4.18.

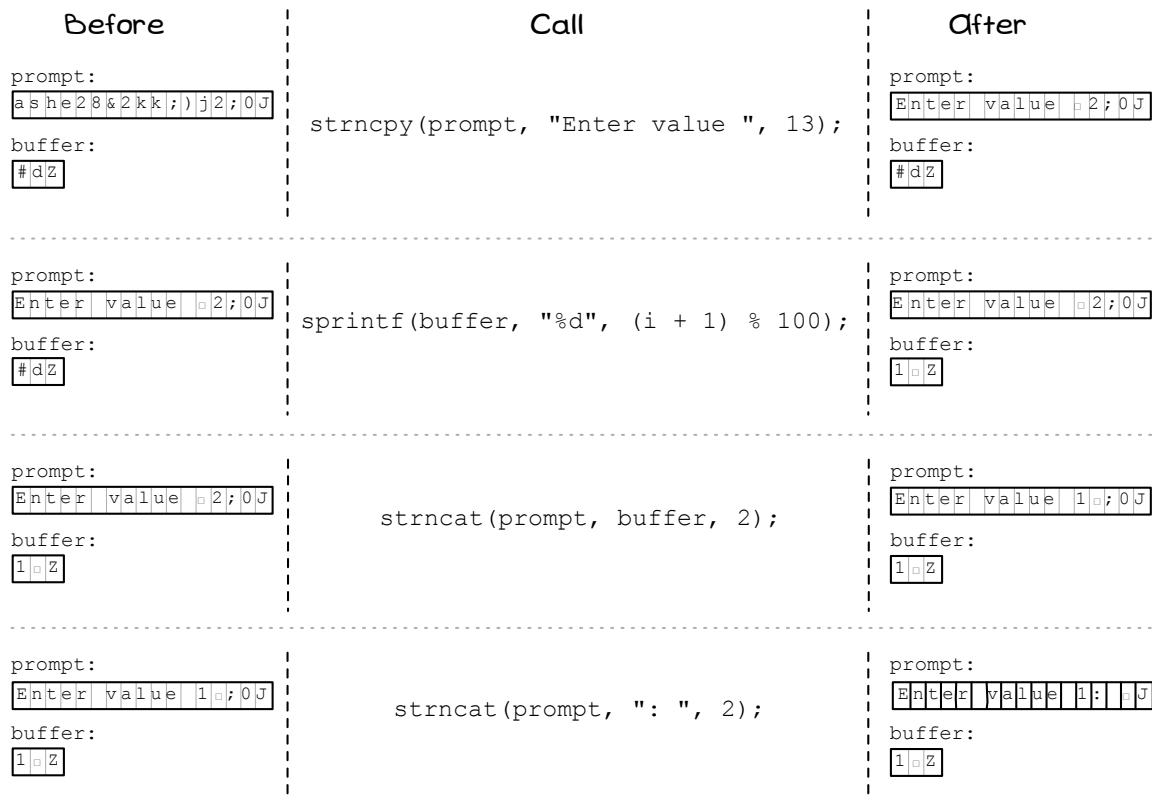


Figure 4.18: Example usage of c-string functions from the Statistics Calculator

Note

- The steps shown in Figure 4.18 perform the following actions:
 1. The first instruction uses `strncpy` to copy the characters in 'Enter value' into the `prompt`. The null character must also be copied so that `strncat` knows where the string currently ends.
 2. Step 2 uses `sprintf` to print the decimal value of $(i + 1) \% 100$ into the `buffer`. This uses $\% 100$ so that only two characters^a and the terminal are ever written into the `buffer`. This assumes that the value of i is 0, so $i + 1$ is 1.
 3. Next `strncat` is used to concatenate `prompt` and `buffer`. This copies the text in `buffer` to the end of `prompt`, and makes sure that there is a null character at the end. So the `n` in this case indicate the maximum number of actual characters to add to the destination, as the null is effectively moved in this function.
 4. The operation is completed by concatenating the final ': ' to the end of the string.
- Notice that there is one additional character left in this `prompt` at the end, this gives space for two digit values, e.g. 'Enter value 10:'.

^a $(i + 1) \% 100$ is the remainder of the value i incremented by one after dividing by 100. This ensures that it is always between 0 and 99. For example, when i is 106, $i + 1$ is 107, and $107 \% 100$ is 7, the remainder of dividing 107 by 100.

4.3.6 C Function (with Array Parameters)

In C you can only use [Pass by Reference](#) to pass an array to a Function or Procedure. There are two ways of passing arrays by reference in C: one uses the bracket notation (type name[]), the other an asterisks notation (type *name). The asterisks notation is more general pass by reference, and will be covered in a later chapter in more details. The brackets notation accomplishes the same task, and indicates that the passed data will be an array.³

The optional **const** operator allows you to indicate that the passed in value will not be changed in the Function or Procedure. This is important with strings, as if you want to pass a string literal to a parameter it must be a **const char ***, as the literal cannot be changed.

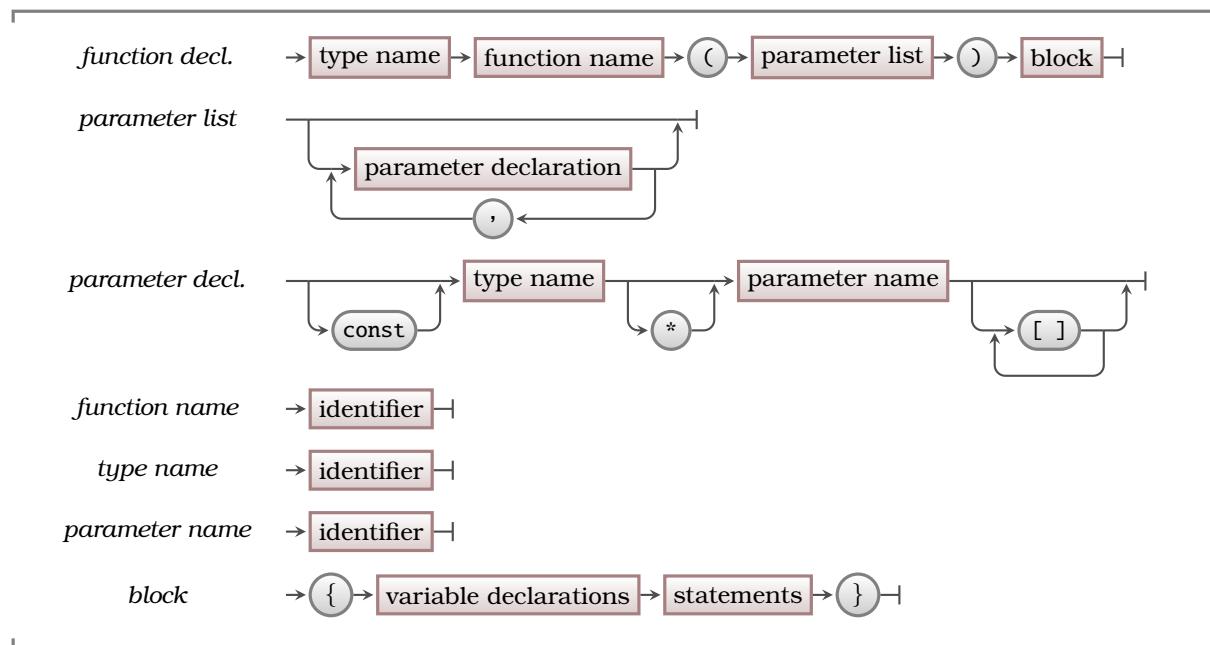


Figure 4.19: C++ Syntax for Functions with Array Parameters

Note

- This syntax shows you how to code [Pass by Reference](#) into your Functions and Procedures in C.
- See Listing 4.18 for examples of the different ways of declaring pass by reference parameters in C.
- Notice that in the call you **do not** need to get the address of arrays, as you would do with other types that are passed by reference. This is because C does this for you in the background. Remember arrays are always passed by reference in C.
- When using the [] syntax you do not specify the size of the array. This allows arrays of varying size to be passed into the Function or Procedure. The size parameter is then used by *convention* to carry across the size of the array.

³Which is passed by reference, as arrays are always passed by reference in C.

C++

```

/* Program: test-array-passing.c */

void test_pass_in_only_v1(const int data[], int size)
{
    printf("Can only read from data -> %d...%d\n", data[0], data[size - 1]);
}

void test_pass_in_and_out_v1(int data[], int size)
{
    printf("Can read and change data\n");
    data[0] = data[0] + 1;           //increment first
    data[size - 1] = data[size - 1] * 2; //double last
}

//-----

void test_pass_in_only_v2(const int *data, int size)
{
    printf("Can only read from data -> %d...%d\n", data[0], data[size - 1]);
}

void test_pass_in_and_out_v2(int *data, int size)
{
    printf("Can read and change data\n");
    data[0] = data[0] + 1;           //increment first
    data[size - 1] = data[size - 1] * 2; //double last
}

//-----

void say_hello_to(const char *name)
{
    printf("Hello %s!\n", name);
}

//-----


int main()
{
    int my_data[] = {1, 2, 3};
    int other_data[] = {1, 2};

    say_hello_to("Fred");

    test_pass_in_and_out_v1(my_data, 3);
    test_pass_in_and_out_v2(other_data, 2);

    test_pass_in_only_v1(my_data, 3);
    test_pass_in_only_v1(other_data, 2);

    return 0;
}

```

Listing 4.18: Code illustrating array passing in C

4.4 Managing Multiple Values in Pascal

4.4.1 Implementing Statistics Calculator in Pascal

Section 4.2 of this Chapter introduced the Statistics Calculator. A partial implementation of this program is shown in Listing 4.19, with the logic in the `max` and `variance` functions still to be implemented. This program reads a number of values from the user into an array, and then calculates and outputs the **sum**, **mean**, **variance**, and **maximum** value from this data.

```
program StasticsCalculator;
uses SysUtils;

const DATA_SIZE = 10;

// Calculate the Sum of the values in the array
function Sum(const data: array of Double): Double;
var
  i: Integer;
begin
  result := 0;

  for i := Low(data) to High(data) do
  begin
    result += data[i];
  end;
end;

// Calculate the Mean of the values in the array
function Mean(const data: array of Double): Double;
begin
  result := Sum(data) / Length(data);
end;

// Find the largest value in the array
function Max(const data: array of Double): Double;
begin
  //todo: add logic here...
  result := 0;
end;

// Find the standard deviation of the values in the array
function Variance(const data: array of Double): Double;
begin
  //todo: add logic here...
  result := 0;
end;

function ReadDouble(prompt: String): Double;
var
  line: String;
begin
  Write(prompt);
  ReadLn(line);
  while not TryStrToFloat(line, result) do
  begin
    WriteLn('Please enter a number.\n');
    Write(prompt);
    ReadLn(line);
  end;
end;
```

```
procedure PopulateArray(var data: array of Double);
var
  i: Integer;
begin
  for i := Low(data) to High(data) do
    begin
      data[i] := ReadDouble('Enter value ' + IntToStr(i) + ': ');
    end;
end;

// Implements a statistics calculator. The program reads in values entered by the user
// and then calculates the Sum, Mean, variance, and max
procedure Main();
var
  data: array [0..DATA_SIZE-1] of Double;
begin
  PopulateArray(data);

  WriteLn('Calculating statistics...');

  WriteLn('Sum: ', Sum(data):4:2);
  WriteLn('Mean: ', Mean(data):4:2);
  WriteLn('Variance: ', variance(data):4:2);
  WriteLn('Max: ', max(data):4:2);
end;

begin
  Main();
end.
```

Listing 4.19: Pascal code for the Statistics Calculator

Note

- SysUtils is used to give access to the TryStrToFloat function. This *tries* to convert a string to a double, and returns a boolean to indicate if it succeeded.
- The arrays in this program are passed by reference using **const** (for in only) or **var** (for in and out).
- The `Low(...)` function gives you the first index of the array, `High(...)` gives you the last index of the array, and `Length(...)` tells you the number of elements in the array.

4.4.2 Pascal Array Declaration

Pascal allows you to declare array variables and parameters.

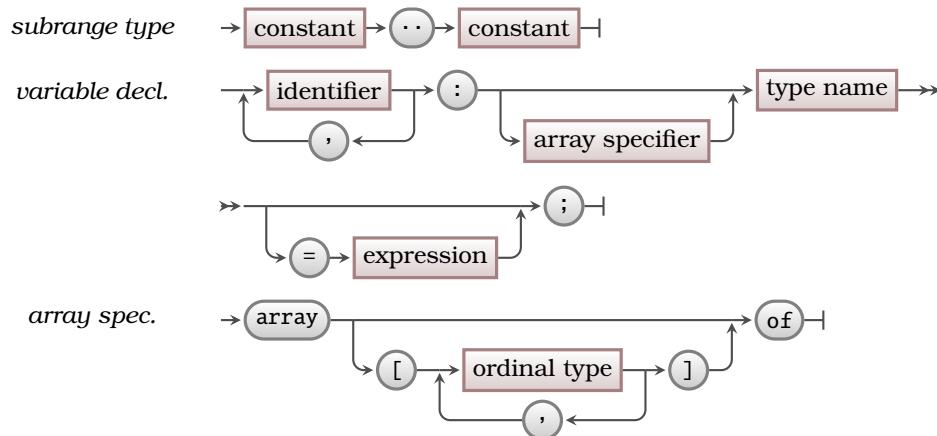


Figure 4.20: Pascal Syntax for Array Variable and Parameter Declarations

Pascal

```

program TestArray;

procedure Main();
var
    data: array [0..2] of Integer = (0, 1, 2);
    data1: array [0..2] of Integer;
    i: Integer;
begin
    data1 := data;           // copy from data into data1...
    data[0] := 1;
    for i := 0 to 2 do
    begin
        WriteLn('data[', i, '] = ', data[i]);
        WriteLn('data1[', i, '] = ', data1[i]);
    end;
end;

begin
    Main();
end.

```

Listing 4.20: Pascal code demonstrating array declaration

Note

- This is the Pascal syntax to declare an **Array**.
- You can initialise an array when it is declared using a list of values in parenthesis `(...)`. This can only be done to initialise arrays, and is not valid elsewhere.
- You can use an assignment statement to copy the contents from one array over another of the same size.

4.4.3 Pascal For Loop

The [For Loop](#) in Pascal can be used to implement the logic to process each element of an array. With the for loop you specify a control variable, and the range of values it will loop over. When the loop is started the control variable is assigned the initial value, at the end of each loop this value is incremented (for `to`) or decremented (for `downto`) until it has processed all values in the indicated range.

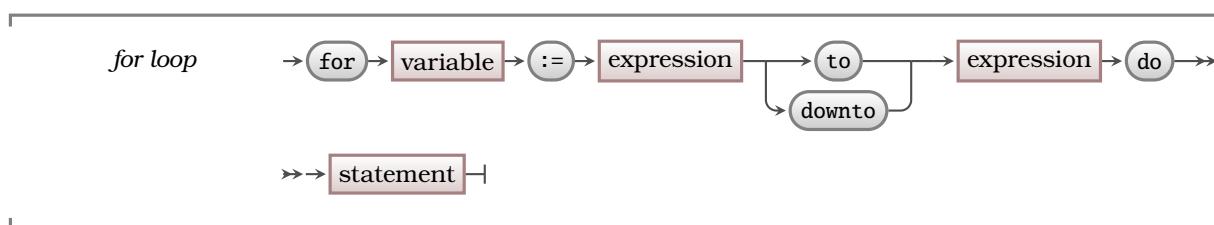


Figure 4.21: Pascal Syntax for a for loop

Pascal

```

program TestFor;

procedure PrintLine(length: Integer);
var
  i : Integer;
begin
  for i := 1 to length do Write('-');
  WriteLn();
end;

procedure PrintCharacters(text: String);
var
  i : Integer;
begin
  for i := 1 to Length(text) do
  begin
    WriteLn(text[i], ' (ASCII ', Integer(text[i]), ') at index ', i);
  end;
end;

begin
  PrintCharacters('Hello World');
  PrintLine(50);
  PrintCharacters('Fred');
end.
  
```



Listing 4.21: Code illustrating the for loop in Pascal

Note

- This is the Pascal syntax for implementing a [For Loop](#).
- The first expression is the *initial value* given to the variable.
- If you use `to` the variable's value is increased by one at the end of each loop, ending the loop when the variable's value is larger than the second expression.
- Alternatively `downto` decreased the variable's value by one at the end of each loop, ending when the variable's value is less than the second expression.



4.4.4 Pascal Array Functions

Pascal includes a number of functions to help you work with arrays. These allow you to determine the length of the array, and its range of indexes. These functions are typically used in conjunction with the for loop.

Function Prototypes	
function Length(arg: array of ...): Integer	
Returns	
Integer	The number of elements in the arg array.
Parameter	Description
arg	The array you want to get the length of.
function Low(arg: array of ...): Integer	
Returns	
Integer	The lowest index in the arg array.
Parameter	Description
arg	The array you want to get the lowest index of.
function High(arg: array of ...): Integer	
Returns	
Integer	The last index in the arg array.
Parameter	Description
arg	The array you want to get the last index of.

Table 4.8: Parameters that must be passed to Length, High, and Low

Pascal

```

procedure Main();
var
  i: Integer;
  myList: array [0..5] of Integer;
  otherArr: array [7..23] of Integer;
begin
  WriteLn('myList has ', Length(myList), ' elements.');
  WriteLn('Its first index is ', Low(myList), ', its last is ', High(myList));
  WriteLn('for i := Low(myList) to High(myList) do ...');

  for i := Low(myList) to High(myList) do
    WriteLn('process myList[', i, ']');

  WriteLn('otherArr has ', Length(otherArr), ' elements.');
  WriteLn('from ', Low(otherArr), ', to ', High(otherArr));
end;

begin
  Main();
end.

```



4.4.5 Pascal Array Parameters

Pascal allows arrays to be passed as parameters to functions and procedures. Like other parameters these can be passed by value or by reference.

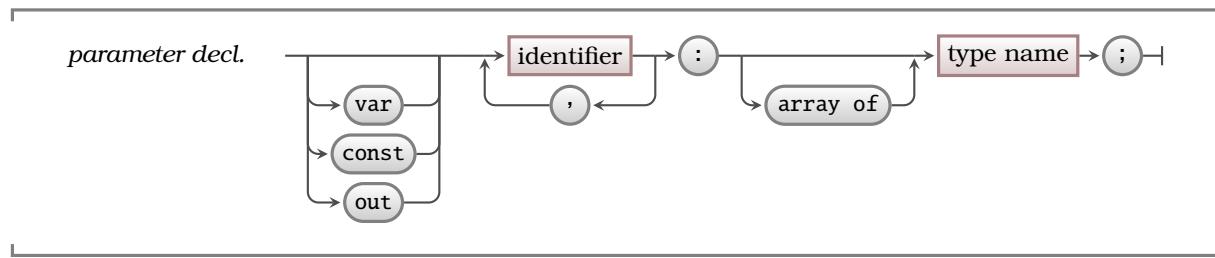


Figure 4.22: Pascal Syntax for array parameters

Note

- This syntax shows you how to code [Pass by Reference](#) into your functions and procedures in Pascal.
- See Listing 4.22 for example code.
- When declaring array parameters you do not specify the size, this allows the parameter to accept arrays of any size.



Pascal

```

program TestArrayPassing;

procedure TestPassInByVal(data: array of Integer);
begin
    WriteLn('Can read and change (local copy of) data -> ',
            data[Low(data)], ' to ', data[High(data)]);
    data[0] := -1000;
    WriteLn('Local copy now -> ', data[Low(data)], ' to ', data[High(data)]);
end;

//-----

procedure TestPassInByRef(const data: array of Integer);
begin
    WriteLn('Can only read from data -> ',
            data[Low(data)], ' to ', data[High(data)]);
end;

//-----

procedure TestPassInOutByRef(var data: array of Integer);
begin
    WriteLn('Can read and change data');
    data[0] := data[0] + 1;           //increment first
    data[High(data)] := data[High(data)] * 2; //double last
end;

//-----

procedure Main();
var
    myData: array [0..2] of Integer = (1, 2, 3);
    otherData: array [0..1] of Integer = (7, 281);
begin
    TestPassInByVal(myData);
    TestPassInByRef(myData);

    TestPassInOutByRef(myData);
    TestPassInByRef(myData);

    TestPassInByRef(otherData);
end;

begin
    Main();
end.

```

Listing 4.22: Code illustrating array passing in Pascal

4.4.6 Pascal String

Pascal has built in support for strings. Behind the scenes Pascal uses an array of characters to store the text for each string. The first element of this array stores an integer that indicates the size of the string, and this is followed by the text characters.

Characters:		F	r	e	d
Bytes Values ^a :	4	70	114	101	100

Table 4.9: The characters and byte values for the string containing the text ‘Fred’ in Pascal

^aByte values are shown as decimal.

Pascal

```
program TestString;

procedure Main();
var
    sampleText, name: String;
    word: String[9];
begin
    sampleText := 'Hello World';

    WriteLn(sampleText, ' has ', Length(sampleText), ' characters');

    WriteLn('Enter a word (upto 9 characters long): ');
    ReadLn(word);
    WriteLn('You entered ', word);

    WriteLn('Enter your full name: ');
    ReadLn(name);
    WriteLn('Welcome ', name);

    if name = 'Fred Smith' then
    begin
        WriteLn('Wow, you have the same name as used in the text!');
    end;
end;

begin
    Main();
end.
```



Listing 4.23: Code illustrating working with strings in Pascal

Note

- You can access individual characters in a string using array notation, the first text character is at index 1.
- Pascal strings can be concatenated used +, for example ‘Hello’ + ‘ World’.



4.5 Understanding Arrays

Arrays offer a means of storing and working with a list of values in your code. Each array has a number of elements, each of which has a value, and can be accessed using an index. Together with the [For Loop](#), arrays provide a means of managing multiple values in your code. The following illustrations show how these work in the computer, and should help you better understand how arrays can be used within your code.

4.5.1 Understanding Populate Array

Section [4.2.1 Designing Statistics Calculator](#) outlined the pseudocode and flowcharts for a small statistics programs. This included a number of functions and procedures that helped divide the program's code into smaller units of work. One of these procedures was [Populate Array](#), discussed in Section [4.2.3 Populating the Array](#). This procedure is responsible for reading values from the user and using these to populate the program's array, and the flowchart for this logic is shown in Figure [4.23](#).

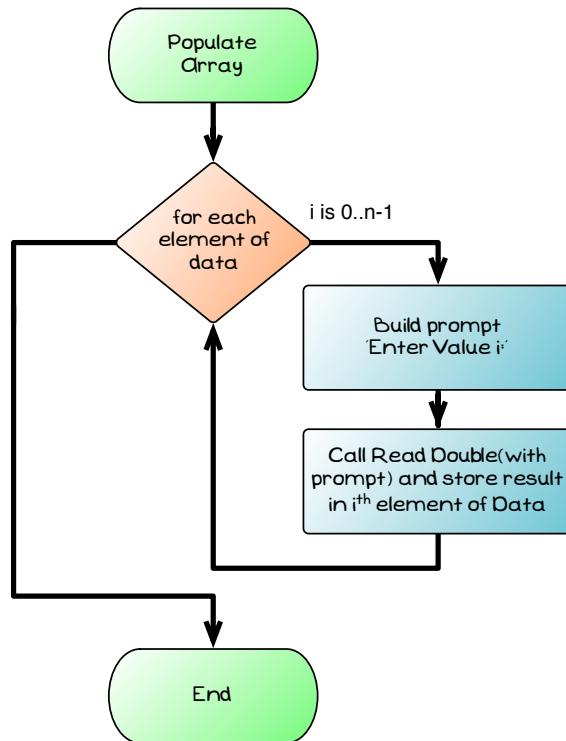


Figure 4.23: Flowchart showing the process for Populate Array, from Figure [4.13](#)

The following illustrations will show this code running to populate an array that contains three values. This will show how the array is passed by reference, and how the for loop works together with the array to populate all elements.

Main starts, and the array is allocated space

All local variables are allocated space on the Stack when the function or procedure they are declared in is called. In this example the Main procedure is executed and space is allocated for its my_data variable. This variable is an **Array** that is used to store three double values. When Main is loaded onto the Stack there is space allocated for three double values associated with the my_data variable.

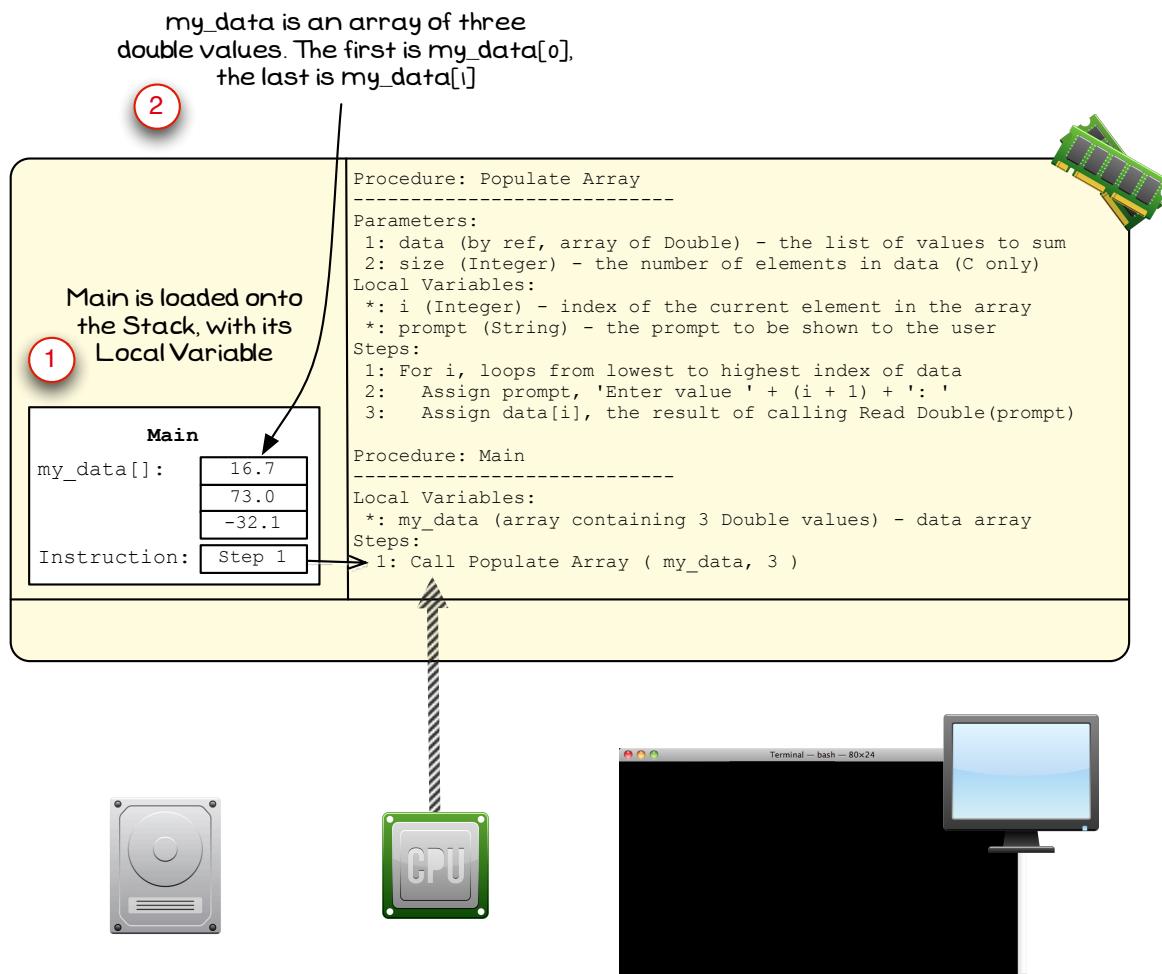


Figure 4.24: When the program starts Main allocates space for its local variables, including the array

Note

- In Figure 4.24 the indicated areas show the following:
 1. The program starts and Main is loaded onto the stack, allocating space for its local variables.
 2. The my_data array is allocated space to store its values.
- Notice that the three values in the array are allocated next to each other.
- The indexes can be used to access the array's elements. The index value determines the number of elements that must be skipped to find where the value is stored.

Populate array is called, and a reference to my_data passed in

Populate Array is called as the first step in Main. This is passed the my_data variable (pass by reference), rather than being passed the values from within that variable. This gives the data parameter access to the memory where my_data is stored.

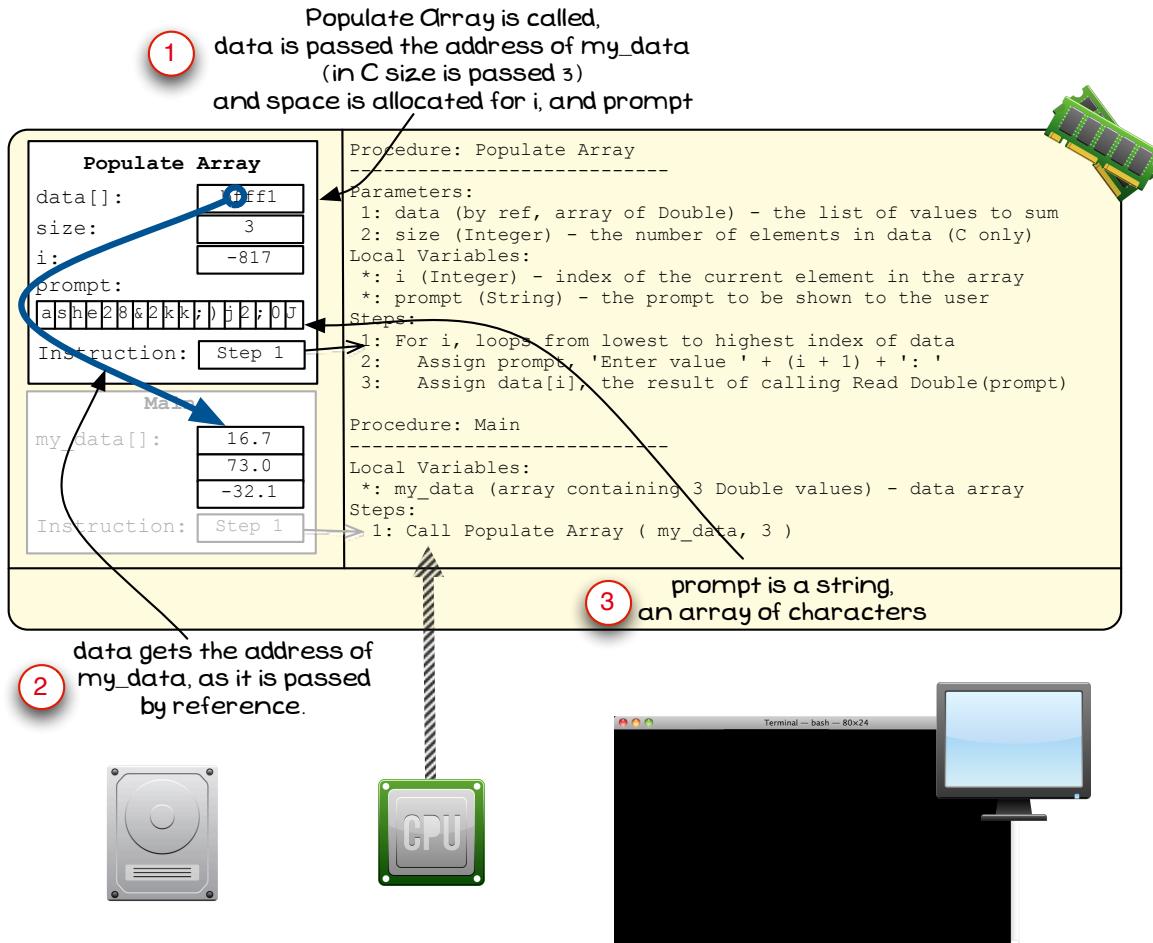


Figure 4.25: Populate array is called, and a reference to the my_data array is pass to its data parameter

Note

- In Figure 4.25 the indicated areas show the following:
 - When `Populate Array` is called it is loaded onto the Stack. Its data parameter receives the address of `my_data` from `Main`. In C the value 3 would also be passed to the size parameter, as C does not keep track of the length of the array for you. At the same time space for `Populate Array`'s local variables `i` and `prompt` are allocated on the Stack.
 - Notice that in `Populate Array` the data parameter only stores the address of the array, as it is passed by reference. This saves time and space, and is needed in this case as the procedure wants to store data into the variable passed to this parameter.
 - The `prompt` local variable is also an array. It is allocated spaces on the stack as is done for all local variables.
- Arrays are allocated a contiguous area of memory to store its elements.
- A String is an array of characters.

Step 1 of Populate Array is run

Step 1 of Populate Array initialises the for loop's control variable (*i* in this case). This variable keeps track of the times the loop body has executed, and can be used to get the *current value* from the array.

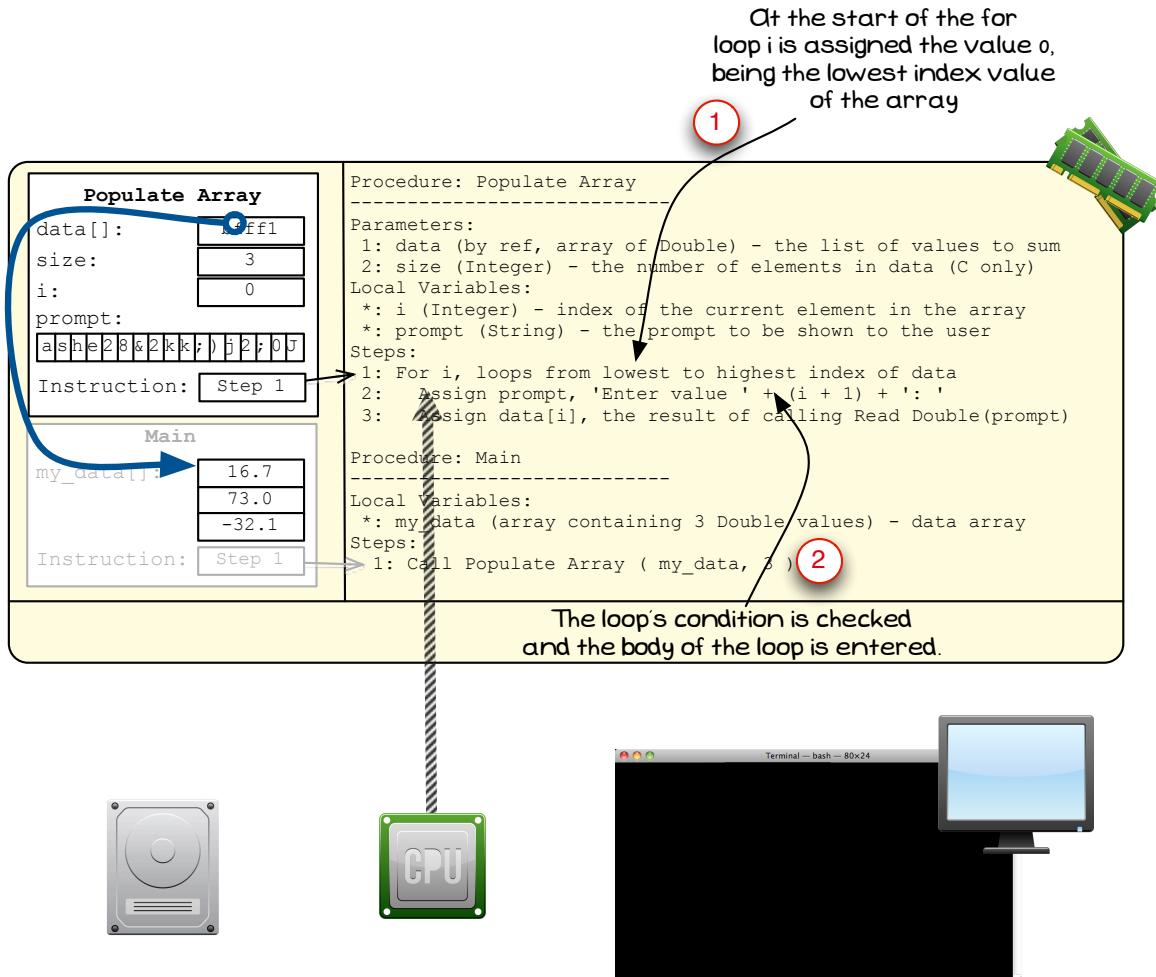


Figure 4.26: Step 1 of Populate Array is called, and the for loop sets *i* to the lowest index of the data array

Note

- In Figure 4.26 the indicated areas show the following:
 - Step 1 of `Populate Array` is a for loop. The first action of the for loop is to initialise the value of *i* to the *lowest index value* of the data array. The lowest index is 0, so *i* is assigned the value 0.
 - Next the loop checks its condition, it is loop from 0 to 2, and has not passed 2 so the body of the loop will be executed. This will be checked again when the for loop ends.
- The first action of a for loop is to initialise the value of the control variable.
- When processing each element of an array the for loop should initialise the control variable to 0, the first index of the array.

Step 2 constructs the prompt to be shown to the user

The user needs to be told what to enter. The prompt is a string that will contain this message so that it can be passed to Read Double. The value for the prompt will use the loop's control variable (the counter) so that the user known which value they are up to.

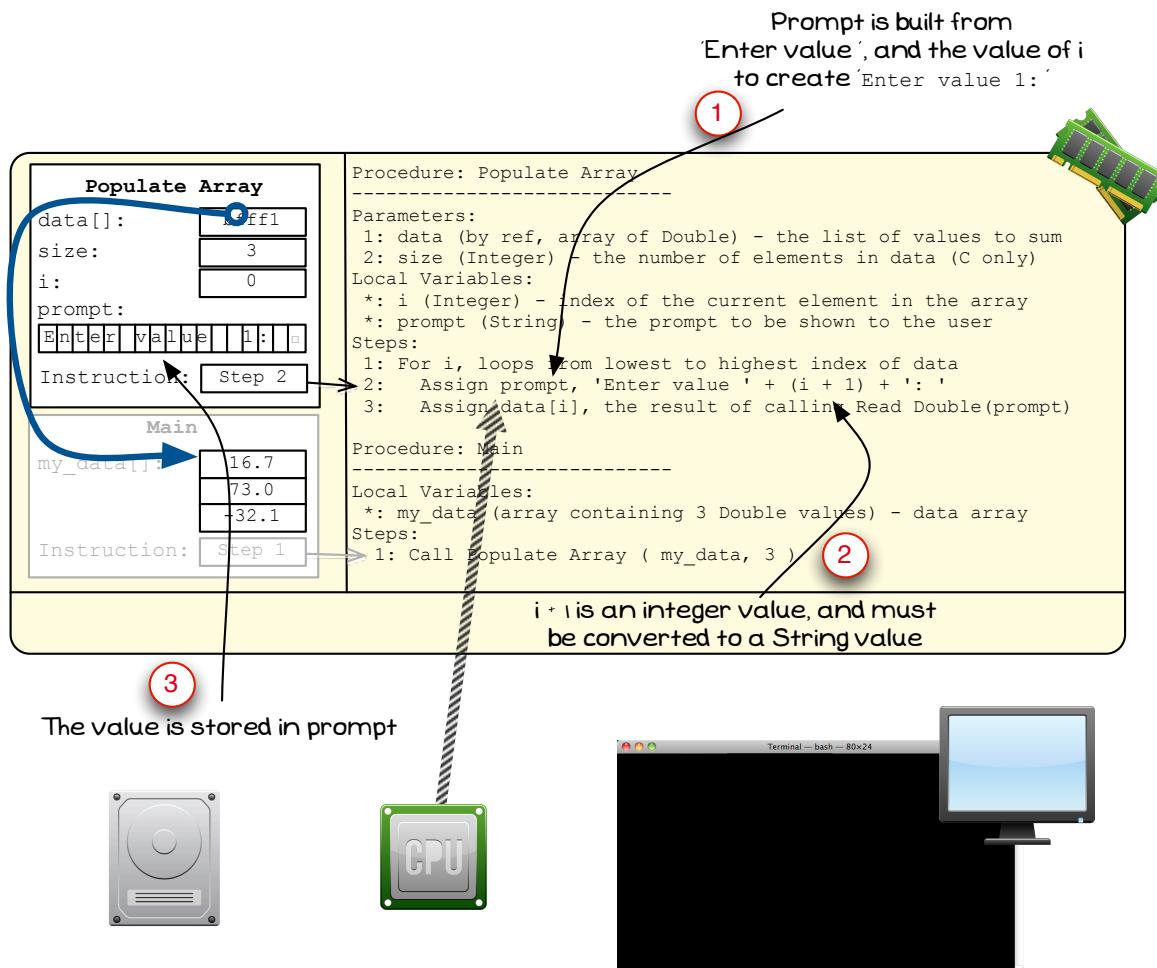


Figure 4.27: Step 2 builds the prompt Enter value 1: which will be shown to the user

Note

- In Figure 4.27 the indicated areas show the following:
 - Step 2 of Populate Array builds the prompt by concatenating 'Enter Value' with $i + 1$ and ':' .
 - To achieve this $i + 1$ must be converted from an Integer to a String.
 - The result is stored in the prompt variable.
- This action will be performed each time the loop executes. In this case the value stored in prompt will be 'Enter value 1: '.
- The small square shown at the end of the prompt represents the overhead. In C this is the *sentinel* value, in Pascal it is the length^a of the array.

^aPascal actually stores the length at the start of the string, but the idea is the same.

Step 3 reads a value from the user and stores it in the array at index 0

The next step calls the Read Double function. This is responsible for reading a value from the user, and returning it to the caller. The value returned is then stored in an element of the array. The *i* variable is read to determine the position where this value should be stored. This means that you can think of *i* as referring to the *current* element of the array.

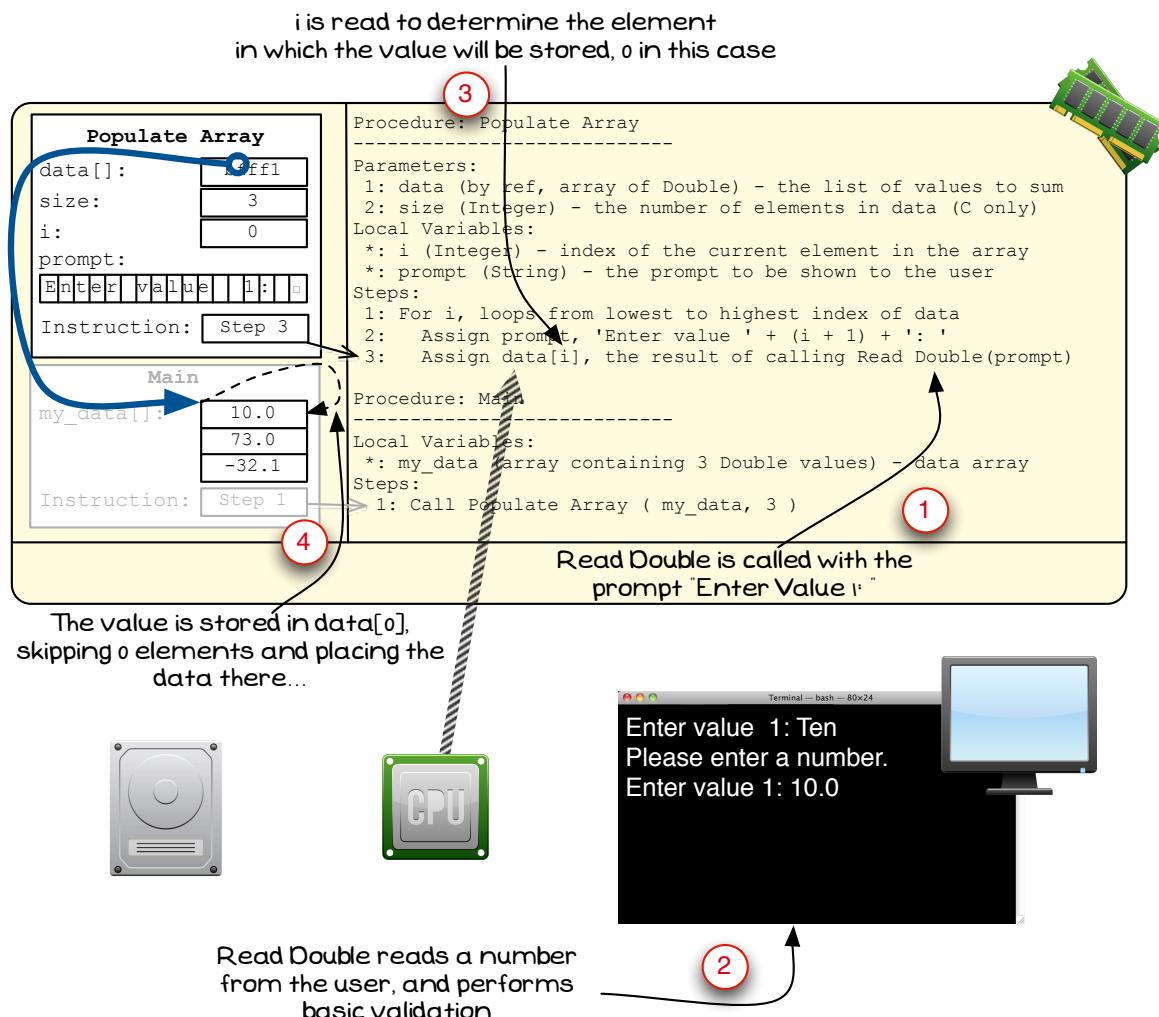


Figure 4.28: Step 3 reads a double value from the user and stores it in `data[i]`

Note

- In Figure 4.28 the indicated areas show the following:
 - Step 3 of `Populate Array` calls `Read Double`, passing across the prompt to be shown to the user.
 - Within `Read Double` the value is read from the Terminal. This includes some validation to make sure that the value entered is a number.
 - To determine where the value is stored the computer needs to evaluate the expression used to index the array. In this case that is the value of the *i* variable.
 - The data reference is followed, and 0 elements skipped, to find the location where the data should be stored.
- Notice that this has stored the value in `my_data`, as `data` is passed by reference.

Control returns to Step 1 as the loop body has ended

At the end of the loop body the for loop performs two actions. It has finished the first pass through the loop, so its control variable (the counter) needs to be incremented to 1. Then it needs to jump back to check its condition. This will determine if the loop's body is executed again or skipped. In this case i is still in the defined range so the loop's body is run again.

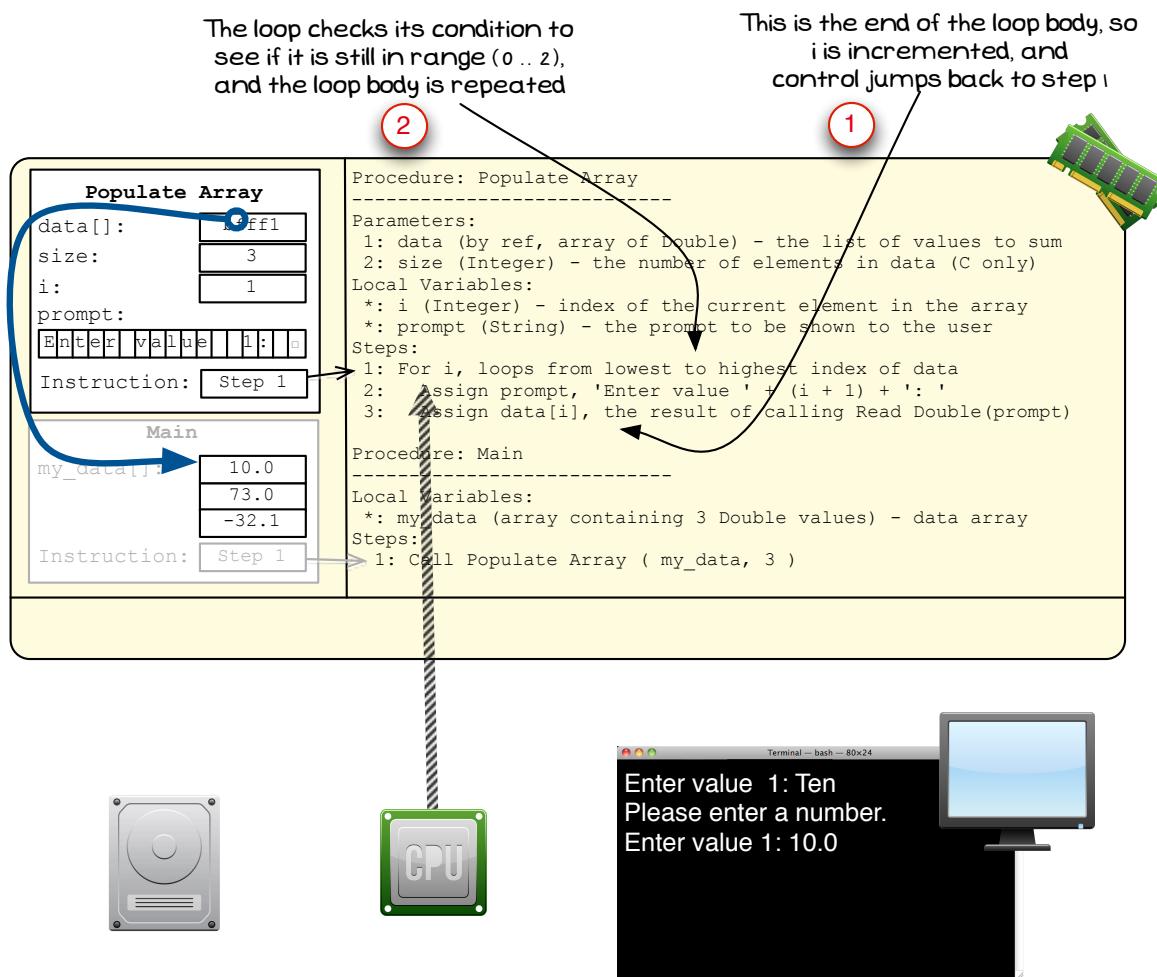


Figure 4.29: At the end of the loop body i is incremented and control jumps back to check the loop's condition

Note

- In Figure 4.29 the indicated areas show the following:
 - The end of the loop body indicates that two things needs to occur. Firstly the value of i needs to be incremented, and then control needs to jump back to check the condition of the loop (step 1).
 - The loop's condition checks if i is still in range (i is in the range $0..2$, coded as $i < size$ in C). As it is still in range the loop's body will execute again.

Second prompt is built asking the user to enter value 2

Back at step 2 again, the prompt needs to be recreated. This time its message will be 'Enter value 2: '. The process to create this is the same, with the value of $i + 1$ being converted to a String, and the three parts concatenated together and stored in `prompt`. This overrides the details in the existing array, reusing the same memory to store these values.

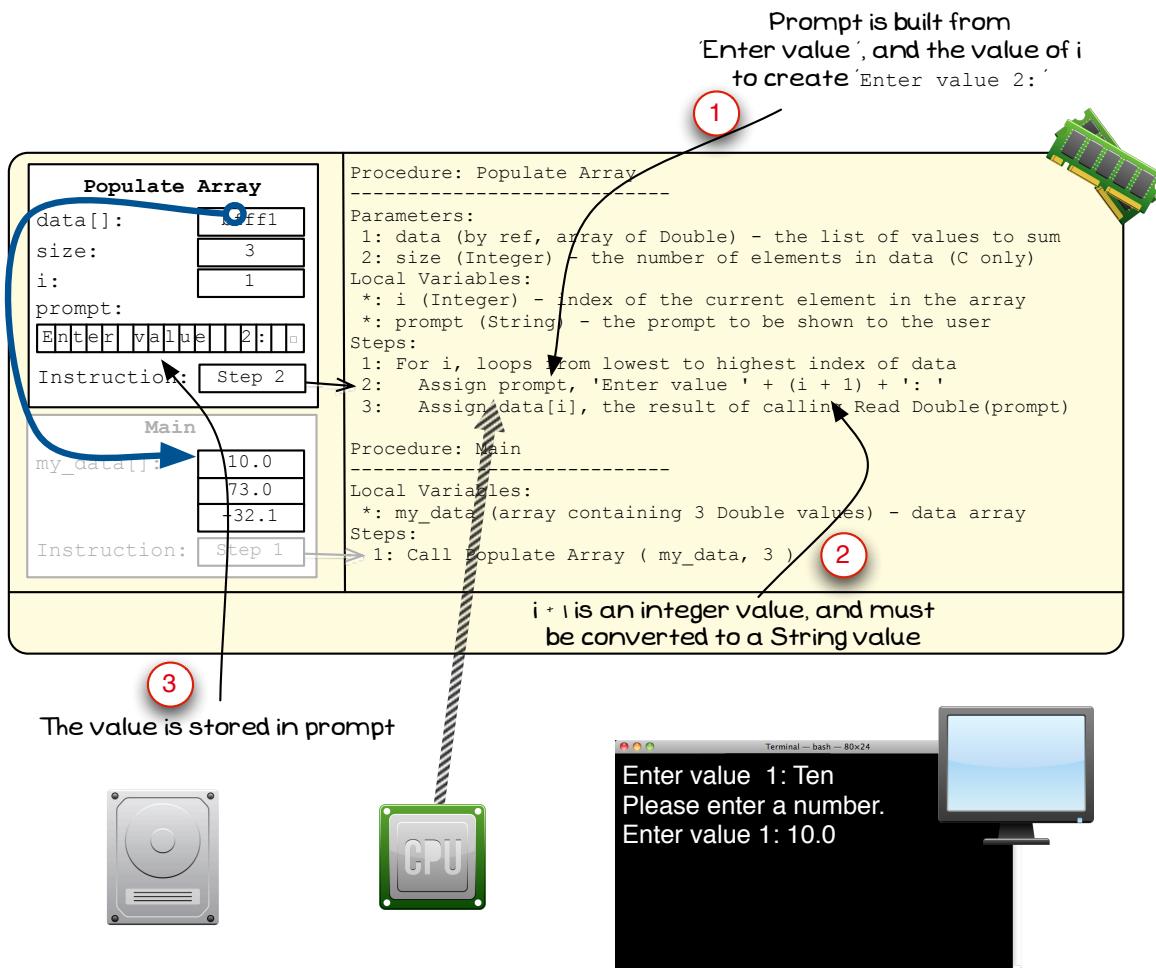


Figure 4.30: Step 2 builds the prompt `Enter value 2:` which will be shown to the user

Note

- In Figure 4.30 the indicated areas show the following:
 - Step 2 of `Populate Array` builds the prompt by concatenating 'Enter Value' with $i + 1$ and ': '.
 - To achieve this $i + 1$ must be converted from an Integer to a String.
 - The result is stored in the `prompt` variable.
- Notice that this is using the same location to store its data.
- The small square shown at the end of the `prompt` represents the overhead. In C this is the *sentinel* value, in Pascal it is the length^a of the array.

^aPascal actually stores the length at the start of the string, but the idea is the same.

Populate Array stores the second value read into data[1]

Step 3 uses Read Double again to get the value to store in the second element of the array. To find where this should be stored the computer calculates the value of the index, reading this from the *i* variable. The value returned from Read Double is then stored in the array referred to by *data*, at index 1.

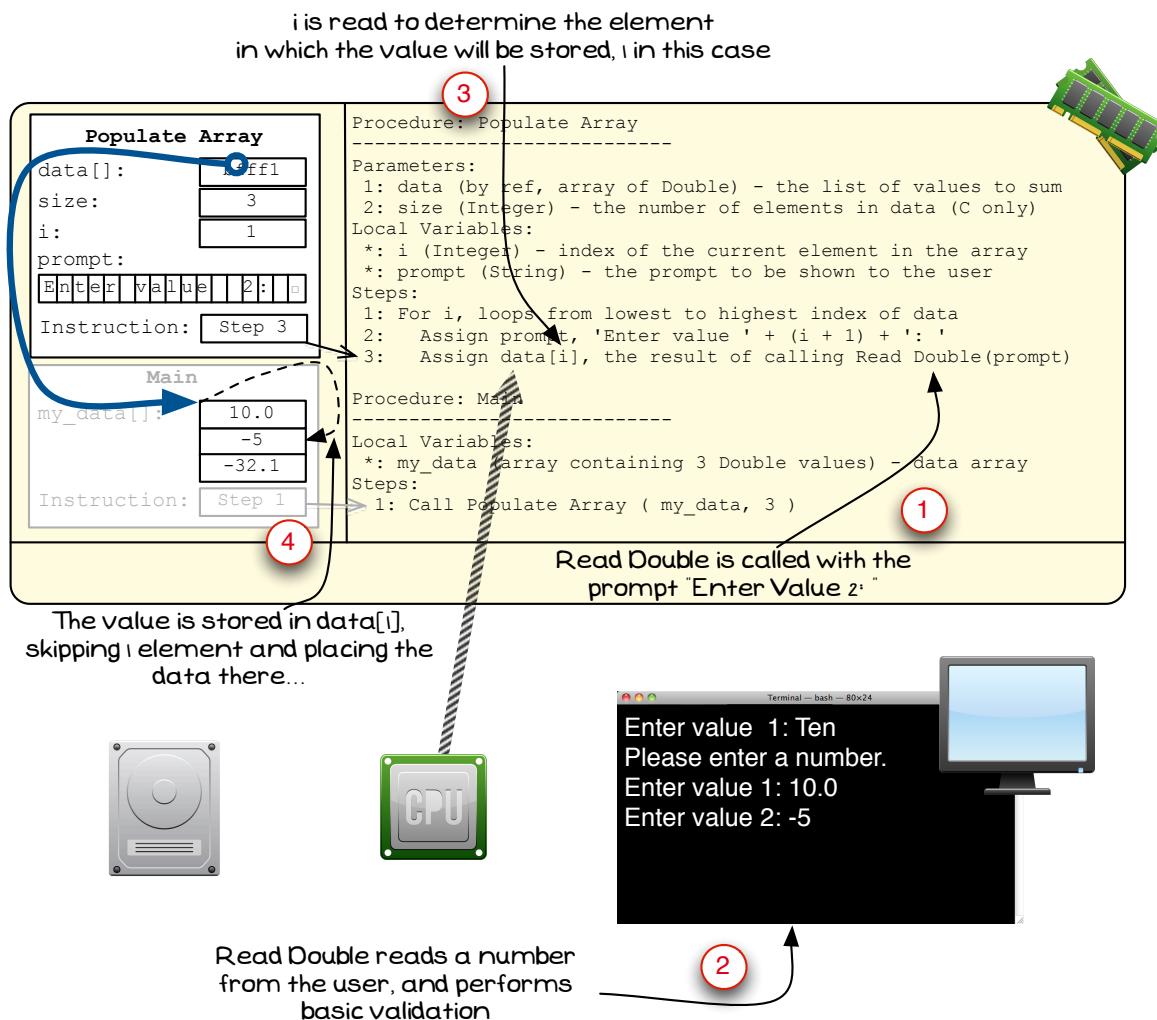


Figure 4.31: The second value read is stored in *data[1]*

Note

- In Figure 4.31 the indicated areas show the following:
 - Step 3 of *Populate Array* calls *Read Double*, passing across the prompt to be shown to the user.
 - Within *Read Double* the value is read from the Terminal. This includes some validation to make sure that the value entered is a number.
 - The value of the index is now 1, as read from variable *i*.
 - The data reference is followed, and 1 element is skipped, to find the location where the data should be stored.

i is incremented again, and control returns to Step 1 to determine if loop runs again

The end of the for loop's body has been reached again, so it performs two actions: it increments its control variable (*i* in this case) and jumps back to check its condition (step 1). The condition then determines if the loop's body is to be executed again or skipped. In this case *i* is still in the defined range so the loop's body is run a third time.

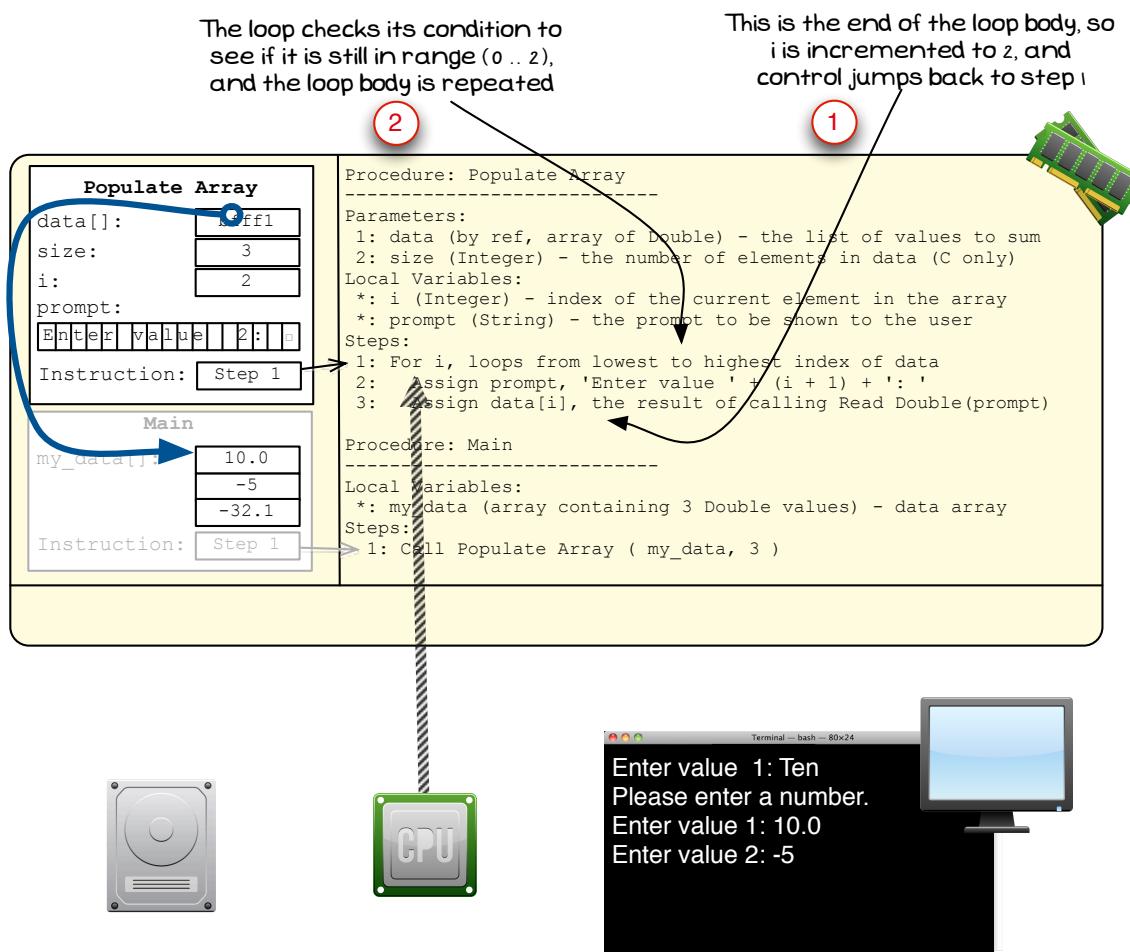


Figure 4.32: At the end of the loop body *i* is incremented and control jumps back to check the loop's condition

Note

- In Figure 4.32 the indicated areas show the following:
 - The end of the loop body indicates that two things needs to occur: the value of *i* is incremented, and control jump back to check the condition of the loop.
 - The loop's condition checks if *i* is still in range (*i* is in the range 0..2, coded as $i < \text{size}$ in C). As it is still in range the loop's body is execute again.
- These same actions always occur at the end of the for loop. It increments its control variable, and jumps back to check its condition.

C++

The C for loop can be used for more than just counting. At the end of the loop the instructions from the **C For Loop's increment** occur, typically something like *i++*.

Third prompt is built asking the user to enter value 3

Back at step 2 for the third time. This step recreates the prompt, this time with the message 'Enter value 3: '. The process to create this is the same as before, with the value of $i + 1$ being converted to a String, and the three parts concatenated together and stored in `prompt`. Remember that this overrides the data currently in the `prompt` array.

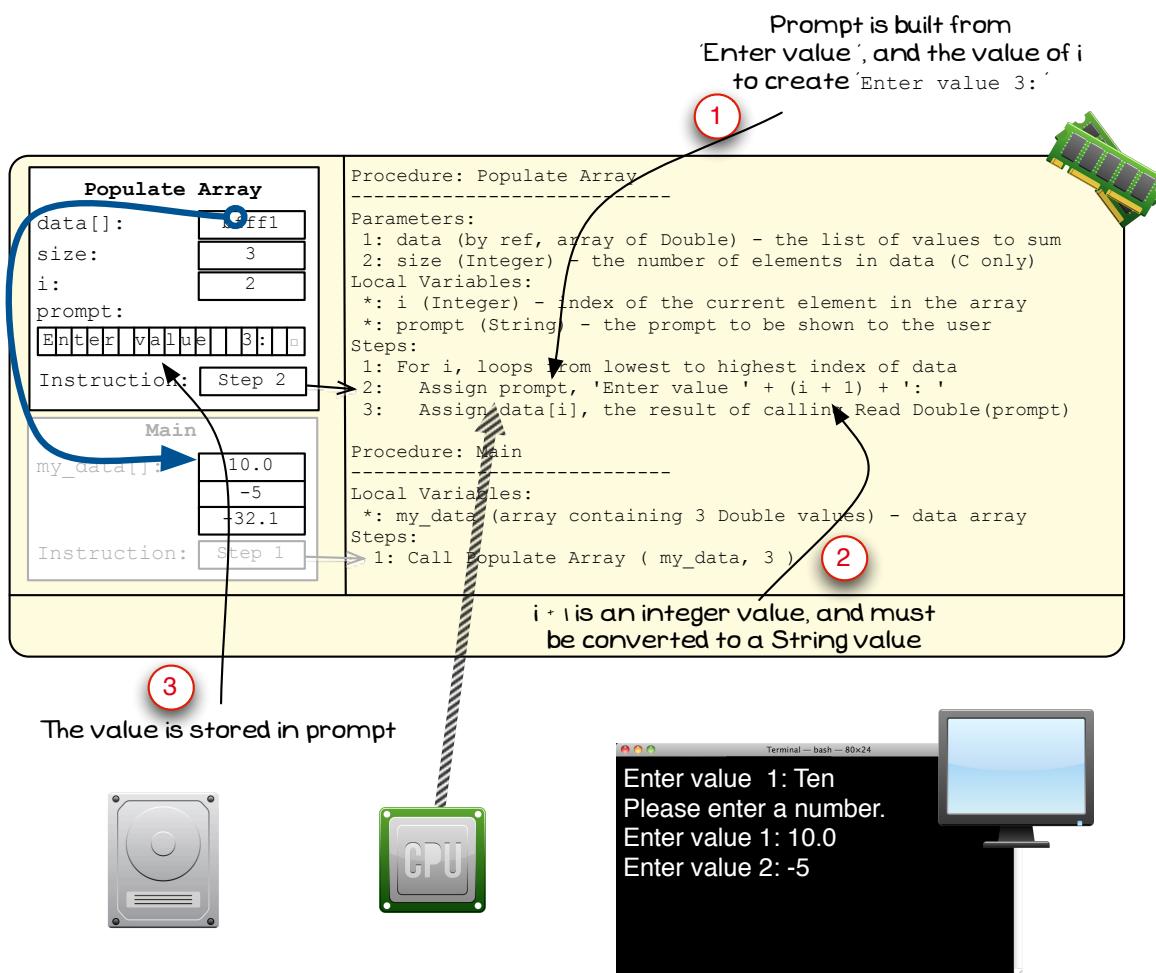


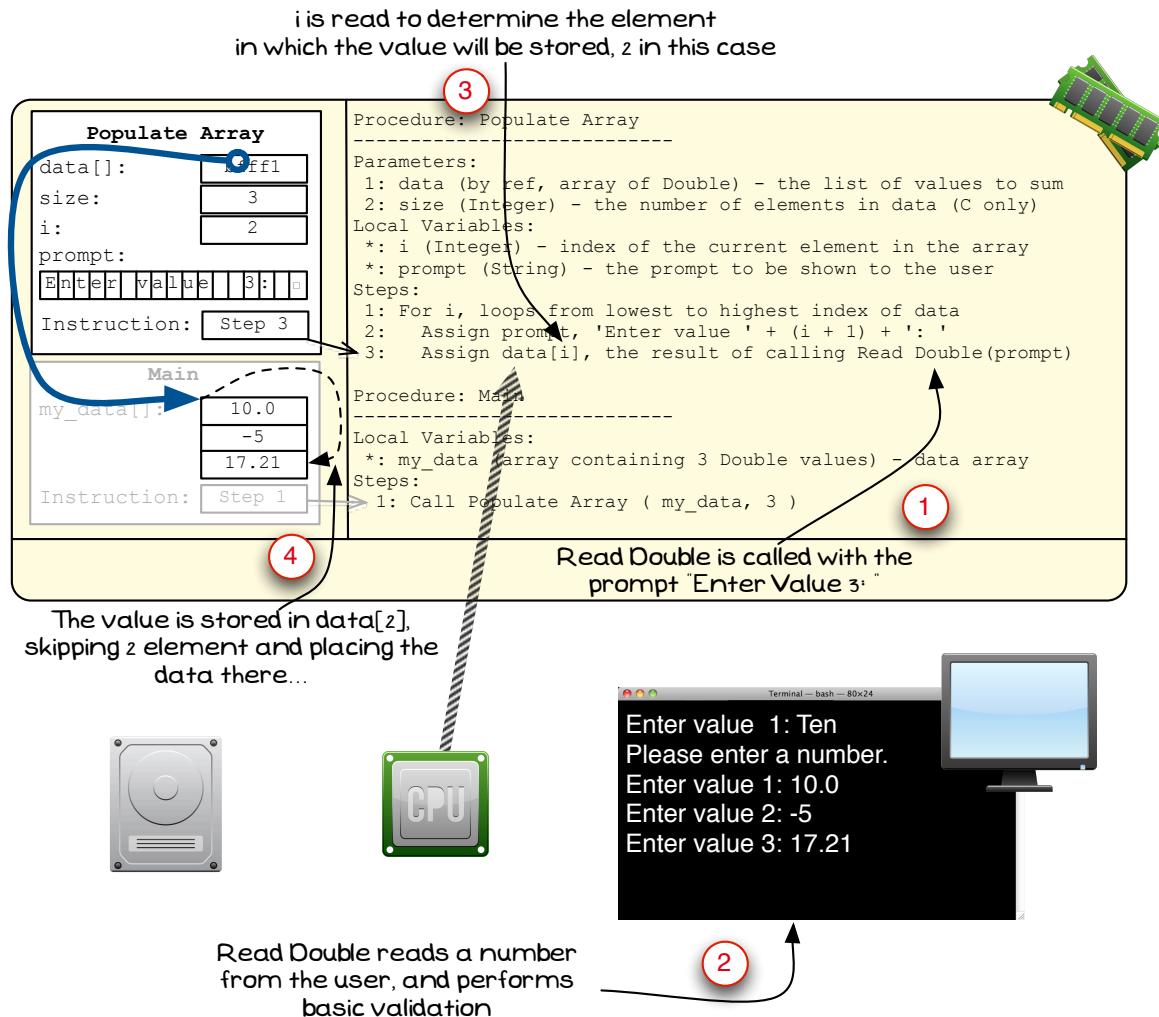
Figure 4.33: Step 2 builds the prompt `Enter value 3:` which will be shown to the user

Note

- In Figure 4.33 the indicated areas show the following:
 - Step 2 of `Populate Array` builds the prompt by concatenating 'Enter Value' with $i + 1$ and ': '.
 - To achieve this $i + 1$ must be converted from an Integer to a String.
 - The result is stored in the `prompt` variable.

Populate Array stores the third value read into data[2]

Once again, step 3 uses Read Double get the value to store in the array. In this case i indicates that this should be stored in the element at index 2 (skipping 2 elements, so storing the value in the third).

**Figure 4.34:** The second value read is stored in `data[1]`**Note**

- In Figure 4.34 the indicated areas show the following:
 - Step 3 of `Populate Array` calls `Read Double` again.
 - Within `Read Double` the value is read from the Terminal.
 - The value of the index is now 2, as read from variable `i`.
 - The data reference is followed, and 2 elements are skipped to find the location where the data should be stored.
- Notice how over these three loops `i` has been used to determine which element is the *current element*. The loop's processing is the same, but changing `i` means that this action is applied to each element in the array one at a time.
- The loop's body determines the action to apply to the i^{th} element of the array.
- The for loop then updates `i` so that the body is applied to *each* element of the array.

i is incremented again, and control jumps back to check the condition a fourth time

The end of the for loop's body has been reached again, so it performs two actions: it increments its control variable (i in this case) and jumps back to check its condition (step 1). This time the value of i is outside the range of the indexes for data (0..2, it is no longer less than 3). This means that the loop's body should not be run again, and control will jump past the body to the next step. As there are no more steps in Populate Array it will end.

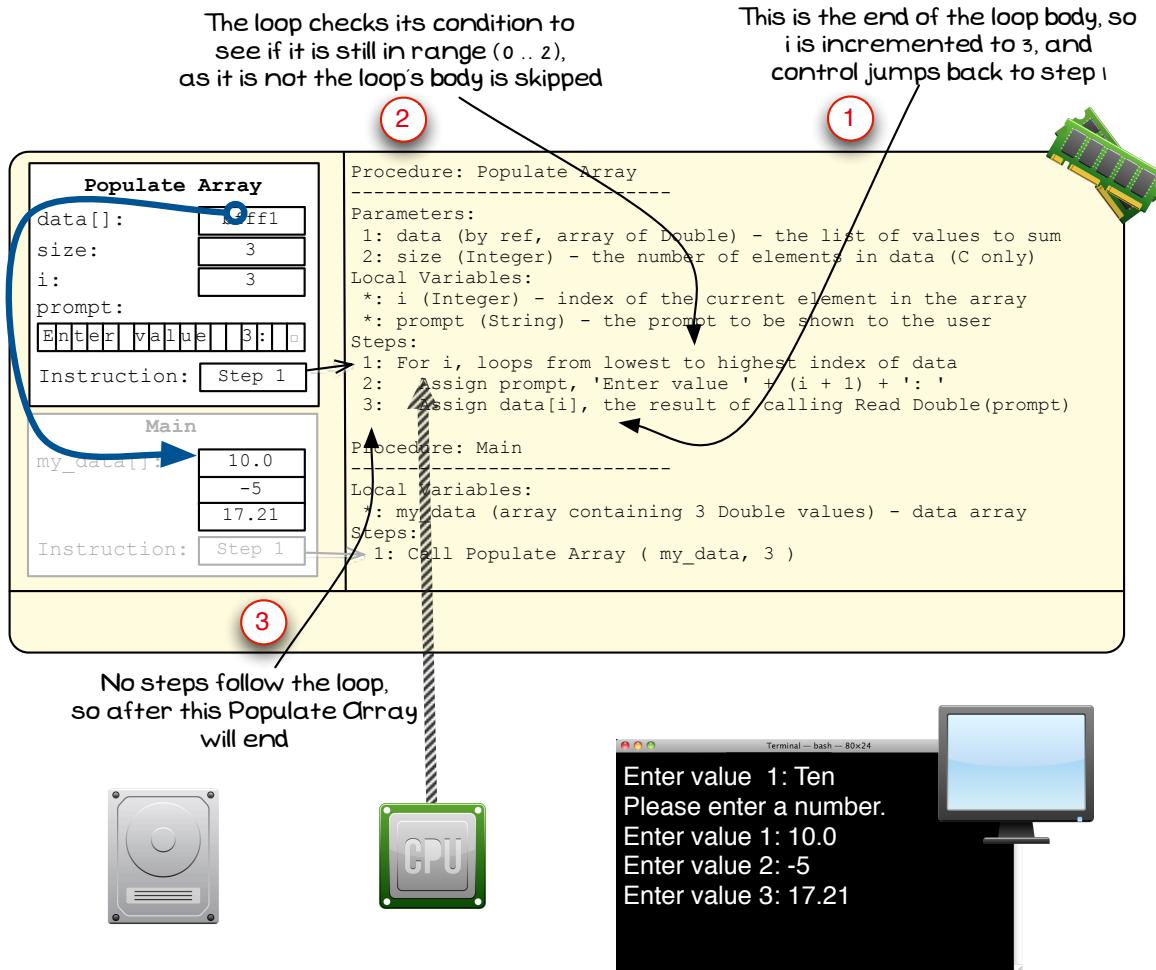


Figure 4.35: At the end of the loop body *i* is incremented and control jumps back to check the loop's condition

Note

- In Figure 4.35 the indicated areas show the following:
 - The end of the loop body causes the value of *i* to be incremented, and control jump back to check the condition of the loop.
 - The loop's condition checks if *i* is still in range (*i* is in the range 0..2, coded as *i < size* in C).
 - i* is **not** in the range 0..2, so the loop body is skipped. As there are no more steps in this procedure it ends.
- The condition in the for loop is responsible for determining if the loop runs again. This time the condition indicates that the loop is not to be run again, so control will jump past the end of the loop.
- The for loop works like a while loop, with additional logic to keep a counter.

Populate Array ends, and has populated Main's my_data array

When Populate Array ends its space on the stack is released so that it can be used again, and control returns to Main. Populate Array was responsible for reading values from the user and storing these in the array passed to it, and if you look at Figure 4.36 you can see that this has been achieved.

The instructions in Populate Array commanded the computer to read a value from the user and store it in the current element (the i^{th} element) of the array. These actions were then repeated by the **For Loop** for each index of the array. Together the for loop and its body allow you to define actions that must be performed on all elements in an array.

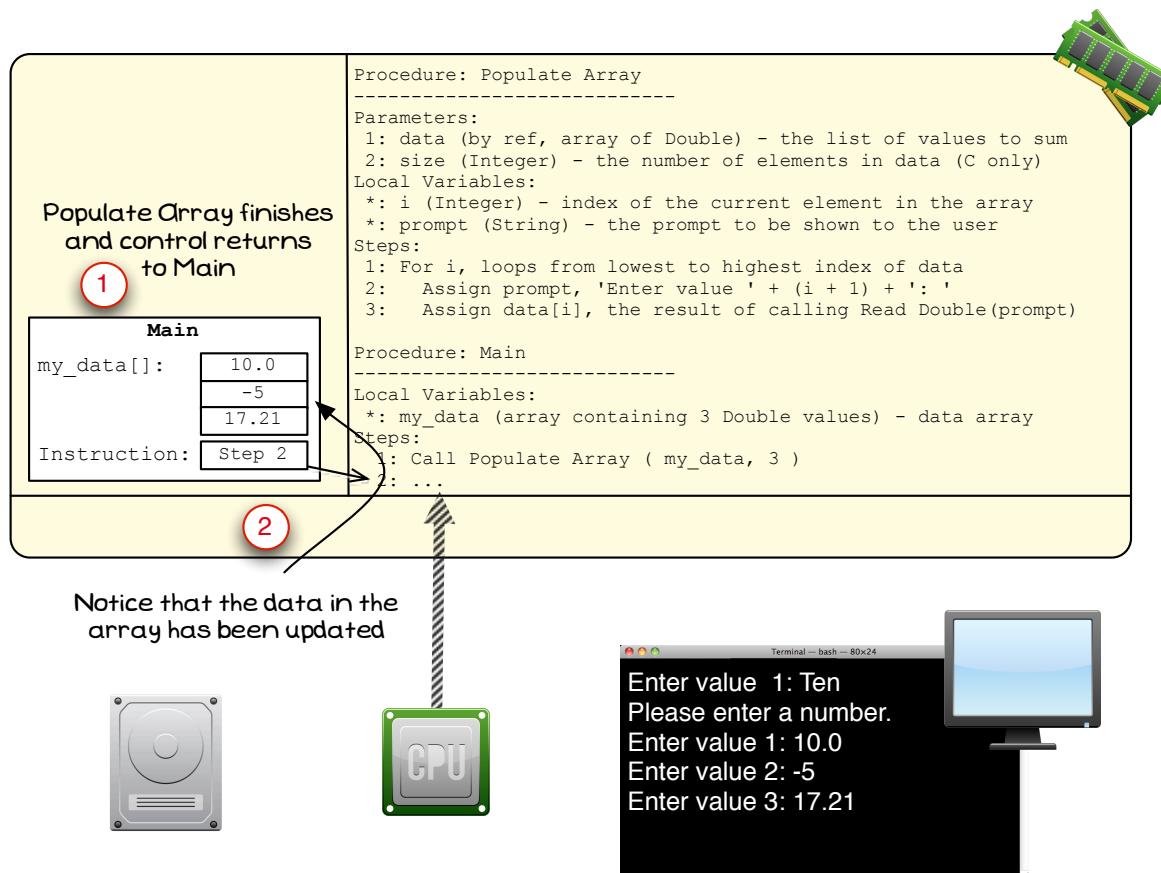


Figure 4.36: Control returns to Main, and its my_data array has been populated

Note

- In Figure 4.36 the indicated areas show the following:
 - The last step in Populate Array has been completed, so it ends and control returns to Main.
 - The values in my_data have been updated. Passing this array by reference to Populate Array allowed it to make changes to this array's values.
- Notice that the data associated with the Populate Array procedure has been released, including the data used for the prompt array.
- Populate Array is responsible for filling the array passed to it with values from the user, and that is what it has done.

4.5.2 Understanding Sum

Figure 4.37 shows the flowchart of the `Sum` function from the Statistics Calculator program. This algorithm was developed in the Section 4.2.2, and its Pseudocode is shown in Listing 4.1. The `Sum` function is responsible for calculating the sum of all of the values in the array passed to it. This is achieved by having a total variable that is initialised to 0, and then has the value of *each element* from the array added to it.

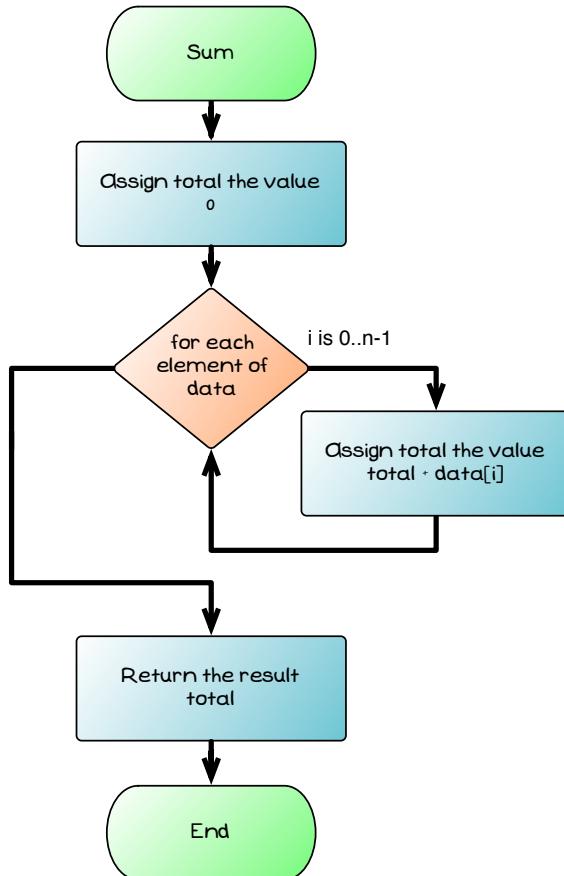


Figure 4.37: Flowchart showing the process for `Sum`

The following code will show how this function is executed on an array with three values in it. This will continue the execution from Section 4.5.1 Understanding Populate Array, though the same process would occur for any array values.

Note

The most important thing to pay attention to in this illustration is the interactions between the `For Loop` and the `Array`. Make sure you can see how this combination allows you to specify the actions to be performed on an element (in the loop's body), and then to have this run for each element of the array (controlled by the loop condition).

Sum is called, and passed the my_data array

When `Sum` is called it is passed the array to read its values from. This is passed in the same way as was done in `Populate Array`. The difference here is that this is passed using a `const` reference, to indicate that `Sum` is not allowed to change the data in the array. This means that `Sum` will not compile if you update values in this array, and provide a guarantee to the caller that their data will not change when given to the `Sum` function.

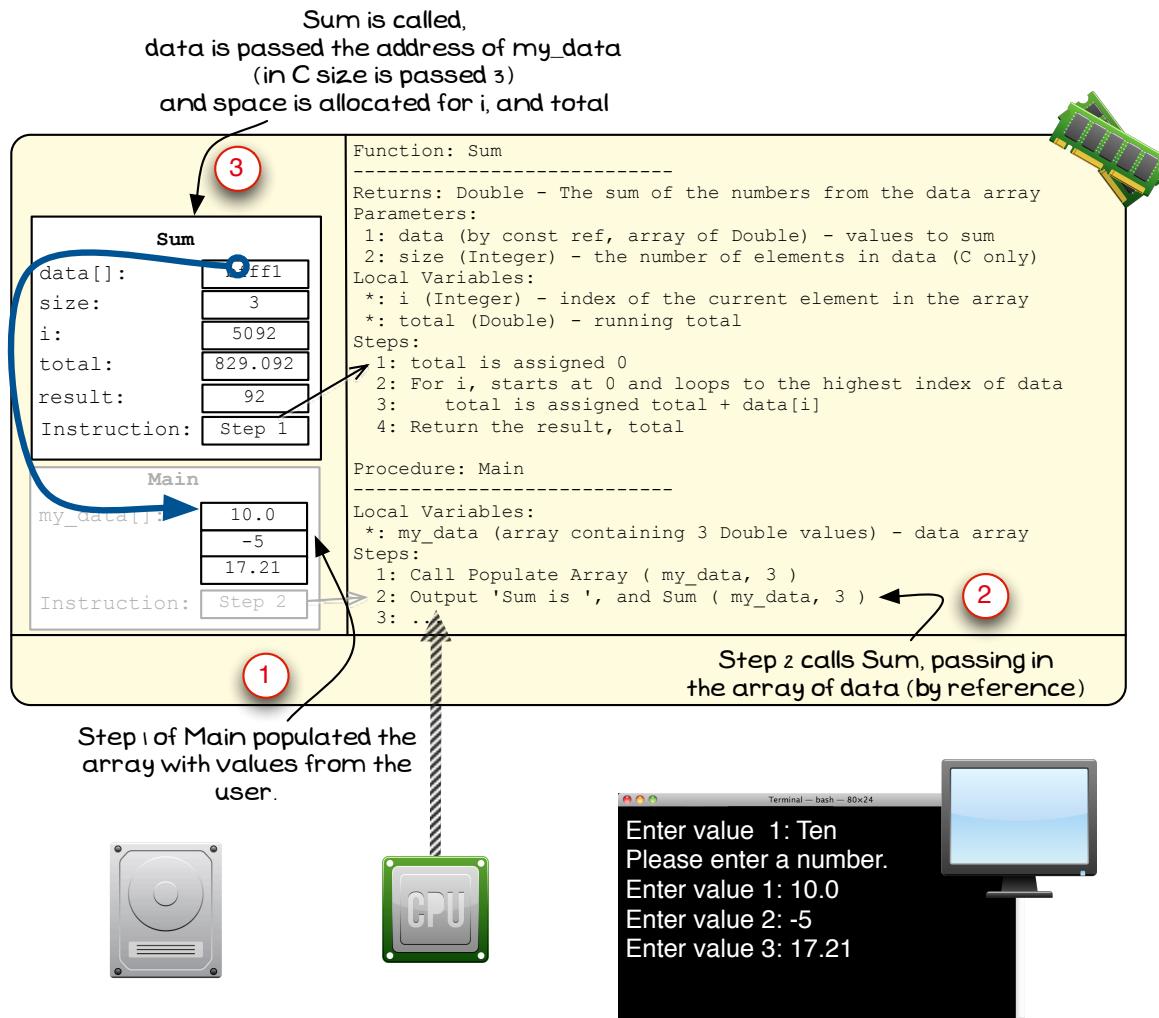


Figure 4.38: Sum is called, and it is passed the array to get its values from

Note

- In Figure 4.38 the indicated areas show the following:
 - Step 1 of Main has populated the array with values.
 - Step 2 calls the `Sum` function, passing in the `my_data` array.
 - When `sum` starts it is allocated space on the heap. Its `data` parameter is passed the address of `my_data`. In C the `size` parameter will be passed the value 3, this isn't needed in Pascal as the language takes care of these details for you. Space is also allocated for the local variables `i` and `total`.
- Passing `my_data` by reference means that `sum` gets a reference that *points* to the start of the array.
- `Sum` is responsible for calculating the sum value of the values in the array.

total is initialised to 0

The first action in `Sum` is to set the value of `total` to 0. `total` will be used to store the running total of the array, and it must start at 0. Remember that the space on the stack was used before, and therefore these variables get seemingly random values initially. It is important to remember to always initialise the variables you are using.

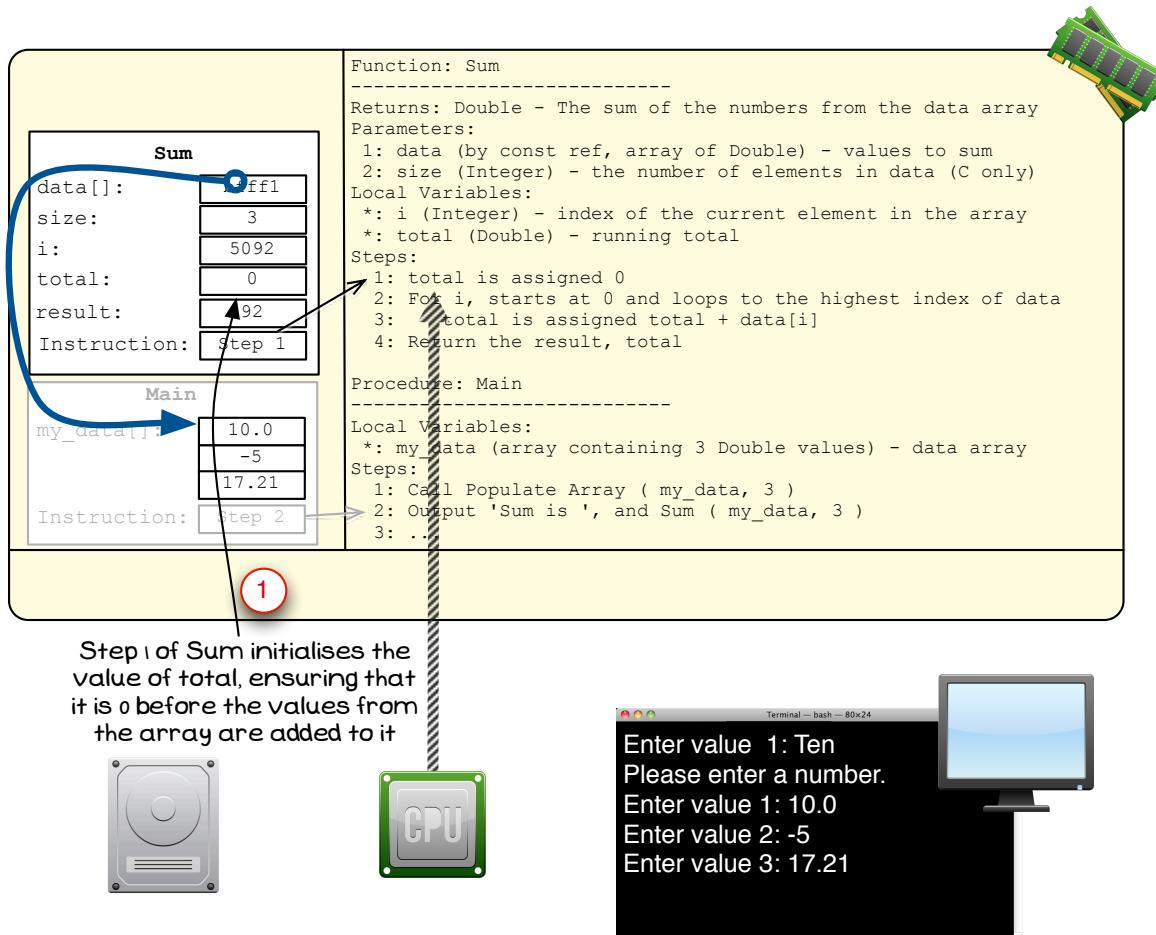


Figure 4.39: Total is initialised, having its value set to 0

Note

- In Figure 4.39 the indicated areas show the following:
 - The total variable is assigned the value 0.
- Local variables do not get a default value when the Function or Procedure starts. You need to make sure you initialise these to appropriate values before you use them.
- total will keep a running total of the elements in the array, it must start at the value 0.

For loop initialises i

Step 2 of Sum starts the for loop that will iterate over the elements of the data array. The for loop's control variable, *i*, is set to the first index of the array and the condition checks if the loop's body should run. As *i* is in the range $0..2$, it is less than 3, control will jump into the loop's body making step 3 the next instruction.

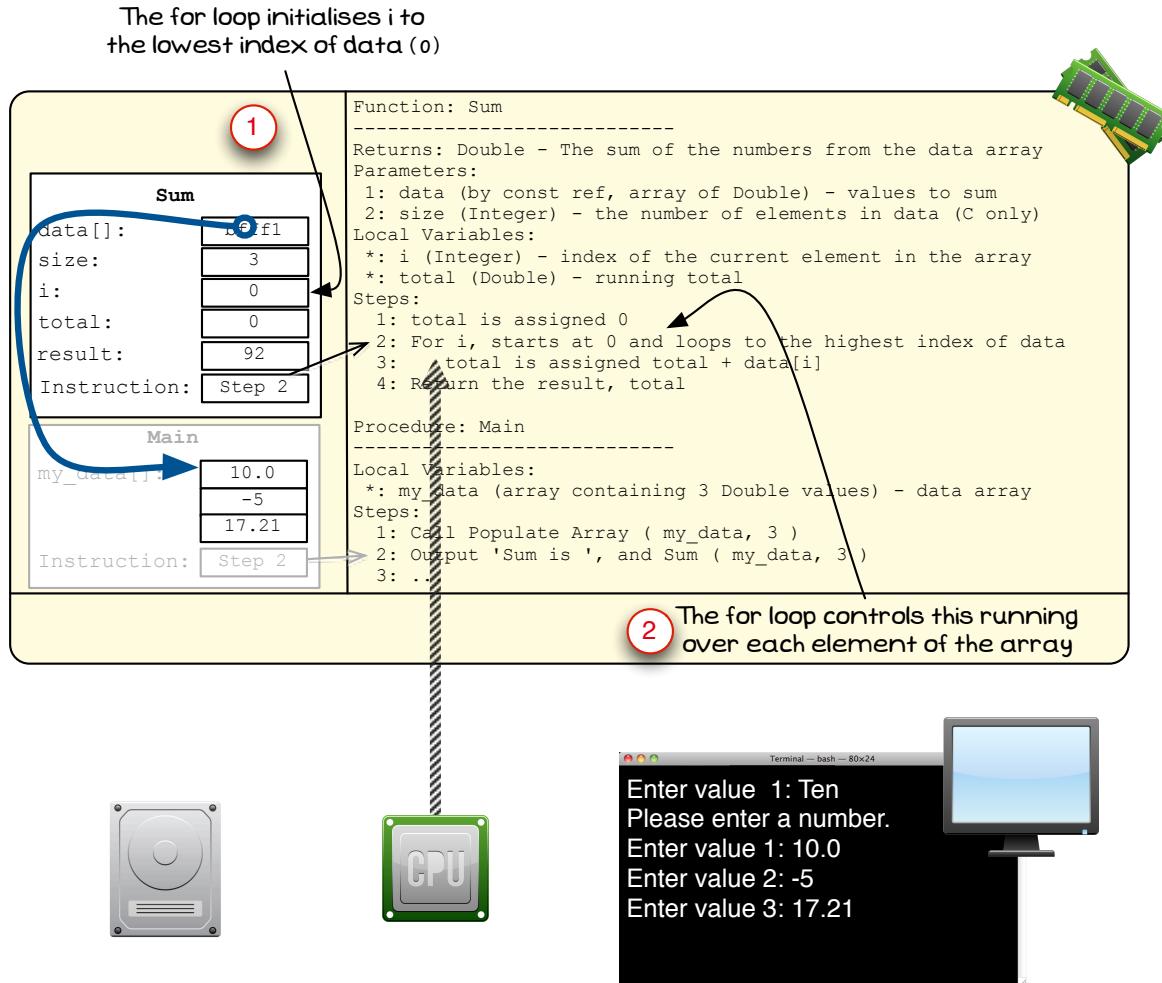


Figure 4.40: *i* is initialised by the for loop, and control jumps to the loop's body

Note

- In Figure 4.40 the indicated areas show the following:
 - The for loop initialises its control variable *i*, assigning it the value 0.
 - The condition of the loop is checked to see if the body should execute. As *i* is still in the range $0..2$, it is less than 3, the loop's body will execute.
- The first action of a for loop is always to initialise its control variable. After this it checks if the body of the loop should run.

total is increased by the value in data[0]

The body of the for loop reads the *current* value from the data array, `data[i]`. As `i` is currently 0 this reads the first element of the array. This reads the value 10.0, which is added to the current total, 0. The resulting value is then stored back into the total variable, giving it the value 10.0, calculated from the expression `total + data[0]`, which is $0 + 10.0$.

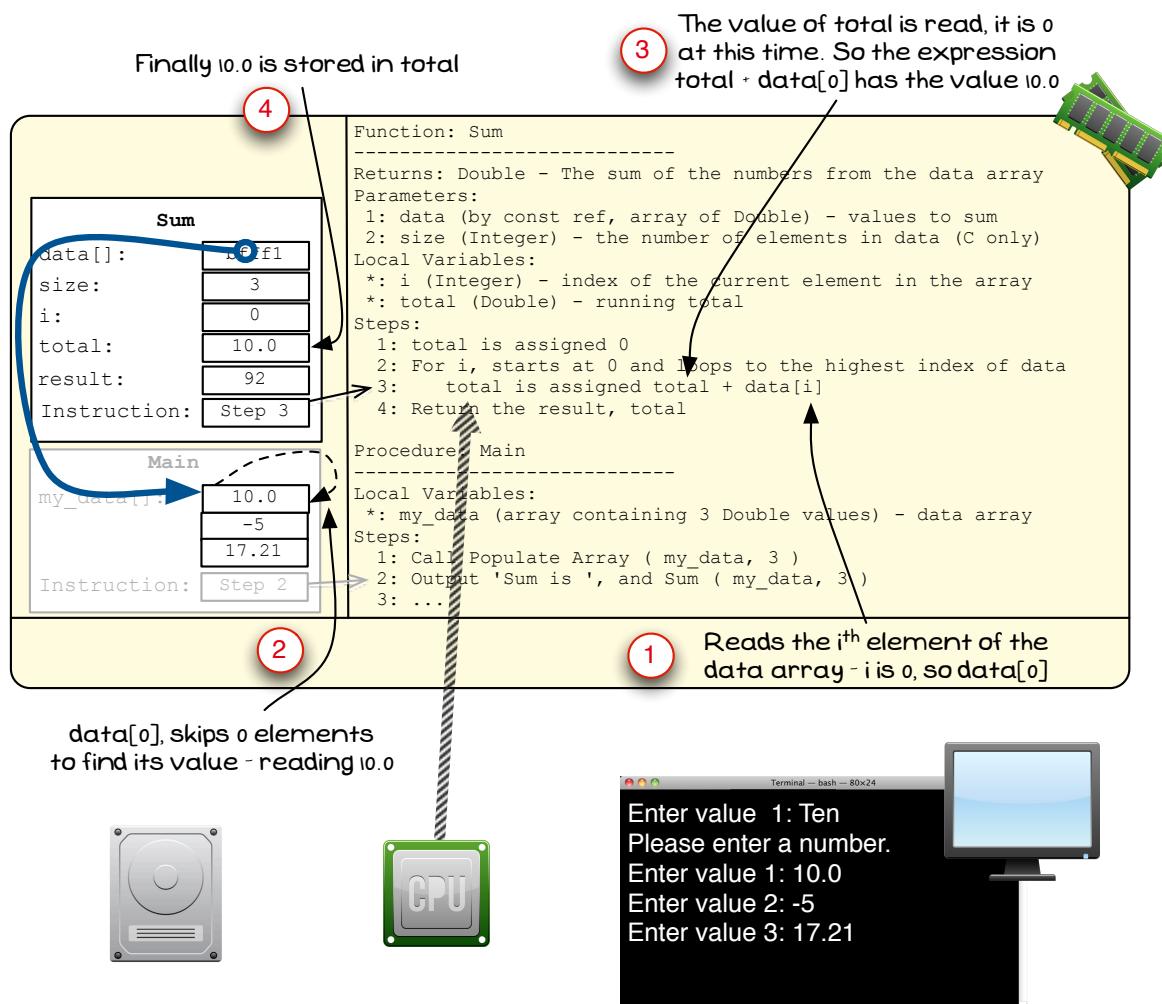


Figure 4.41: Total is increased by the value in `data[0]`

Note

- In Figure 4.41 the indicated areas show the following:
 - The value of `data[i]` must be read in this expression. At this point `i` is 0, so `data[0]` must be read.
 - `data[0]` is found in the array referred to by `data`, after skipping 0 elements. This reads the value of the first array element.
 - The expression evaluates `totala + data[i]`, giving `total + data[0]`, which is $0 + 10.0$, with the final value being 10.0.
 - The value 10.0, calculated above, is then stored in `total`.
- The body of the for loop provides the instructions that are run for each element of the array.

^aRemember that at this point `total` has not been changed, so it has the value 0.0 as shown in Figure 4.40.

For loop increases the value of i and runs the loop body a second time

At the end of the for loop it increments the value of its control variable, assigning i the value 1, and then jumps back to check its condition. As i is still in the range 0..2 the loop body will be run again, making step 3 the next action.

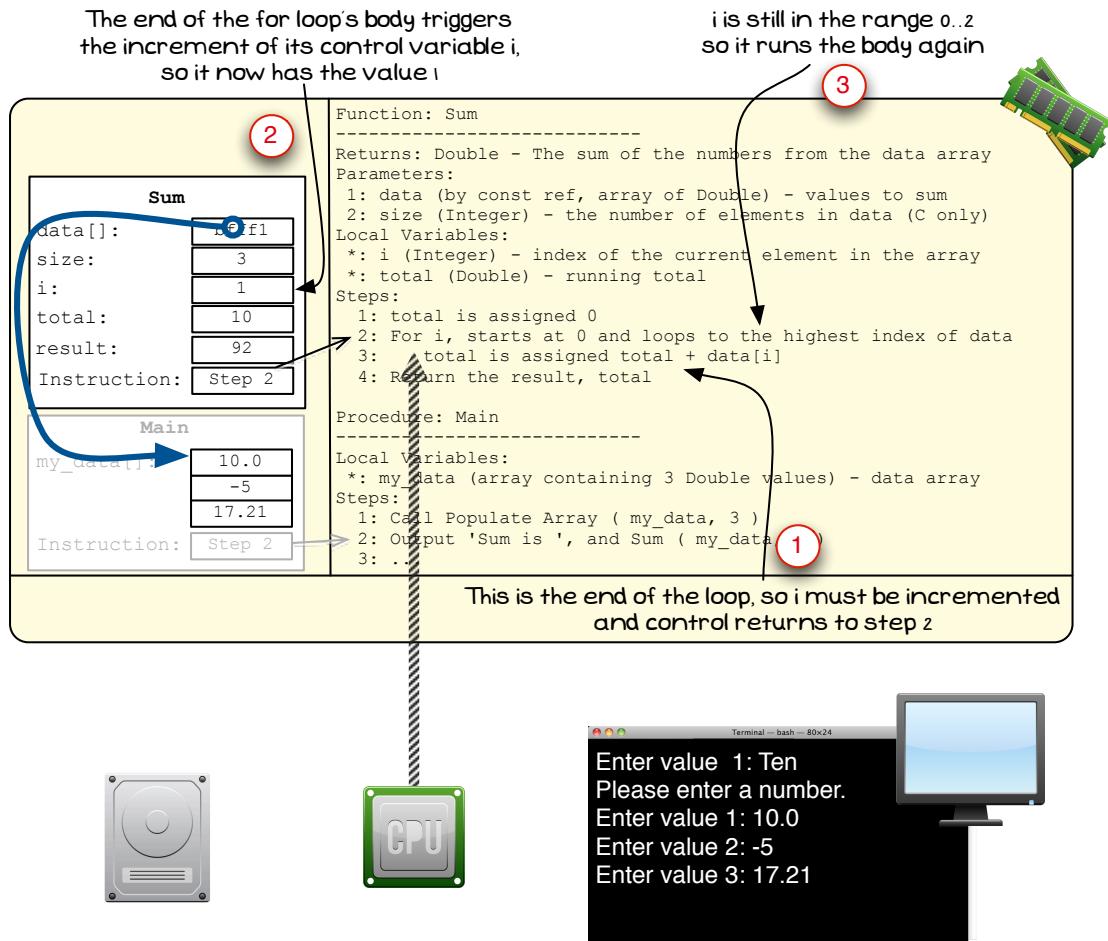


Figure 4.42: At the end of the loop body i is incremented, and control jumps back to check the condition

Note

- In Figure 4.42 the indicated areas show the following:
 - The end of the loop has been reached. The for loop increments its control variable, and jumps back to check its condition.
 - In this case i is the control variable, so its value is increased to 1.
 - The condition of the loop is checked to see if the body should execute. As i is still in the range 0..2 (it is less than 3) the loop's body will execute, making step 3 the next action.
- The end of each for loop always performs these steps. Increasing the value of its control variable, and jumping back to check its condition.

C++

Remember C can do more than just counting in the for loop, though this is its main use. ♦

The value of total is increased by the value in data[1]

The body of the for loop reads the *current* value from the data array, $\text{data}[i]$. Now i has the value 1 it reads the second element of the array. This reads the value -5, which is added to the current total, 10.0. The resulting value is then stored back into the total variable, giving it the value 5.0, calculated from the expression $\text{total} + \text{data}[1]$, which is $10.0 + -5.0$.

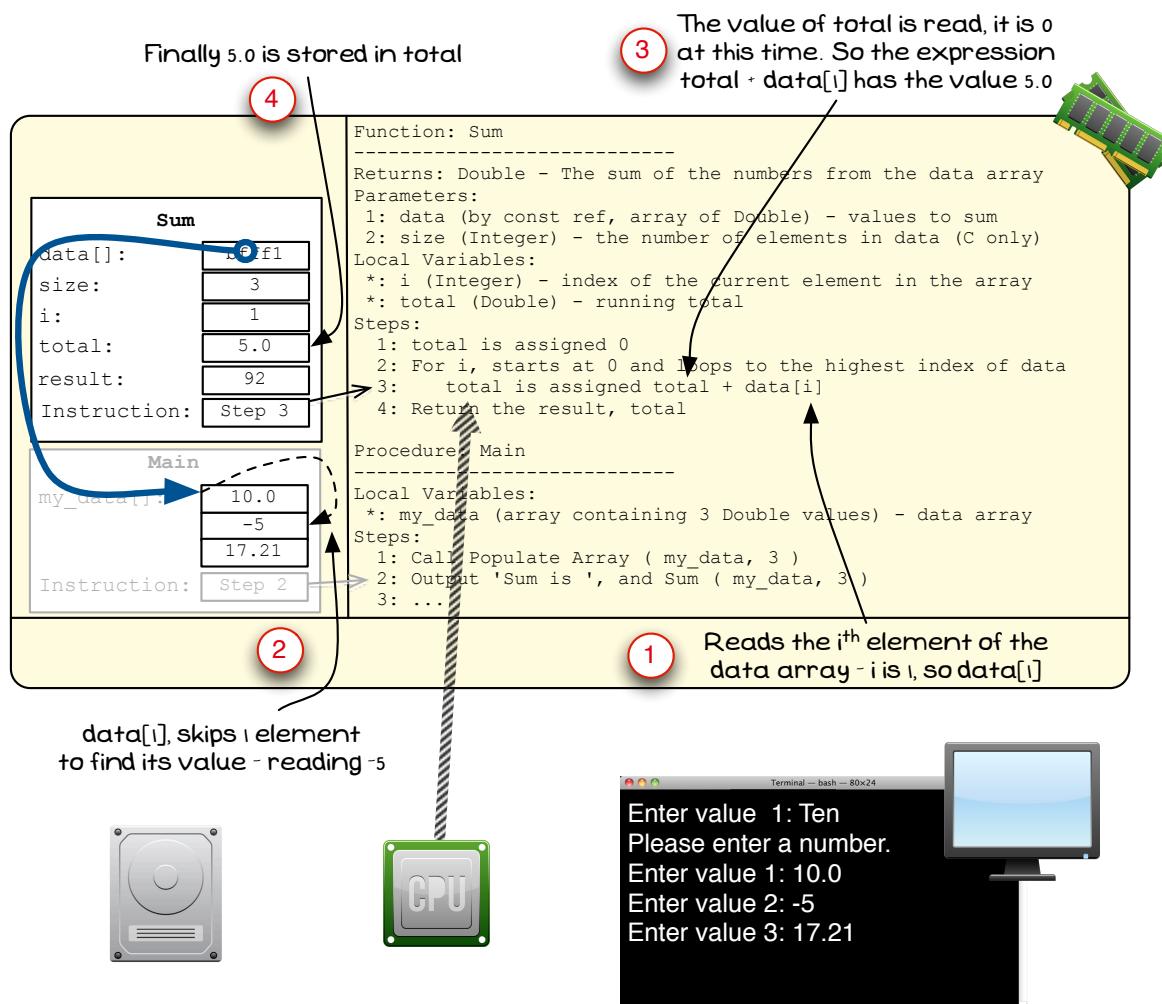


Figure 4.43: Total is increased by the value in $\text{data}[1]$

Note

- In Figure 4.43 the indicated areas show the following:
 - The value of $\text{data}[i]$ must be read in this expression. At this point i is 1, so $\text{data}[1]$ must be read.
 - $\text{data}[1]$ is found in the array referred to by data , after skipping 1 element. This reads the value of the second array element.
 - The expression evaluates $\text{total}^a + \text{data}[i]$, giving $\text{total} + \text{data}[1]$, which is $10.0 + -5.0$, with the final value being 5.0.
 - The value 5.0, calculated above, is then stored in total .
- Notice that this is performing the same task, just using the next element from the array.

^aAs before, total has not been changed at this point, so it has the value 10.0 as shown in Figure 4.42.

For loop increases the value of i and runs the loop body a third time

The end of the for loop has been reached again, so it increments the value of its control variable, assigning i the value 2, and then jumps back to check its condition. As i is still in the range 0..2 the loop body will run a third time, making step 3 the next action.

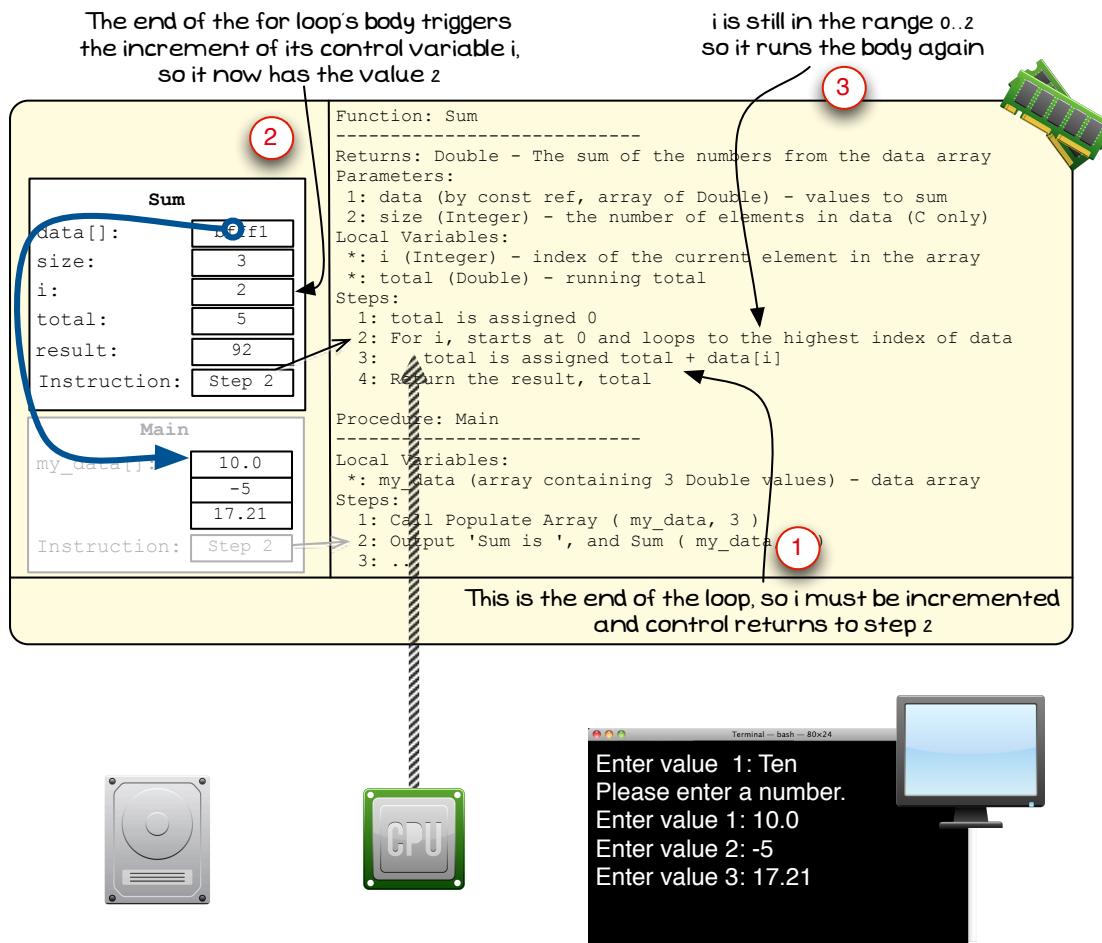


Figure 4.44: At the end of the loop body i is incremented, and control jumps back to check the condition

Note

- In Figure 4.44 the indicated areas show the following:
 - The end of the loop has been reached. The for loop increments its control variable, and jumps back to check its condition.
 - In this case i is the control variable, so its value is increased to 2.
 - The condition of the loop is checked to see if the body should execute. As i is still in the range 0..2 (it is less than 3) the loop's body will execute, making step 3 the next action.
- The end of each for loop always performs these steps. Increasing the value of its control variable, and jumping back to check its condition.

The value of total is increased by the value in data[2]

The body of the for loop reads the *current* value from the data array, `data[i]`. Now `i` has the value 2 it reads the second element of the array. This reads the value 17.21, which is added to the current total, 5.0. The resulting value is then stored back into the total variable, giving it the value 22.21, calculated from the expression `total + data[2]`, which is $5.0 + 17.21$.

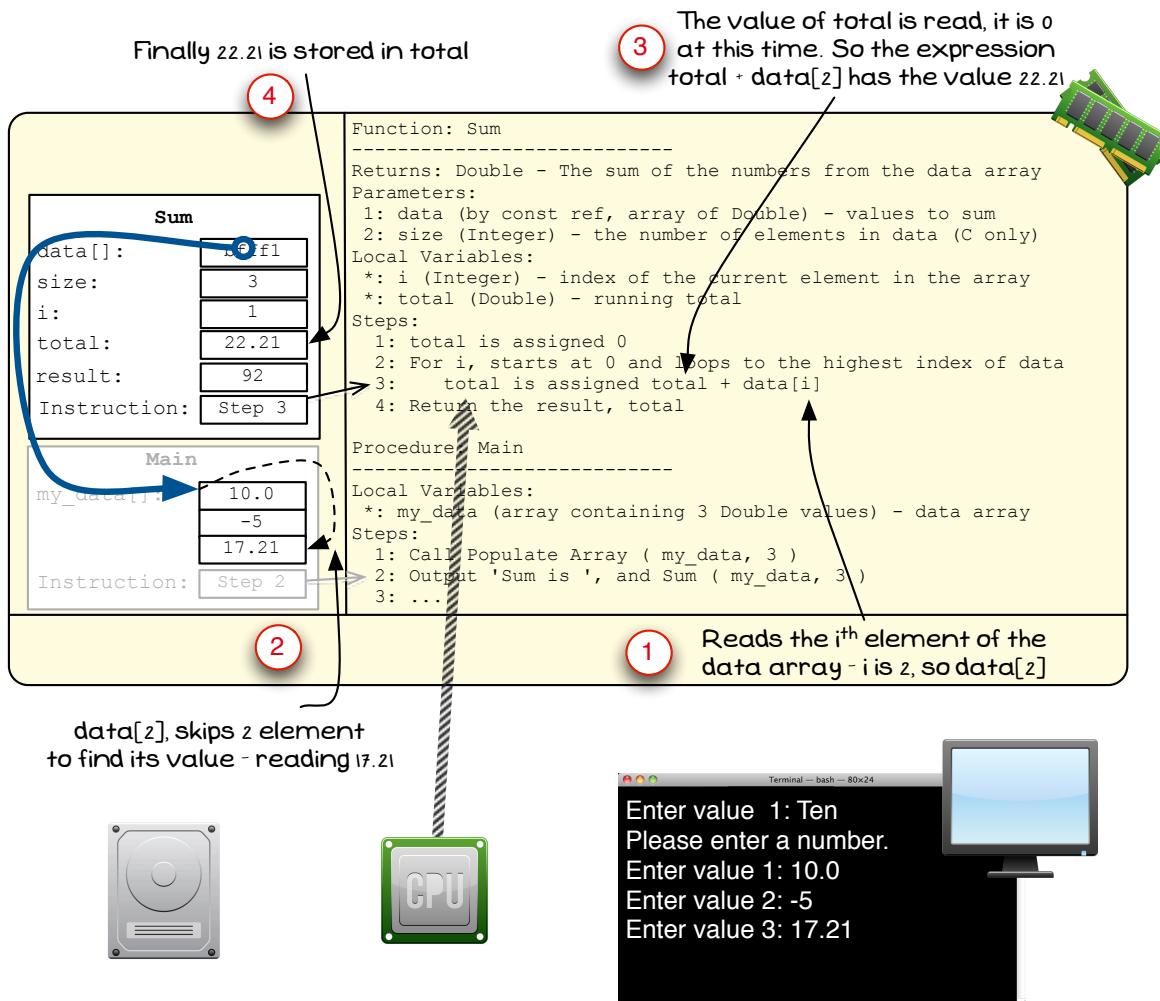


Figure 4.45: Total is increased by the value in data[1]

Note

- In Figure 4.45 the indicated areas show the following:
 - The value of `data[i]` must be read in this expression. At this point `i` is 2, so `data[2]` must be read.
 - `data[2]` is found in the array referred to by `data`, after skipping 2 elements. This reads the value of the third array element.
 - The expression evaluates `totala + data[i]`, giving `total + data[2]`, which is $5.0 + 17.21$, with the final value being 22.21.
 - The value 22.21, calculated above, is then stored in `total`.
- Notice that this is performing the same task, just using the next element from the array.

^aAs before, `total` has not been changed at this point, so it has the value 5.0 as shown in Figure 4.44.

For loop increases the value of i, and this time the loop finishes

The end of the for loop has been reached again, so it increments the value of its control variable, assigning *i* the value 3, and then jumps back to check its condition. This time *i* is no longer in the range 0..2 (it is not less than 3), so the loop body will now be skipped, making step 4 the next action.

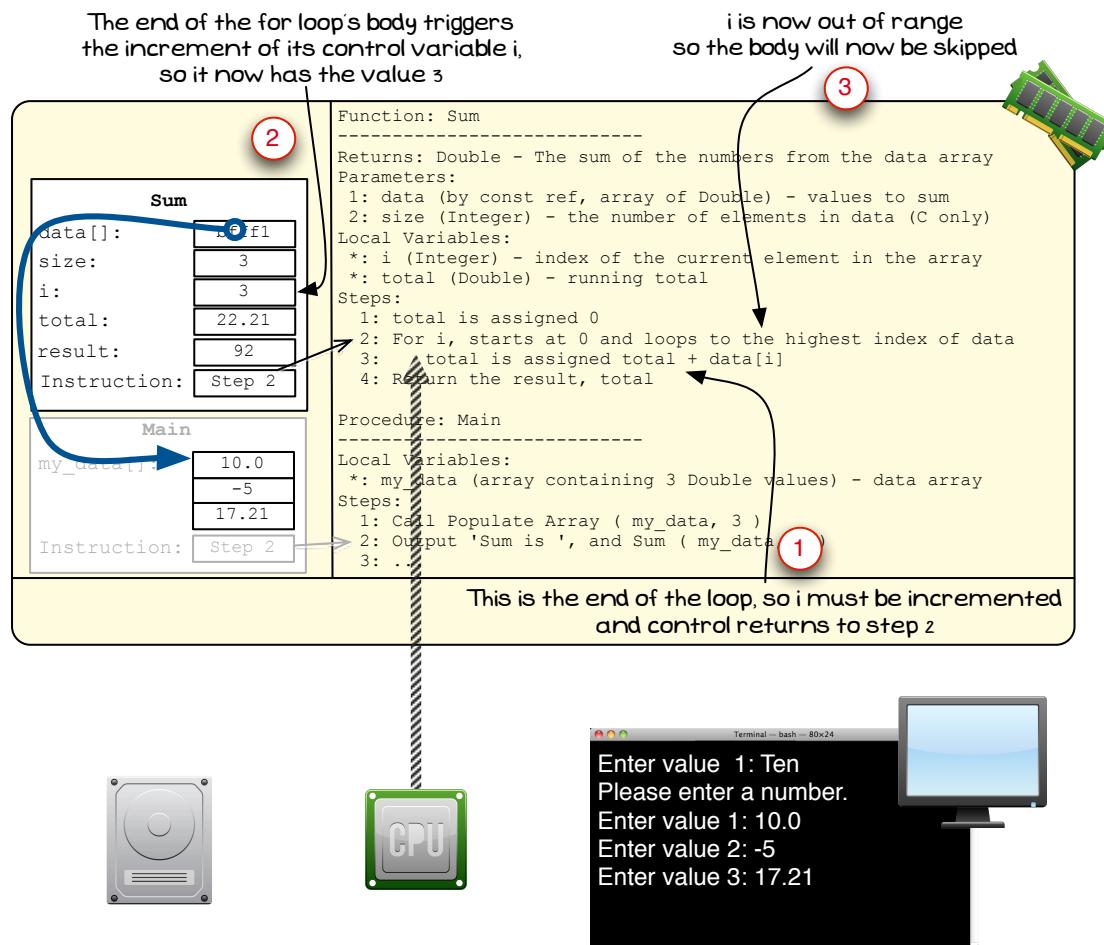


Figure 4.46: At the end of the loop body *i* is incremented, and control jumps back to check the condition

Note

- In Figure 4.46 the indicated areas show the following:
 - The end of the loop has been reached. The for loop increments its control variable, and jumps back to check its condition.
 - In this case *i* is the control variable, so its value is increased to 3.
 - The condition of the loop is checked to see if the body should execute. *i* is no longer in the range 0..2 (it is not less than 3), so the loop's body will be skipped, making step 4 the next action.
- The for works just like a while loop, checking its condition each loop and skipping the body when the condition is not met (when it is false).

Sum function returns the total to the expression in Main

The end of the for loop has been reached again, so it increments the value of its control variable, assigning *i* the value 3, and then jumps back to check its condition. This time *i* is no longer in the range 0..2 (it is not less than 3), so the loop body will now be skipped, making step 4 the next action.

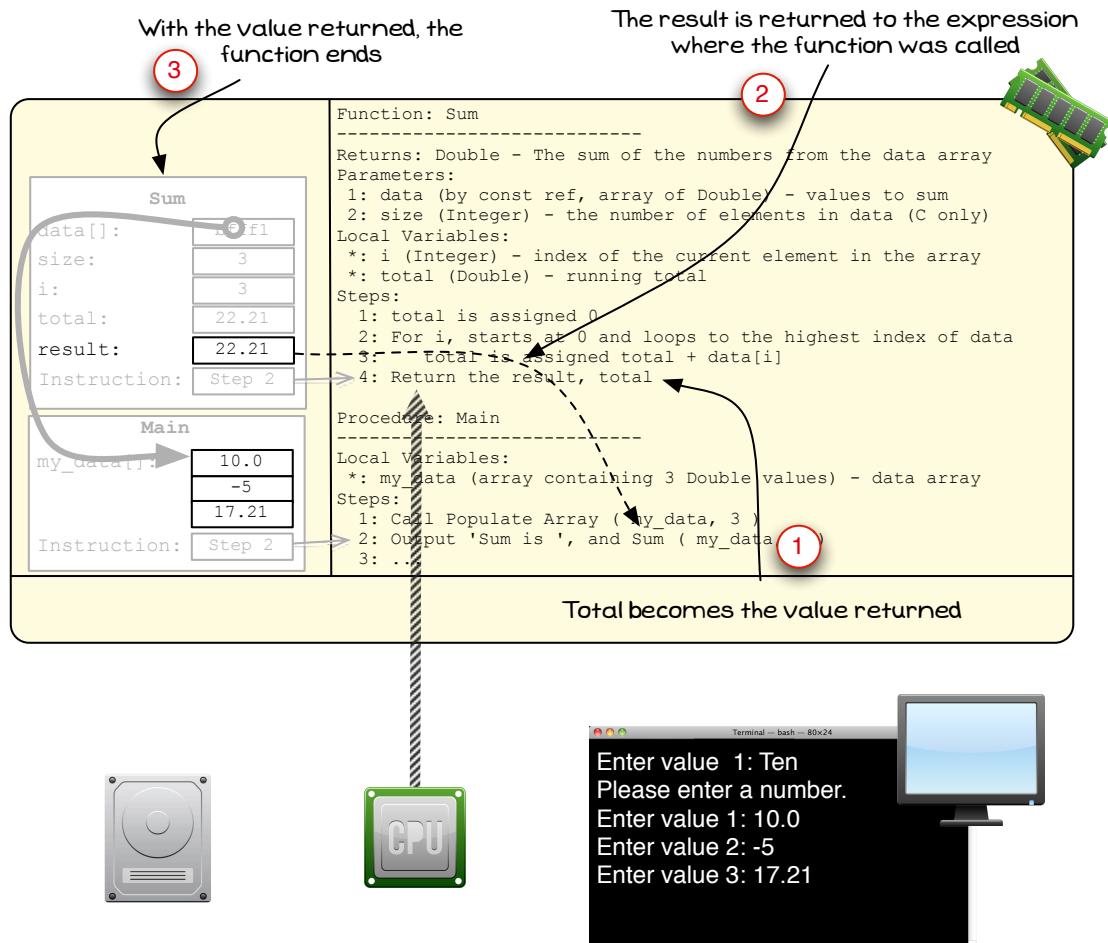


Figure 4.47: Step 4 indicates that the value in total is returned to Main

Note

- In Figure 4.47 the indicated areas show the following:
 - Step 4 indicates that the value in the *total* variable should be returned to the caller.
 - The result returned by the **Sum** function is used in the expression in **Main**.
 - As this is the end of the function its space on the stack is released, allowing control to return to **Main**.
- The for works just like a while loop, checking its condition each loop and skipping the body when the condition is not met (when it is false).

Main outputs the sum to the Terminal

Sum returns its value to be used in step 2 of Main. This step outputs the value returned to the Terminal. So by the end of Step 2 in Main the sum has been calculated, and written to the Terminal.

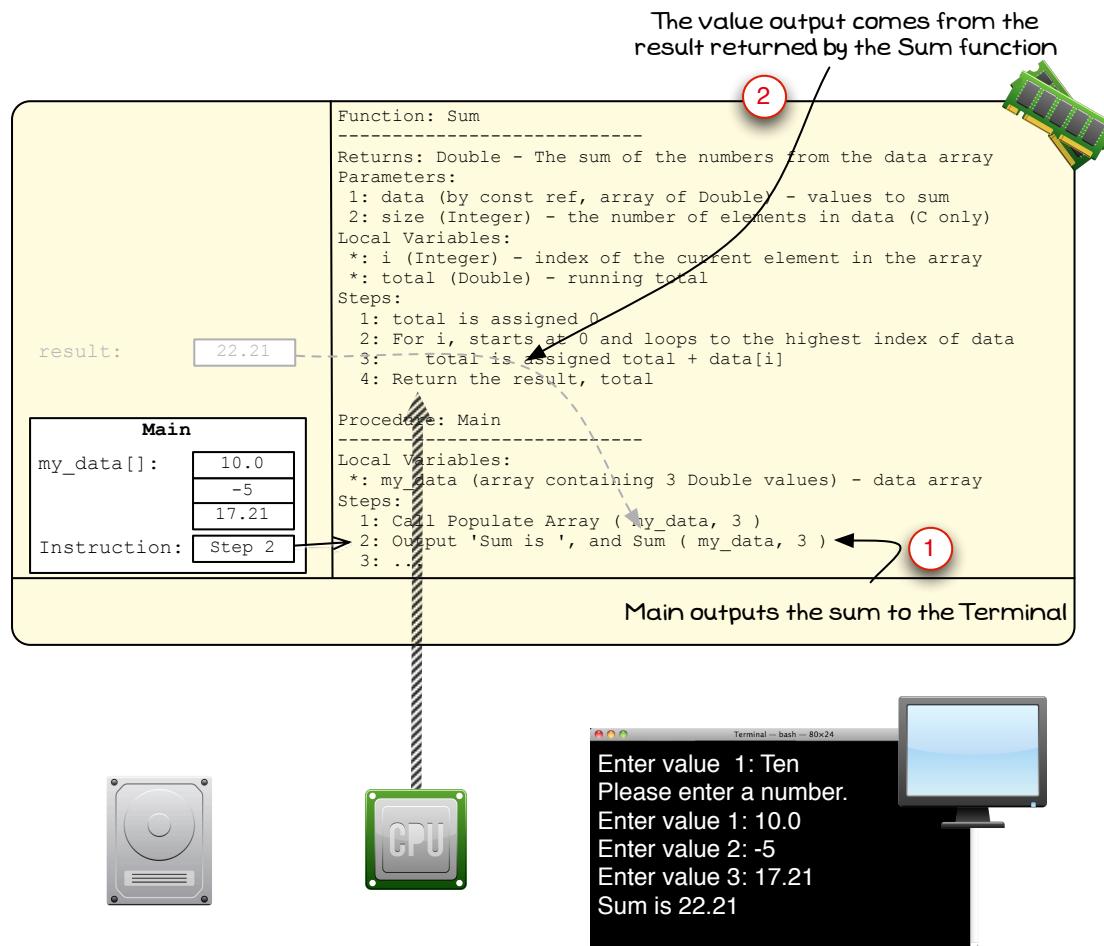


Figure 4.48: Step 4 indicates that the value in total is returned to Main

Note

- In Figure 4.48 the indicated areas show the following:
 - Back in Main, the sum is output to the Terminal.
 - The value that is output is the value returned from the Sum function.

4.6 Array Examples

4.6.1 Bubble Game (start)

The bubble game has ten floating bubbles for the user to ‘pop’. Each bubble is a sprite, which gives it a location on the screen and a movement vector. When the program runs the bubbles move in random directions, reappearing at a new random location if they go off the screen.

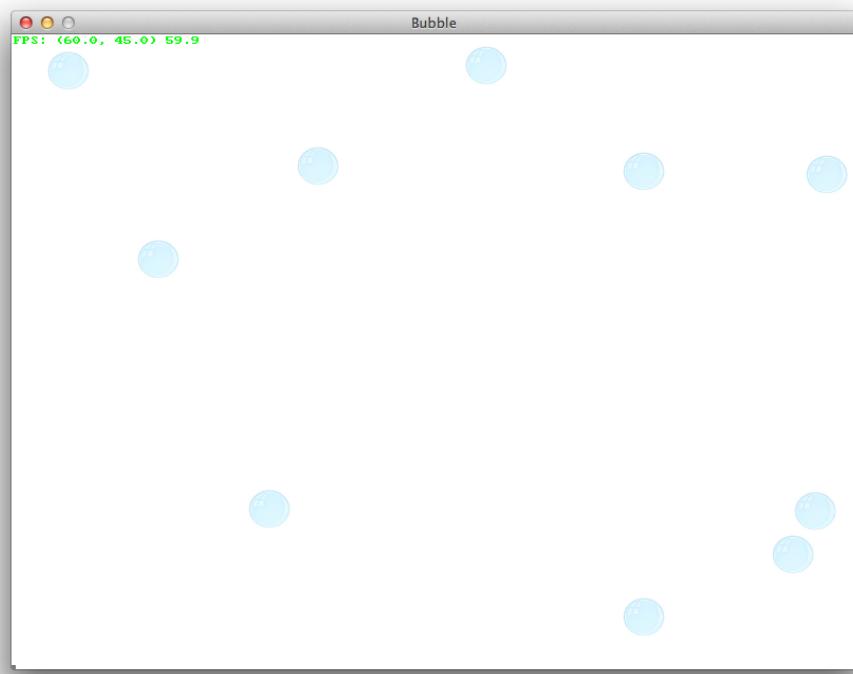


Figure 4.49: Example execution of the Bubbles program

C++

```
#include <stdio.h>
#include <stdbool.h>
#include "SwinGame.h"

#define BUBBLE_COUNT 10

// Load the bubble image
void load_resources()
{
    load_bitmap_named("bubble", "bubble.png");
}

// Place a bubble somewhere on the screen,
// and give it a random movement
void place_bubble(sprite bubble)
{
    // Set the bubble's position
    sprite_set_x(bubble, rnd(screen_width() - sprite_width(bubble)));
    sprite_set_y(bubble, rnd(screen_height() - sprite_height(bubble)));

    // Set the bubble's movement
    sprite_set_dx(bubble, (rnd() * 2) - 1); // between +1 and -1
    sprite_set_dy(bubble, (rnd() * 2) - 1); // between +1 and -1
}

// Create bubbles, and place them on the screen to start
void populate_bubbles(sprite bubbles[], int sz)
{
    int i;

    for (i = 0; i < sz; i++) //For each bubble
    {
        bubbles[i] = create_sprite(bitmap_named("bubble")); // Create it
        place_bubble(bubbles[i]); // Place it on the screen
    }
}

// Update the bubble, move it and check if it is off screen
void update_bubble(sprite bubble)
{
    update_sprite(bubble); // Moves based on sprites dx,dy

    if (sprite_offscreen(bubble)) // is it off screen?
    {
        place_bubble(bubble); // put it back on screen
    }
}
```

Listing 4.24: Starting code for a Bubble Game in C++, continues in Listing 4.25

C++

```

// Update all of the bubbles...
void update_bubbles(sprite bubbles[], int sz)
{
    int i;

    for (i = 0; i < sz; i++)
    {
        update_bubble(bubbles[i]);
    }
}

// Draw all of the bubbles
void draw_bubbles(const sprite bubbles[], int sz)
{
    int i;

    for (i = 0; i < sz; i++)
    {
        draw_sprite(bubbles[i]);
    }
}

// A start of a bubble game...
// Requires "bubble.png" to be placed in Resources/images
int main()
{
    // Create an array of bubbles
    sprite bubbles[BUBBLE_COUNT];

    open_audio();
    open_graphics_window("Bubble Pop!", 800, 600);
    load_default_colors();

    load_resources();
    populate_bubbles(bubbles, BUBBLE_COUNT);      // Load the bubbles

    do
    {
        // Update the game...
        process_events();
        update_bubbles(bubbles, BUBBLE_COUNT);

        // Draw the game
        clear_screen();

        draw_framerate(0,0);
        draw_bubbles(bubbles, BUBBLE_COUNT);

        refresh_screen();
    } while ( ! window_close_requested() );

    close_audio();

    release_all_resources();
    return 0;
}

```

Listing 4.25: Starting code for a Bubble Game in C++ (continued)

Pascal

```

program BubbleGame;
uses
  sgTypes, sgInput, sgAudio, sgGraphics, sgResources, sgUtils, sgText,
  sgSprites, sgImages;

const BUBBLE_COUNT = 10;

// Load the bubble image
procedure LoadResources();
begin
  LoadBitmapNamed('bubble', 'bubble.png');
end;

// Place a bubble somewhere on the screen, and give it a random movement
procedure PlaceBubble(bubble : Sprite);
begin
  SpriteSetX(bubble, Rnd(ScreenWidth() - SpriteWidth(bubble)));
  SpriteSetY(bubble, Rnd(ScreenHeight() - SpriteHeight(bubble)));
  SpriteSetDx(bubble, (Rnd() * 2) - 1); // between +1 and -1
  SpriteSetDy(bubble, (Rnd() * 2) - 1); // between +1 and -1
end;

// Create bubbles, and place them on the screen to start
procedure PopulateBubbles(var bubbles: array of Sprite);
var
  i, bubbleWidth, bubbleHeight: integer;
begin
  bubbleWidth := BitmapWidth(BitmapNamed('bubbles'));
  bubbleHeight := BitmapHeight(BitmapNamed('bubbles'));

  for i := Low(bubbles) to High(bubbles) do
  begin
    bubbles[i] := CreateSprite(BitmapNamed('bubble'));
    PlaceBubble(bubbles[i]);
  end;
end;

// Update the bubble, move it and check if it is off screen
procedure UpdateBubble(bubble: Sprite);
begin
  UpdateSprite(bubble); // Moves based on sprites dx,dy

  if (SpriteOffscreen(bubble)) then // is it off screen?
    PlaceBubble(bubble) // put it back on screen
end;

// Update all of the bubbles...
procedure UpdateBubbles(var bubbles: array of Sprite);
var
  i: integer;
begin
  for i := Low(bubbles) to High(bubbles) do
  begin
    UpdateBubble(bubbles[i]);
  end;
end;

```

Listing 4.26: Starting code for a Bubble Game in Pascal, continues in Listing 4.27



Pascal

```
// Draw all of the bubbles
procedure DrawBubbles(const bubbles: array of Sprite);
var
  i: integer;
begin
  for i := Low(bubbles) to High(bubbles) do
  begin
    DrawSprite(bubbles[i]);
  end;
end;

// A start of a bubble game...
// Requires 'bubble.png' to be placed in Resources/images
procedure Main();
var
  // Create an array of bubbles
  bubbles: array [0 .. BUBBLE_COUNT - 1] of Sprite;
begin
  OpenAudio();
  OpenGraphicsWindow('Bubble Pop!', 800, 600);

  LoadResources();
  PopulateBubbles(bubbles);      // Load the bubbles

  repeat
    // Update the game...
    ProcessEvents();
    UpdateBubbles(bubbles);

    // Draw the game
    ClearScreen();

    DrawFramerate(0,0);
    DrawBubbles(bubbles);

    RefreshScreen();
  until WindowCloseRequested();

  CloseAudio();

  ReleaseAllResources();
end;

begin
  Main();
end.
```

Listing 4.27: Starting code for a Bubble Game in Pascal (continued)



4.7 Array Exercises

4.7.1 Concept Questions

Read over the concepts in this chapter and answer the following questions:

1. How is an array different to a standard variable?
2. How do arrays make it easier to work with multiple values?
3. Why is 0 the index of the first element in an array?
4. How many bytes in memory would an array of 10 integers require?
5. Draw a picture to show how the 10 integer values are stored in memory. Indicate how these values relate to the array.
6. How can you access an element from an array?
7. How can you copy the contents of one array into another array?
8. Why should you pass an array to a parameter by reference, rather than by value?
9. Can you perform an action on all elements of an array?
10. How should you *rethink* actions that needs to be performed on all elements in an array?
11. How does the for loop work together with an array to perform an action on each element in an array?
12. Why is a string an array? What values are stored in the elements of a string?
13. Strings in C and Pascal have a single byte overhead. What is this overhead for? Why is it needed?
14. Can you read/write past the end of an array (i.e. reading/writing to the 11th element when the array only contains 10 elements)? What can happen if you do this?

C++

1. In C you need to pass an additional size parameter for any arrays passed in a function/procedure call. Explain why this is needed.



4.7.2 Code Reading Questions

Use what you have learnt to read and understand the following code samples.

1. Read the C code in Listing 4.28, or the Pascal code in Listing 4.29, and answer the following questions:

- (a) What is returned by the function if it is passed ...

data[]	sz (C only)	max
{ 1, 2, 3, 4 }	4	10
{ -4, 6, 20, 11, 92 }	5	50
{ 2, 7, 1 }	3	5

- (b) What would be a good name for this function?
- (c) Why is the array passed in using the const modifier?

C++

```
bool ????(const int data[], int sz, int max)
{
    int i;
    bool result = true;

    for (i = 0; i < sz; i++)
    {
        if (data[i] > max)
        {
            result = false;
            break;
        }
    }

    return result;
}
```



Listing 4.28: What does this C function do?

Pascal

```
function ????(const data: array of Integer; max: Integer): Boolean;
var
    i: Integer;
begin
    result := true;

    for i := Low(data) to High(data) do
    begin
        if data[i] > max then
        begin
            result := false;
            break;
        end;
    end;
end;
```



Listing 4.29: What does this Pascal function do?

2. Read the C code in Listing 4.30, or the Pascal code in Listing 4.31, and answer the following questions:

- (a) What does the code do?
- (b) What would be a good name for this function?
- (c) What is returned by the function if it is passed ...

data[]	sz (C only)	val
{ 1, 2, 3, 4 }	4	3
{ 7, 12, 20, 51, -6 }	5	10
{ 2, 7, 1 }	3	1
{ -1, -2, -3, -4 }	4	-3
{ 6, 12, 18, 24, 30 }	5	22

C++

```
bool ????(const int data[], int sz, int val)
{
    int i;
    bool result = false;

    for (i = 0; i < sz; i++)
    {
        if (data[i] == val)
        {
            result = true;
            break;
        }
    }

    return result;
}
```

Listing 4.30: What does this C function do?

**Pascal**

```
function ????(const data: array of Integer; val: Integer): Boolean;
var
    i: Integer;
begin
    result := False;

    for i := Low(data) to High(data) do
    begin
        if data[i] = val then
        begin
            result := True;
            exit;
        end;
    end;
end;
```



Listing 4.31: What does this Pascal function do?

3. The following code is designed to find the median (middle value) of a array of numbers, but does it work? Read the C code in Listing 4.32, or the Pascal code in Listing 4.33, and answer the following questions:
- Assume that data contains the values [1, 2, 3, 4, 5, 6, 7, 8], execute this code by hand and show the steps involved. Explain any shortcuts you take.
 - What value is returned when the function ends?
 - You should have noticed that there is a bug in this program. How can it be fixed?
 - What does this program assume about the data in the array?
 - Can you think of a simpler solution? Explain your solution.

C++

```
int median (const int data[], int sz)
{
    int start_idx, end_idx;

    start_idx = 0;
    end_idx = sz - 1; // index of last element

    while (start_idx != end_idx && start_idx < end_idx)
    {
        start_idx++;
        end_idx--;
    }

    if (start_idx == end_idx)
        return data[start_idx];
    else
        return data[start_idx] + data[end_idx] / 2;
}
```

**Listing 4.32:** A median function written using C

Pascal

```
function Median (const data: array of Integer): Integer;
var
    startIdx, endIdx: Integer;
begin
    startIdx := Low(data);
    endIdx := High(data);

    while (startIdx <> endIdx) and (startIdx < endIdx) do
    begin
        startIdx := startIdx + 1;
        endIdx := endIdx - 1;
    end;

    if startIdx = endIdx then result := data[startIdx]
    else result := data[startIdx] + data[endIdx] / 2;
end;
```

**Listing 4.33:** A median function written using Pascal

4. Read the C code in Listing 4.34, or the Pascal code in Listing 4.35, and answer the following questions:
- This procedure alters the array passed to it. Hand execute the procedure for the values shown in the following table. Record your workings as well as the final answer.
 - What does the code do?
 - What would be a good name for this procedure?
 - What would be good names for param3 and param4?

data[]	sz (C only)	param3	param4
{ 1, 2, 3, 4 }	4	2	10
{ 8, 10, 11 }	3	1	9
{ -1, -2, -3, -4 }	4	0	0
{ 42, 42 }	2	1	73

C++

```
void ???(int data[], int sz, int param3, int param4)
{
    int i;

    for (i = sz - 1; i > param3; i--)
    {
        data[i] = data[i - 1];
    }

    data[param3] = param4;
}
```

Listing 4.34: What does this C function do?

**Pascal**

```
procedure ???(var data: array of Integer; param3, param4: Integer);
var
    i: Integer;
begin
    for i := High(data) downto param3 + 1 do
    begin
        data[i] := data[i - 1];
    end;

    data[param3] := param4;
end;
```



Listing 4.35: What does this Pascal function do?

4.7.3 Code Writing Questions: Applying what you have learnt

Apply what you have learnt to the following tasks:

1. Create a program that reads in the user's name and then outputs the message 'Hello ' and the users name. For example, if the user enters the name 'Fred' then the program outputs the message 'Hello Fred!' to the Terminal. See Table 4.10.

Program Description	
Name	Hello User
Description	Asks the user to enter their name, and then outputs 'Hello ' and their name to the Terminal.

Table 4.10: Description of the *Hello User* program.

2. Create a program that reads in the user's name and then determines if the name is a *silly name*. For example, if the user enters the name 'Fred' the program could output, 'Fred is an awesome name!', but if the user enters any other name they get a message like 'Andrew is a silly name!'. Customise the message for your friends and tutor. See Table 4.11.

Program Description	
Name	Silly Name Test
Description	Asks the user to enter their name, and then outputs a message based on the name entered.

Table 4.11: Description of the *Silly Name* program.

3. Complete the implementation of the Statistics Calculator. Then add the following additional functionality:

(a) Add a **Print All** procedure to print all of the values stored in the array to the Terminal.

C++

```
void print_all(const int data[], int sz) ...
```



Pascal

```
procedure PrintAll(const data: array of Integer); ...
```



(b) Add a **Frequency** function that calculates the frequency of a value in the array.

C++

```
int frequency(const int data[], int sz, int n) ...
```



Pascal

```
function Frequency(const data: array of Integer; n: Integer): Integer; ...
```



(c) Add a **Standard Deviation** function.

C++

```
int stddev(const int data[], int sz) ...
```



Pascal

```
function Stddev(const data: array of Integer): Integer; ...
```



(d) Add a **Minimum** function.

C++

```
int min(const int data[], int sz) ...
```



Pascal

```
function Min(const data: array of Integer): Integer; ...
```



4. Implement the Bubble Game from Section 4.6.1 Bubble Game (start). You will need to do the following:
- Download the appropriate SwinGame template.
 - Extract the template into a folder called 'Bubbles'
 - Find a picture of a bubble that is about 30 pixels wide and high.
 - Place the bubble image in the Resources/images folder in your SwinGame project.
 - Implement the code from Section 4.6.1.
 - Run the game and you should see something similar to Figure 4.50.

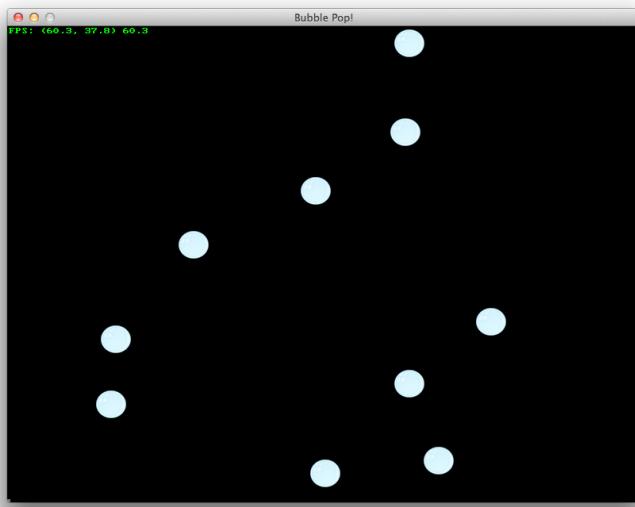


Figure 4.50: The start of a bubble game

- (g) Extend the game in one or more of the following ways:
- Play a sound effect when the bubbles appear.
 - Add a background image.
 - Check if the user has clicked the bubble. If they have, play a pop sound effect and place the bubble.
 - Add a score, and give points for the number of bubbles popped.
 - Try changing the game dynamics... for example: Start new bubbles at the top of the screen, have them move down the screen, and pop when they hit the ground, ending the game.
 - ...

4.7.4 Extension Questions

If you want to further your knowledge in this area you can try to answer the following questions. The answers to these questions will require you to think harder, and possibly look at other sources of information.

1. Write the code to sort the values in your statistic calculator.
2. Write a function that calculates the **Mode**, the most frequent value.
3. Get your bubbles to bounce off each other. Hint: have a look at SwinGame physics **Collide Circles** procedure.

5

Custom Data Types

ou are progressing well! You have learnt to structure your spells, guide the flow of magic, and apply it to many targets. Now it is time to see how you can also structure the components that go into your spells. Fetch the snake and dragon scale, now turn your thoughts to their structure. Take your wand and...

Chapter 4 introduced the array, allowing you to store multiple values of the same type in a single variable. This greatly expanded the ways in which you can work with data in your code, but there are more tools you can use to model data in your programs.

This chapter will show you how to create your own data types, allowing you to model the entities¹ related to your program. This means that your code can work with more meaningful values, making it easier to create larger and more complicated programs.

When you have understood the material in this chapter you will be able to define your own data types to model the entities you want to work with in your program.

Contents

5.1 Custom Data Type Concepts	423
5.1.1 Type (recap)	424
5.1.2 Program (with Type Declarations)	425
5.1.3 Type Declaration	426
5.1.4 Declaring Variables (with custom types)	430
5.1.5 Assignment Statement (with Fields and Elements)	431
5.1.6 Expression (with custom types)	434
5.2 Using Custom Types	438
5.2.1 Designing Small DB	438
5.2.2 Understanding Small DB	438
5.2.3 Choosing Artefacts for Small DB	439
5.2.4 Writing the Code for Small DB	444
5.2.5 Compiling and Running Small DB	445
5.3 Custom Data Types in C	447
5.3.1 Implementing Small DB in C	447
5.3.2 C Type Declaration	452
5.3.3 C Record/Structure Declaration	454
5.3.4 C Enumeration Declaration	456
5.3.5 C Union Declaration	457

¹A fancy way of saying the ‘things’ associated with your program.

5.3.6 C Variable Declaration (with Types)	459
5.4 Custom Data Types in Pascal	461
5.4.1 Implementing Small DB in Pascal	461
5.4.2 Pascal Type Declaration	463
5.4.3 Pascal Record Declaration	465
5.4.4 Pascal Enumeration Declaration	468
5.5 Understanding Custom Types	469
5.5.1 Understanding Read Row	469
5.5.2 Understanding Print Row	478
5.6 Example Custom Types	481
5.6.1 Lights	481
5.6.2 Shape Drawing Program	490
5.7 Custom Type Exercises	500
5.7.1 Concept Questions	500
5.7.2 Code Writing Questions: Applying what you have learnt	501
5.7.3 Extension Questions	501

5.1 Custom Data Type Concepts

To this point, data has been about single values that are either numbers, text, or Boolean values. These values can be used in [Expressions](#) and stored in [Variables](#) and [Array](#) elements. Now as we move to creating larger and more complicated programs you need a more effective means of modelling the data in your code. This chapter introduces concepts that you can use to more accurately model the data and entities associated with your program.

This chapter introduces the following **artefacts**:

- [Enumeration](#): a kind of type used to store one of a list of available options.
- [Record](#): a kind of type used to store multiple fields in a single composite value.
- [Union](#): a kind of type used to store one value of different possible types.

You may need to revise the following programming artefacts:

- [Variable](#): The idea of storing data within your code.
- [Array](#): Allowing you to store multiple values within your code.
- [Local Variable](#): Storing data in a [Function](#) or [Procedure](#).
- [Parameter](#): Passing data to a Function or Procedure.

The following programming terminology will also be used in this Chapter:

- [Expression](#): A value used in a statement.
- [Type](#): A kind of data used in your code.

The example program for this chapter will be Small DB, a program that allows the user to enter a number of integer, double, and text values.

```
acain-mbp:types acain$ ./SmallDB
-----
Welcome to Small DB
-----
Please enter 3 values.
They can be text
or numbers.
Enter value: Fred Smith
Stored in row with id 0
Enter value: 123456
Stored in row with id 1
Enter value: 3.1415
Stored in row with id 2
Row with id 0: has text 'Fred Sm'
Row with id 1: has integer 123456
Row with id 2: has double 3.141500
acain-mbp:types acain$
```

Figure 5.1: Small DB run from the Terminal

5.1.1 Type (recap)

This chapter is all about types, so it's important to have a good understanding of what a type is. A type is a specification for a class of data, describing how it is stored and interpreted.

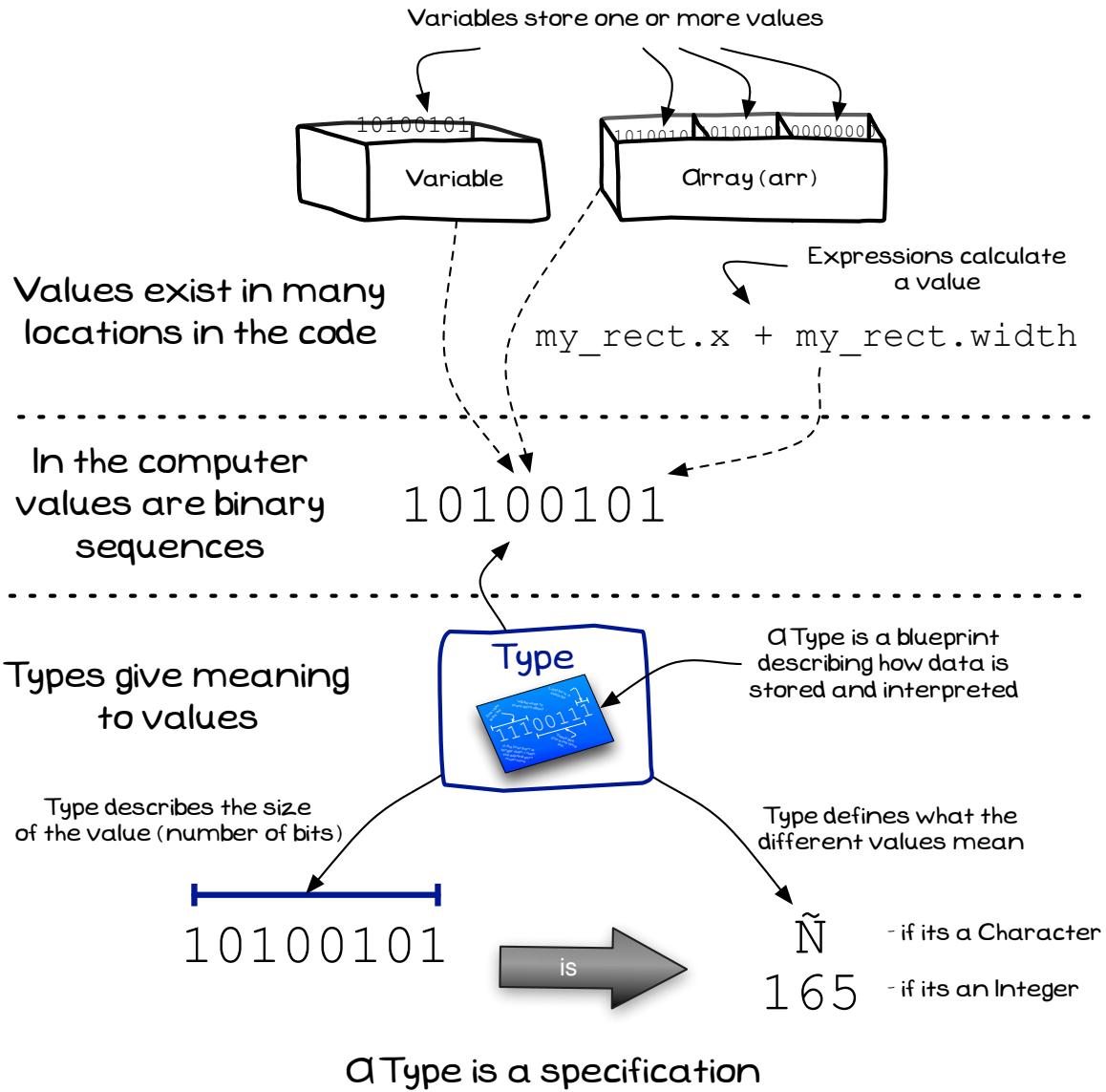


Figure 5.2: Type defines the size and interpretation of values in your code

Note

- A Type is a kind of artefact, describing the format and interpretation of values.
- Types specify the following:
 - The size (number of bits) needed to store values of this type.
 - How the bits of the type are interpreted.
 - The operations that can be performed on values of this type.
- You can think of a Type as a blueprint, specifying data layout.
- The interpretation of a value depends on its Type, for example `10100101` is `~N` if the value is a Character type, but the same value would be 165 if it is an Integer type.
- You can create your own types...

5.1.2 Program (with Type Declarations)

The Program is the overarching artefact, containing the code that defines the other artefacts that we have been creating. When you want to declare your own types, you do so in your program's code. This makes it possible to model the data, in the same way as you model the actions in your code.

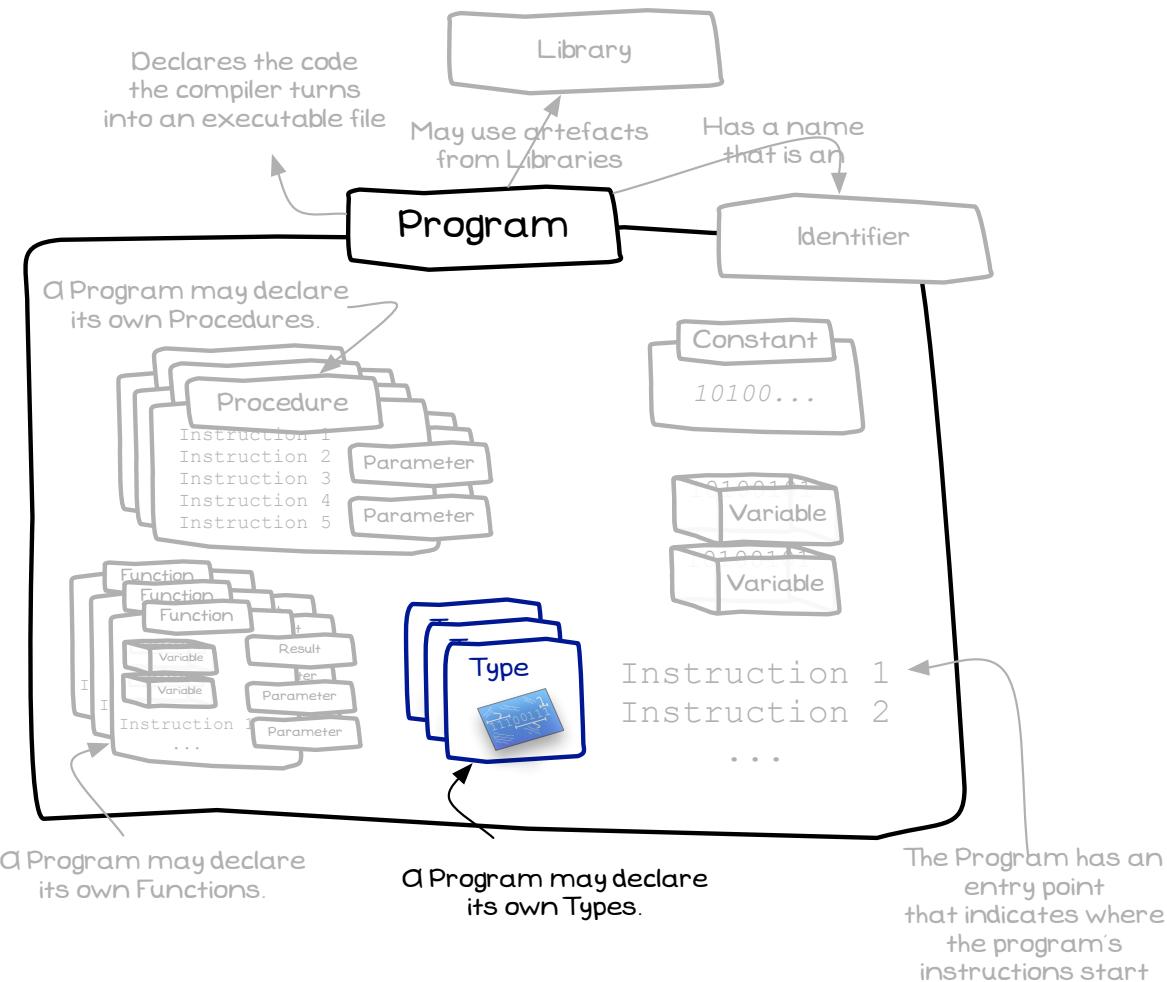


Figure 5.3: A Program can contain Type Declarations

Note

- A program is an **artefact**, it is the container within which you code other artefacts such as your functions, procedures, constants, and now types.
- The types you declare in your program will be available for use within the program's code, and your functions and procedures.
- You will be able to create **Variables** (including **Arrays**) of the types you create. They will store the values in the format you specified. This includes **Parameters** and **Local Variables**.

5.1.3 Type Declaration

Type declarations allow you to define your own types. Programming Languages offer a range of different code structures you can use to build your own types.

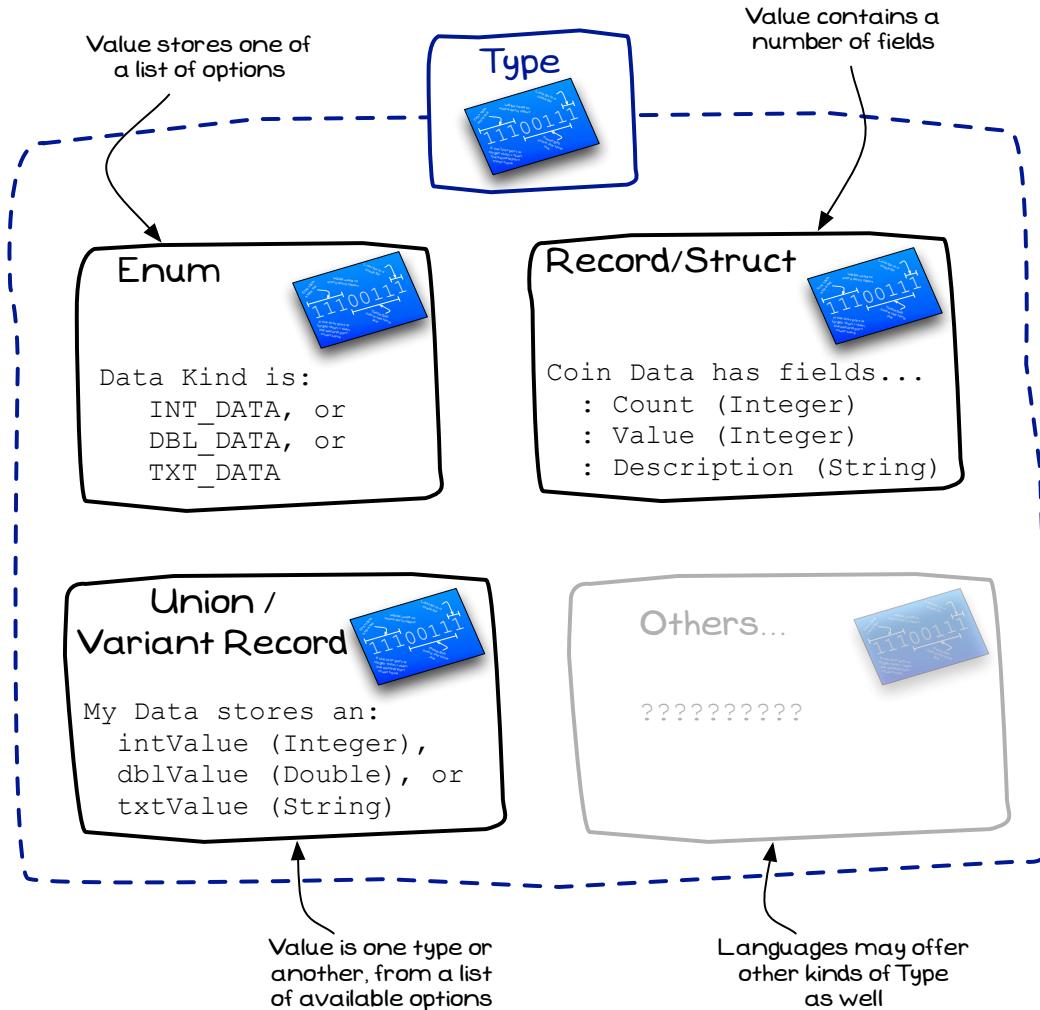


Figure 5.4: You can declare your own Data Types

Note

- A type declaration is the **term** given for declaring your own type *artefact* in code.
- Programming Languages offer a range of different kinds of types that you can create.
- Each kind of type declaration allows you to create a type for different kinds of values:
 - **Enum**: An enumeration creates a type where the value will be one of a list of available options.
 - **Record** or **Struct**: A structured record, where each value of this type is made up of a number of fields.
 - **Union** or **Variant Record**: Makes it possible to create a type where the value could be one of a number of other types.
 - Languages may also offer other kinds of type you can declare.
- You use these different kind of types to design the structure for the values you will work with in your code.
- There is no data associated with a type declaration, you are declaring a new format, not another value.

Record

A Record (Structure in C) is a composite type whose value is made up of a number of **fields**. Each field stores a value of a certain type. A value of the record's type stores data for each field described in the Record/Structure.

In your code, records can be used to model data associated with the *things*, the *entities*, associated with your program. For example, a financial application can have records to store Account, Customer, and Transaction values. A murder mystery game may have Player, Clue, and Scene values, whereas a Space Invaders game would have Player, Alien, and Bullet values. Each of these would be modelled in code using a Record/Structure.

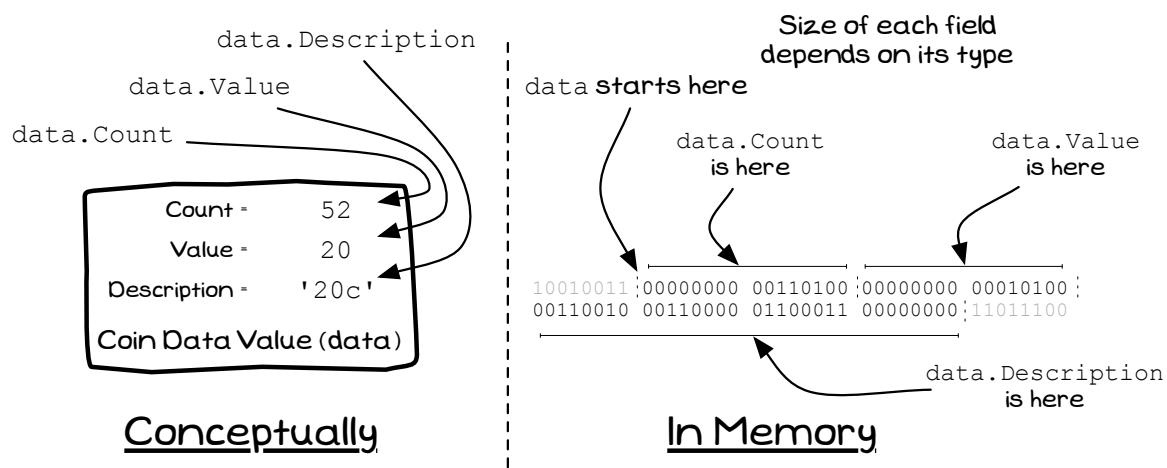
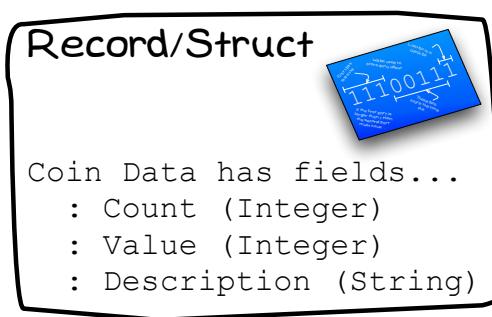


Figure 5.5: A Record or Structure contains Fields

Note

- A Record/Structure is a kind of **artefact** you can declare.
- You can create your own Record/Structure types, these can then be used to define the data stored in **Variables** in your code.
- Remember that this is declaring a new data format, it is not declaring a new data value: for that you need to declare a **Variable**.
- The **size** of a record is based on the sum of the sizes of its fields.

Enumeration

An Enumeration allows you to create a type where the value is one of a list of available options. When you declare an enumeration you are listing the values that are available for data of this type. The example in Figure 5.6 declares a type that can have the value ADD_COINS or REMOVE_COINS.

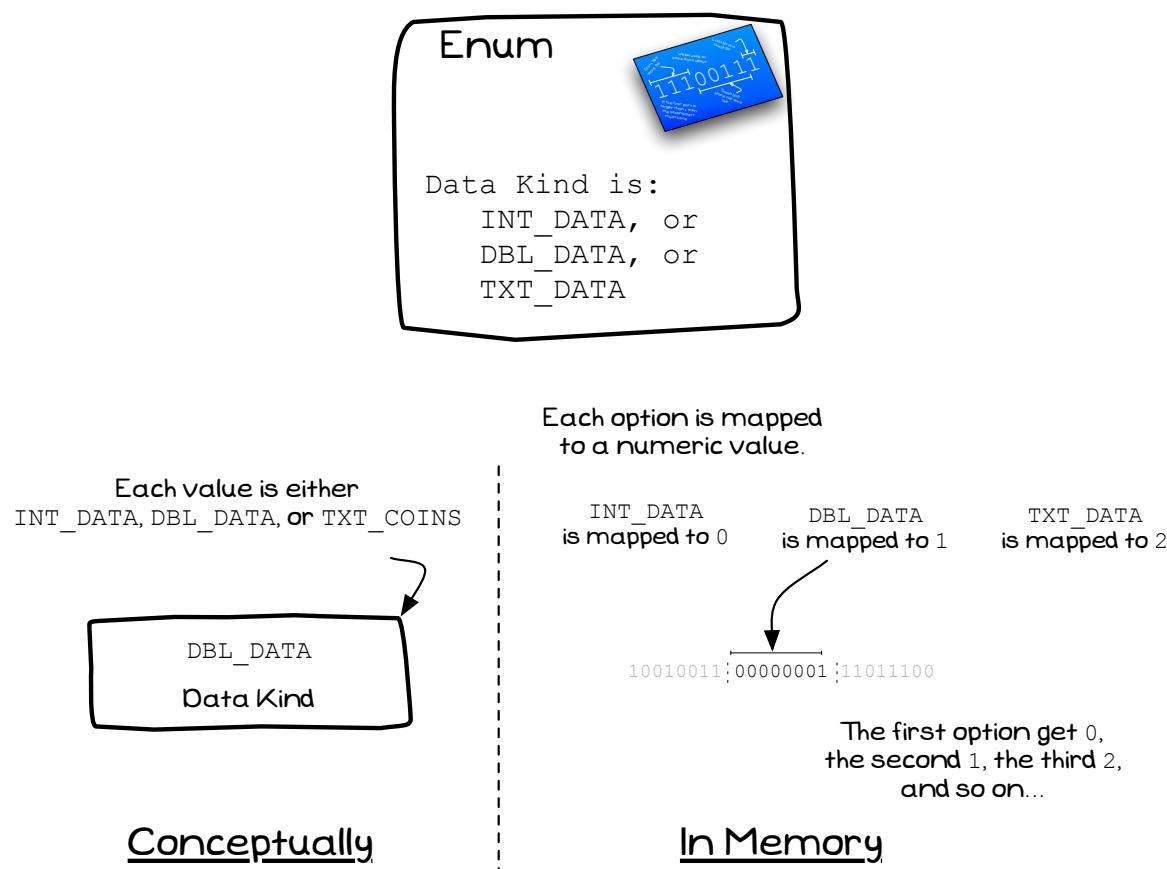


Figure 5.6: An Enumeration allows you to define related constants

Note

- An Enumeration is a kind of **artefact** that you can declare.
- Using an enumeration you can declare a kind of value that must come from a list of available options.
- When you declare the enumeration you list the *values* that it may have.
- This is like a list of constants, where values of this type will match one of these constants.
- Internally the compiler will map your values to numeric values. The first option is represented by the value 0, the second is represented by the value 1, and so on.
- You can specify the values for each option in the enumeration, this can be useful if you want to be able to combine options in a single value. In these cases you give each option a bit unique value (first at 1, then 2, 4, 8, 16, etc).
- The **size** of an enumeration is based on the size of the integer type used to represent its values.

Union

A Union (Variant Record in Pascal) allows you to declare a type where the values may be one of a range of alternative types. In effect, you can declare a type where the value may be one of a number of different types. This is useful if you want to store different types of values at a location in your program.

A Union often works best by accompanying it with a **tag** value. This value then records the kind of data currently being stored at the location in memory. A good option is to use an **Enumeration** for the tag's type, giving you a range of value, that describe the range of types stored.

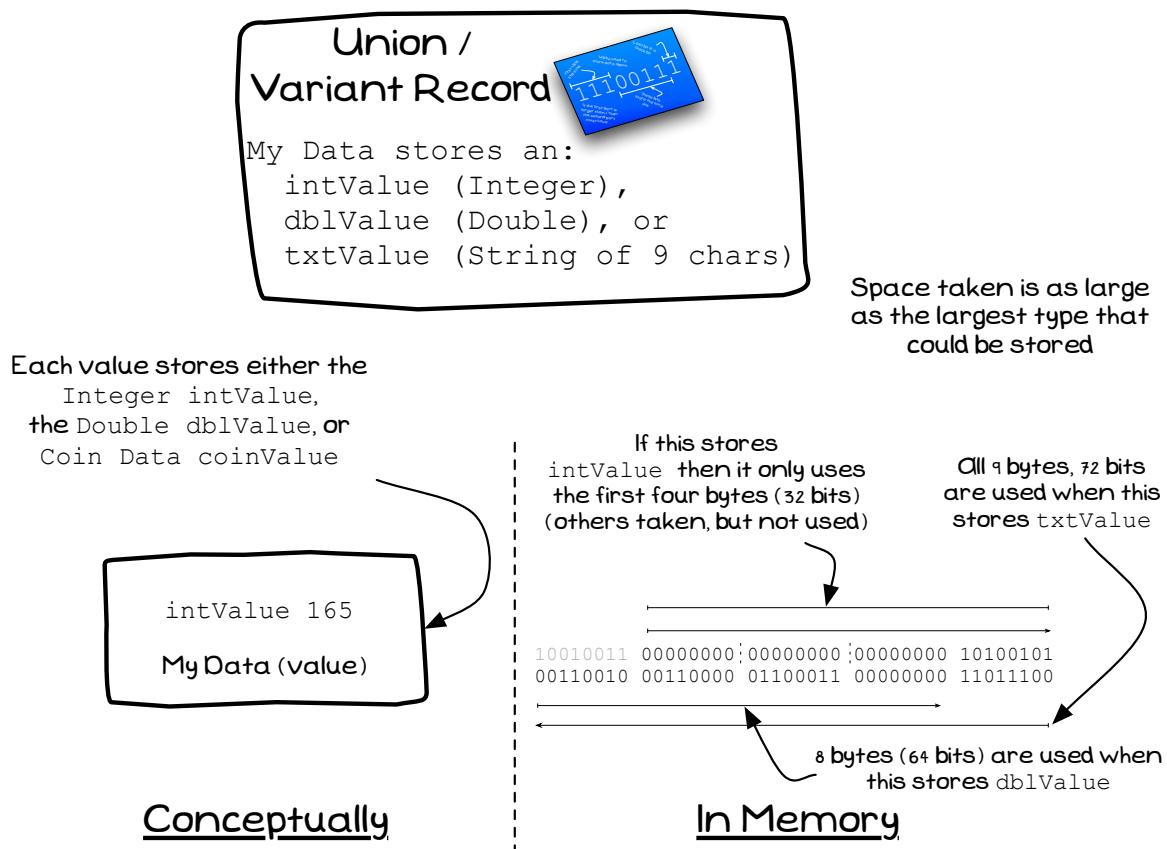


Figure 5.7: A Union is one type that can store one of a range of other types

Note

- A Union is a kind of artefact that you can declare.
- The Union allows you to use one location in memory to store one of a number of types of value.
- At any one time this data can be used to store **one** of these values.
- It is the developers responsibility to ensure they access the right kind of value when it is being used.
- A **tag** value can be used to store a marker that indicates the type of data being stored in the union.
- The **size** of a union is the size of its *largest* option. For example a union of a Character (1 byte), a Integer (4 bytes), and a Double (4 bytes) is 4 bytes, the size of the largest kind of value it needs to store.

5.1.4 Declaring Variables (with custom types)

The custom types allow you to specify a data format. To make use of this format you must declare variables that use this type. The type can be used to declare [Local Variables](#) and [Parameters](#), allowing you to store values in this format and pass them around between your functions and procedures.

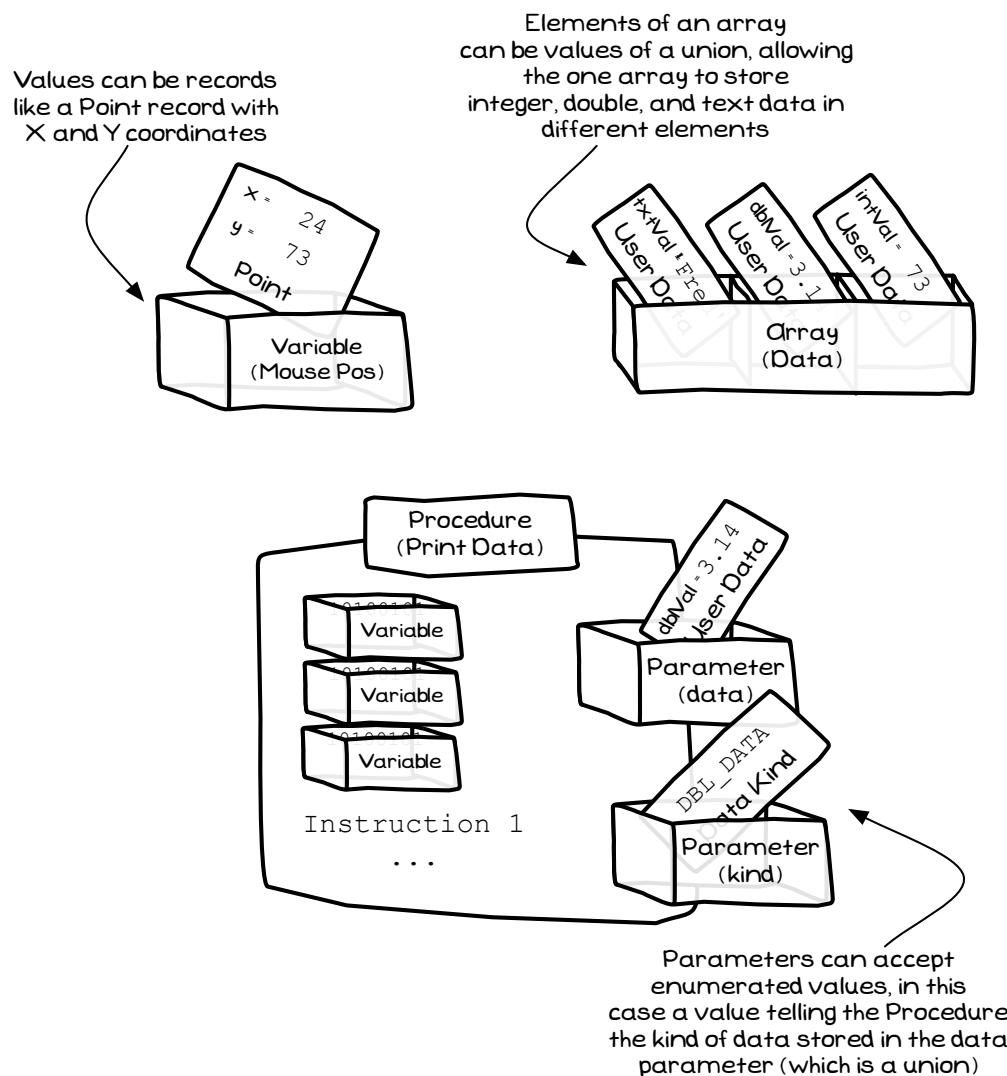


Figure 5.8: Some examples of how you can use your types to declare the kind of data stored in variables

Note

- Variables declaration is the **term** given to the code that creates Variables.
- The Variables you create store data as described in your custom type definitions.
- In Figure 5.8 there are the following examples:
 - Mouse Pos: A Variable that stores Point Record data.
 - Data: [Array](#) that stores User Data values which are either integer, double, or text values using a [Union](#).
 - kind: [Parameter](#) that passes in an [Enumeration](#) value from the Data Kind type. This value can then be used by
- You can combine these types in a huge variety of ways. The best idea is to try and model the entities related to your program.

5.1.5 Assignment Statement (with Fields and Elements)

The Assignment Statement allows you to store a value in a Variable or Array. The righthand side of the assignment is an expression that calculates the value to be stored. The lefthand side is a variable or array element, a place into which the value can be stored. With the addition of the custom types you can now also store values in **fields** of a record or union.

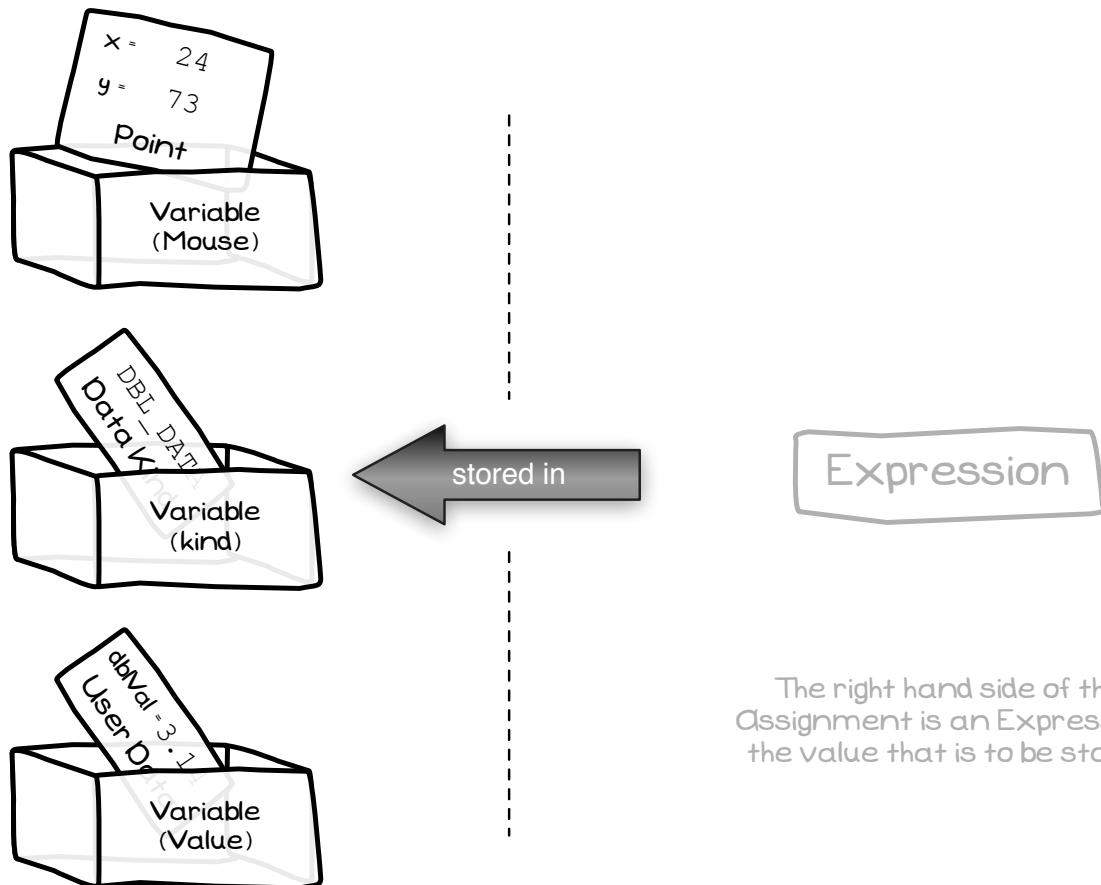


Figure 5.9: You can assign values to a Record or Union's fields

Note

- The Assignment Statement is an **action**, you can command the computer to store a value in a variable, array element, record's field, or union's field.
- Enumeration values are stored in a single variable, so they work in the same way as shown in Section [2.1.8 Assignment Statement](#).
- With a record you can assign values to its fields individually, or you can assign all of the values from another matching record.
- A union can have its value set via its fields, or you can copy the value from another matching union.

Record Assignment

The assignment statement can be used to assign a value to a record's fields, or to copy an existing record's values.

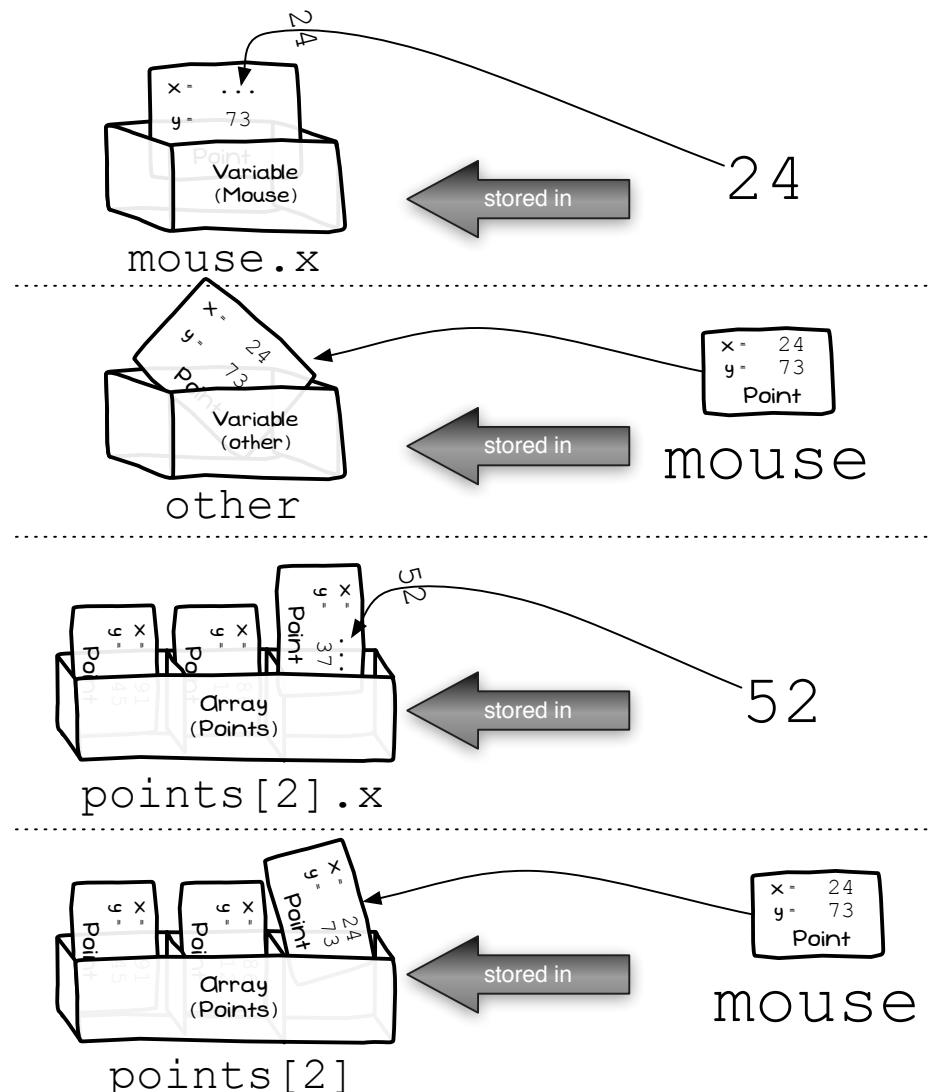


Figure 5.10: You can assign an individual field or the entire record in one assignment statement

Note

- The four examples from 5.10 show the following:
 - You can assign a value to a field of a record. In this case 24 is assigned to `mouse.x`.
 - A Point expression can be assigned to a Point variable. This copies the entire record into the Variable.
 - It doesn't matter if the record is in an array, you can still assign a value to an record's fields.
 - You can also assign an entire record into an element of an array.
- If the language allows arrays to be copied then you can also copy an entire array of records to a destination.

Union Assignment

The **Union** is similar to a Record in that you can assign values to a union via its fields or by copying another union value into the variable or array element. The difference with the Union is that it has only a single value, with the different fields giving you different interpretations of that data.

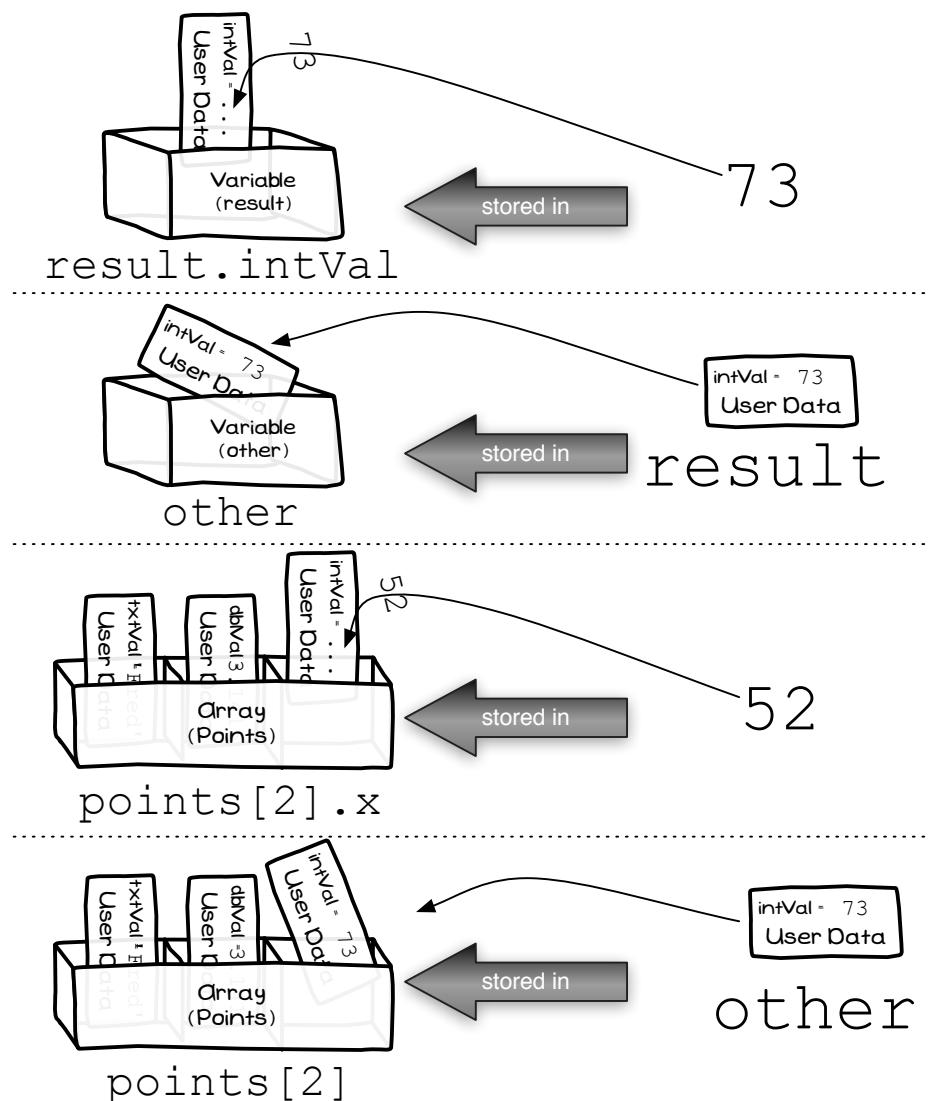


Figure 5.11: You can assign an individual field or the entire union in one assignment statement

Note

- The four examples from 5.11 show the following:
 1. You can assign a value to the fields of a Union. This overrides any value currently stored in the Variable.
 2. It is possible to copy an entire Union value in the assignment.
 3. This works in the same way with arrays, you can write a value to an Union.
 4. You can also copy an existing union value into an element.
- When accessing the data in a Union you are responsible for ensuring you read back the value you stored as it does not remember the kind of value you stored in the union.

5.1.6 Expression (with custom types)

The types you define allow you to specify how data values can be formatted, allowing you to declare variables that contain data in this new format. You can then read data back from your variables in expressions.

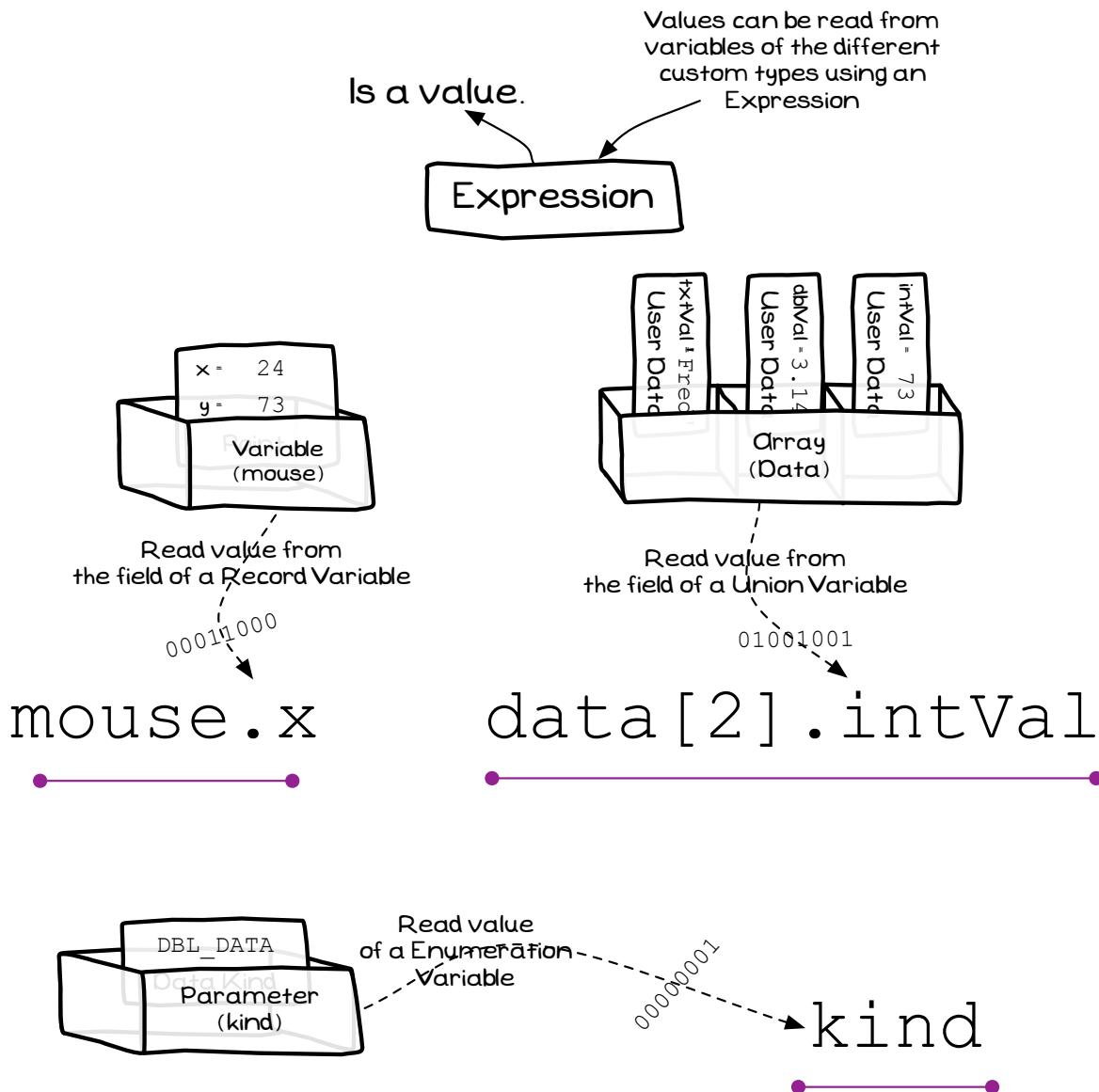


Figure 5.12: An expression can read the value of a record's field, a union's field, and from an enumeration

Note

- Expression is the **term** given to the code that calculates values within your statements.
- Within an expression you can read the value from...
 - a field of a record.
 - a field of a union.
 - an enumeration.
- The dot (.) notation is used to indicate which field you want to access from a record or union.

Record Expressions

A **Record** is a type that contains a number of fields. When using a record value you can use either an individual fields from the record, or the record in its entirety.

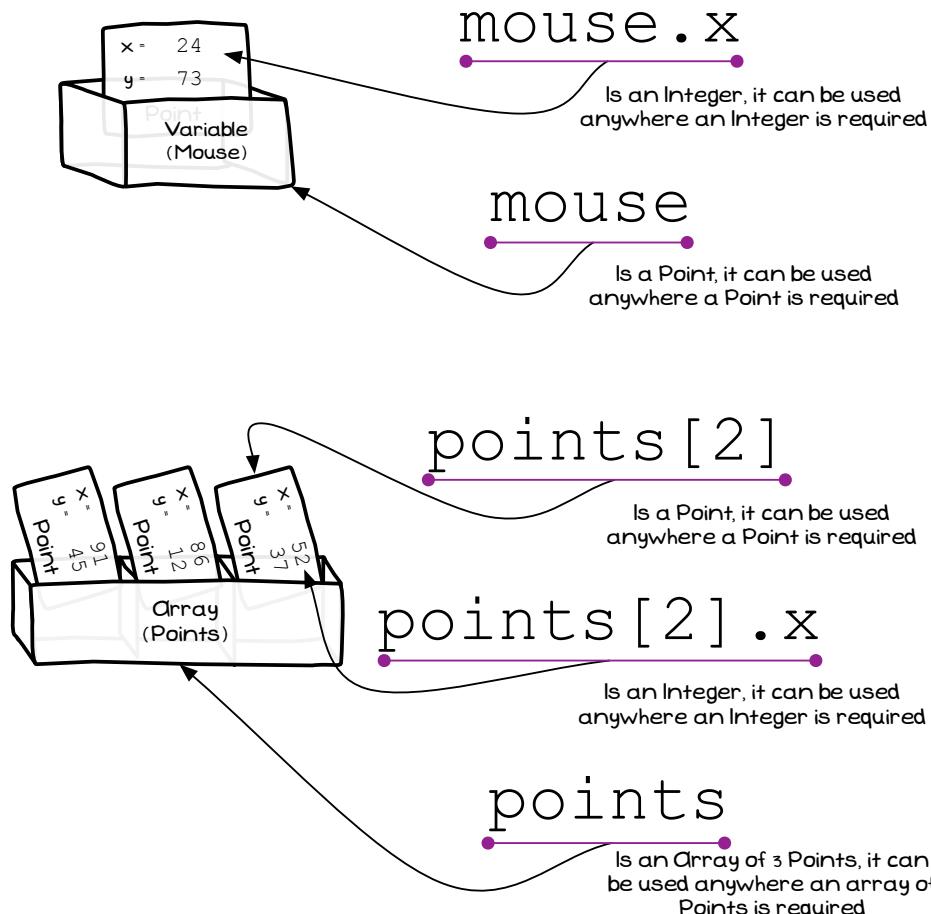


Figure 5.13: A field of a record can be used, or the record can be used in its entirety

Note

- Figure 5.13 shows some examples of expressions on an record variable and array.
- The Point record in the illustration has an `x` field that stores an integer value.
- You can access a field of the record from its variable using the dot notation. So `mouse.x` reads the `x` value form the record stored in the `mouse` variable. This value is then an Integer, and can be used anywhere an Integer is allowed. For example, you could have this in an equation where the value was subsequently stored in an Integer variable or passed to an integer parameter.
- You can access the entire record using just `mouse`. This expression has the Point type. It can be used anywhere a Point can be used. For example, it could be stored in another Point variable, or passed to a Point parameter.
- In an array of records, each element has the records type. In Figure 5.13 the `points` array is storing 3 Point values. This means the `points` is an expression to access the entire array, `points[2]` accesses the 3rd element and therefore has a Point type, and `points[2].x` accessed the `x` value of the 3rd element of the `points` array.

Union Expressions

A **Union** has multiple fields that all give access to the same piece of memory. In effect, the union stores only *one* of the values from its available fields. This allows you to create a type that can be used to store one of a selection of available values.

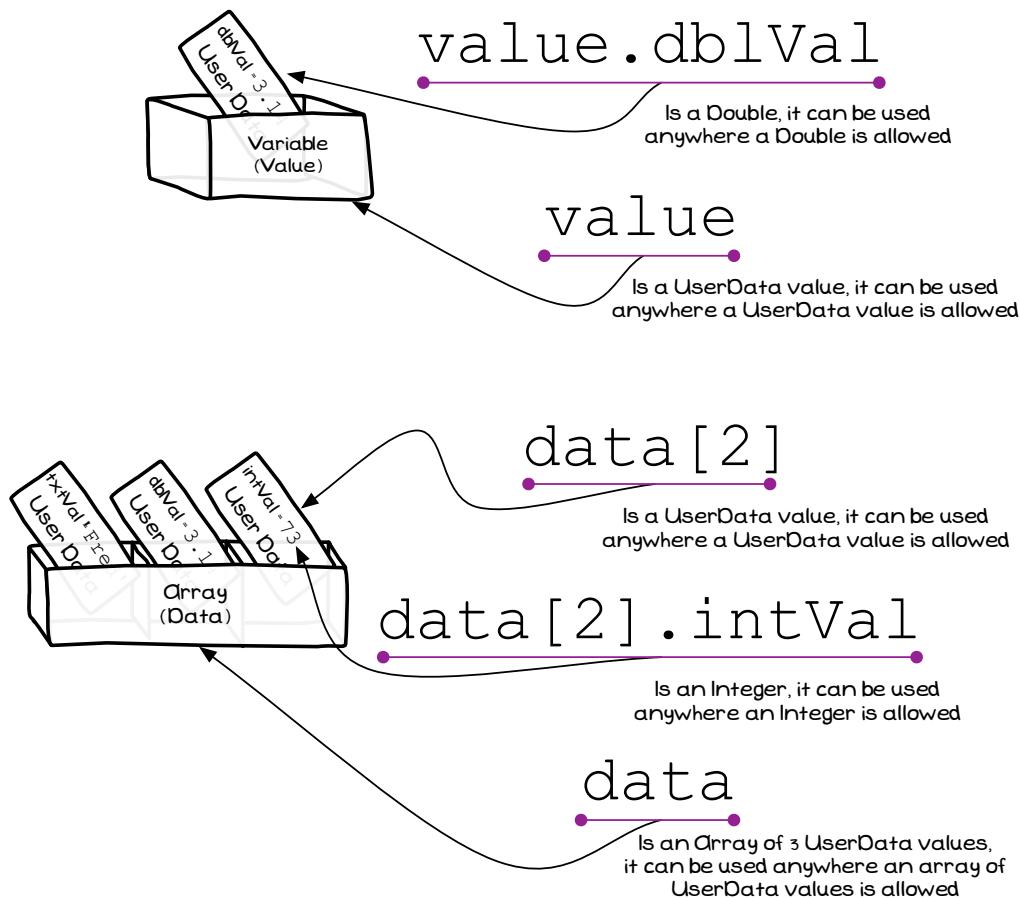


Figure 5.14: A field of a union can be used, or the union can be used in its entirety

Note

- A Union is very similar to a Record, the only difference is that the union only stores one of the field values.
- Figure 5.14 shows an example of a union variable and array.
- The expression `value` gives access to the union stored in the variable. This has a `UserData` type and can be used anywhere a `UserData` value can be accepted.
- The expression `value dblVal` is a `Double` value, and can be used anywhere a `Double` is allowed.
- Accessing a union value from an array is similar to accessing a record value. In Figure 5.14 you can access the array in its entirety using `data`, you can access the first `UserData` value using `data[0]`, and you can access its text value using `data[0].txtVal`.
- When accessing the data in a Union you are responsible for ensuring you read it using the correct field. For example, it is possible to read the data stored in `value` using `value.intVal`. This will not cause any errors during compiling or running, but the value read will be the Integer interpretation of the `Double` value stored in the variable.

Enumeration Expression

The **Enumeration** is the simplest of the custom types to make use of. It defines a list of available options for values of this type. This means that enumerations are just like standard values.

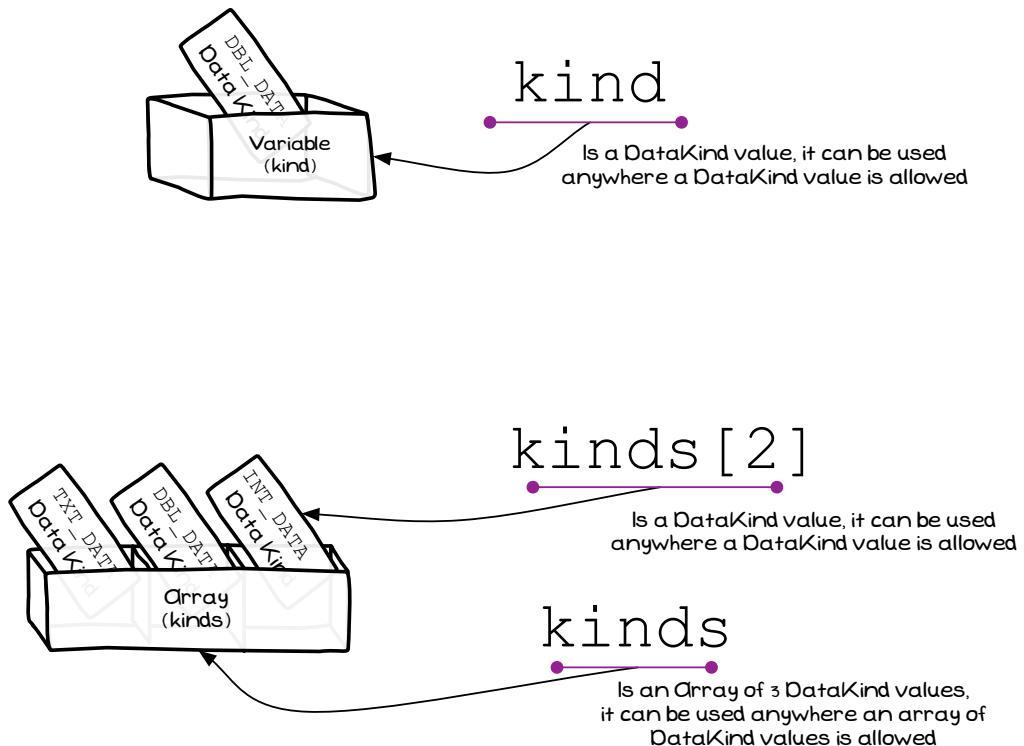


Figure 5.15: You interact with an Enumeration just like other simple data types (Integers, and Doubles for example)

Note

- Accessing a value of an Enumeration type is just like accessing an Integer or Double value.
- In Figure 5.14 the kind variable is storing a Data Kind value. This value can be read from the variable using the variable's name (its **Identifier**).
- The kinds variable is an array of Data Kind values. The expression kinds is an array of Data Kind values and can be used anywhere the array would be accepted. kinds[0] is a Data Kind value and can be used anywhere a Data Kind value can be used.

5.2 Using Custom Types

Designing your own data types means that your code can work with more meaningful values. You can design the data that is stored in the program so that it is organised in ways that will help make the processing simpler.

5.2.1 Designing Small DB

Table 5.1 contains a description of the Small DB program that will be explored in this chapter. This is a small program that will read, store, and output values entered by the user.

Program Description	
Name	<i>Small DB</i>
Description	Reads text (up to 7 characters), integer, and double values from the user, storing them within the program and then outputting their values to the Terminal along with the type being stored in the program. Each value will be stored in a row with a unique identifier that will be assigned by the system. The first value will be assigned the id 0, the second will be assigned the id 1, and so on.

Table 5.1: Description of the Small DB program.

As before, the process to design and implement this program will follow a number of steps:

1. Understand the problem, and get some ideas on the tasks that need to be performed.
2. Choose the artefacts we will create and use
3. Design the control flow for the procedural² artefacts
4. Map these artefacts to code
5. Compile and run the program

5.2.2 Understanding Small DB

This program does not perform any complex functionality, so it does not require much analysis to understand the tasks that need to be performed.

Data identified:

- **Row:** has a unique identifier, and a value.
- **Column Value:** each value in a row is either an integer, a double, or a text value.

Tasks to be performed:

- **Read Row:** The program needs to be able to read a row from the user.
- **Print Row:** After reading the value the program need to output the value to the Terminal.

²The Program, and any Functions and Procedures.

5.2.3 Choosing Artefacts for Small DB

The process of choosing the artefacts for a program involves determining both the structure of the data, as well as the structure of the functions and procedures. In many cases the structure of the data is more important than the structure of the functionality, as getting the data right will make the processing easier. Therefore, the first task is to consider how the data can be structured.

The three main tools that you have for designing the structure of the data in your program are **records**, **unions**, and **enumerations**. A record allows you to create a composite data type that is made up of a number of fields. The union allows you to create a type that stores one kind of data, or another. Finally the enumeration allows you to create a list of available options.

Looking for records in Small DB

The most common of these is the **record**, a **struct** in C. This type allows you to create a single composite value that is composed of a number of related field values. This can be used to model the *entities* in your program. When designing with records you think about the things you want to model, and the data associated with these things.

In the case of the Small DB program there appears to be one kind of record: the **row**. The program needs to store **row** values, where each row has a unique id (an integer), and a data value. These two values *together* make up a row.

This data type will also work nicely with the planned functionality for the program. The code needs to be able to read and print **row** values. This means that this code can accept/return **row** values. The **Print Row** procedure will take in a **row** parameter, and print out its details to the Terminal. The **Read Row** function can read values from the user and return a **row** value to the caller.

Looking for unions in Small DB

The **union**³ is going to be less common than records, but they can offer some useful flexibility when designing your code. The union gives you the ability to have a type that stores one of a selection of types. If your program requires the ability to store different types at the one location then a union is a useful way of modelling this.

Reading the description of the Small DB program there does appear to be the need for a union. Each row needs to be able to store *either* a Integer, a Double, or a text value. Using a union it will be possible to create a type to model this. This can be called the **Column Value** type, and will be the union of these three values.

The great thing about a union is that it stores only one value, the one that you assign to it. This means that it takes only the size needed to store the largest kind of value. In our case the **Column Value** will need space to store an Integer (4 bytes), a Double (8 bytes), or 7 Characters (8 bytes, 7 + 1 overhead). This will only need **8 bytes** of space, as at any one time it can only have one of these values.

Looking for enumerations in Small DB

The last type to look for is the enumeration. This can be used to code a list of available options. Reading through the description of the program there is no really obvious list of options, but on further analysis there may be.

Remember that the Union does not know which value you stored in it. So you would be able to store a value in a Row, but then you would not know which value to read back from the union. This is where an enumeration can come in handy. You can create an enumeration that gives options for each of the kinds of values that the union can store. In this case the options can be **INT_VAL**, **DBL_VAL**, and **TXT_VAL**, and can be called **Data Kind**.

³Variant record in Pascal

The Data Kind enumeration allows you to declare variables that will have one of the available options as its value. This value will need to be stored for each Row in the program, so a field needs to be added to the Row type. This kind field can then store a marker that indicates the kind of value that is stored in the record.

This is a common pattern you will find for working with unions. It is called a **tagged union**. The enumeration value is the **tag** and stores an indicator of the kind of value stored in the union. This is the model behind the implementation of unions in Pascal, but must be manually coded in C.

Overall data design

Table 5.2 shows the structures chosen for the data Small DB program. These provide the data model that will be used by the code to implement the program's logic. Having planned out the structure for the data of this program, the next step will be to design its logic.

Data	Details	
Row	A record/struct with the following fields:	
	id	An Integer value that stores the unique id.
	kind	A Data Kind value indicating the type of data being stored in this record.
	data	Stores Column Value data containing the actual data.
Data Kind	A enumeration with the following options:	
	INT_VAL	Indicates the row is storing an Integer value.
	DBL_VAL	Indicates that the row is storing a Double value.
	TXT_VAL	Indicates that the row is storing a text value.
Column Value	A union that stores one of the following:	
	Int Val	An Integer value.
	Dbl Val	A Double value.
	Txt Val	A text value, with up to seven characters.

Table 5.2: Data Dictionary for Small DB

Reading a Row

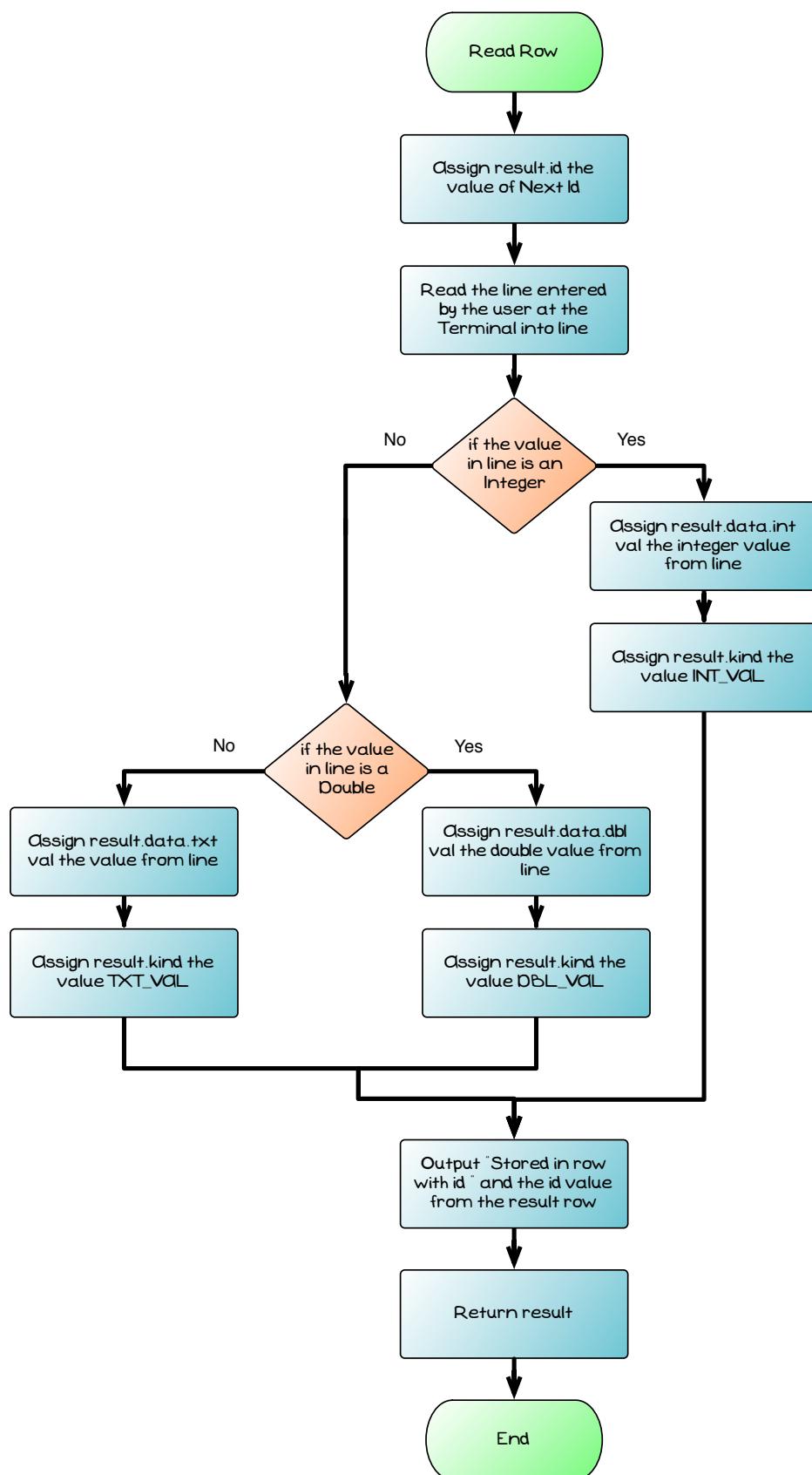
The first piece of functionality to implement can be the Read Row function. This function will be responsible for reading a value from the user, and determining if that value is an integer, double, or string and then storing it in the row with the correct tag value.

To implement this will require the ability to check if the value in a string is an integer or a double. These two tasks can be coded into functions Is Integer and Is Double. Other than this the remaining code just needs to copy values into the fields of the Row that will be returned.

Read Row will need to accept a single parameter, Next Id. This will be the value assigned to the id field of the Row, and will be passed in as this data will be managed elsewhere. At the end of the function, Read Row will return a single Row value. As this is a **record**, it will contain id, kind, and data values.

Figure 5.16 shows the flowchart for the steps in the Read Row function. Notice that all three fields of the Row are assigned values. The id is assigned in the first statement, whereas the data and kind fields are assigned values in the branches of the if statements. All of this data is then returned when the result value is returned.

The **union** is being used when the data field is assigned a value. When the *integer* branch is taken the union is assigned a value via its int val field. When the *double* branch is taken the

**Figure 5.16:** Flowchart for Read Row

union is assigned a value via its `dbl_val` field. When neither of these branches is taken, the `text` value is assigned to the union via its `txt_val` field.

Finally, one of the options from the **enumeration** is stored in the `kind` field of the record alongside the union's value. This means that the `INT_VAL` value is stored in the `kind` field when the *integer* branch is taken, and the `DBL_VAL` value is stored when the *double* branch is taken, and the `TXT_VAL` value is stored when the *text* path is taken.

Figure 5.17 shows three examples of the kinds of values that could be the result of this function when it returns. In each case there are three field values in the row. These are defined in the `Row` record, and include the `id`, the `kind`, and the `data`.

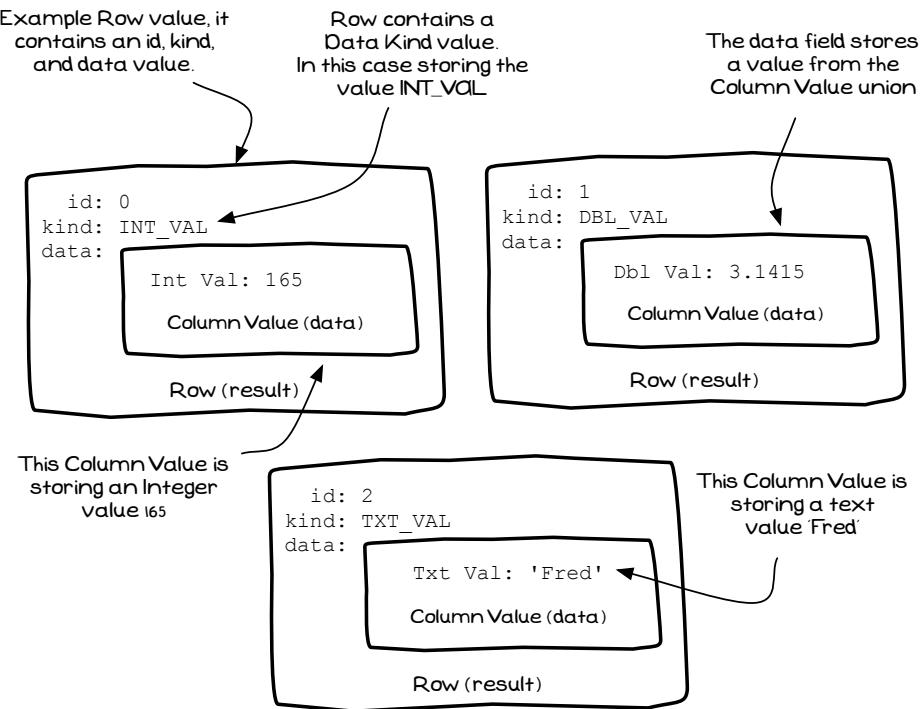


Figure 5.17: Examples of data that could be read into a `Row` value in `Read Row`

Printing a Row

Having read data into a Row it is now possible to output that to the Terminal. The steps required to do this can be coded into a Print Row procedure. The required steps are shown in the flowchart in Figure 5.18.

The Print Row procedure will take a single Row parameter. This parameter will contain the data related of the row that is to be output to the Terminal. The procedure will output the id value and the data values from the Row, using the kind value to determine which field to access from the Column Value union.

The row's id can be output straight away as it is an Integer value. The actual data that needs to be output depends on the kind of value that is stored in the Row. A [Case Statement](#) can be used to select a path based upon the value stored in the row's kind field. The four paths here cater for the three options from the Data Kind enumeration, plus an additional path in case the value in To Print's kind field does not match⁴ one of these. This path would indicate a bug in the software, but should be included just to be safe.

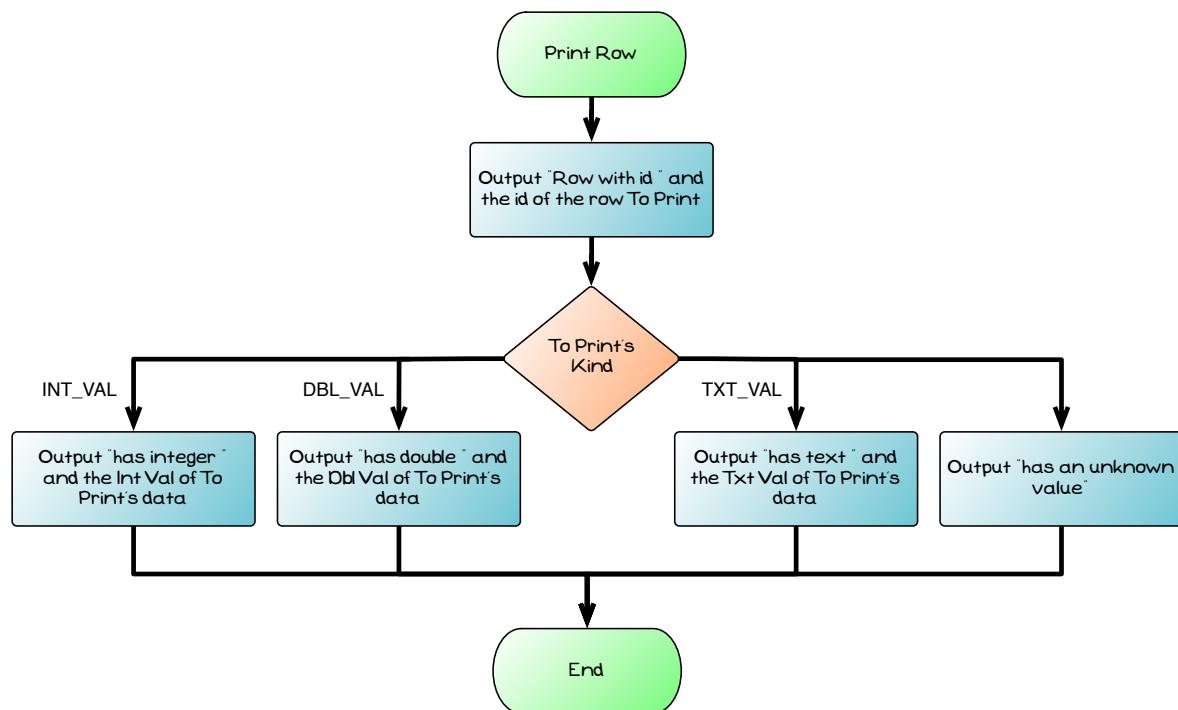


Figure 5.18: Flowchart of the steps needed to print a Row to the Terminal

⁴A enumeration is stored as an Integer value, meaning it is possible to store other values in here.

Overview of Small DB's design

Figure 5.19 shows the structure chart for the design of the Small DB program. The functionality is split between the Read Row function and the Print Row procedure, with Is Integer and Is Double providing useful utility functions to test the data read from the user.

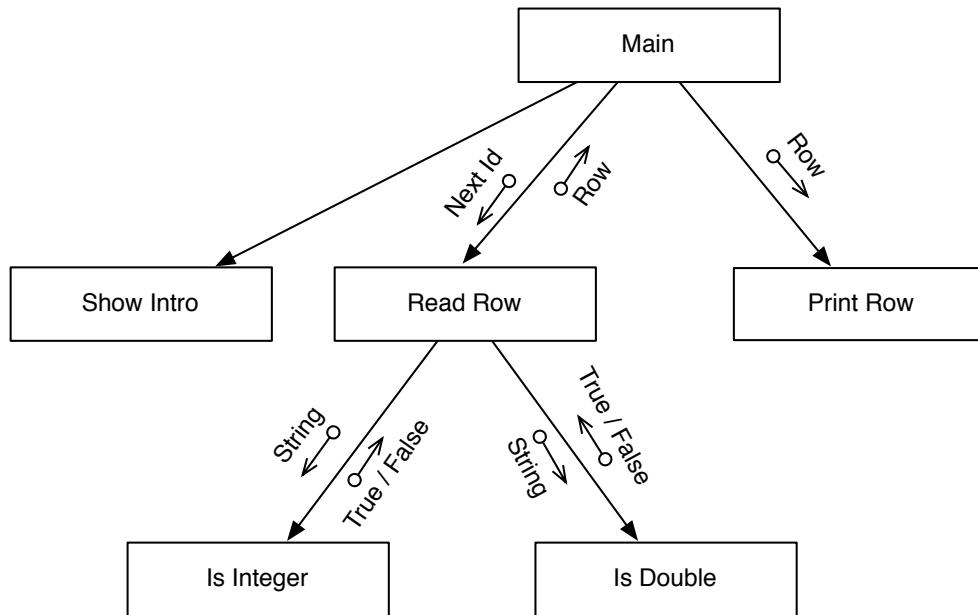


Figure 5.19: Structure chart showing the overview of the Small DB program

The Main procedure will be responsible for storing the data read from the user in an [Array](#) of Row values. The logic in Main will then loop over this array once to read in a value *for each* element of the array, using Read Row to get this value. Main will then loop over the array a second time, this time calling Print Row *for each* element in the array. This will allow Main to read in, and then print out, all of the rows.

A Show Intro procedure has also been added to this design to house the code to show the startup message to the user. This moves this code out of the `main` procedure allowing it to focus on coordinating the tasks involved in working with the array of Row values.

5.2.4 Writing the Code for Small DB

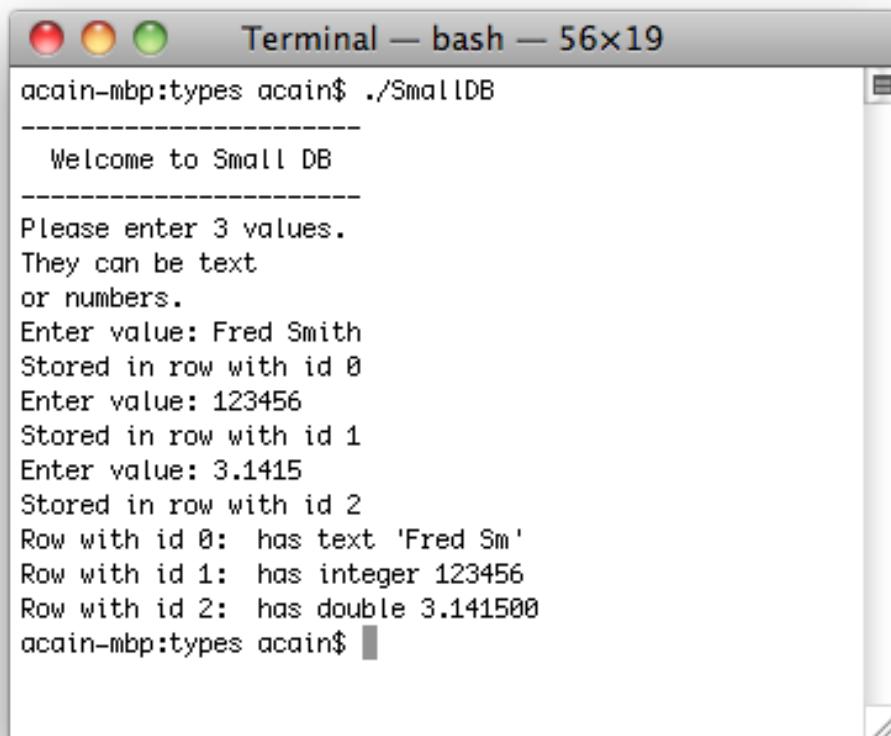
The flowcharts and Pseudocode shown communicate the logic that needs to be coded into the Functions and Procedures of this Program. The following two sections, Section 5.3 Custom Data Types in C and Section 5.4 Custom Data Types in Pascal, contain a description of the syntax needed to code your own types in the C and Pascal programming languages. This information can be used to write the code for the Small DB program, and others.

5.2.5 Compiling and Running Small DB

Remember to code and run this one small piece after the other. For this you could start by writing the `Print Row` procedure, and pass it values that you hard code in the program. This will allow you to experiment with storing different values in the fields of the record and union without having to deal with the user input and testing functions. You can test this by checking that the output matches what you expect based on the values in the `Row`.

Once this is complete the next task would be to work on the `Read Row` function, and its helpers. These have a bit more logic and will require that you test it more carefully. Think about the kind of test data that you can use to check each of the paths through these functions, and use this to check your code as you progress.

Figure 5.20 shows the program in operation.



A screenshot of a Mac OS X terminal window titled "Terminal — bash — 56x19". The window contains the following text:

```
acain-mbp:types acain$ ./SmallDB
-----
Welcome to Small DB
-----
Please enter 3 values.
They can be text
or numbers.
Enter value: Fred Smith
Stored in row with id 0
Enter value: 123456
Stored in row with id 1
Enter value: 3.1415
Stored in row with id 2
Row with id 0: has text 'Fred Sm'
Row with id 1: has integer 123456
Row with id 2: has double 3.141500
acain-mbp:types acain$
```

Figure 5.20: Small DB run from the Terminal, repeated from Figure 5.1

5.3 Custom Data Types in C

5.3.1 Implementing Small DB in C

Section 5.2 of this Chapter introduced the Small DB program. A partial implementation of this program is shown in Listing 5.1. The type definitions, and code are missing the details needed to store and display double values. This program reads a number of rows of data from the user (determined by the DB_SIZE constant). Each row stores a single value, being either a double value, an integer value, or a text value.

```
/* Program: small-db.c */
#include <stdio.h>
#include <strings.h>
#include <stdbool.h>
#include <stdlib.h>
#include <errno.h>
#include <limits.h>

// The number of elements in the row array
#define DB_SIZE 3

// The Column Value union. Stores either an integer, a
// double or 8 (7 + 1) characters.
typedef union {
    int      int_val;
    //todo: Add double as an option
    char     txt_val[8];
} column_value;

// The Data Kind enumeration indicates the kind of data
// stored in a row, matches the options available in the
// Column Value union.
typedef enum {
    INT_VAL,
    //todo: Add double as an option
    TXT_VAL,
    UNK_VAL // an unknown value
} data_kind;

// The Row record/structure. Each row contains an id
// a kind, and some data (a Column Value).
typedef struct {
    int      id;
    data_kind kind;
    column_value data;
} row;

// Trim spaces from the start/end of a string (in place)
// This is passed the string to trim, and the number of characters it contains
void trim(char* text, int n)
{
    int i, j;
    int first_non_space = 0;

    // Get the position of the last character
    int last_char = strlen(text);
    if (last_char > n) last_char = n;

    // Move back one character - past the null terminator
    if (text[last_char] == '\0') last_char--;
}
```

```

// for each character, back from the last char to the first
for(i = last_char; i >= 0; i--)
{
    if (text[i] == ' ') text[i] = '\0'; //replace spaces with null
    else break; // found a non-space so break out of this loop
}

// remember the new position of the last character
last_char = i;

// Search forward from the start...
for(first_non_space = 0; first_non_space < last_char; first_non_space++)
{
    // Break at the first character that is not a space
    if (text[first_non_space] != ' ') break;
}

if (first_non_space > 0)
{
    // Need to copy characters back to start of text...
    // j will track from the start of the text
    j = 0;

    // i will track the index of the non-white space characters
    // starting at the first_non_white space and looping
    // until it gets to the last char (include last char so <= not <)
    for(i = first_non_space; i <= last_char; i++)
    {
        text[j] = text[i];
        j++;
    }
    text[j] = '\0'; // add a null terminator to the end
}
}

// Test if the passed in text refers to an integer
bool is_integer(const char* text)
{
    char * p;
    long val;

    // If the text is empty there is no integer
    if (text == NULL || *text == '\0')
        return false;

    // Test that it can be converted to an integer
    val = strtol (text, &p, 10); // base 10

    // It is an integer if all characters were used in
    // the conversion, and there was no range error
    // and the result is in the
    return *p == '\0' && errno != ERANGE && val <= INT_MAX && val >= INT_MIN;
}

// Test if the passed in text refers to a double
bool is_double(const char* text)
{
    char * p;

    // IF the text is empty there is no double
    if (text == NULL || *text == '\0')

```

```

    return false;

    // Test that it converts to a double
    strtod (text, &p);

    // It is a double if the next character in the text
    // after the conversion is the end of the string
    return *p == '\0';
}

// Clear anything from the input, upto the end of the current line
void clear_input()
{
    scanf("%*[^\n]"); // skip anything is not not a newline
    scanf("%*1[\n]"); // read the newline
}

// Display the intro message.
void show_intro()
{
    printf("%s%s%s%s%d%s%s%s",
        "-----\n",
        " Welcome to Small DB\n",
        "-----\n",
        "Please enter ", DB_SIZE, " values.\n",
        "They can be text\n",
        "or numbers.\n");
}

// Read a row in from the user and return it. The next_id
// is the id number for the newly created row.
row read_row(int next_id)
{
    char line[16] = "", temp[2];
    row result = {0, UNK_VAL, {0}};

    //store the id
    result.id = next_id;

    // Read the value from the user into the line
    printf("Enter value: ");

    // Read at most 15 characrters up to a new line
    // check if only one of the two inputs is matched
    if (scanf("%15[^\\n]%1[\\n]", line, temp) != 2)
    {
        // If the next character was not a newline, read
        // any remaining text and skip it
        clear_input();
    }

    // Remove any leading or trailing spaces
    trim(line, 16);

    // test int first
    if (is_integer(line))
    {
        // read the integer from the line, and store in row
        sscanf(line, "%d", &result.data.int_val);
        // store the kind in the row
        result.kind = INT_VAL;
    }
}

```

```

else if (is_double(line)) // test dbl
{
    // todo: Add handling of double...
}
else
{
    // copy the text into the row (at most 7 + 1 characters)
    strncpy(result.data.txt_val, line, 7); // 7 + 1
    // store the kind in the row
    result.kind = TXT_VAL;
}

printf("Stored in row with id %d\n", result.id);
return result;
}

// Print the row to the Terminal
void print_row(row to_print)
{
    // Print the row's id
    printf("Row with id %d: ", to_print.id);

    // Branch based on the kind, and output the data
    switch (to_print.kind)
    {
        case INT_VAL:
            printf(" has integer %d\n", to_print.data.int_val);
            break;
        // Add double as an option
        case TXT_VAL:
            printf(" has text '%s'\n", to_print.data.txt_val);
            break;
        default:
            printf(" has an unknown value\n");
    }
}

// Entry point
int main()
{
    // Create array of row values
    row db_data[DB_SIZE];
    int i;

    show_intro();

    // For each row in the array
    for (i = 0; i < DB_SIZE; i++)
    {
        // Read the current row's value from the Terminal
        db_data[i] = read_row(i);
    }

    // For each row in the array
    for (i = 0; i < DB_SIZE; i++)
    {
        // Print the row to the Terminal
        print_row(db_data[i]);
    }

    return 0;
}

```

```
}
```

Listing 5.1: C code for the Small DB program**Note**

- `strtol` is a function from the `stdlib.h` header. This converts the input to a integer, and updates `p` to point to the character the conversion got to before ending. Checking that this matches the end of string terminator (`'\0'`) indicates that the input matches the entire string.
- `strtod` is a similar function that converts text to a double. As with `strtol`, this updates `p` to refer to the character it got in the conversion. If this refers to the end of the text then the data does contain a double.
- The `trim` function removes spaces from the start and the end of the input, ensuring that numbers with leading or trailing spaces are still detected as numbers.
- The type definitions must appear before they are used, as a result they commonly appear at the start of the code after the includes.
- `limits.h` gives access to the constants `INT_MAX` and `INT_MIN`, these can be used to check if the integer read is in the range of an integer value.
- `errno.h` gives access to the `errno` global variable that contains an error number when the integer is outside of the range of a long.
- The code in `Read Row` includes code that skips any unwanted input on the current line. This requires two `scanf` calls: the first reading in any characters other than a newline, and the next reading in the newline. This method of skipping unwanted data in input is safe and works across multiple platforms.
- A number of online resources refer to using `fflush` to skip processing input between reads. This **should not** be used. The `fflush` function is used to ensure that output is written to the underlying hardware. The behaviour of this function **is not defined** for input streams. As a result it will not work as expected on all platforms.

5.3.2 C Type Declaration

In C you can declare your own record/structure, union, and enumeration types using the `typedef` declaration. It is also possible to create an alias type declaration, in which you assign a new name to an existing type.

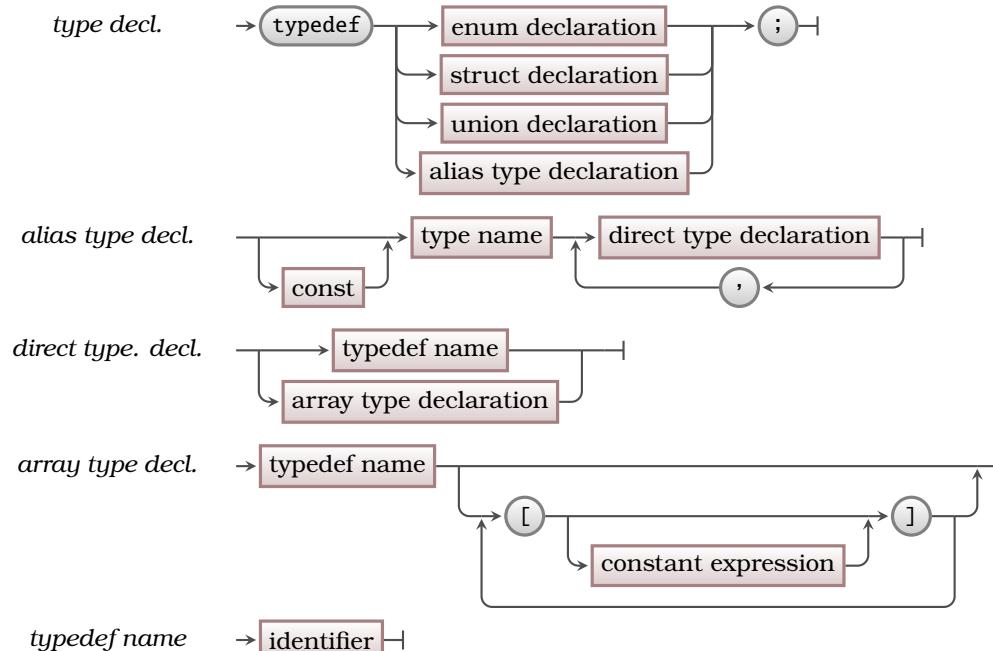


Figure 5.21: C++ Syntax for Type Declarations

Note

- This syntax allows you to declare your own data types.
- The following sections contain the details of declaring different kinds of custom types:
 - Records/structures are shown in [C Record/Structure Declaration](#).
 - Enumerations are shown in [C Enumeration Declaration](#).
 - Unions are shown in [C Union Declaration](#).
- Listing 5.2 shows how to declare alias types. An alias type give a new name to an existing type. Included in this are examples of the following:
 - `number` is an alias for `int`.
 - `five_numbers` is an alias for an array of five `int` values.
 - `my_string` is an alias for a string of 256 characters.
 - `c_string` is an alias for a variable length string.
 - `const_c_string` is an alias for a read-only c-string.
- Notice that these new types can be used to declare variables, arrays, and parameter.
- `grid2` is an array that contains `five_numbers` in each element. Each element of this array is an array of five integer values.

C++

```

/* program: test-alias.c */

// Basic alias - number is an alias for int
typedef int number;

// Array alias - five_numbers is an alias for an array of five integers
typedef int five_numbers[5];

// Basic, and Array, c_string and my_string are alias
typedef char *c_string, my_string[256];

// Declaration of const_c_string, a constant c string
typedef const char *const_c_string;

void test(const_c_string text, number idx, five_numbers data)
{
    printf("%s - %d \n", text, data[idx]);
}

int main()
{
    number var1 = 3;
    five_numbers arr1 = {1, 2, 3, 4, 5};
    five_numbers grid[2] = {{1, 2, 3, 4, 5},
                           {6, 7, 8, 9, 10} };
    my_string name = "Fred";

    printf("%s\n", name);

    test("Hello World", var1, arr1);
    test("Grid 0 - ", var1, grid[0]);
    test("Grid 1 - ", var1, grid[1]);

    return 0;
}

```

Listing 5.2: Testing the alias declarations in C

5.3.3 C Record/Structure Declaration

A record is a type that allows you to store multiple field values. In C this is implemented using a struct. The struct defines a list of fields and their types. The field declarations are similar to other variable declarations, with you specifying the type and then the name of the field.

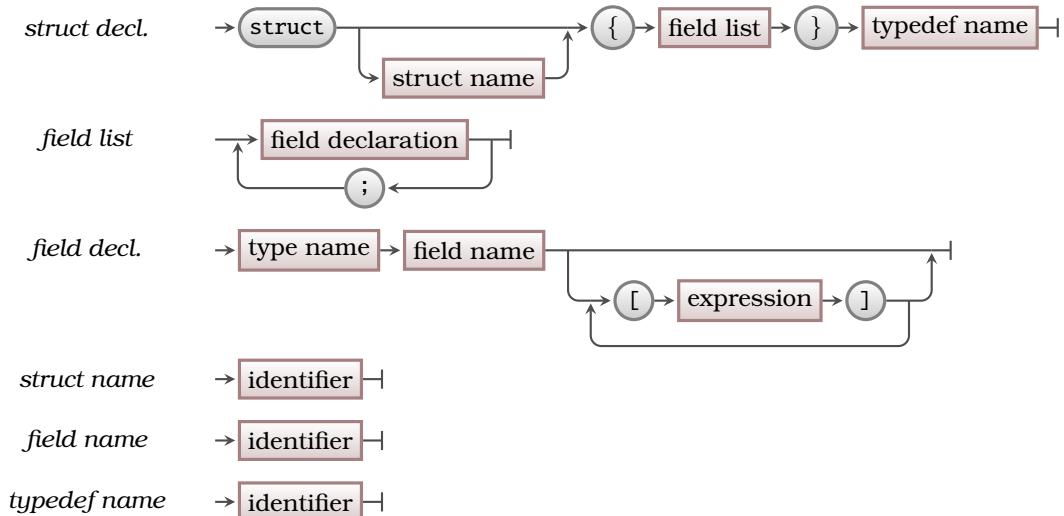


Figure 5.22: C++ Syntax for Structure Declarations

Note

- This is the syntax for declaring your own custom record.
- The declaration starts with **typedef**, which indicates that this is a declaration for a custom type, then **struct** indicating the declaration of a structured record.
- Next comes an *option* **struct name**. This identifier can be used to refer to the struct^a but requires the keyword **struct** before it. For example, the declaration in Listing 5.3 declares a person, or a **struct person_struct**, depending on if you use the **struct name** or the **typedef name**.
- Following this is a **list of fields** between braces (i.e. `{ ... }`). Each field has its own type that may be of any type, including other structures, arrays, standard types, enumerations, and unions.
- Finally the **typedef** ends with the **name** of the type you are declaring.
- Listing 5.3 shows an example of a record in C. The person record contains an array of fifty characters called `name`, and an integer called `age`.
- Remember that the type declaration is creating a new type. After declaring the **struct** you can now create variables of the `person` type.
- Additional fields could be added to the record, and these will be added to all variables declared from this type.

^aThis enables you to declare a struct outside of a **typedef**.

C++

```

/* Program: test-struct.c */
#include <stdio.h>

typedef struct person_struct {
    char    name[50];
    int     age;
} person;

void print_person(person p)
{
    printf("%s (aged %d)\n", p.name, p.age);
}

void clear_input()
{
    scanf("%*[^\n]");
    scanf("%*1[\n]");
}

person get_person(const char *prompt)
{
    person result = { "", 0 };

    printf("%s\n", prompt);

    printf("Enter name: ");
    scanf("%49[^\\n]", result.name);

    printf("Enter age: ");
    scanf(" %d", &result.age);
    clear_input();

    return result;
}

int main()
{
    person me = {"Fred Smith", 20};
    person friends[2];

    friends[0] = get_person("Enter details for a friend.");
    friends[1] = get_person("Enter details for another friend.");

    print_person(me);
    print_person(friends[0]);
    print_person(friends[1]);

    return 0;
}

```

Listing 5.3: C for working with a structure

5.3.4 C Enumeration Declaration

An enumeration is a list of available options for the type. A variable of an enumeration type can store one of these values. In C you declare the enumeration using a `typedef`, and list the available constants within the braces.

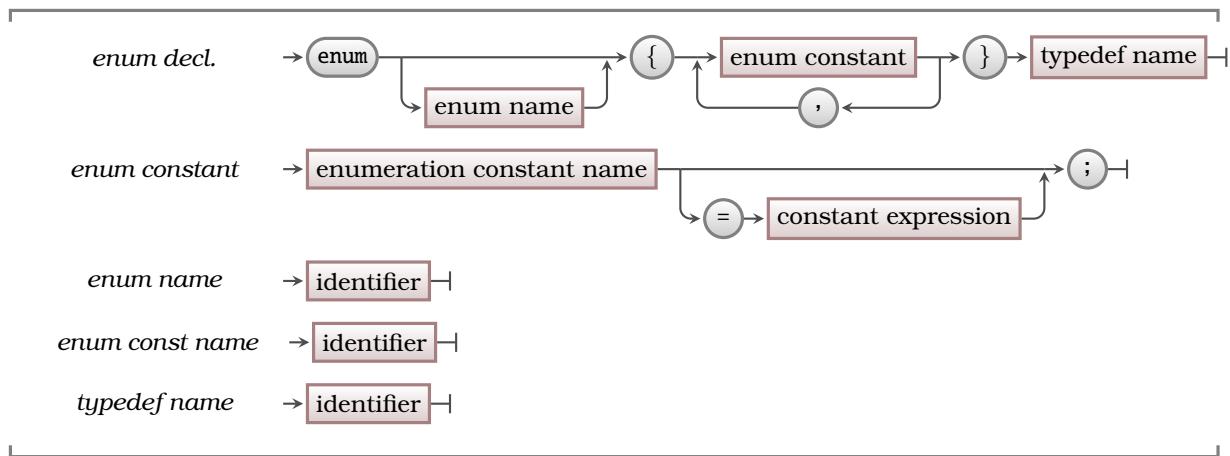


Figure 5.23: C++ Syntax for Enumeration Declarations

```
C++ /* Program: test-enum.c */

typedef enum
{
    SAFE,
    DANGER,
    EXTREME_DANGER
} warning_level;

int main()
{
    warning_level situation = SAFE;

    switch (situation)
    {
        case SAFE: printf("Safe\n"); break;
        case DANGER: printf("Danger!\n"); break;
        case EXTREME_DANGER: printf("Run!\n"); break;
        default: printf("Unknown...\\n");
    }
    return 0;
}
```

Listing 5.4: C code illustrating enumeration declarations.

- This is the syntax that allows you to declare an enumeration in C.
 - The declaration starts with `typedef`, which indicates that this is a declaration for a custom type, and `enum` indicating that this custom type is an enumeration.
 - Following this is a list of constants between braces (i.e. `{ ... }`). Each constant must have a unique name, and the by convention is all UPPERCASE.
 - Finally the `typedef` ends with the name of the type you are declaring.

5.3.5 C Union Declaration

A union declaration is very similar to a record/structure declaration. The difference is in the way these are represented in memory. The structure stores a value for each field, whereas the union only stores a single value, allowing you to choose which of the fields has a value.

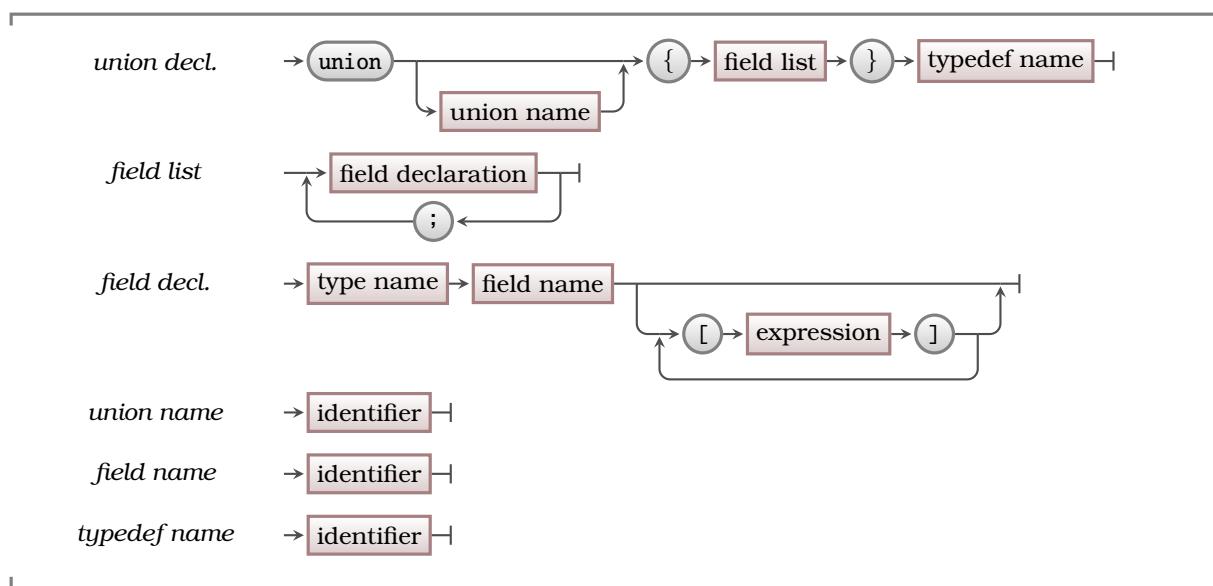


Figure 5.24: C++ Syntax for Union Declarations

C++

```

/* program: test-union.c */

typedef unsigned char byte;

typedef union color_union
{
    unsigned int value;
    byte components[4];
} color;

int main()
{
    color red;
    red.components[0] = 255; // r
    red.components[1] = 0;   // g
    red.components[2] = 0;   // b
    red.components[3] = 255; // a

    // Warning: Unsafe use, assumes knowledge of underlying layout
    // of the integer that will not work on all CPUs.
    printf("%x\n", red.value);
    return 0;
}

```

Listing 5.5: C code demonstrating union declaration and use

Note

- This is the syntax for declaring your own custom union.
- The declaration starts with **typedef**, which indicates that this is a declaration for a custom type, then **union** indicating the declaration of a union.
- Next comes an *option* **union name**. This identifier can be used to refer to the union^a but requires the keyword `union` before it. For example, the declaration in Listing 5.5 declares a `color`, or a `union color_union`, depending on if you use the *union name* or the *typedef name*.
- Following this is a **list of fields** between braces (i.e. `{ ... }`). Each field has its own type that may be of any type, including other structures, arrays, standard types, enumerations, and unions. This is the same as with a [C Record/Structure Declaration](#), except that when it is stored in memory only one of these fields will have a value.
- Finally the `typedef` ends with the **name** of the type you are declaring.
- Listing 5.5 shows an example of a union in C. The `color` union contains *either* an unsigned integer called `value`, or an array of four bytes called `components`.
- Remember that the type declaration is creating a new type. After declaring the union you can now create variables of the `color` type.
- Please note that when you store a value in a union via one field, that is the field that has a reliable value. If you access the union's value via another field the results are **unreliable**. This behaviour is demonstrated in Listing 5.5 where a value is stored in the union via the `components` field, but accessed via the `value` field. This should be avoided in 'real' code as it relies upon an understanding of how the data is being laid out in memory which can differ by platform, and can be a source of hard to locate issues.
- For an example of good usage see Listing 5.1.

^aThis enables you to declare a union outside of a `typedef`.

5.3.6 C Variable Declaration (with Types)

In C you can declare variables from any of the types that you have declared. The *type name* in the variable's declaration can contain the names of the types that you declare using C's `typedef` declaration. See [C Type Declaration](#).

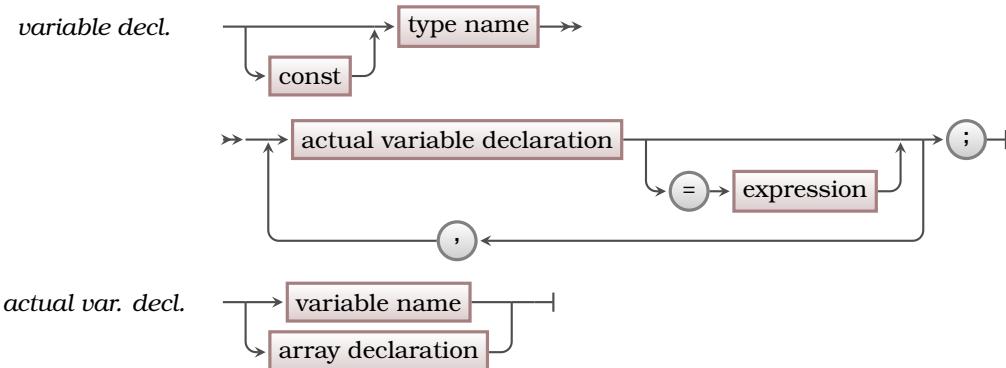


Figure 5.25: C++ Syntax for Variable Declarations

Note

- This shows the syntax for declaring variables that use the types you have created.
- In C the type declaration must appear before you can use the type to declare variables.
- The code in Listing 5.6 demonstrates how variables can be declared using custom defines records, unions, and enumerations.
- In C it is possible to initialise a record/structure using similar notation to that used to initialise arrays (see [C Array Declaration](#)). This uses braces (i.e. `{...}`) to surround the expression. Within the braces you place one value for each field, in order. These values are then used to initialise the fields of the variable. See the declaration of `var2` in Listing 5.6.
- Unions can also have their values initialised. This also uses the brace notation, but only the first declared field can be initialised.

C++

```

#include <stdio.h>
#include <strings.h>

typedef struct
{
    int      field1;
    double   field2;
    char     field3[10];
} my_struct;

typedef enum
{
    OPT_1,
    OPT_2
} my_enum;

typedef union
{
    long long  long_val;
    double    double_val;
} my_number;

int main()
{
    // Declare variables, structures, enums, and unions.
    my_struct var1;
    my_struct var2 = {1, 3.1415, "Fred"};
    my_struct var3 = var2;

    my_enum var4 = OPT_1;

    my_number var5;

    // Play with record/structures
    var1.field1 = 2;
    var1.field2 = 6.5;
    strncpy(var1.field3, "Wilma", 9);

    printf("%s %d %f %s\n", "var1", var1.field1, var1.field2, var1.field3);
    printf("%s %d %f %s\n", "var2", var2.field1, var2.field2, var2.field3);
    printf("%s %d %f %s\n", "var3", var3.field1, var3.field2, var3.field3);

    // Play with enums
    printf("Int value of var4 is %d\n", var4);
    var4 = OPT_2;
    printf("Int value of var4 is %d\n", var4);

    // Play with unions
    var5.long_val = 123456;
    printf("%lld\n", var5.long_val);

    var5.double_val = 3.1415;
    printf("%f\n", var5.double_val); //note: long_val has been written over...

    return 0;
}

```

Listing 5.6: C code illustrating variable declaration and use with Custom Types

5.4 Custom Data Types in Pascal

5.4.1 Implementing Small DB in Pascal

Section 5.2 of this Chapter introduced the Small DB program. A partial implementation of this program is shown in Listing 5.7. The type definitions, and code are missing the details needed to store and display double values. This program reads a number of rows of data from the user (determined by the DB_SIZE constant). Each row stores a single value, being either a double value, an integer value, or a text value.

```

program SmallDb;
uses SysUtils;

// The number of elements in the row array
const DB_SIZE = 3;

type
    // The Data Kind enumeration indicates the kind of data stored in a row, matches
    // the options available in the Column Value union.
    DataKind = ( INT_VAL, DBL_VAL, TXT_VAL );

    ColumnValue = record // The Column Value union.
        case kind: DataKind of
            INT_VAL: ( intValue: Integer ); // Stores either an Integer,
            // todo: add double as an option
            TXT_VAL: ( txtVal: String[7]; ); // 7 characters. (8 bytes)
    end;

    Row = record           // The Row record/structure.
        id: Integer;      // Each row contains an id
        data: ColumnValue; // and a single column
    end;

procedure ShowIntro(); // Display the intro message.
begin
    WriteLn('-----');
    WriteLn(' Welcome to Small DB');
    WriteLn('-----');
    WriteLn('Please enter ', DB_SIZE, ' values.');
    WriteLn('They can be text');
    WriteLn('or numbers.');
end;

// Read a row in from the user and return it.
function ReadRow(nextId: Integer): Row;
var
    line: String = '';
begin
    //store the id
    result.id := nextId; // The nextId is the id number for the newly created row

    Write('Enter value: ');
    ReadLn(line); // Read the value from the user into the line

    // test int first, read the Integer from the line, and store in row
    if TryStrToInt(line, result.data.intValue) then
    begin
        result.data.kind := INT_VAL; // was an Integer, so set kind to INT_VAL
    end
    else if TryStrToFloat(line, result.data dblVal) then // test dbl

```

CHAPTER 5. CUSTOM DATA TYPES

```

        //todo: add double as an option...
else
begin
    // is not Integer or Double, so its text. Copy the text into the row
    result.data.txtVal := line;
    result.data.kind := TXT_VAL;      // was text, so set kind to TXT_VAL
end;

    WriteLn('Stored in row with id ', result.Id);
end;

// Print the row to the Terminal
procedure PrintRow(toPrint: row);
begin
    WriteLn('Row with id ', toPrint.Id);      // Print the row's id

    // Branch based on the kind, and output the data
    case (toPrint.data.kind) of
        INT_VAL:   WriteLn(' has Integer ', toPrint.data.intValue);
        // DBL_VAL: todo: add double as an option
        TXT_VAL:   WriteLn(' has text ', toPrint.data.txtVal);
        else       WriteLn(' has an unknown value ');
    end;
end;

// Entry point
procedure Main();
var
    dbData: array[0..DB_SIZE - 1] of Row;    // Create array of Column Values
    i: Integer;
begin
    ShowIntro();

    // For each row in the array
    for i := Low(dbData) to High(dbData) do
        // Read the current row's value from the Terminal
        dbData[i] := ReadRow(i);

    // For each row in the array
    for i := Low(dbData) to High(dbData) do
        // Print the row to the Terminal
        PrintRow(dbData[i]);
end;

begin
    Main();
end.

```

Listing 5.7: Pascal code for the Small DB program**Note**

- The type definitions must appear before they are used, as a result they commonly appear at the start of the code.
- The column's kind has been moved into the ColumnValue record.

5.4.2 Pascal Type Declaration

In Pascal you can declare your own record/structure, union, and enumeration types using a type declaration. It is also possible to create an alias type declaration, in which you assign a new name to an existing type.

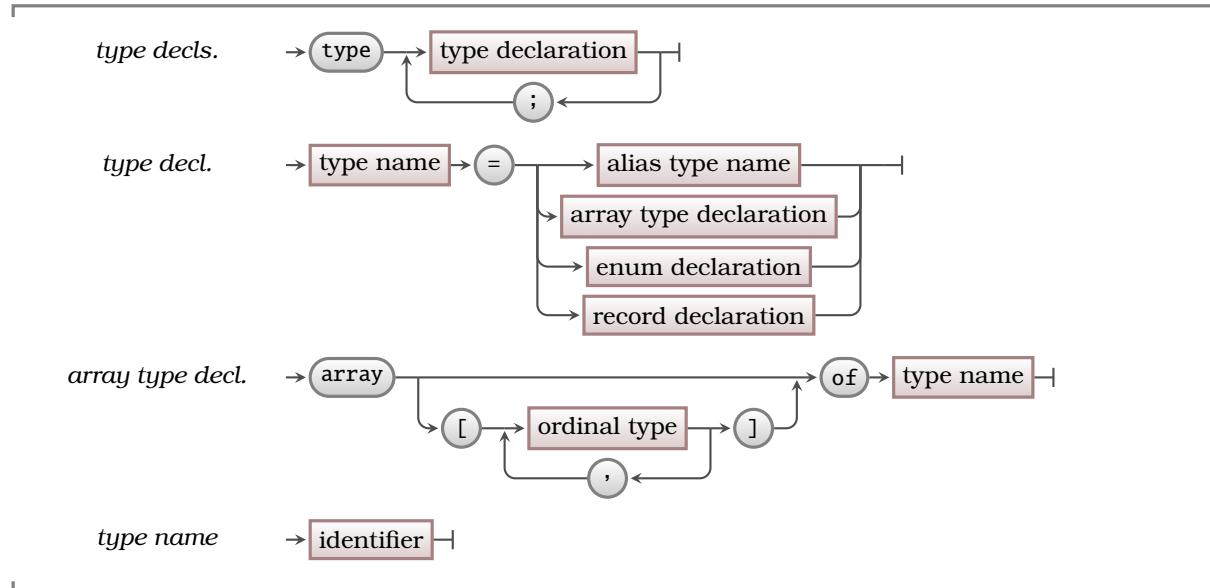


Figure 5.26: Pascal Syntax for Type Declarations

Note

- This syntax allows you to declare your own data types.
- The following sections contain the details of declaring different kinds of custom types:
 - Records/structures and unions are shown in [Pascal Record Declaration](#).
 - Enumerations are shown in [Pascal Enumeration Declaration](#).
- Listing 5.8 shows how to declare alias types. An alias type give a new name to an existing type. Included in this are examples of the following:
 - Number is an alias for Integer.
 - FiveNumbers is an alias for an array of five int values.
- Notice that these new types can be used to declare variables, arrays, and parameter.
- grid2 is an array that contains FiveNumbers in each element. Each element of this array is an array of five integer values.

Pascal

```
program TestAlias;

type
    // Basic alias - number is an alias for int
    Number = Integer;

    // Array alias - FiveNumbers is an alias for an array of five integers
    FiveNumbers = array [0..4] of Integer;

procedure Test(idx: Number; data: FiveNumbers);
begin
    WriteLn(' -> ', data[idx]);
end;

procedure Main();
var
    var1: Number = 3;
    arr1: FiveNumbers = (1, 2, 3, 4, 5);
    grid: array [0..1] of FiveNumbers =
        ((1, 2, 3, 4, 5),
         (6, 7, 8, 9, 10));
begin
    Test(var1, arr1);
    Test(3, grid[0]);
    Test(var1, grid[1]);
end;

begin
    Main();
end.
```



Listing 5.8: Testing the alias declarations in Pascal

5.4.3 Pascal Record Declaration

A record is a type that allows you to store multiple field values. In Pascal the record defines a list of fields and their types. The field declarations are similar to other variable declarations, with you specifying the name and then the type of the field.

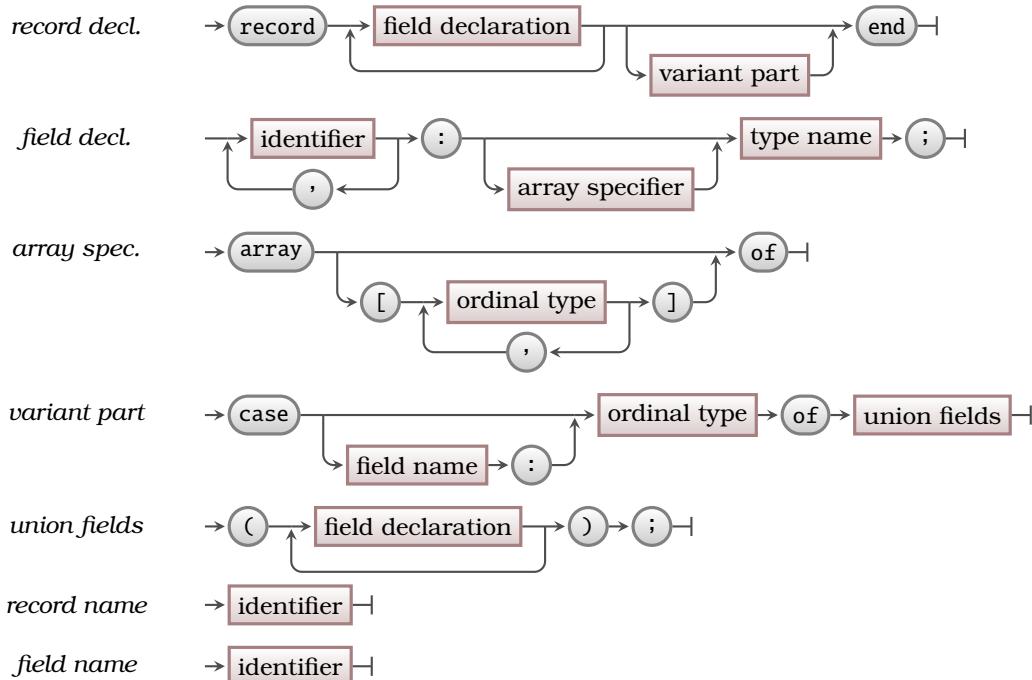


Figure 5.27: Pascal Syntax for Record Declarations

Note

- This is the syntax for declaring your own custom record.
- The declaration starts with **typedef**, which indicates that this is a declaration for a custom type, then **struct** indicating the declaration of a structured record.
- Next comes an *option struct name*. This identifier can be used to refer to the struct^a but requires the keyword **struct** before it. For example, the declaration in Listing 5.3 declares a person, or a struct **person_struct**, depending on if you use the **struct name** or the **typedef name**.
- Following this is a **list of fields** between braces (i.e. `{...}`). Each field has its own type that may be of any type, including other structures, arrays, standard types, enumerations, and unions.
- Finally the **typedef** ends with the **name** of the type you are declaring.
- Listing 5.3 shows an example of a record in C. The person record contains an array of fifty characters called `name`, and an integer called `age`.
- Remember that the type declaration is creating a new type. After declaring the **struct** you can now create variables of the `person` type.
- Additional fields could be added to the record, and these will be added to all variables declared from this type.

^aThis enables you to declare a struct outside of a **typedef**.

Pascal

```

program TestRecord;

type
  Person = record
    name: String;
    age: Integer;
  end;

procedure PrintPerson(const p: Person);
begin
  WriteLn(p.name, ' (aged ', p.age, ')');
end;

function GetPerson(prompt: String): Person;
begin
  WriteLn(prompt);

  WriteLn('Enter name: ');
  ReadLn(result.name);

  WriteLn('Enter age: ');
  ReadLn(result.age);
end;

procedure Main();
var
  me: Person;
  friends: array [0..1] of Person;
begin
  me.name := 'Fred Smith';
  me.age := 20;

  friends[0] := GetPerson('Enter details for a friend.');
  friends[1] := GetPerson('Enter details for another friend.');

  PrintPerson(me);
  PrintPerson(friends[0]);
  PrintPerson(friends[1]);
end;

begin
  Main();
end.

```

**Listing 5.9:** Pascal for working with a record

Pascal Variant Records (Unions)

Pascal records can include a *variant* part, which stores a single value from a list of field options. The variant part comes at the end of the record, starting with the `case` keyword. The variant parts is matched with a ordinal type (e.g. enumeration) that can also be stored as a *tag* field, indicating which field option is currently storing a value. See Listing 5.10 for an example.

Pascal

```

program TestUnion;

type
  MyDataOptions = ( IS_INTEGER, IS_FLOAT, IS_TEXT );

  MyUnionType = record
    otherField: String;
    case kind: MyDataOptions of
      IS_INTEGER: ( asInt: Integer );
      IS_FLOAT: ( asFloat: Single );
      IS_TEXT: ( asString: String[4] );
  end;

procedure Main();
var
  data: MyUnionType; // create data of this type
begin
  WriteLn('data is ', SizeOf(data), ' bytes (stores three data values...)');

  data.otherField := 'a standard field...';
  data.kind := IS_FLOAT; //should match value stores (programmer managed!)
  data.asFloat := 10.0; //can store as Integer/Float/String based on field used

  case data.kind of
    IS_INTEGER: WriteLn(data.asInt);
    IS_FLOAT: WriteLn(data.asFloat:4:2);
    IS_TEXT: WriteLn(data.asString);
  end;
end;

begin
  Main();
end.

```

Listing 5.10: Pascal variant record (union)

Note

- The variant part of the record stores a **single** value.
- MyUnionType stores three values: `otherField`, `kind`, and one value from the variant part.
- The `kind` field of the MyUnionType record indicates which of the three field options is storing a value. This is known as a *tag*.
- You have to manage the *tag* field yourself.



5.4.4 Pascal Enumeration Declaration

An enumeration is a list of available options for the type. A variable of an enumeration type can store one of these values. In Pascal you declare the enumeration by listing the available constants within parenthesis.

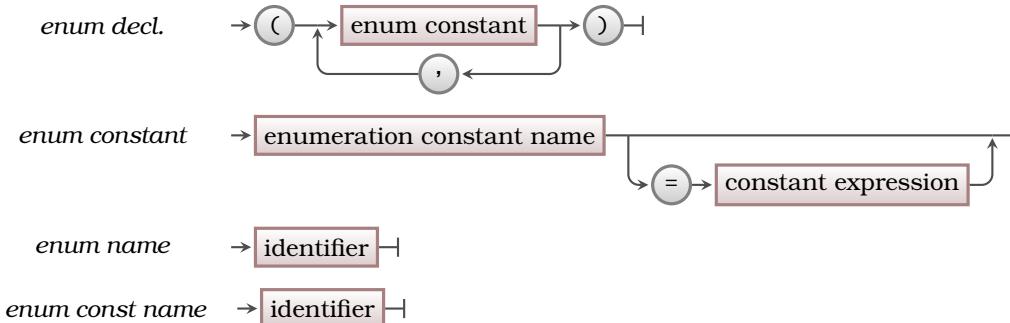


Figure 5.28: Pascal Syntax for Enumeration Declarations

Pascal

```

program TestEnum;

type
  WarningLevel = ( SAFE, DANGER, EXTREME_DANGER );

procedure Main();
var
  situation: WarningLevel = SAFE;
begin
  case situation of
    SAFE: WriteLn('Safe');
    DANGER: WriteLn('Danger!');
    EXTREME_DANGER: WriteLn('Run!');
    else WriteLn('Unknown...');
  end;
end;

begin
  Main();
end.
  
```

Listing 5.11: Pascal code illustrating enumeration declarations.

Note

- This is the syntax that allows you to declare an enumeration in Pascal.
- The enumeration type contains a list of constants between parenthesis (i.e. (...)). Each constant must have a unique name, and the convention is all UPPERCASE.

5.5 Understanding Custom Types

Custom data types offer you the opportunity to define how data is organised in your program. You can create records that contain a number of fields, unions that store a single field value, and enumerations that define their own list of values. To help you understand these concepts better the following illustrations demonstrate how these values are stored in the variables in your code.

5.5.1 Understanding Read Row

Section 5.2.1 Designing Small DB described the design of a Small DB program. The program allows the user to enter some values that were then stored in variables in the program, making use of records, unions, and enumerations. The design for this program included a number of functions and procedures, one of which was the Read Row function. This function was responsible for reading a row value from the user and returning it in a Row record. The flowchart for this function is shown in Figure 5.29, which is a repeat of Figure 5.16.

The following illustrations will show this code running to read in a value from the user. This will demonstrate how the computer stores values in record, union, and enumeration variables.

The illustrations will show the following:

1. [Code is loaded for Small DB](#)
2. [Read Row is called to read in row with id 0](#)
3. [Step 1 stores the value 0 in result's id field](#)
4. [A double value is entered by the user](#)
5. [The double data is stored in the row](#)
6. [The result row is returned to Main](#)
7. [This process is repeated for each element of the array](#)

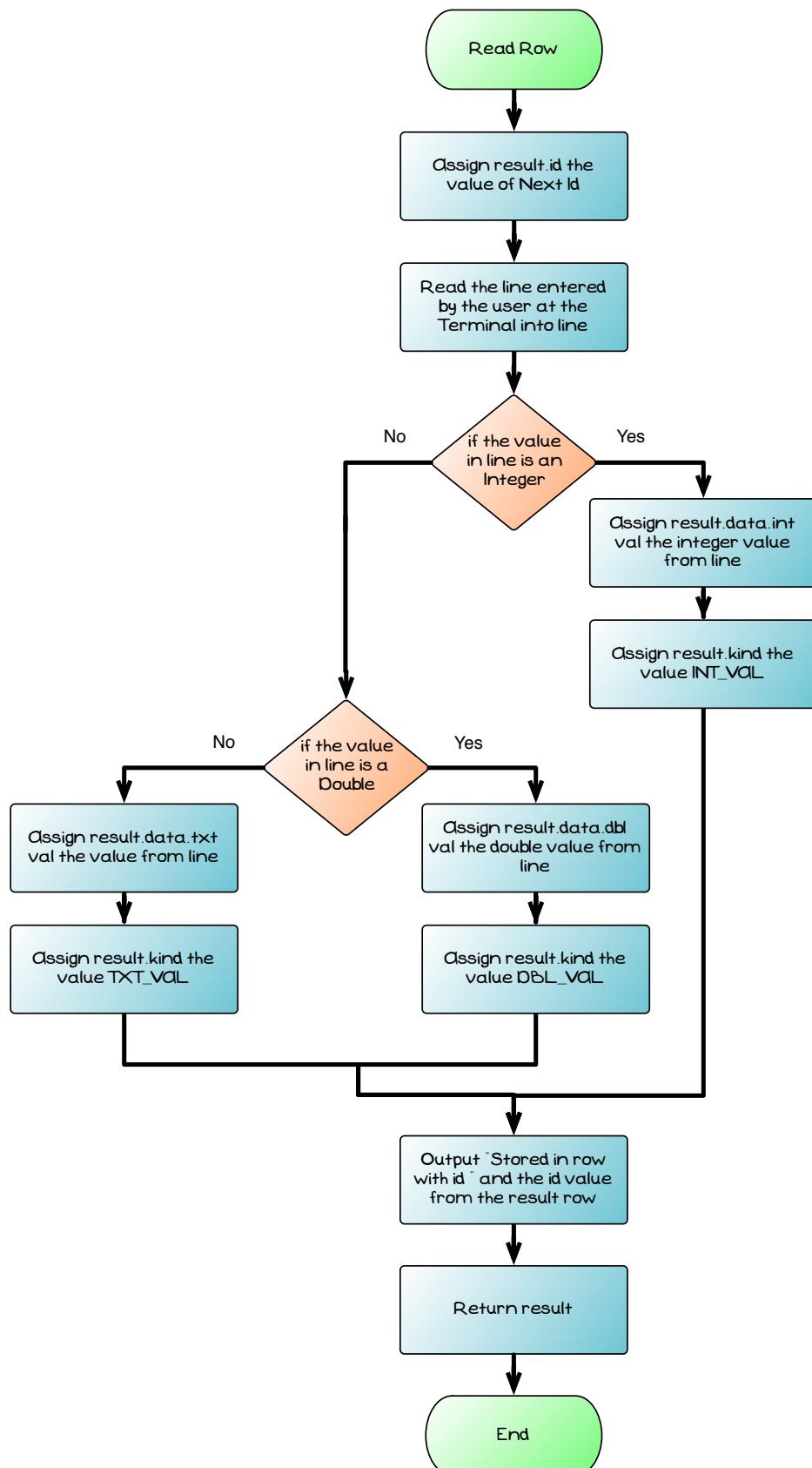


Figure 5.29: Flowchart for Read Row, from Figure 5.16

Code is loaded for Small DB

When the program starts its code is loaded into memory and its `main` procedure is started.

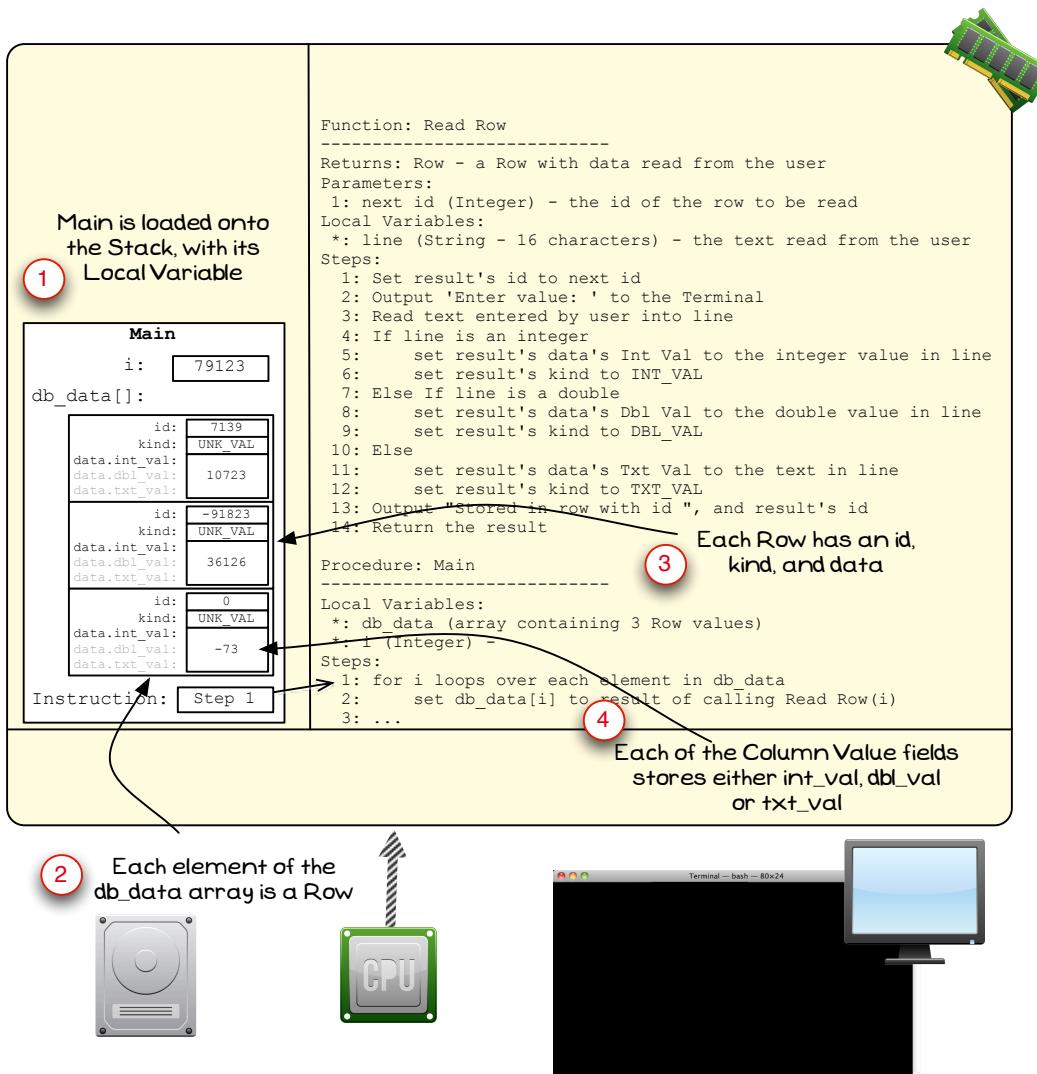


Figure 5.30: When the program starts Main allocates space for its local variables, including the array

Note

- In Figure 5.30 the indicated areas show the following:
 - The Program starts and `Main` is loaded onto the stack, allocating space for its local variables.
 - The `db_data` array is allocated space to store its values. Each element of the array has the fields declared in the `Row` record structure.
 - Each `Row` has an `id`, `kind`, and `data` value.
 - Each of these data values has *either* a `int_val`, a `dbl_val` or a `txt_val`.
- Notice that the values in the array are allocated next to each other.
- In the illustration the `Row`'s `data` field will have only one of its fields highlighted, indicating which field is currently stored in the union.

Read Row is called to read in row with id 0

The Read Row function is called to read a row value from the user. This will check what the user has entered and store an appropriate value in the Row it returns.

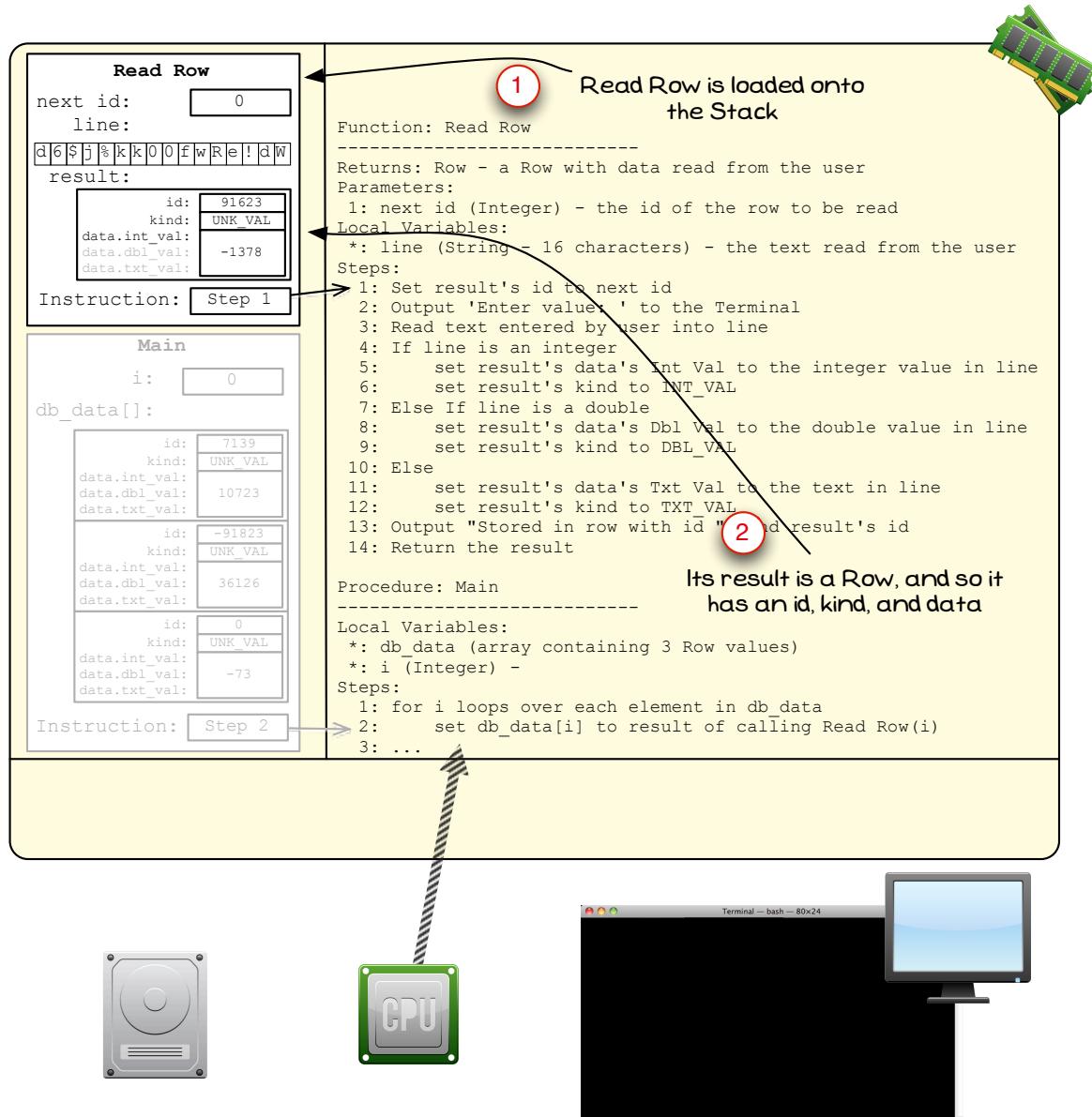
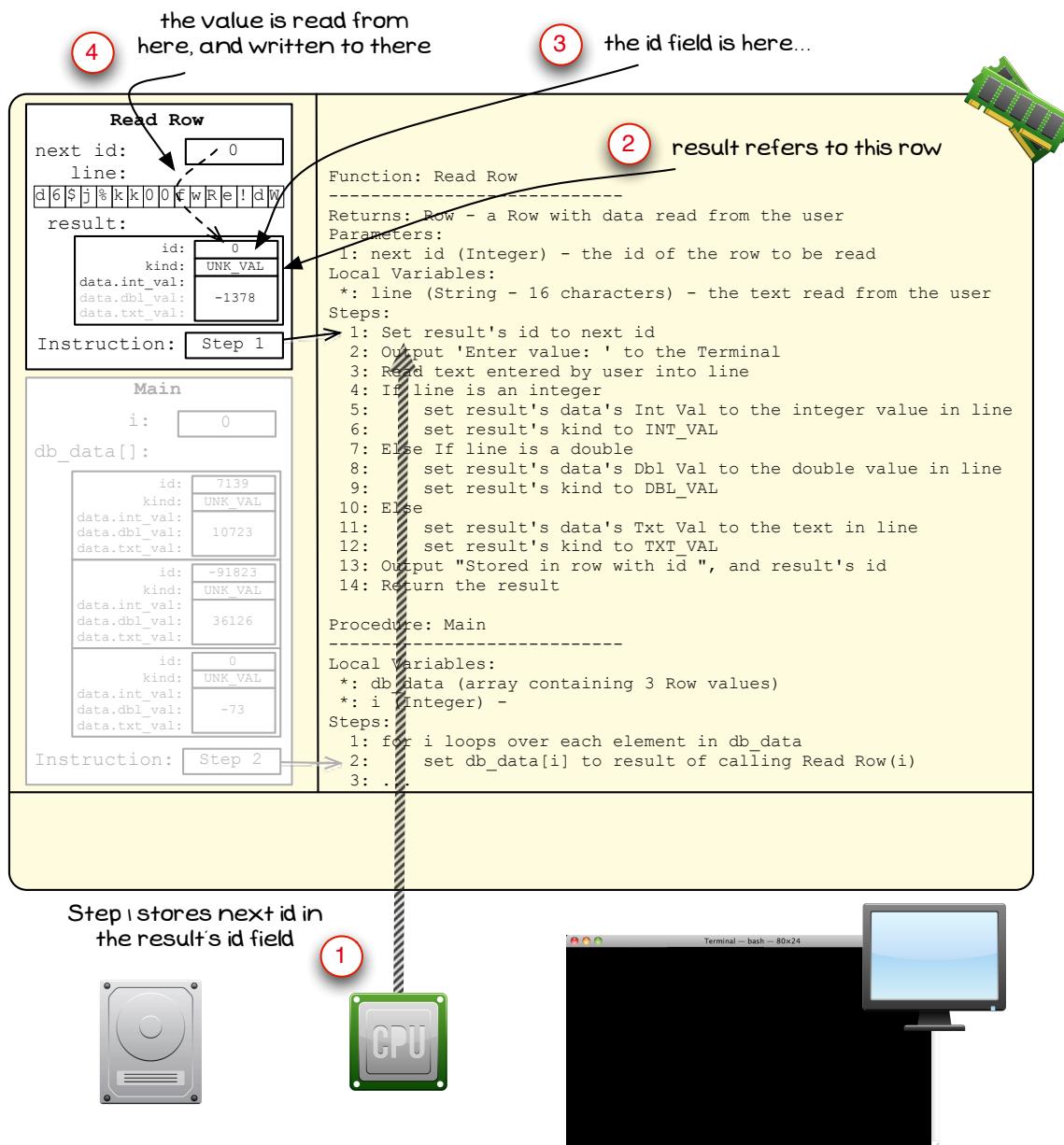


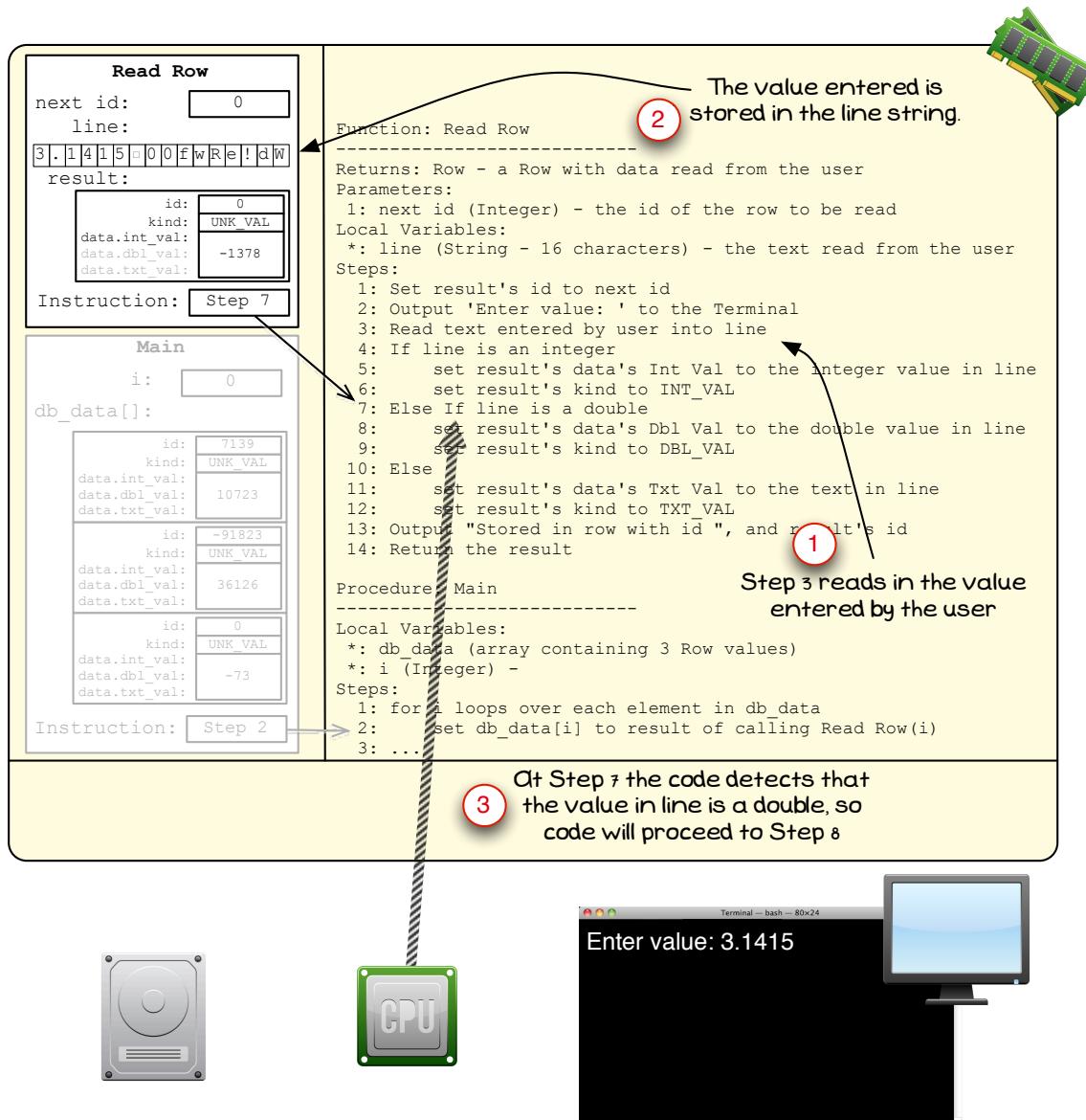
Figure 5.31: At step 2 Main calls Read Row, getting it to read in the i^{th} row from the user

Note

- In Figure 5.31 the indicated areas show the following:
 - When `Read Row` is called it is allocated space on the stack.
 - `Read Row` will be returning a `Row` value, so its result will have `id`, `kind`, and `data` values as these are what is specified in the `Row` record's definition.
- Each row will have the same kind of data stored within it. The details of this are specified in the `Row` record's definition.

Step 1 stores the value 0 in result's id field**Figure 5.32:** Step 1 of Read Row stores a value in result's id**Note**

- In Figure 5.32 the indicated areas show the following:
 - Step 1 of Read Row assigns a value to `result.id`.
 - In this code `result` refers to this variable in `Read Row`.
 - The `id` part then refers to this field.
 - As a result the value from the `next_id` parameter is read, and its value is assigned to `result.id`.
- Each part of `result.id` refers to a different kind of data.
- `result` is a row, this has `id`, `kind`, and `data` fields.
- `result.id` is an Integer, it is the `id` field of the `result Row`.

A double value is entered by the user**Figure 5.33:** A double value is entered by the user, so the code must store the double in the row**Note**

- In Figure 5.33 the indicated areas show the following:
 - At Step 3 the computer reads the text entered by the user.
 - The value read is stored in the `line` variable.
 - At Step 7 the code determines that the data is a double, and execution will proceed to step 8.

The double data is stored in the row

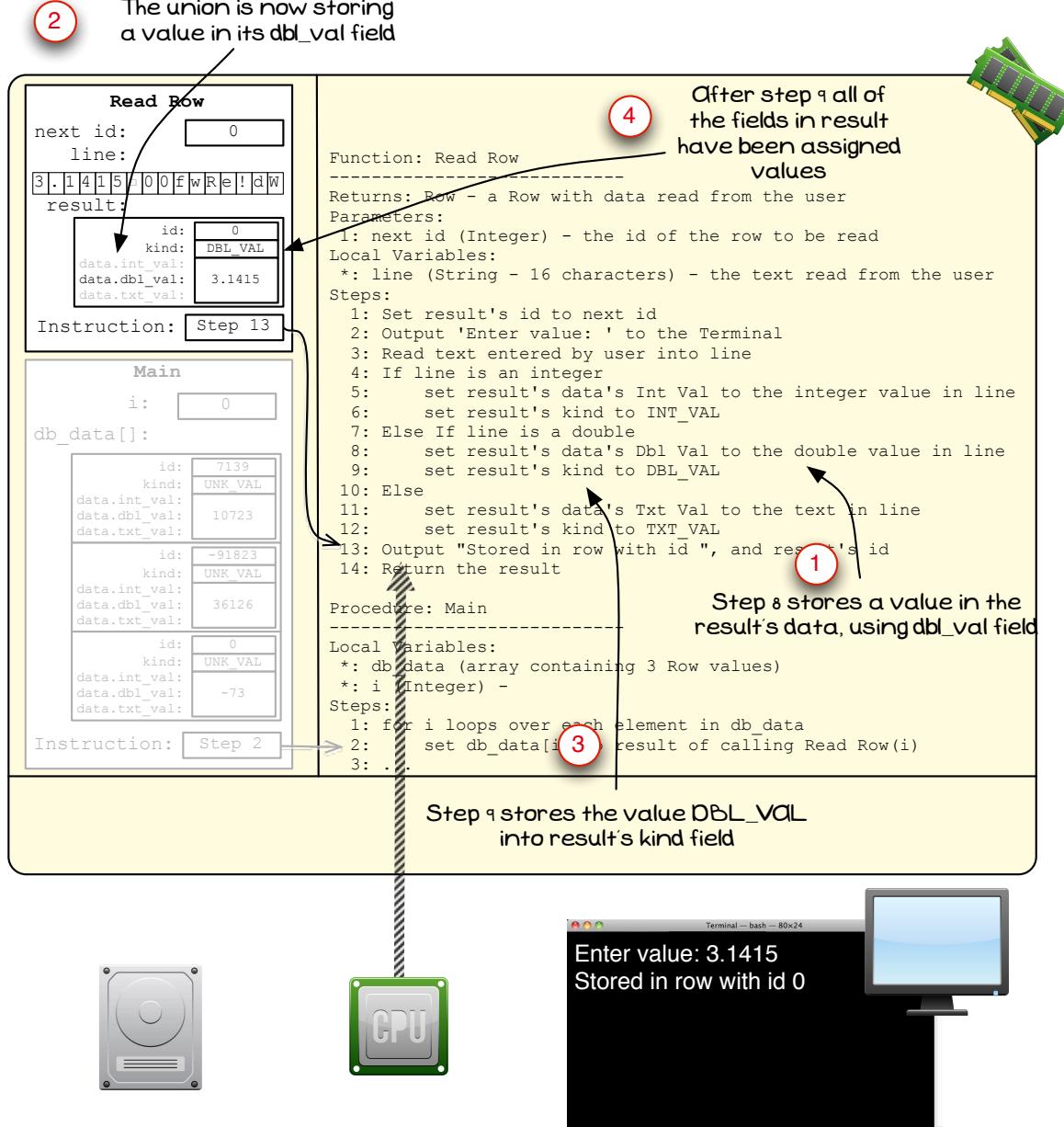
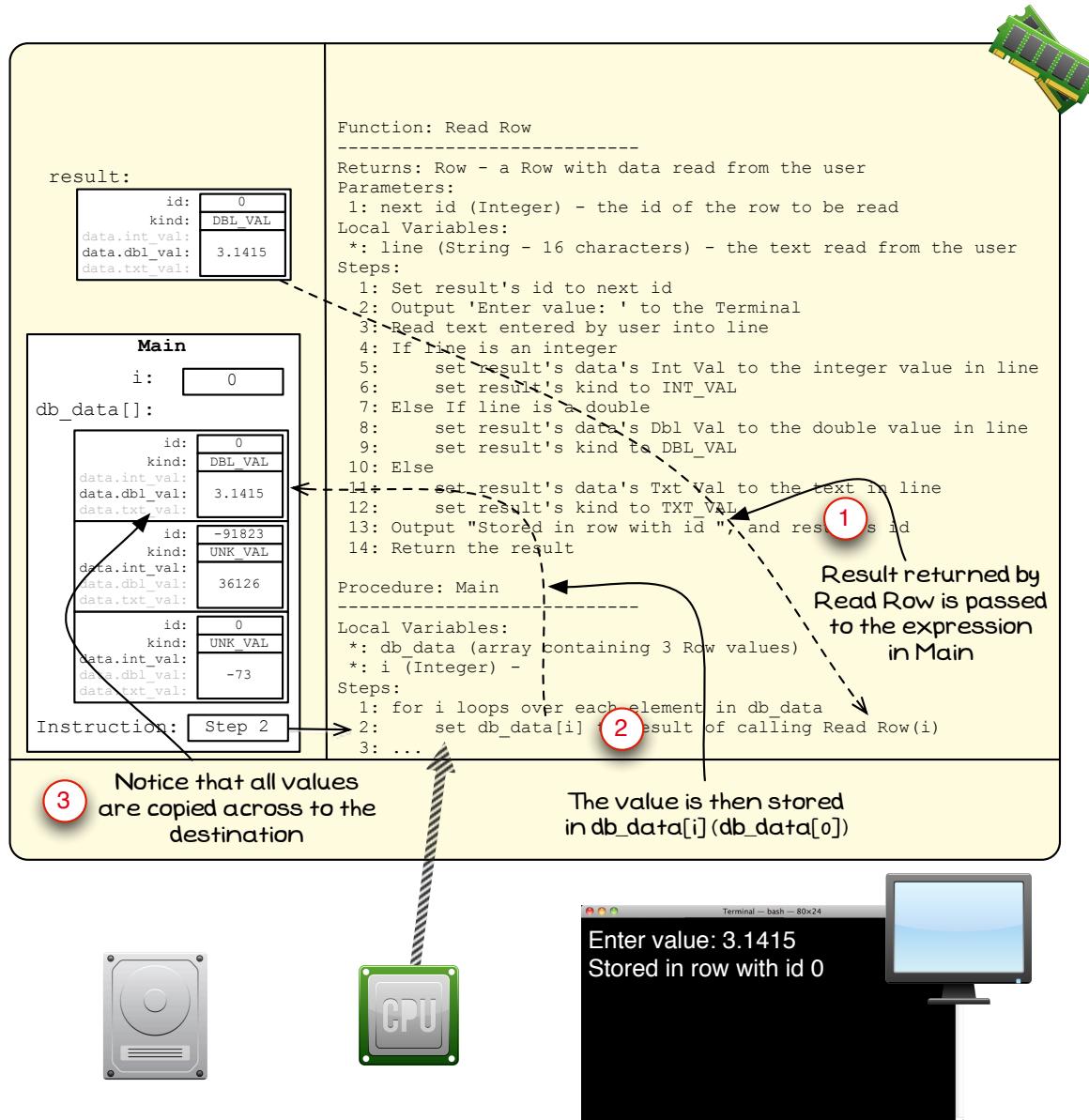


Figure 5.34: A double value is entered by the user, so the code must store the double in the row

Note

- In Figure 5.34 the indicated areas show the following:
 - Step 8 of `Read Row` stores the double value entered by the user into the `dbl_val` field of the data field of the `result` Row.
 - Notice that the union is now shown as storing a value in its `dbl_val` field.
 - Step 9 stores the `DBL_VAL` value in the `kind` field of the `result` Row. This is one of the values from the `Data Kind` enumeration.
 - At this point all of the fields of `result` have been assigned values.

The result row is returned to Main**Figure 5.35:** The result Row is returned to Main**Note**

- In Figure 5.35 the indicated areas show the following:
 - At the end of **Read Row** the result Row is returned to **Main**.
 - In **Main** the value is used in an assignment statement, that assigns it to the i^{th} value in the **db_data** array. As **i** is currently 0, this stores the **entire** row in **dd_data[0]**.
 - When this assignment occurs all of the data from the result of **Read Row** is copied into **db_data[0]**.
- With Records and Unions you can read/write to individual fields using the dot notation, or you can access the entire record.

This process is repeated for each element of the array

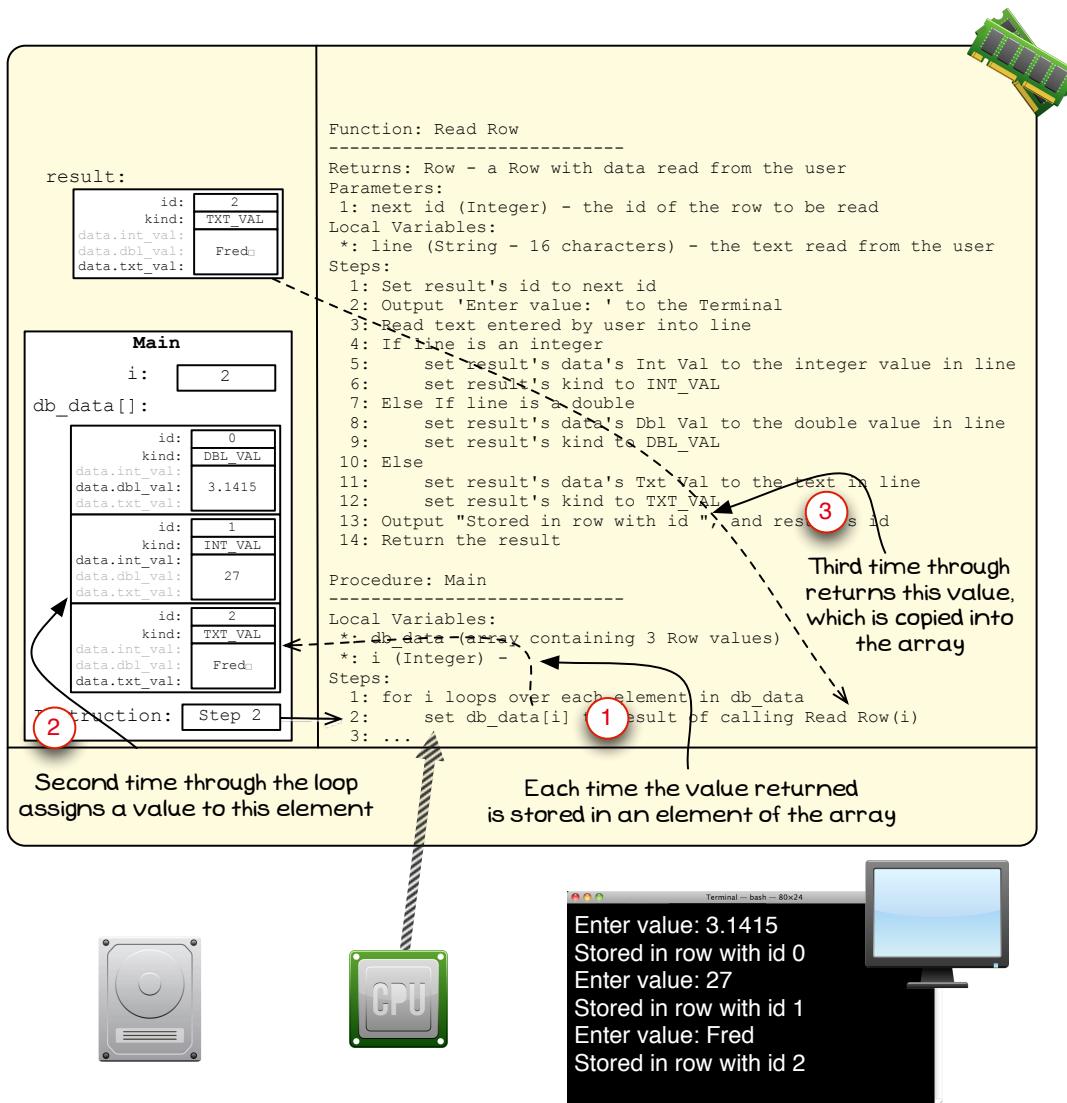


Figure 5.36: The for loop ensures that a Row value is read in for each element of the array

Note

- In Figure 5.36 the indicated areas show the following:
 - Each time through the loop the value is written to an element in the array. This is showing the last iteration where i is 2.
 - The second time this loop was executed the user entered an integer value, this is now stored in the second element of the array.
 - The third time through the loop, the result returned is storing a string value. This is returned to Main, and then stored in the `db_data` array.
- Notice that there is only ever one value in each Row's data. This is a **union**, and only stores one of its field values.
- See how the enumeration values indicate the field the data has been stored in. This is why the enumeration's constants were named in a similar^a way to the union's fields.

^aThey could have been named anything, but this reflects their purpose well.

5.5.2 Understanding Print Row

Print Row is the other key piece of logic in the Small DB program. This procedure outputs the values read from the user to the Terminal. It uses the data stored in the Row's fields to determine how this value is output, and how the data can be read. The flowchart of this logic is shown in Figure 5.37.

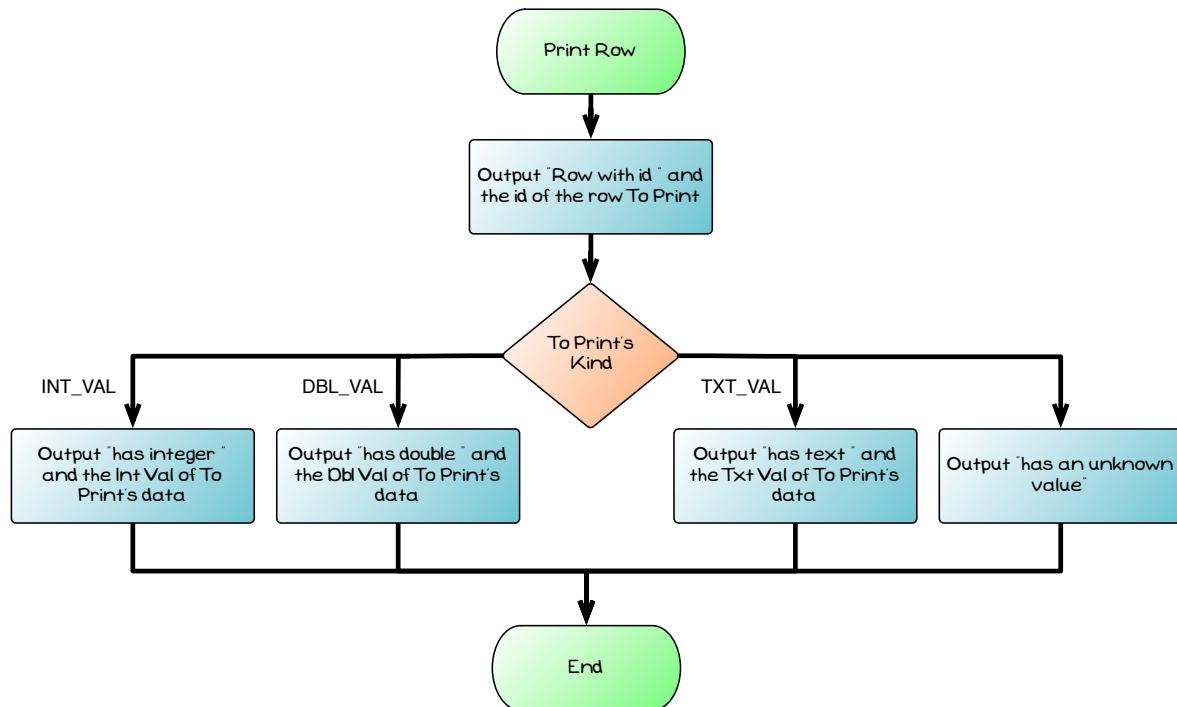


Figure 5.37: Flowchart of the steps needed to print a Row to the Terminal, from Figure 5.18

Within Main, Print Row is called once for each Row in the db_data array. The following illustrations demonstrate the third and final call to Print Row.

The illustrations will show the following:

1. Print Row is called for each element in the array
2. The text value is output to the Terminal

You can use these details to determine how the other data was written to the Terminal.

Print Row is called for each element in the array

This illustration starts part way through the third call to Print Row. At this stage the first two rows have been output to the Terminal, as has the id of the third Row.

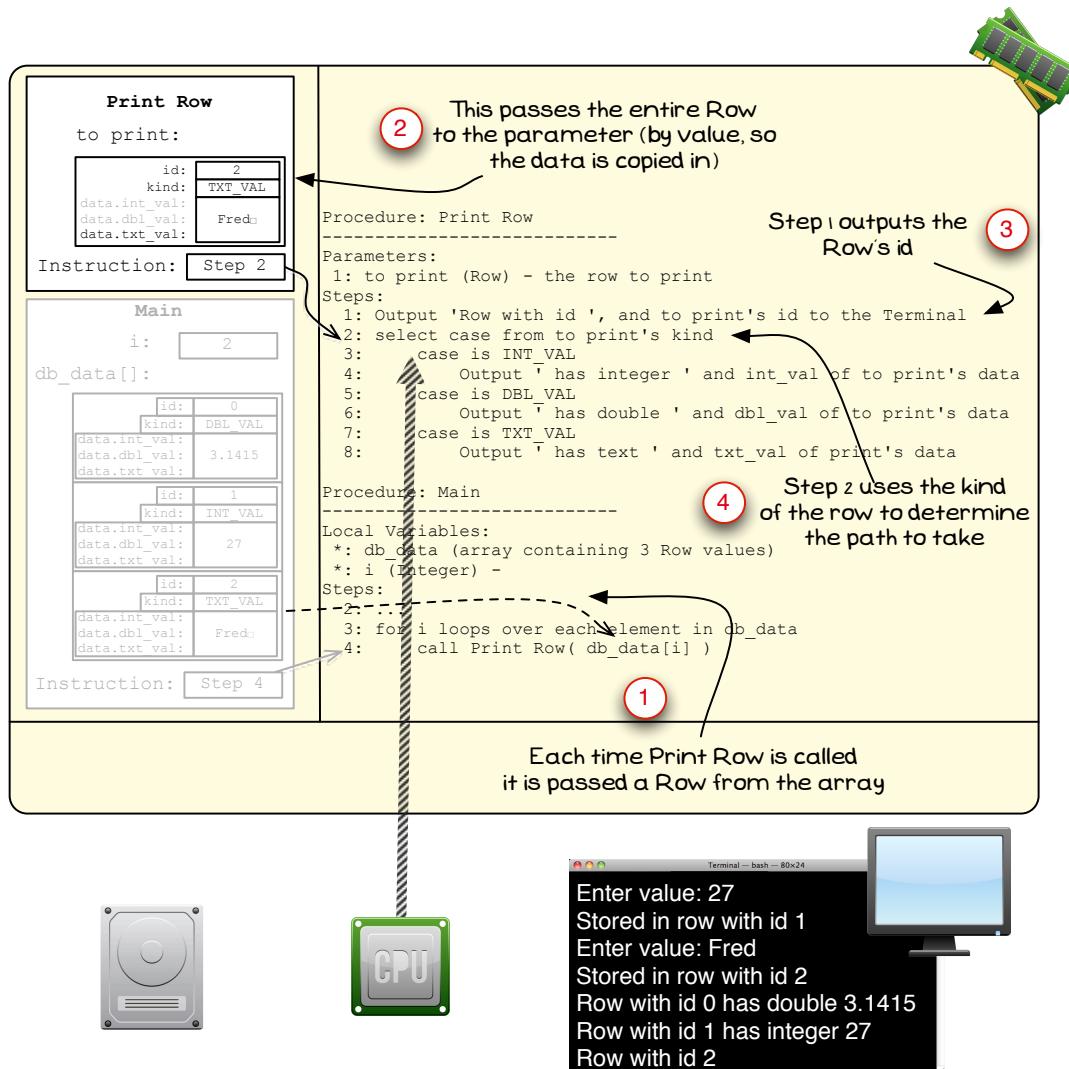
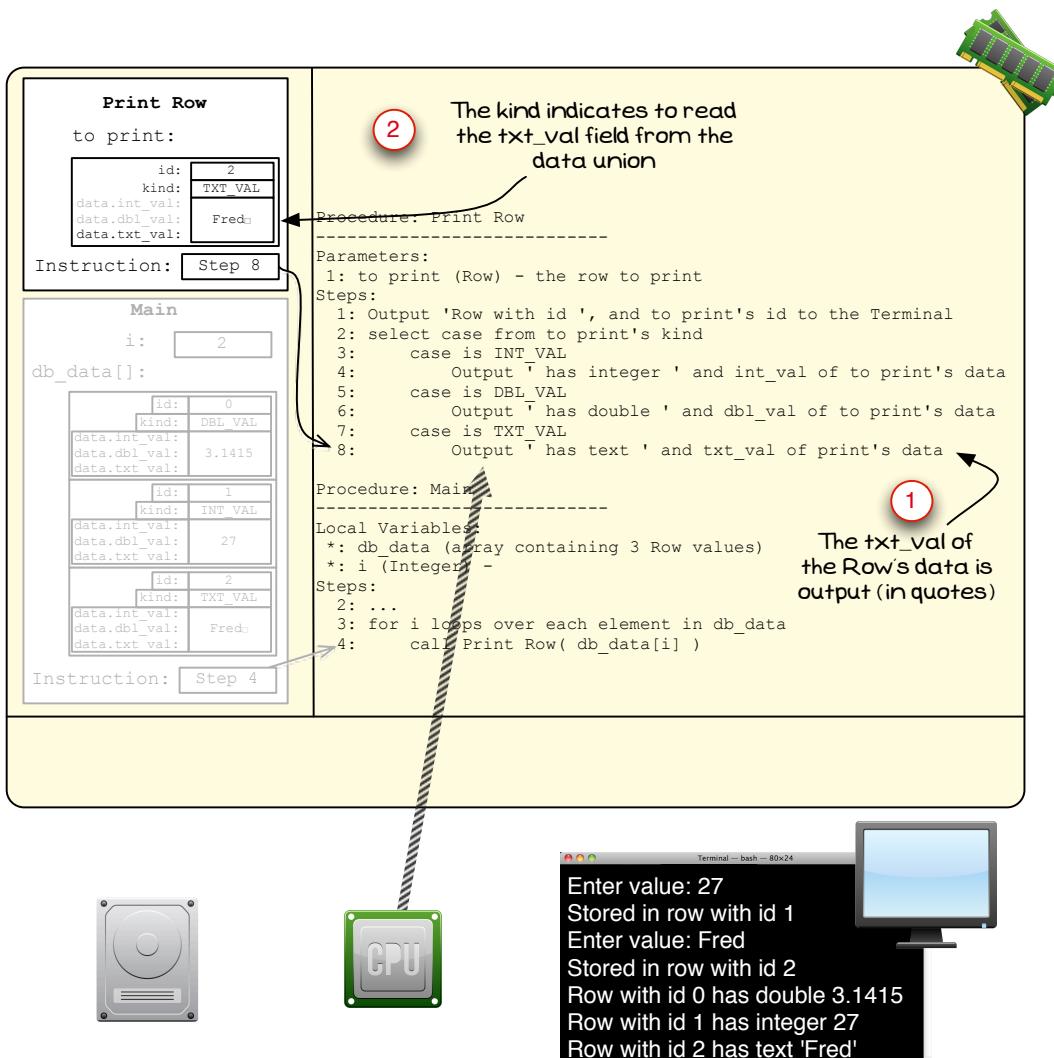


Figure 5.38: Print Row is called for each of the Row elements in db_data

Note

- In Figure 5.38 the indicated areas show the following:
 - This is the third call to Print Row. Each time this procedure is called it receives a copy of the data from the element passed to it.
 - The parameter is a copy of the data from the array element.
 - The first action in Print Row is to output the `id` value from the `to print` Row.
 - The illustration is showing the computer at the stage where it is reading to `print's kind` to determine which path to take. As `print's kind` is currently `TXT_VAL` it will take the path at Step 8.
- The case statement will allow the code to output the message that matches the kind of data stored in `to print`.

The text value is output to the Terminal**Figure 5.39:** `Print Row` is called for each of the Row elements in `db_data`**Note**

- In Figure 5.39 the indicated areas show the following:
 - Step 8 reads the `txt_val` field of `to print`'s `data` field. This reads the text value from within that field, and this is output to the Terminal.
 - The `kind` field was used to determine which field to read from the union.
- In this example the record, union, and enumeration are all working together to enable the required functionality.
- Without the enumeration it would not be possible to determine which field to read from the union. Reading any of the fields on the union would return data, but only the field that was written to can be relied upon to have a meaningful value.
- Without the union you would need to waste space storing all three data values, but only ever using one.
- Without the record it would be hard to relate the values stored within a single Row. The row only existed because of the record's declaration.
- Back in `Main`, the array allows you to store multiple of these values.

5.6 Example Custom Types

5.6.1 Lights

This example draws three light bulbs to the screen. These lights can be clicked to turn them on and off. The code includes the declaration of a record/structure and an enumeration.

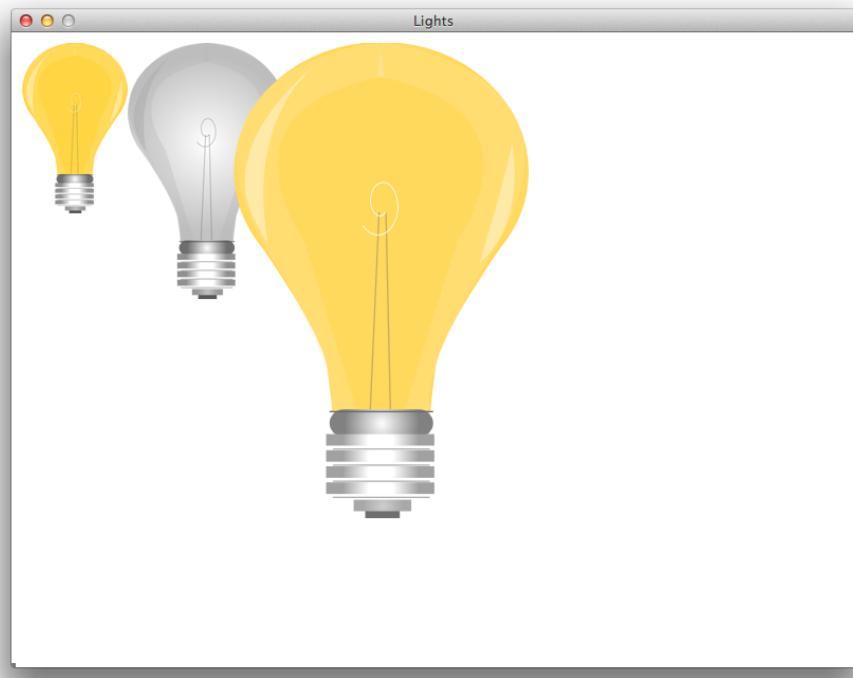


Figure 5.40: Example execution of the Lights program

C++

```
/* Program: lights (C++, SwinGame) */
#include <stdbool.h>
#include <strings.h>
#include <stdio.h>
#include "SwinGame.h"

// =====
// = Delcare Constants =
// =====

#define NUM_LIGHTS 3

// =====
// = Declare Types =
// =====

// There are three sizes of light...
typedef enum
{
    SMALL_LIGHT,
    MEDIUM_LIGHT,
    LARGE_LIGHT
} light_size;

// A light is on/off, have a size, and a position
typedef struct
{
    bool      is_on;      // is the light on?
    light_size size;      // size of the light
    point2d   position;   // location of the light (top left)
} light;
```

Listing 5.12: Lights code in C++, continues in Listing 5.13



C++

```

// =====
// = Declare functions and procedures =
// =====

// Sets up a new light, and returns its data
light create_light(bool on, light_size sz, point2d pos)
{
    light result;

    result.is_on = on;           // sets the light on/off
    result.size = sz;            // sets the size of the light
    result.position = pos;       // sets its position

    return result;               // return the initialised light
}

// Get the bitmap to use for the light "l"
bitmap light_bitmap(light &l)
{
    char name[17] = ""; // construct the name of the bitmap... 16+1

    // the start of the name is based on the size of the bitmap
    switch (l.size)
    {
        case SMALL_LIGHT:
            strncat(name, "small light", 11);
            break;
        case MEDIUM_LIGHT:
            strncat(name, "medium light", 12);
            break;
        case LARGE_LIGHT:
            strncat(name, "large light", 11);
            break;
        default:
            return NULL;
    }

    // the end of the name is based on if the light is on/off
    if (l.is_on)
        strncat(name, " on", 4);
    else
        strncat(name, " off", 4);

    // return the bitmap with that name
    return bitmap_named(name);
}

// Draw the light "l" to the screen
void draw_light(light &l)
{
    draw_bitmap(light_bitmap(l), l.position);
}

```

Listing 5.13: Lights code in C++, continues in Listing 5.14

C++

```
// Draw all of the lights in "lights"
void draw_lights(light lights[], int count)
{
    int i;

    for(i = 0; i < count; i++)
    {
        draw_light(lights[i]);
    }
}

// Is the light currently under the mouse?
bool light_under_mouse(light &l)
{
    point2d mouse;
    bitmap light_bmp;

    // get the mouse position
    mouse = mouse_position();
    // get the light bitmap, to determine its size etc.
    light_bmp = light_bitmap(l);

    // Check if the point in the bitmap is drawn...
    return bitmap_point_collision(light_bmp, l.position.x, l.position.y, mouse);
}

// Check if the lights have been changed (clicked)
void update_lights(light lights[], int count)
{
    int i;

    // only change if the mouse was clicked
    if (mouse_clicked(LEFT_BUTTON) )
    {
        // for all of the lights
        for (i = 0; i < count; i++)
        {
            // if the light is under the mouse
            if (light_under_mouse(lights[i]))
            {
                // change state (on = off, off = on)
                lights[i].is_on = ! lights[i].is_on;
            }
        }
    }
}
```

Listing 5.14: Lights code in C++, continues in Listing 5.15

C++

```

// Load all of the bitmaps name is based on "size" + "state"
void load_bitmaps()
{
    // Load "on" lights
    load_bitmap_named("small light on", "on_sml.png");
    load_bitmap_named("medium light on", "on_med.png");
    load_bitmap_named("large light on", "on.png");

    // Load "off" lights
    load_bitmap_named("small light off", "off_sml.png");
    load_bitmap_named("medium light off", "off_med.png");
    load_bitmap_named("large light off", "off.png");
}

// =====
// = Main - Entry Point =
// =====

int main(int argc, char* argv[])
{
    // Create a number of lights
    light lights[NUM_LIGHTS];

    open_audio();
    open_graphics_window("Lights", 800, 600);
    load_default_colors();

    load_bitmaps();

    // Setup the lights
    lights[0] = create_light(true, SMALL_LIGHT, point_at(10, 10));
    lights[1] = create_light(true, MEDIUM_LIGHT, point_at(110, 10));
    lights[2] = create_light(true, LARGE_LIGHT, point_at(210, 10));

    do
    {
        // Update
        process_events();
        update_lights(lights, NUM_LIGHTS);

        //Draw
        clear_screen();
        draw_lights(lights, NUM_LIGHTS);
        refresh_screen();
    } while ( ! window_close_requested() );

    release_all_resources();
    close_audio();
    return 0;
}

```

Listing 5.15: Last of the Lights code in C++

Pascal

```
program Lights;
uses
    sgTypes, sgInput, sgGraphics, sgUtils, sgImages, sgGeometry;

// =====
// = Delcare Constants =
// =====

const NUM_LIGHTS = 3;

// =====
// = Declare Types =
// =====

type
    // There are three sizes of light...
    LightSize = (SMALL_LIGHT, MEDIUM_LIGHT, LARGE_LIGHT);

    // A light is on/off, have a size, and a position
    Light = record
        isOn: Boolean;           // is the light on?
        size: LightSize;         // size of the light
        position: Point2d;       // location of the light (top left)
    end;

// =====
// = Declare functions and procedures =
// =====

// Sets up a new light, and returns its data
function CreateLight(switchOn: boolean; sz: LightSize; pos: Point2d): Light;
var
    checkLight : Light;
begin
    checkLight.isOn := switchOn;      // sets the light on/off
    checkLight.size := sz;            // sets the size of the light
    result.position := pos;          // sets its position

    result := checkLight;             // return the initialised light
end;
```

Listing 5.16: Lights code in Pascal, continues in Listing 5.17



Pascal

```
// Get the bitmap to use for the light 'l'
function LightBitmap(const one: Light): bitmap;
var
  name: String;
begin
  // the start of the name is based on the size of the bitmap
  case(one.size) of
    SMALL_LIGHT: name := 'small light';
    MEDIUM_LIGHT: name := 'medium light';
    LARGE_LIGHT: name := 'large light';
  else
    result := nil;
    exit;
  end;

  // the end of the name is based on if the light is on/off
  if (one.isOn) then name += ' on'
  else name += ' off';

  // return the bitmap with that name
  result := BitmapNamed(name);
end;

// Draw the light 'l' to the screen
procedure DrawLight(const l: Light);
begin
  DrawBitmap(LightBitmap(l), l.position);
end;

// Draw all of the lights in 'lights'
procedure DrawLights(const lights: array of Light);
var
  i: integer;
begin
  for i := Low(lights) to High(lights) do
  begin
    DrawLight(lights[i]);
  end;
end;
```

Listing 5.17: Lights code in Pascal, continues in Listing 5.18



Pascal

```

// Is the light currently under the mouse?
function LightUnderMouse(var l: light): boolean;
var
  mouse: Point2d;
  lightBmp: Bitmap;
begin
  // get the mouse position
  mouse := MousePosition();
  // get the light bitmap, to determine its size etc.
  lightBmp := LightBitmap(l);

  // Simple version using a bounded rectangle
  result := PointInRect(mouse,
                        BitmapRectangle(l.position.x, l.position.y, lightBmp));
end;

// Check if the lights have been changed (clicked)
procedure UpdateLights(var lights: array of Light);
var
  i: integer;
begin
  // only change if the mouse was clicked
  if (MouseClicked(LeftButton) ) then
  begin
    // for all of the lights
    for i:= Low(lights) to High(lights) do
    begin
      // if the light is under the mouse
      if (LightUnderMouse(lights[i])) then
        // change state (on = off, off = on)
        lights[i].isOn := not lights[i].isOn;
    end;
  end;
end;

// Load all of the bitmaps name is based on 'size' + 'state'
procedure LoadBitmaps();
begin
  // Load 'on' lights
  LoadBitmapNamed('small light on', 'on_sml.png');
  LoadBitmapNamed('medium light on', 'on_med.png');
  LoadBitmapNamed('large light on', 'on.png');

  // Load 'off' lights
  LoadBitmapNamed('small light off', 'off_sml.png');
  LoadBitmapNamed('medium light off', 'off_med.png');
  LoadBitmapNamed('large light off', 'off.png');
end;

```

Listing 5.18: Lights code in Pascal, continues in Listing 5.19



Pascal

```
// =====
// = Main - Entry Point =
// =====

procedure Main();
var
    // Create a number of lights
    lights: array [0..NUM_LIGHTS-1] of Light;
begin
    OpenGraphicsWindow('Lights', 800, 600);

    LoadBitmaps();

    // Setup the lights
    lights[0] := CreateLight(true, SMALL_LIGHT, PointAt(10, 10));
    lights[1] := CreateLight(true, MEDIUM_LIGHT, PointAt(110, 10));
    lights[2] := CreateLight(true, LARGE_LIGHT, PointAt(210, 10));

    repeat
        // Update
        ProcessEvents();
        UpdateLights(lights);

        //Draw
        ClearScreen();
        DrawLights(lights);
        RefreshScreen();
    until WindowCloseRequested();

    ReleaseAllResources();
end;

begin
    Main();
end.
```

Listing 5.19: Lights code in Pascal



5.6.2 Shape Drawing Program

The Shape Drawing program allows the user to create simple shape drawings using circles, triangles, rectangles, and ellipses.

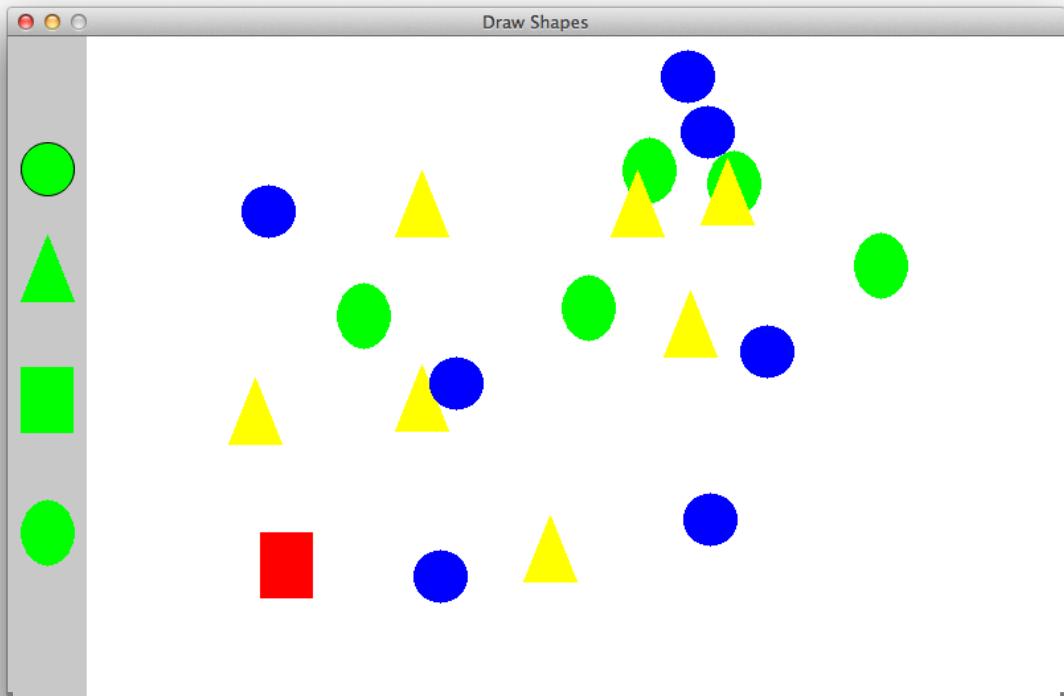


Figure 5.41: Example execution of the Shape Drawing program

C++

```

/* Program: Shape Drawer - SwinGame source file. (C++) */
#include "SwinGame.h"

// =====
// = Declare Constants =
// =====
#define MAX_SHAPES 20
#define MENU_RIGHT_X 60

// Menu Shape Constants
const rectangle MENU_RECT      = {10,250,40,50};
const circle   MENU_CIRCLE     = {{30,100},20};
const triangle MENU_TRIANGLE   = {{30,150},{10,200},{50,200}};
const rectangle MENU_ELLIPSE   = {10,350,40,50};
const rectangle DRAWING_PAD    = {61,0,790,600};

// =====
// = Declare Types =
// =====

// The Shape Type is one of these options
typedef enum
{
    CIRCLE,
    RECTANGLE, //TODO: Add Ellipse and Triangle
    UNKNOWN
} shape_type;

// The Shape Data is either...
typedef union
{
    rectangle rect;
    circle   circ; //TODO: Add Ellipse (as rectangle) and Triangle
} shape_data;

// The shape records...
typedef struct
{
    shape_type type;        // the option selected
    color      fill_color; // the fill color
    shape_data data;       // the shape's data
} shape;

// A Drawing records...
typedef struct
{
    shape      shapes[MAX_SHAPES]; // a number of shapes
    int         index;           // the index of the last shape used
    shape_type selected_shape;  // the selected shape type
} drawing;

```

Listing 5.20: Shape Drawing code in C++, continues in Listing 5.21

C++

```
// =====
// = Drawing Procedures =
// =====

// Draw the shapes in the menu
void draw_shapes_menu(shape_type selected_shape)
{
    // Draw the "toolbar" area
    fill_rectangle(ColorLightGrey, 0, 0, 60, 600);

    // Draw the menu shapes.
    fill_circle(ColorGreen, MENU_CIRCLE);
    fill_rectangle(ColorGreen, MENU_RECT);
    //TODO: Add call to draw Ellipse and Triangle

    // Redraw the selected shape
    switch(selected_shape)
    {
        case RECTANGLE:
            draw_rectangle(ColorBlack, MENU_RECT);
            break;
        case CIRCLE:
            draw_circle(ColorBlack, MENU_CIRCLE);
            break;      //TODO: Add code to draw selected Ellipse and Triangle
        case UNKNOWN:
            break;
    }
}

// Draw a shape onto the screen
void draw_shape(shape &s)
{
    switch(s.type)
    {
        case RECTANGLE:
            fill_rectangle(s.fill_color, s.data.rect);
            break;
        case CIRCLE:
            fill_circle(s.fill_color, s.data.circ);
            break;      //TODO: Add code to draw Ellipse and Triangle shape
        case UNKNOWN:
            break;
    }
}

// Draw the drawing
void draw_drawing(drawing &d)
{
    clear_screen();
    draw_shapes_menu(d.selected_shape);

    // Draw all shapes
    for(int i = 0; i < MAX_SHAPES; ++i)
    {
        draw_shape(d.shapes[i]);
    }
}
```

Listing 5.21: Shape Drawing code in C++, continues in Listing 5.22

C++

```
// =====
// = Procedures to interact with Drawing... =
// =====

// Clear the drawing
void clear_drawing(drawing &d)
{
    // Start adding at index 0
    d.index = 0;

    // All shapes are unknown...
    for(int i = 0; i < MAX_SHAPES ;++i)
    {
        d.shapes[i].type = UNKNOWN;
    }
}

// Add a rectangle to the drawing
void make_rectangle(shape &s, point2d pt)
{
    // Set the shape
    s.type = RECTANGLE;
    s.fill_color = ColorRed;

    // Copy in the menu rectangle
    s.data.rect = MENU_RECT;

    // Change its position
    s.data.rect.x = pt.x;
    s.data.rect.y = pt.y;
}

// Add a circle to the drawing
void make_circle(shape &s, point2d pt)
{
    // Set the shape
    s.type = CIRCLE;
    s.fill_color = ColorBlue;

    // Copy in the menu circle radius
    s.data.circ.radius = MENU_CIRCLE.radius;

    // Set the position
    s.data.circ.center = pt;
}

//TODO: Add procedures to make Ellipse and Triangle shapes
```

Listing 5.22: Shape Drawing code in C++, continues in Listing 5.23

C++

```
// =====
// = Procedures to handle user interactions =
// =====

// Check key presses - c clears, q quits
bool process_key(drawing &d)
{
    if(key_down(VK_Q))
        return true;

    if(key_down(VK_C))
        clear_drawing(d);

    return false;
}

// Check if user clicked in a shape in the toolbar
void process_menu_click(drawing &d, point2d pt)
{
    if(point_in_rect(&pt, &MENU_RECT))
    {
        d.selected_shape = RECTANGLE;
    }
    else if(point_in_circle(&pt, &MENU_CIRCLE))
    {
        d.selected_shape = CIRCLE;
    }
    //TODO: Add code to test if user clicked in the Ellipse or Triangle
}

// Add a shape to the drawing canvas
void process_canvas_click(drawing &d, point2d pt)
{
    // Try to add a shape... is the current index < maximum?
    if(d.index < MAX_SHAPES)
    {
        // Select the shape to add...
        switch(d.selected_shape)
        {
            case RECTANGLE:
                make_rectangle(d.shapes[d.index], pt);
                break;
            case CIRCLE:
                make_circle(d.shapes[d.index], pt);
                break;
            //TODO: Add code to call make the shape an Ellipse / Triangle
            default:
                return; // exit as no selected shape... (doesn't increment index)
        }

        // Increment the index
        d.index++;
    }
}
```

Listing 5.23: Shape Drawing code in C++, continues in Listing 5.24

C++

```

// Check if the user has performed any actions...
bool process_input(drawing &d)
{
    if(mouse_clicked(LEFT_BUTTON))
    {
        point2d pt = mouse_position();

        if(pt.x < MENU_RIGHT_X)
            process_menu_click(d, pt);
        else
            process_canvas_click(d, pt);
    }

    if(any_key_pressed())
        return process_key(d);
    else
        return false;
}

// =====
// = Main ... =
// =====

int main()
{
    // Create the drawing...
    drawing my_drawing;
    bool quit = false;

    // Initialise the drawing with empty data...
    clear_drawing(my_drawing);

    open_graphics_window("Draw Shapes", 800, 500);
    load_default_colors();

    do
    {
        process_events(); // read user interactions...
        quit = process_input(my_drawing);

        draw_drawing(my_drawing);
        refresh_screen();
    } while (!window_close_requested() && !quit);

    release_all_resources();
    return 0;
}

```

Listing 5.24: Last of the Shape Drawing code in C++

Pascal

```

program ShapeDrawer;
uses sgTypes, sgInput, sgAudio, sgGraphics, sgResources, sgGeometry;

// =====
// = Declare Constants =
// =====
const MAX_SHAPES = 20;

const MENU_RECT      : Rectangle = (x: 10; y: 250; width: 40; height: 50);
const MENU_CIRCLE    : Circle = (center: (x: 30; y: 100); radius: 20);
const DRAWING_PAD   : Rectangle = (x: 61; y: 0; width: 790; height: 600);
// Todo: Add Ellipse & Triangle

// =====
// = Declare Types =
// =====
type
  // The Shape Type is one of these options
  // Todo: add Ellipse & Triangle
  ShapeType = ( CIRCLE_TYPE, RECTANGLE_TYPE, UNKNOWN_TYPE );

  Shape = record
    fillColor: Color;           // Had a color, and
    case kind: ShapeType of     // Shape Data is either...
      CIRCLE_TYPE: ( circ: Circle; );
      RECTANGLE_TYPE: ( rect: Rectangle; );
    //Todo: Add Ellipse and Triangle
  end;

  Drawing = record
    Shapes: array[0..MAX_SHAPES-1] of shape; // A Drawing records has
    Index: integer;                      // a number of shapes
    SelectedShape: ShapeType;           // the index of the last shape
    // the selected shape type
  end;

// =====
// = Drawing Procedures =
// =====

// Draw the shapes in the menu
procedure DrawShapesMenu(selectedShape: ShapeType);
begin
  // Draw the "toolbar" area
  FillRectangle(ColorLightGrey, 0, 0, 60, 600);

  // Draw the menu shapes.
  FillCircle(ColorGreen, MENU_CIRCLE);
  FillRectangle(ColorGreen, MENU_RECT);
  //Todo: Add Ellipse and Triangle

  // Redraw the selected shape
  case(SelectedShape) of
    RECTANGLE_TYPE: DrawRectangle(ColorBlack, MENU_RECT);
    CIRCLE_TYPE: DrawCircle(ColorBlack, MENU_CIRCLE);
  //Todo: Add Ellipse and Triangle
end;
end;

```

Listing 5.25: Shape Drawing code in Pascal, continues in Listing 5.26



Pascal

```

// Draw a shape onto the screen
procedure DrawShape(var s: Shape);
begin
  case s.kind of
    RECTANGLE_TYPE: FillRectangle(s.FillColor, s.rect);
    CIRCLE_TYPE: FillCircle(s.FillColor, s.circ);
    //Todo: Add Ellipse and Triangle
  end; //end case
end;

procedure DoDrawing(var d: Drawing); // Draw the Drawing
var
  i: integer;
begin
  // Clear screen and redraw menu
  ClearScreen(ColorWhite);
  DrawShapesMenu(d.SelectedShape);

  // Draw the shapes
  for i := 0 to MAX_SHAPES do
    DrawShape(d.Shapes[i]);
end; //do Drawing

// =====
// = Procedures to Add Shapes =
// =====

procedure AddRectangle(var s: Shape; pt: Point2d);
begin
  // Set the shape
  s.kind := RECTANGLE_TYPE;
  s.FillColor := ColorRed;

  // Copy in the menu Rectangle
  s.rect := MENU_RECT;

  // Change its position
  s.rect.x := pt.x;
  s.rect.y := pt.y;
end;

procedure AddCircle(var s: shape; pt: Point2d);
begin
  // Set the shape
  s.kind := CIRCLE_TYPE;
  s.FillColor := ColorBlue;

  // Copy in the menu Circle radius
  s.circ.Radius := MENU_CIRCLE.Radius;

  // Set the position
  s.circ.Center := pt;
end;//add Circle

//Todo: Add AddEllipse and AddTriangle

```

Listing 5.26: Shape Drawing code in Pascal, continues in Listing 5.27



Pascal

```
//input processing procedures
function ProcessKey(var d: Drawing): Boolean;
var
  i: integer;
begin
  if KeyDown(VK_Q) then
    result := true;
    exit;
  if KeyDown(VK_C) then
  begin
    d.Index := 0;
    for i := 0 to MAX_SHAPES do
      d.Shapes[i].kind := UNKNOWN_TYPE;
  end;
  result := false;
end;

// Check if the user has clicked in a shape in the toolbar
procedure ProcessMenuClick(var d: Drawing; pt: Point2D);
begin
  if PointInRect(pt, MENU_RECT) then
    d.SelectedShape := RECTANGLE_TYPE;

  if PointInCircle(pt, MENU_CIRCLE) then
    d.SelectedShape := CIRCLE_TYPE;

  //Todo: Add Ellipse (check in rect) and Triangle
end;

// Add a shape to the canvas
procedure ProcessCanvasClick(var d: Drawing; pt: Point2D);
begin
  // Try to add a shape... is the current index < maximum?
  if(d.Index < MAX_SHAPES) then
  begin
    // Select the shape to add...
    case(d.SelectedShape) of
      RECTANGLE_TYPE: AddRectangle(d.Shapes[d.Index], pt);
      CIRCLE_TYPE: AddCircle(d.Shapes[d.Index], pt);
      //Todo: Add triangle and ellipse
    else
      exit; // dont add to index
    end; //end case

    d.Index := d.Index + 1;
  end; //end if
end; // process pad click
```

Listing 5.27: Shape Drawing code in Pascal, continues in Listing 5.28



Pascal

```
//input processing procedures
function ProcessKey(var d: Drawing): Boolean;
var
  i: integer;
begin
  if KeyDown(VK_Q) then
    result := true;
    exit;
  if KeyDown(VK_C) then
  begin
    d.Index := 0;
    for i := 0 to MAX_SHAPES do
      d.Shapes[i].kind := UNKNOWN_TYPE;
  end;
  result := false;
end;

// Check if the user has clicked in a shape in the toolbar
procedure ProcessMenuClick(var d: Drawing; pt: Point2D);
begin
  if PointInRect(pt, MENU_RECT) then
    d.SelectedShape := RECTANGLE_TYPE;

  if PointInCircle(pt, MENU_CIRCLE) then
    d.SelectedShape := CIRCLE_TYPE;

  //Todo: Add Ellipse (check in rect) and Triangle
end;

// Add a shape to the canvas
procedure ProcessCanvasClick(var d: Drawing; pt: Point2D);
begin
  // Try to add a shape... is the current index < maximum?
  if(d.Index < MAX_SHAPES) then
  begin
    // Select the shape to add...
    case(d.SelectedShape) of
      RECTANGLE_TYPE: AddRectangle(d.Shapes[d.Index], pt);
      CIRCLE_TYPE: AddCircle(d.Shapes[d.Index], pt);
      //Todo: Add triangle and ellipse
    else
      exit; // dont add to index
    end; //end case

    d.Index := d.Index + 1;
  end; //end if
end; // process pad click
```

Listing 5.28: Last Shape Drawing code in Pascal



5.7 Custom Type Exercises

5.7.1 Concept Questions

Read over the concepts in this chapter and answer the following questions:

1. What is a type?
2. What is the relationship between a type and a value?
3. When you create your own type what have you created? A value, or something else?
4. Why would you want to create your own type?
5. What are the three main kinds of type you can create?
6. What kind of data type(s) could you create to model the following in a program?
 - (a) An address book, containing names, phone numbers, and email addresses.
 - (b) The kind of a '*power up*' in a game, e.g. health pack, upgrade, bonus, etc.
 - (c) A field that is either an integer, a double, or some text.
 - (d) A button that has a location on the screen, a width and height, and some text that is drawn on the button.
7. What is a record? What can it be used to model?
8. What is an enumeration? What can it be used to model?
9. What is a union? What can it be used to model?
10. Why is it a good idea to use an enumeration in conjunction with a union?
11. Explain the different ways you can store/read a value when you are using a record.
12. Explain the different ways you can store/read a value when you are using a enumeration.
13. Explain the different ways you can store/read a value when you are using a union.
14. Open one of your SwinGame projects and have a look in the lib folder. This folder contains a number of source code files used to access SwinGame functionality. Have a look in the **Types** file (types.c or sgTypes.pas), and examine the follow types. For each type write a short description of what it contains.
 - (a) Rectangle
 - (b) Circle
 - (c) LineSegment
 - (d) Triangle
 - (e) Point 2D
 - (f) Vector

5.7.2 Code Writing Questions: Applying what you have learnt

Apply what you have learnt to the following tasks:

1. Create a Contact record that stores a person's name (50 characters), their phone number (20 characters), and email address (50) characters.
2. Use the Contact record to create a small address book program that lets you enter in the details of four contacts, and then outputs these to the Terminal.
3. Implement the Lights program from Section 5.6.1.
4. Implement the Shape Drawing program, add the code to create Ellipse and Triangle shapes.
5. Implement the Small DB program, including the support for the double data in the row.

5.7.3 Extension Questions

If you want to further your knowledge in this area you can try to answer the following questions. The answers to these questions will require you to think harder, and possibly look at other sources of information.

1. Extend the Small DB program so that each '*row*' has three '*columns*'.
2. Explore the sizes of the different data types you have created using the `Size Of` function: `sizeof` in C, or `SizeOf` in Pascal.

6

Dynamic Memory Allocation

There are many places you can draw upon to power your spells. So far you have been limited by the constraints of this realm. The tools I give you now will let you stretch beyond this realm, and will open your mind to even greater powers. You will need your orb, your wand, and ...

So far data has been limited by the constraints of the Stack. With the stack, the compiler must know how much space to allocate to each variable ahead of time. This means you are limited to working with a fixed number of values, whether those values are stored in a number of variables or stored in an array. This constraint is not a problem for small programs, but most programs will require the flexibility to work with a variable number of data elements.

This chapter introduces the tools needed to dynamically allocate additional memory for your program to use. With these tools you will be able to dynamically allocate additional space for your program to use as you need it. As memory is finite you will also see how you can release this memory back to the computer when you no longer require it.

When you have understood the material in this chapter you will be able to dynamically allocate memory for your program, increasing and decreasing the number of values that you are storing.

Contents

6.1 Dynamic Memory Allocation Concepts	505
6.1.1 Heap	505
6.1.2 Pointer	508
6.1.3 Allocating Memory	514
6.1.4 Freeing Memory Allocations	518
6.1.5 Issues with Pointers	519
6.1.6 Linked List	522
6.1.7 Summary of Dynamic Memory Allocation Concepts	524
6.2 Using Dynamic Memory Allocation	525
6.2.1 Designing Small DB 2	525
6.2.2 The Analysis Phase: Understanding Small DB 2	526
6.2.3 The Design Phase: Choosing artefacts and designing control flow	526
6.2.4 The Implementation Phase: Writing the code for Small DB 2	533
6.2.5 The Testing Phase: Compiling and running Small DB 2	533
6.2.6 Designing Linked Small DB 2	534
6.2.7 The Design Place: Designing Linked Small DB 2	535
6.3 Dynamic Memory Allocation in C	547
6.3.1 Small DB 2, the dynamic array version in C	547
6.3.2 Small DB 2, the linked version in C	551

6.3.3 C Variable Declaration (with pointers)	554
6.3.4 C Pointer Operators	556
6.3.5 C Type Declarations (with pointers)	557
6.3.6 C Memory Allocation Functions	561
6.4 Dynamic Memory Allocation in Pascal	565
6.4.1 Small DB 2, the dynamic array version in Pascal	565
6.4.2 Small DB 2, the linked version in Pascal	568
6.4.3 Pascal Variable Declaration (with pointers)	571
6.4.4 Pascal Pointer Operators	572
6.4.5 Pascal Type Declarations (with pointers)	573
6.4.6 Pascal Memory Allocation Functions	575
6.5 Exercises for Dynamic Memory Allocation	578
6.5.1 Concept Questions	578
6.5.2 Code Reading Questions	579
6.5.3 Code Writing Questions: Applying what you have learnt	580
6.5.4 Extension Questions	580

6.1 Dynamic Memory Allocation Concepts

6.1.1 Heap

When your program is executed it allocated memory to work with. This memory is divided into different areas based on the kind of values that will be stored there. Previously all of the program's data was housed on the Stack, dynamically allocated memory is allocated into a separate area known as the Heap. Any memory that you allocate to your program will come from this location.

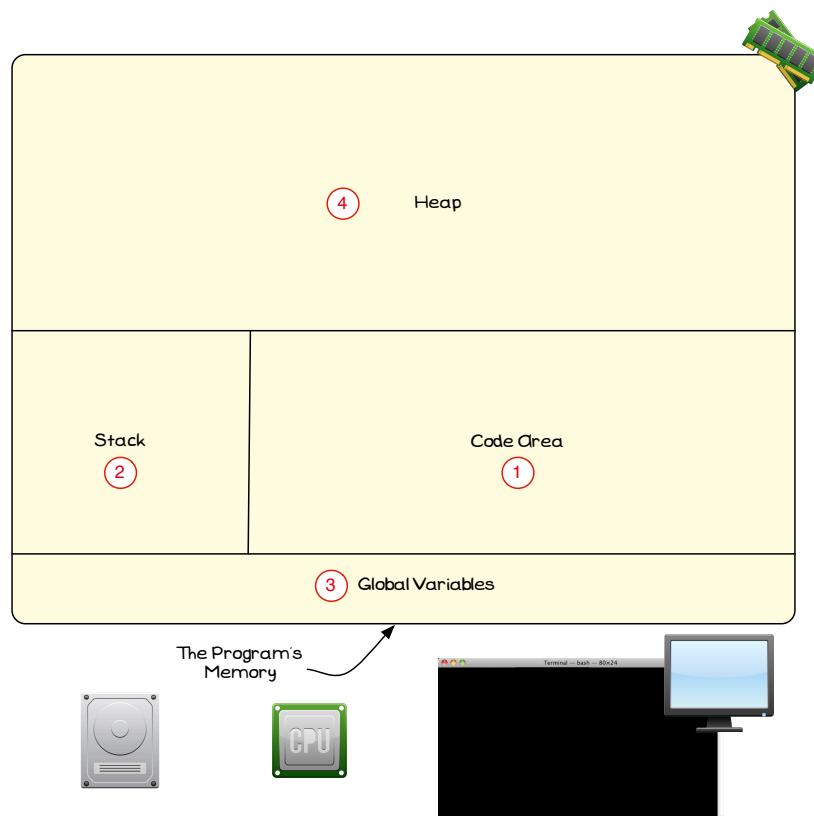


Figure 6.1: The Heap is used to store all dynamically allocated values

Note

- Figure 6.1 includes the following areas:
 1. Your program's machine code is loaded into the **Code Area**.
 2. The **Stack** is used to manage the execution of the program's functions and procedures.
 3. **Global Variables** are allocated their own space.
 4. The new area is the **Heap**. This is used to store all dynamically allocated values.
- Values can be stored in the *global variables*, in local variables on the *Stack*, and on the *Heap* using dynamic memory allocation functions and procedures.
- The space taken up by the **global variables** is fixed based on the size of the variables you have declared.
- Each function/procedure takes a fixed amount of space on the stack. The space allocated is enough to store each of the local variables, plus some additional space for various overheads.
- The compiler take care of managing memory in the stack and for the global variables.
- **You** are responsible for any memory allocation done on the heap.

Allocating memory on the heap

Dynamic memory allocation is performed with a couple of operations that will be provided by the programming language you are using. These operations allow you to do the following:

- **Allocate Space:** You ask the Operating System to allocate you some space into which you want to store a certain value. The Operating System will then allocate you space on the Heap that is large enough to store the value you require.
- **Free Allocation:** When you have finished using a piece of memory you have been allocated on the Heap, you can tell the operating system that you have finished with this memory, and that it is free to allocate this to some other value.

These are the two basic actions that exist for performing dynamic memory management. Basically, you can ask for memory, and you can give it back. Once you have been allocated space, that space will be reserved for your use until you free that allocation. So it is important to remember to free the memory you have been allocated when you no longer require it.

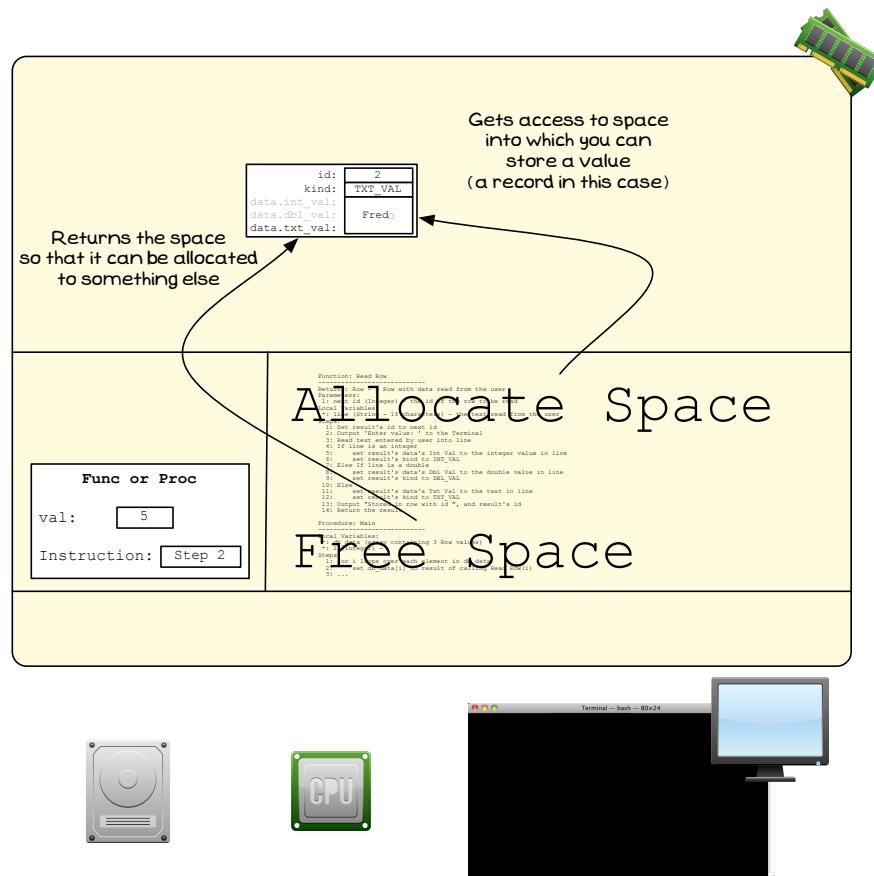


Figure 6.2: You can ask for space, and return the space you were allocated

Note

- Figure 6.2 shows the idea behind the two operations.
- You can ask to be allocated space, this will give you access to a space on the heap. You can then use this to store a value.
- You can tell the Operating System when you are finished with the space, so that it can allocate it to something else.

Accessing dynamically allocated memory

By its very nature, dynamic memory allocation must work a little differently to the way we have been working with values so far. So far, when you wanted to work with a value you declared a variable, or an array. This would have been a **Local Variable**, with its value allocated on the stack along with the other variables you were working with in the current function or procedure. The variable and its value were closely related, with the value being located within the variable. With dynamic memory allocation the values you are allocated are on the heap. This means that their values are not bound within a variable, but exist entirely outside of any variables that appear in your code.

One of the challenges of working with dynamically allocated memory is that you can no longer ‘see’ these values in your code. When you were working with variables, they were in the code, you could see them and think about the value they stored. With dynamically allocated memory you do not have this advantage, these values will be allocated as a result of the operations that are performed while the code is running. This is why it is called **dynamically** allocated memory. It is *not* memory allocated to variables, it is **memory allocated upon request**.

This raises one very important question, as illustrated in Figure 6.3:

If the values exist outside of variables, how do you access them?

For this we require a new kind of data, a new **Type**. This type is used to store a value that tells you where the data you want can be located. It is like an address, telling you where the data can be found. This is the **Pointer** type.

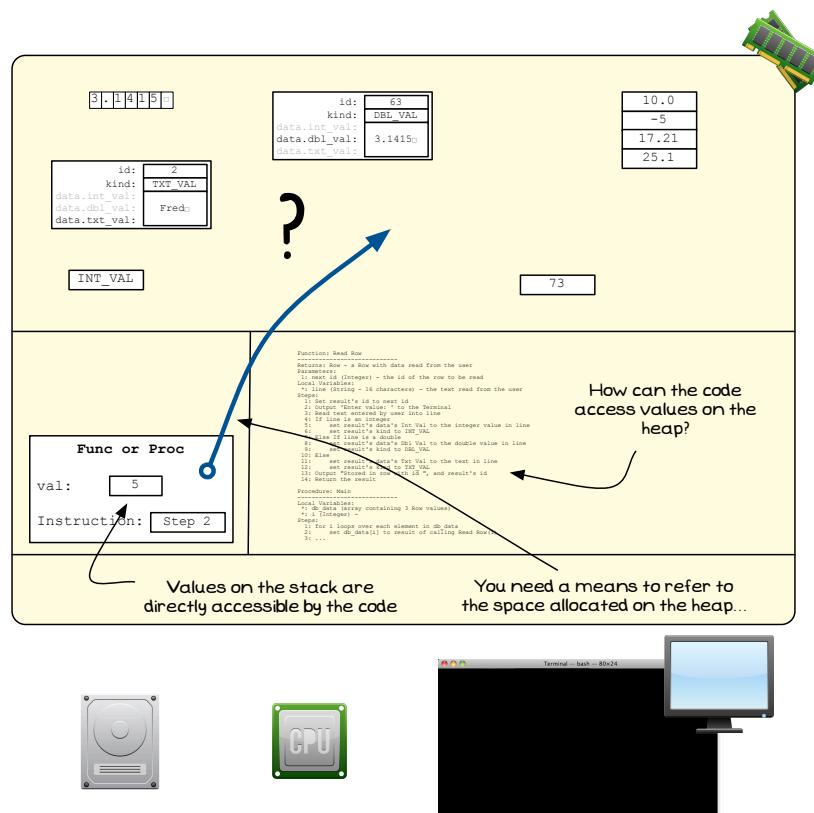


Figure 6.3: How can you access these dynamically allocated values?

6.1.2 Pointer

A Pointer is a new kind of data type, just like Integer, Double, and Boolean. A Pointer Value is an address, a location in memory where a value can be found. The name ‘*Pointer*’ is very descriptive, a *Pointer* points to a value. It tells you, ‘The data I refer to is over there...’.

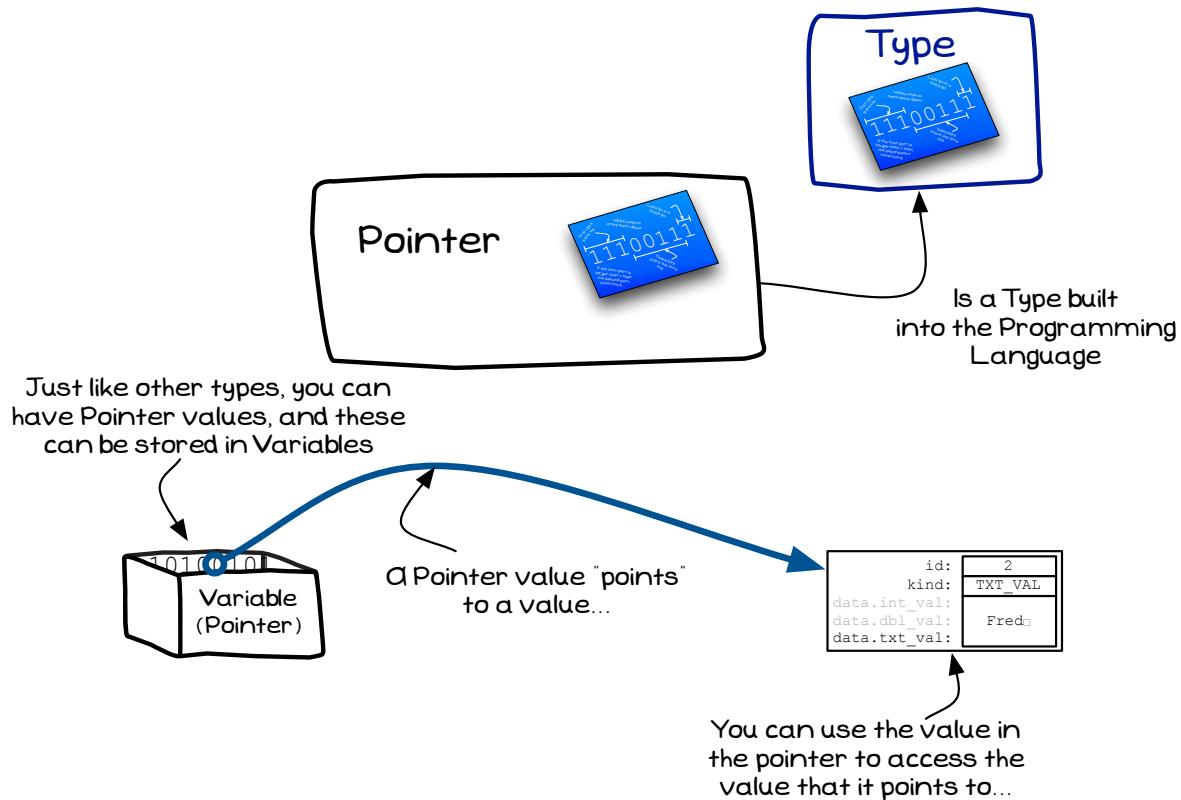


Figure 6.4: A Pointer Value is the address of a value, in effect it *points* to a value

Note

- A pointer is an **existing artefact**, a data type that is built into the programming language.
- A pointer has a value, that stores the location of another value.
- It is a good idea to picture a pointer as a value that *points* to another value.
- The pointer's value is the memory address of the value it *points* to.
- The CPU architecture tells you the size of its pointers. A 32bit machine has 32bit pointers. A 64bit machine has 64bit pointers.

Using pointer to access the heap

Pointers can be used to point to locations in the heap. When you ask the Operating System to allocate you space on the heap, it will give you a pointer value that *points* to the space you were allocated. You can use this pointer to access the value at that location.

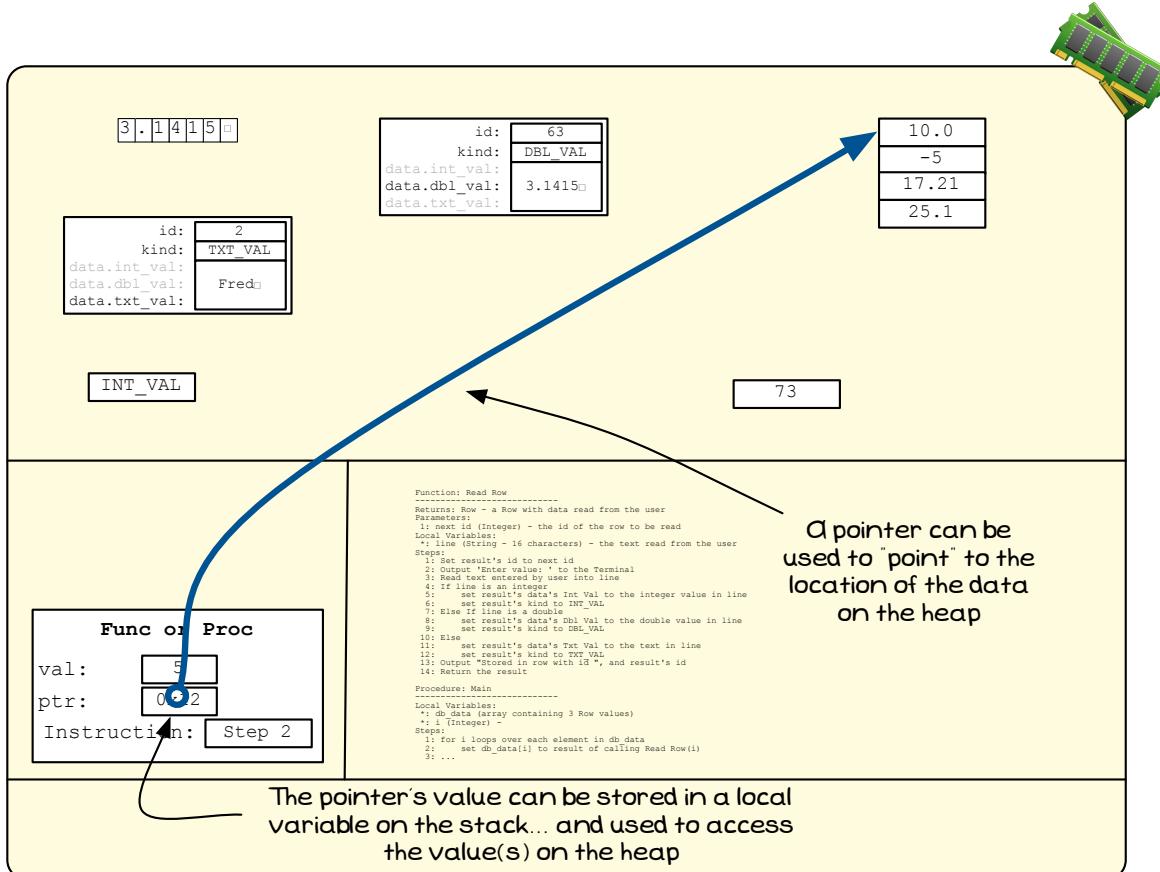


Figure 6.5: You can use pointers to access values on the heap

Note

- Your code can access values stored on the stack, in its local variables and parameters.
- There is no way to directly access values on the heap.
- The memory allocation functions will give you a pointer to the space you were allocated.
- Storing the pointer in a local variable will mean you can use it to access the value on the heap.

What can a pointer point to?

Pointers store a value that is an address of the value that it points to. This means that you can point to *any* value in memory, regardless of where it is. You can have pointer values that point to **Local Variables**, **Global Variables**, **Parameters**, **Array** elements, fields of **Records** or **Unions**. One of its key ability, however, is the ability to point to values on the **Heap**.

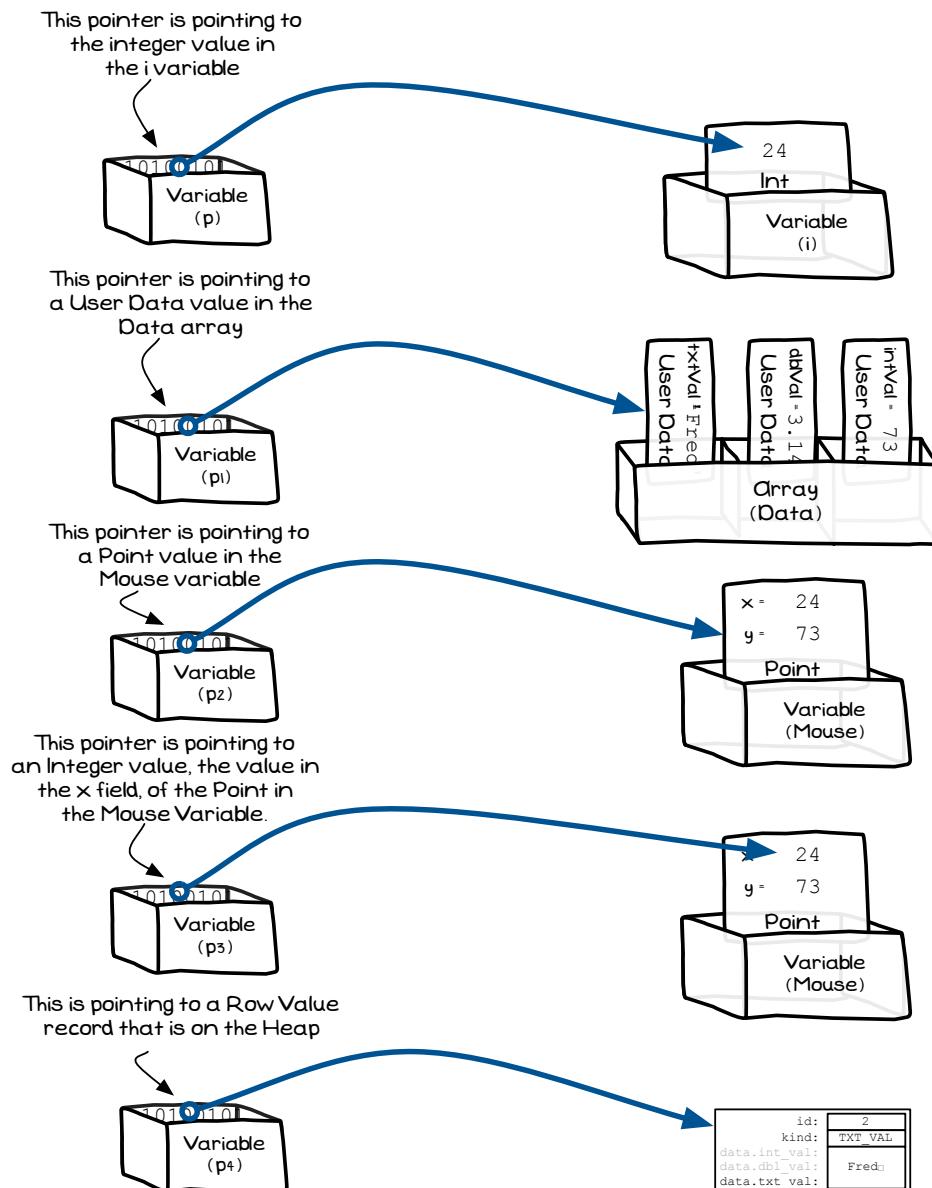


Figure 6.6: A Pointer can point to any value, at any location in memory

Note

- Languages usually require you to declare the kind of data that a Pointer Value will refer to. So rather than just having a generic *pointer*, you will have things like a *pointer to an Integer*, or a pointer to a User Data value. This makes it easier to work out what you can do with the value the pointer points to.

Where can pointer values be stored?

A Pointer value is the same as any other value. It can be stored in [Local Variables](#), [Global Variables](#), it can be passed to a function in a [Parameter](#), it can be returned from a [Function](#), and it can also exist on the [Heap](#).

Figure 6.7 shows an illustration of some values in memory. The start variable is located somewhere on the stack as a local variable. This variable is storing a pointer value that points to a Node¹ that is on the Heap. Each of the nodes on the heap are also storing pointer values that refer to other values that are also on the heap.

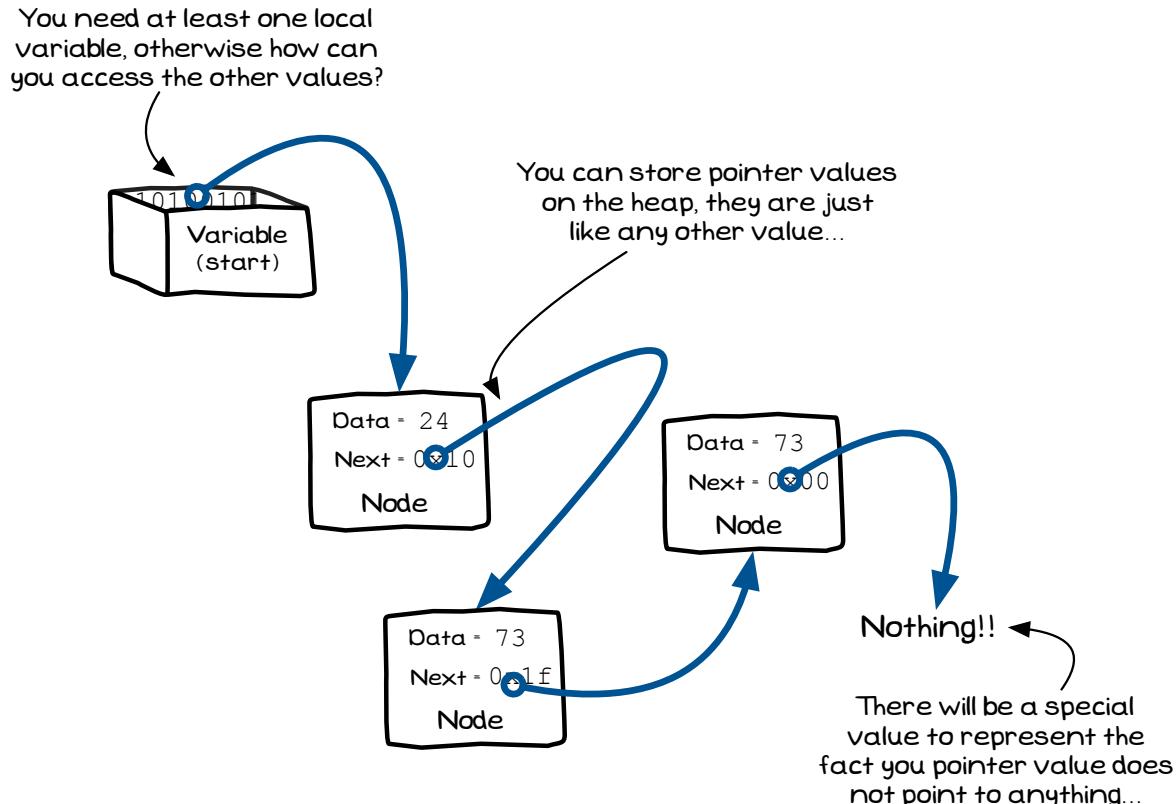


Figure 6.7: Pointers can be stored anywhere a value can be stored

Note

- A pointer value is no different from any other value, and can be stored on the stack, the heap, or in global variables.
- Languages provide a special value for pointers that do not point to a value. In C this is the NULL value, in Pascal it is the nil value, in both cases it is a value that points to nothing.

¹The Node would be a record type declared in the code. This type would contain an Integer value field named data, and a pointer field named next.

How are pointers used?

You need to be able to perform certain actions to make pointers useful. These include:

- You must be able to get a pointer to a value. For example, you should be able to get a pointer to a value stored in a Variable.
- Once you have a Pointer value, you must be able to follow that pointer to its value so that you can ...
 - Read the value it points to.
 - Store a value at the location pointed to.

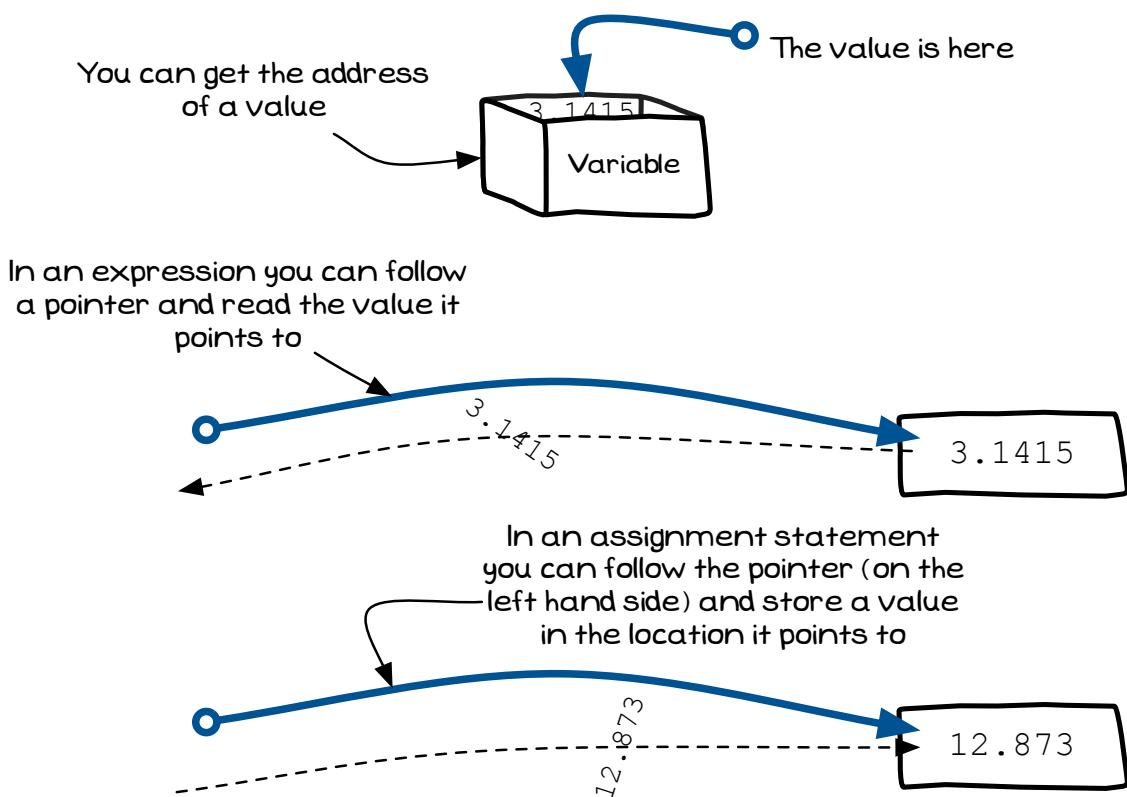


Figure 6.8: You can get pointers to values, and you can follow pointers to values

Note

- You can get the address of values in **Local Variables**, **Global Variables**, **Parameters**, fields of **Records** and **Unions**. Basically, you can get the address of any value you can read.
- Once you have the address (the Pointer value), you can store, or you can use it.
- You need to follow the pointer, called **dereferencing** the pointer, to read its value or to assign a new value to the location it refers to.
- Remember there are two values with pointers:
 1. There is the value of the pointer itself. This is the address that is pointed to. The circle at the start of the line in the illustrations.
 2. There is the value that pointed to. The one at the end of the arrow in the illustrations.
- You can interact with both of these values, depending on whether you *follow* the pointer or use the pointer's value directly.

What is the pointer value? What can you do with it?

Memory is laid out as a sequence of bytes, into which values can be stored. The bytes can be thought of as being in a long line, with each being given numbered based on its position in that line. So the first byte would be byte 0, the next is byte 1, the next byte 2, and so on. This number is then the **address** of that byte. So byte 975708474 is next to byte 975708475, which is next to byte 975708476, etc. This number is also unique, so there is only one byte 975708474. It is this number that is the Pointer value, the number of the byte that it is pointing to.

Figure 6.8 shows an example of memory used by an array of three values. Each value is a Double, so each one occupies 8 bytes. If the first is at address 975708474, then the second starts at address 975708482 (975708474 + 8 bytes). This Figure also shows a pointer, p, that points to this value. That means that p has the value 975708474, being the address of this value, stored within it.

One feature that languages have is called **pointer arithmetic**. When you add, or subtract, a value from a pointer the compiler will work in terms of the kinds of values the pointer points to. So in Figure 6.8 p is a pointer to a Double, this means that when you add one to p you get the value 975708482, which is 1 **Double** past p. Therefore, p + 2 would be 2 *doubles* past p, at 975708490, and so on.

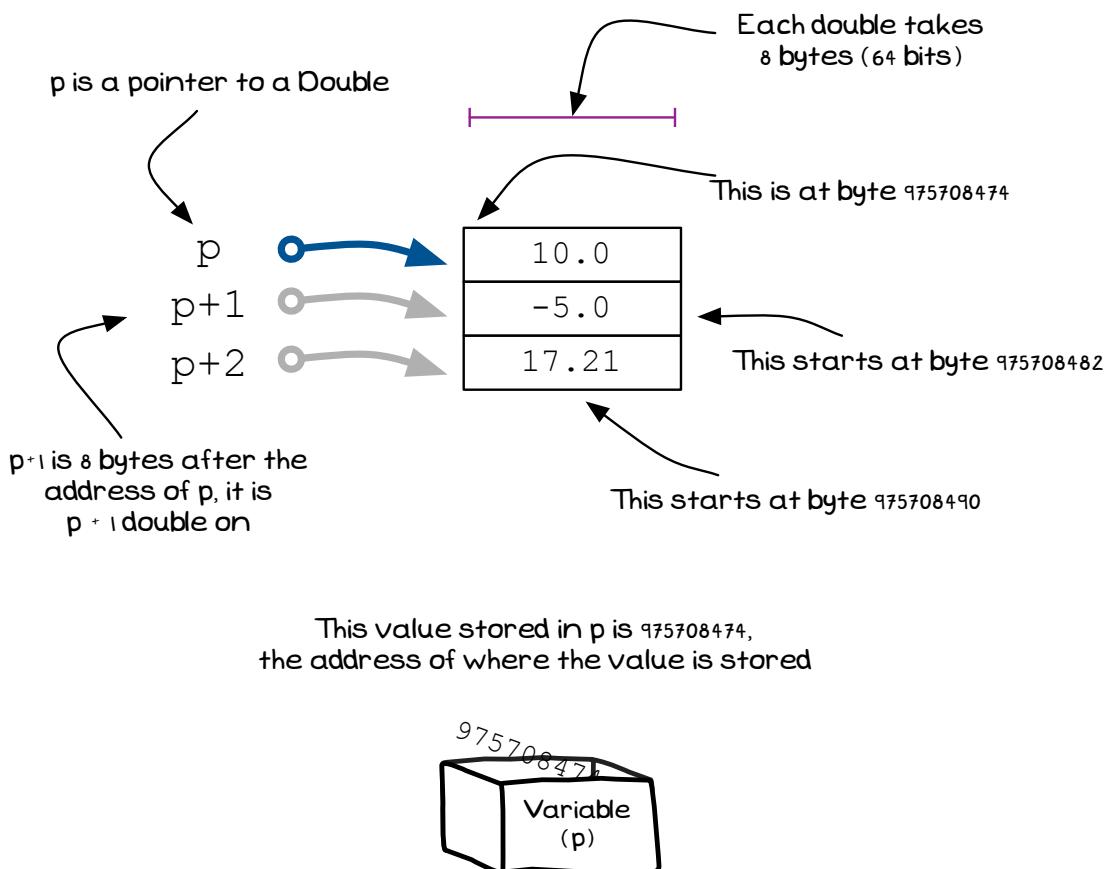


Figure 6.9: The pointer value is the *address* of the value it points to

Note

- Pointer arithmetic is something you need to know exists, but not something that you would work with frequently.

6.1.3 Allocating Memory

With dynamic memory management, one of the tasks you can perform is to request space from the heap. With this request the Operating System will locate available space and allocate this to you for use in your code. The only thing the Operating System really needs to know is how much space do you require? It can then search for a free space of that size, allocate this to you, and then give you a pointer to this newly allocated area of memory.

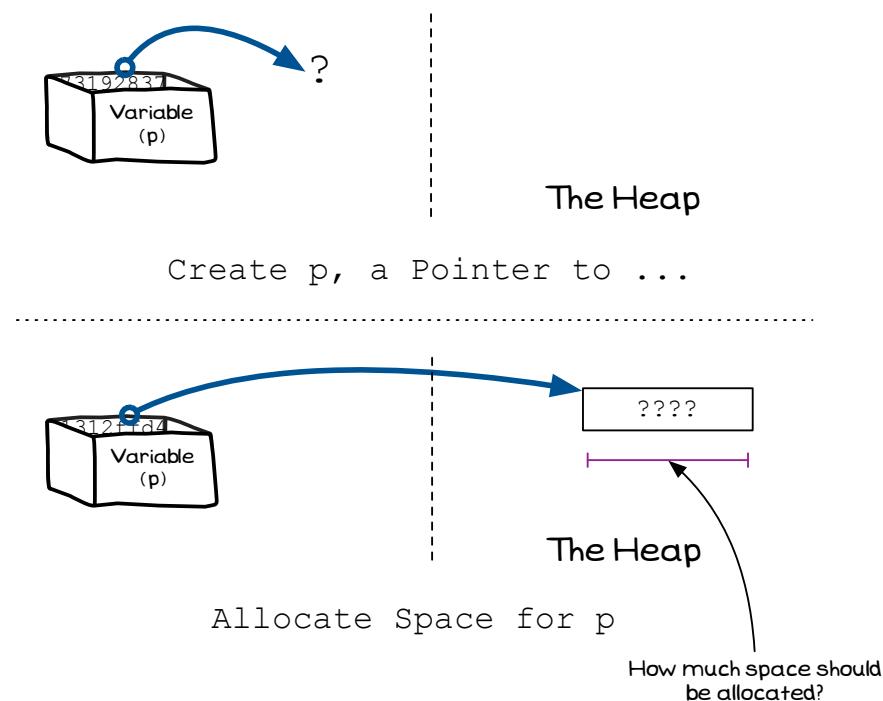


Figure 6.10: When requesting a memory allocation you need to specify the size you want

Note

- Allocating memory is an **action** you can perform by calling appropriate Functions or Procedures offered to you by the Programming Language.
- Your request for memory must include an indicating of the amount of memory that you require.
- You also need to have a pointer that will refer to the memory you are allocated.
- It is possible that your request will be denied, this occurs when the computer has run out of memory to allocate.

Explicitly allocating memory for a single value

If you want to store a single value on the heap you can ask to be allocated enough space for a single value. Figure 6.11 shows a Pointer (p) that points to an Integer value. If you want this value to be on the Heap you can ask to be allocated enough space to store an integer (4 bytes). The Operating System will then allocate you 4 bytes of space from the Heap, and give you the address of this space.

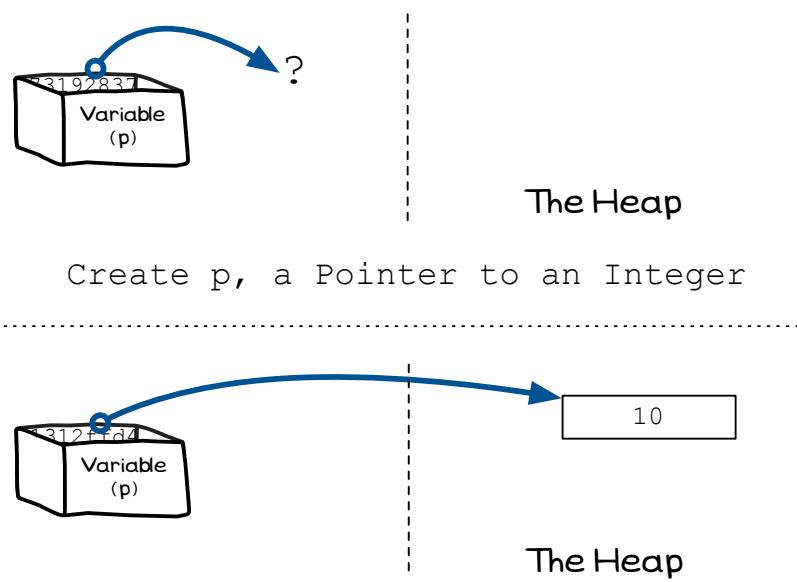


Figure 6.11: You can ask to be allocated enough space to store one value

C++

In C the `malloc` function from `stdlib.h` is used to allocate memory. With `malloc` you must specify the size of the memory you require in bytes. The `sizeof` function can be used to give you the size of the value you require. For the above example you would perform the following, with `sizeof(int)` giving you the number of bytes needed to store an `int` value:

```
p = (int*) malloc(sizeof(int));
```

Pascal

In Pascal the `New` procedure is used to allocate memory to a pointer. The `New` procedure is passed the pointer to allocate, and Pascal uses the information about what it points to, to determine how much space to allocate. For the above example you would perform the following, with `New` working out that you need to be allocated space for an Integer (the type of value that `p` points to): `New(p);`



Explicitly allocating memory for an array

Storing single values on the Heap can be useful, but often you want to be able to allocate enough space for a number of values. Arrays on the Stack must be of a fixed length, so this dynamic allocation allows you to have **variable length arrays**.

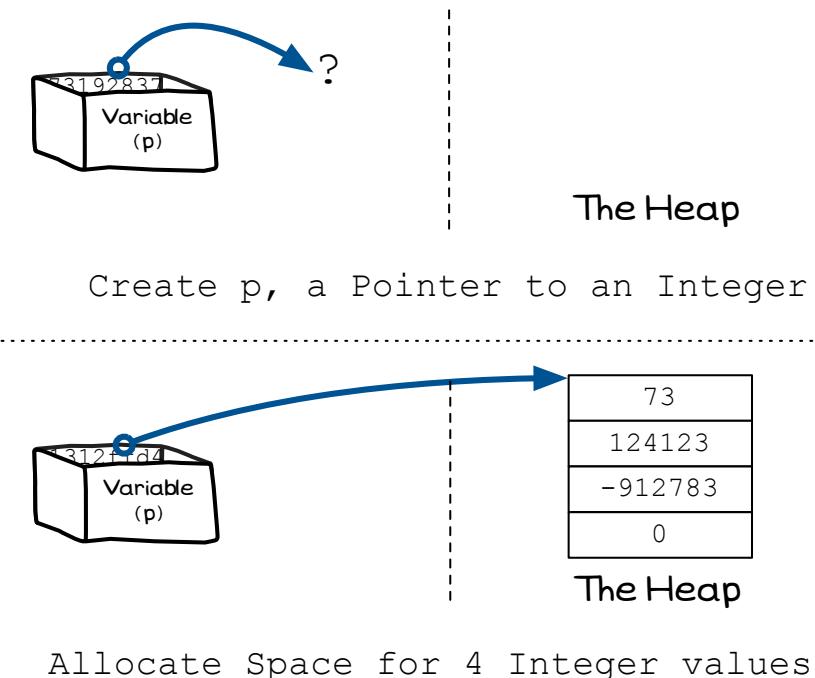


Figure 6.12: You can ask to be allocated a number of values

C++

In C you can use `malloc` to allocate space for a number of elements. Alternatively the `calloc` function from `stdlib.h` provides direct support for allocating space for an array. This function takes two parameters, the first takes the number of elements to allocate, the second the size of those elements. `calloc` also *clears* its allocation, setting each byte allocated to 0. The following code allocates 4 integer values as shown in Figure 6.12: `p = (int*) malloc(4 * sizeof(int));` or using `calloc`, which would set all values to 0, you could use `p = (int*) calloc(4, sizeof(int));`

In C you can use the standard array access mechanisms with pointers to access subsequent elements. So `p[0]` is the value in the first element of the array dynamically allocated in Figure 6.12, `p[1]` is the value of the second element, `p[2]` is the value of the third element, and so on. This relates back to Pointer arithmetic discussed in Section 6.1.2.

Pascal

Pascal has built in support for **dynamic arrays**. You can declare an array without a length, and then call `SetLength` to specify the size you want allocated. Behind the scenes Pascal will allocate the space for this array on the heap, and will take care of ensuring that it is cleaned up when you no longer have access to it. If you just want a block of memory to work with Pascal also offers a `GetMem` function.

Changing the size of a dynamically allocated array

The advantage of dynamic memory allocation is that you can change your allocations. If you asked for an array of two values, you may later want to be able to expand that array to three or four elements. Alternatively, an array with twenty elements may have some data removed and be shrunk down to only 5 elements. All of this is possible with dynamic memory allocation. You can ask to have the memory you were allocated changed to a different size.

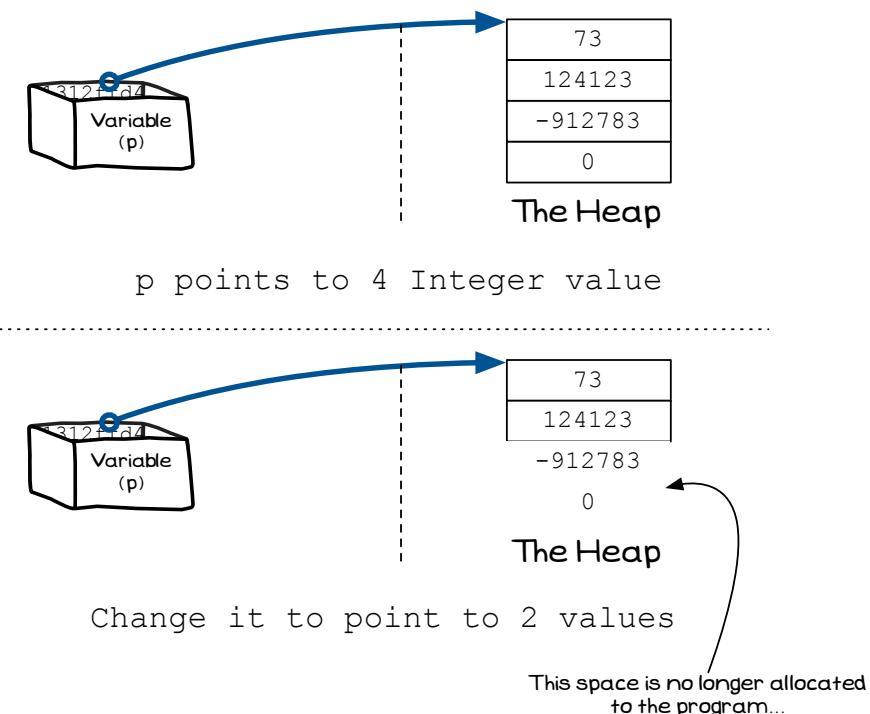


Figure 6.13: You can change the size of the allocation, growing or shrinking the number of element

Note

- These reallocations will keep the data that was in the array previously. Obviously if the new allocation is smaller than than existing one you will lose some values, but the others are kept.
- It is possible that the Operating System will need to move your new allocation, so if you change the size of an array you need to be careful if other pointers refer to elements in their old locations.

C++

The `realloc` function from `stdlib.h` allows you to change the memory allocation of a pointer. The following C code performs the reallocation shown in Figure 6.13:

```
p = (int*) realloc(p, 2 * sizeof(int));
```

Pascal

Pascal **dynamic arrays** perform this task for you. The `SetLength` procedure allows you to change the number of elements allocated to a dynamic array as you see fit.

6.1.4 Freeing Memory Allocations

Dynamic memory allocation requires that you manage the memory you are allocated yourself. You ask to be allocated memory, and it is your responsibility to tell the Operating System when you are finished with that memory. This is one of the main challenges of working with dynamically allocated memory. You need to take care to ensure that you do free the memory you have been allocated when you are finished with it, but at the same time you must make sure that you do not free the memory while it is still needed.

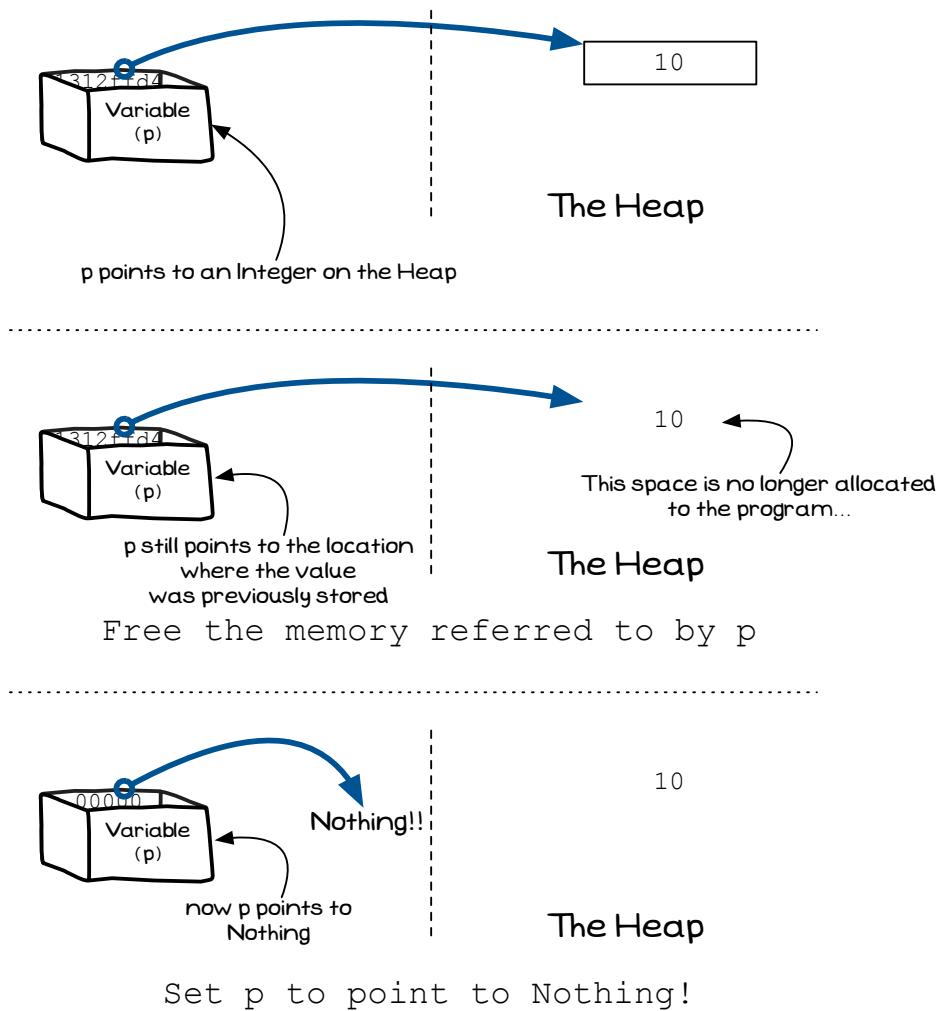


Figure 6.14: You can ask to be allocated enough space to store one value

C++

The `free` function releases the memory allocated to a pointer. To free the memory allocated to `p` as shown in Figure 6.14 you would use: `free(p); p = NULL;` This will both free the memory allocation and point `p` to nothing so that you cannot accidentally access it later.

Pascal

The `Dispose` procedure releases the memory allocated to a pointer. To free the memory allocated to `p` as shown in Figure 6.14 you would use: `Dispose(p); p := nil;` This will both free the memory allocation and point `p` to nothing so that you cannot accidentally access it later.

6.1.5 Issues with Pointers

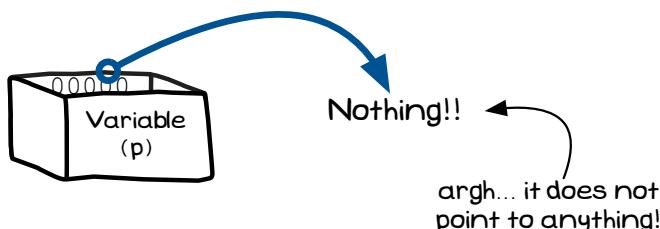
With great power comes great responsibility.

Pointers give you great flexibility in your programs, allowing you to allocate your program more memory as you need it, and return that allocation when you are finished with it. Conceptually this seems very simple, but pointers are a source of many issues in programs.

Access Violations

The first kind of error you are likely to encounter is caused by trying to accessing memory that does not exist. This will cause your program to crash. Figure 6.15 shows a common example where this occurs. Trying to follow a pointer to *Nothing* will crash the program with an access violation. This applies whether you are reading or writing to the value at the end of the pointer. The common name for this kind of error is a **segmentation fault**, *segfault* for short.

The only way to avoid these access violations is to **take care** with your pointers, see Figure 6.16. When you start working with pointers you need to go a little slower, and think a little more carefully about what it is you are doing. Having a good understanding of how these dynamic memory allocation tools work is the first step toward achieving this.



Follow the Pointer p, and ...

Figure 6.15: Trying to follow a pointer that goes nowhere is a runtime error

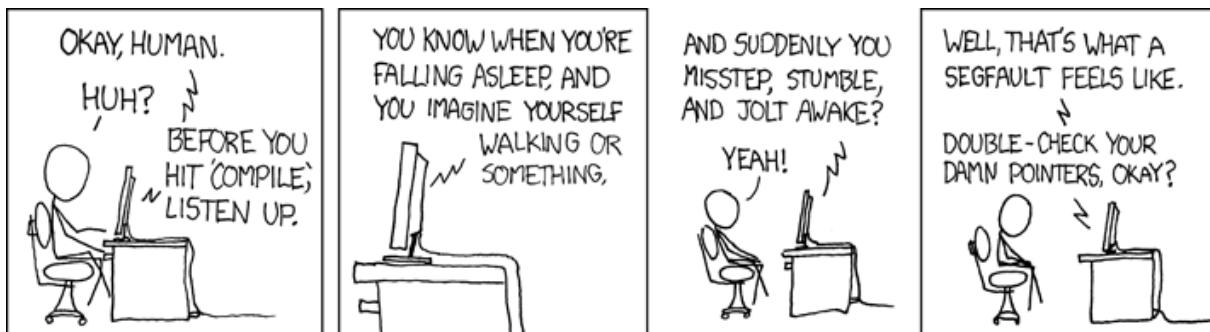


Figure 6.16: To avoid access violation, take care with your pointers. From <http://xkcd.com/371/>

Note

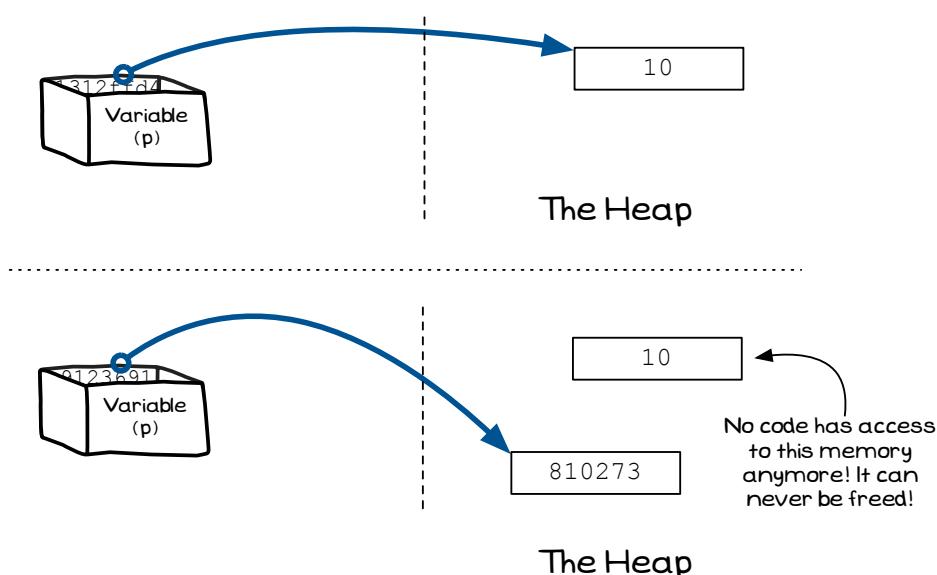
Here are some tips to help you avoid these access violations:

- If there is any doubt, check your pointers before using them.
- Always initialise your pointers to *Nothing*, as uninitialised pointers may point to something, but it won't be something useful.
- You can not see dynamically allocated memory in your code, so use a pencil and paper to sketch this out as you think through the code.

Memory Leaks

The next error is one that will not cause your program to crash, but will consume all of the computer's memory if it is allowed to run for an extended time. Remember that with dynamic memory allocation you are responsible for releasing the memory back to the system. If you do not do this there will come a time when there is no memory left to allocate... Memory leaks are hard to detect, as they do not cause your program to crash or generate any errors in its calculations. All that happens is that over time it consumes more and more memory.

Once again, the only way to avoid these issues is to **take care** with your pointers. You need to make sure that you know where the values are allocated, and where they are released. There should be reasons why you would the memory was allocated, and reasons why it is being released.



Allocate some memory, and point p at it

Figure 6.17: If you *forget* a piece of allocated memory, it can never be freed!

Note

Here are some tips to help you avoid memory leaks:

- Have a clear idea of where memory is allocated, and where it is freed
- Think about the pointer values in local variables at the end of each function and procedure. Do any of these values refer to something that no other pointer does? When the function or procedure ends, the variable's value will be lost. If it is the only thing referring to some allocated memory then that memory can no longer be freed, and you have a memory leak.

Accessing Released Memory

The next error occurs when you are overly zealous about releasing memory. You must not release memory before you are finished with it. The problem occurs when you continue to access a value, after its memory has been released. This is one of the most difficult problems to locate, as it will not cause any problems initially.

Take Figure 6.18 as an example. This demonstrates a case where two pointers refer to one value. It is possible to free that value via one pointer, and then forget that the second refers to the same location. When you read the value from p2 later, it is *likely* to still be 10, so the program will continue to run as normal. The issue will only appear later when something else is allocated to use that piece of memory. All of a sudden the value you thought was allocated to p2 is now changing apparently on its own. Worst of all, the actual cause of the bug could be hundred of lines of code away from where the problem appears. This is what makes this kind of error very difficult to find.

The solution, once again, is to **take care** with pointers.

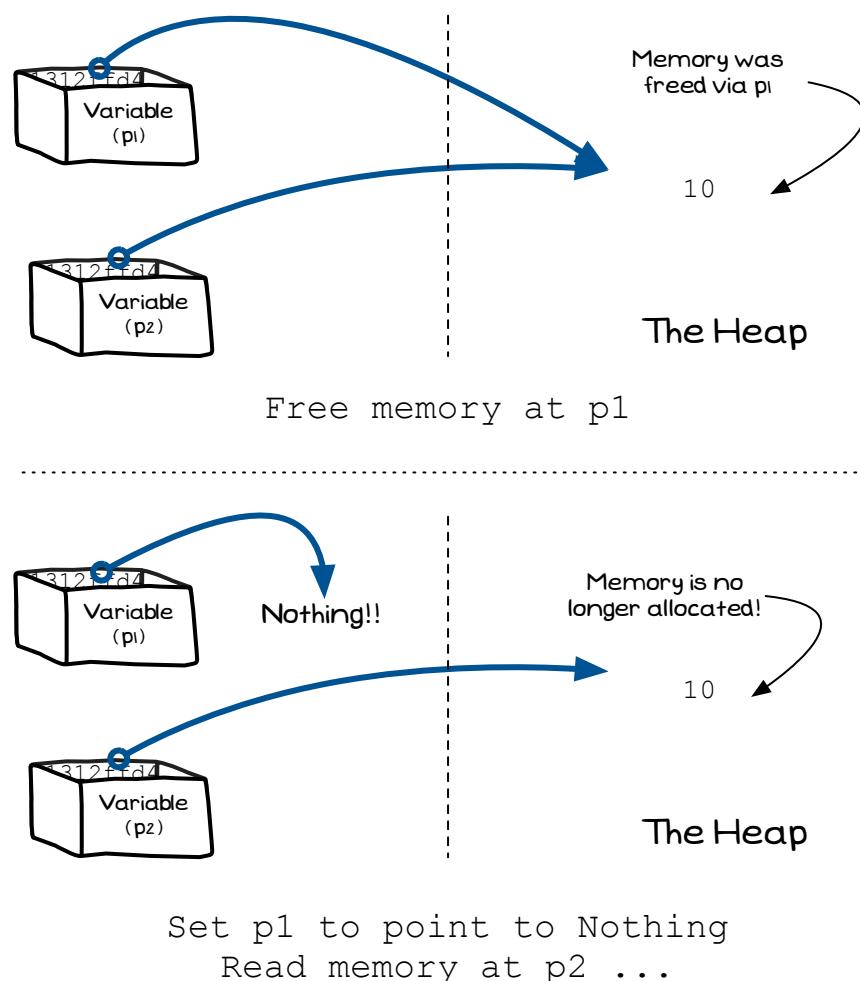


Figure 6.18: You can still read values from memory even when they are unallocated...

Note

Here are some tips to help you avoid accessing released memory:

- When you free memory, spend some time thinking about the things that could be referring to the value you just released.

6.1.6 Linked List

Pointers and dynamic memory allocation make it possible to store values in new and interesting ways. One way of structuring this data is to dynamically allocate each value, and link these together using pointers. An illustration of this is shown in Figure 6.19.

Linked lists have the advantage of being very fast to perform insert and delete actions, when compared with arrays. The disadvantage is an increase in storage size to keep all the pointers, and the fact you must loop through the nodes to access any value in the list.

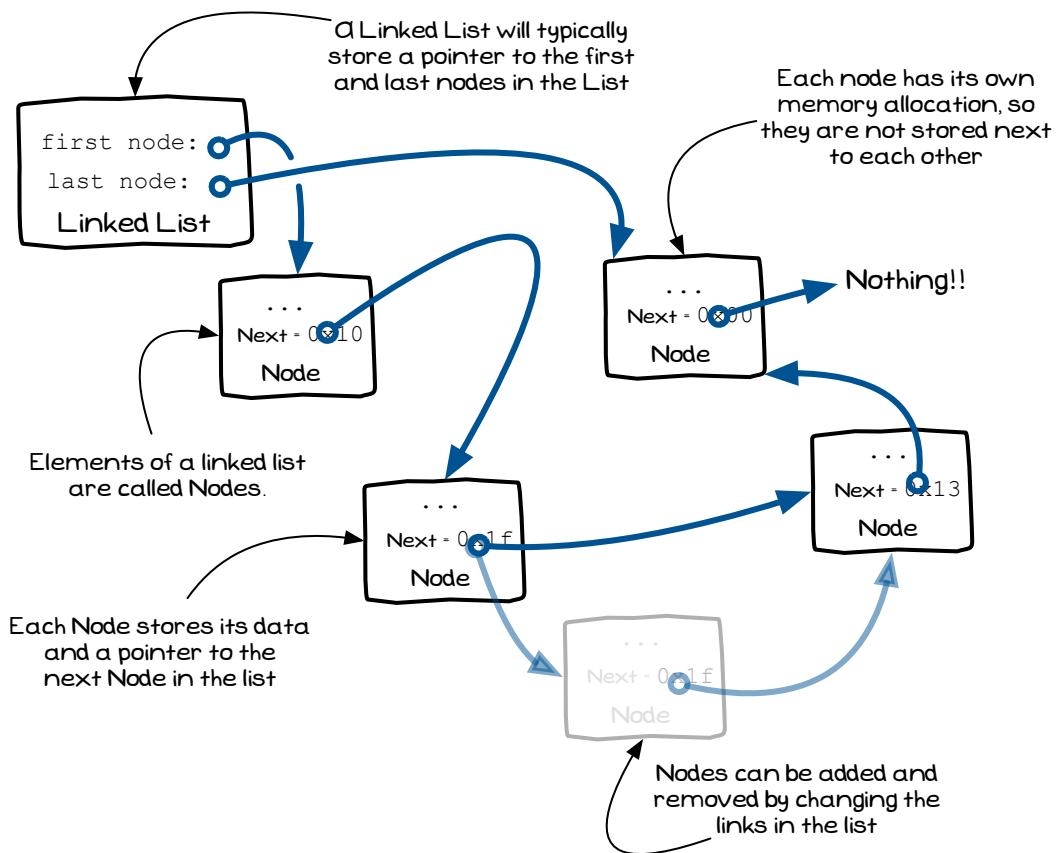


Figure 6.19: Illustration of a linked list in memory

Note

- A Linked List is a **term** given to a certain way data can be structured in memory.
- A Linked List has **Nodes**, the equivalent of the elements of an array.
- Each **Node**, has some data and a pointer to the **Next** element in the list.
- The **Last** element in the list has **Nothing** as its next node.
- To access a Node in the list you must loop through from the first node until you reach the node you are after.
- You can insert and delete elements by changing the links in the list.
- If the grey node in Figure 6.19 is being *inserted* then the previous node must be adjusted to point to it, and it to point to the next element of the list.
- If the grey node in Figure 6.19 is being *deleted* then the previous node changes its link to skip that node and point to the next node in the list.
- The pseudocode in Listing 6.1 shows the standard way of applying an action to each node of a Linked List.

Pseudocode

```
Procedure: ... Work with all nodes in a Linked List
-----
Parameters:
1: List (Linked List)
-----
Locals:
- Current (pointer to a Node)
-----
Steps:
1: Assign Current, the pointer to List's first Node
2: While Current is not Nothing
3:   ... Work with Current's data
4:   Assign Current, Current's Next
```

Listing 6.1: Pseudocode for looping through a linked list



6.1.7 Summary of Dynamic Memory Allocation Concepts

This chapter has introduced a number of concepts related to working with pointers and performing dynamic memory allocation.

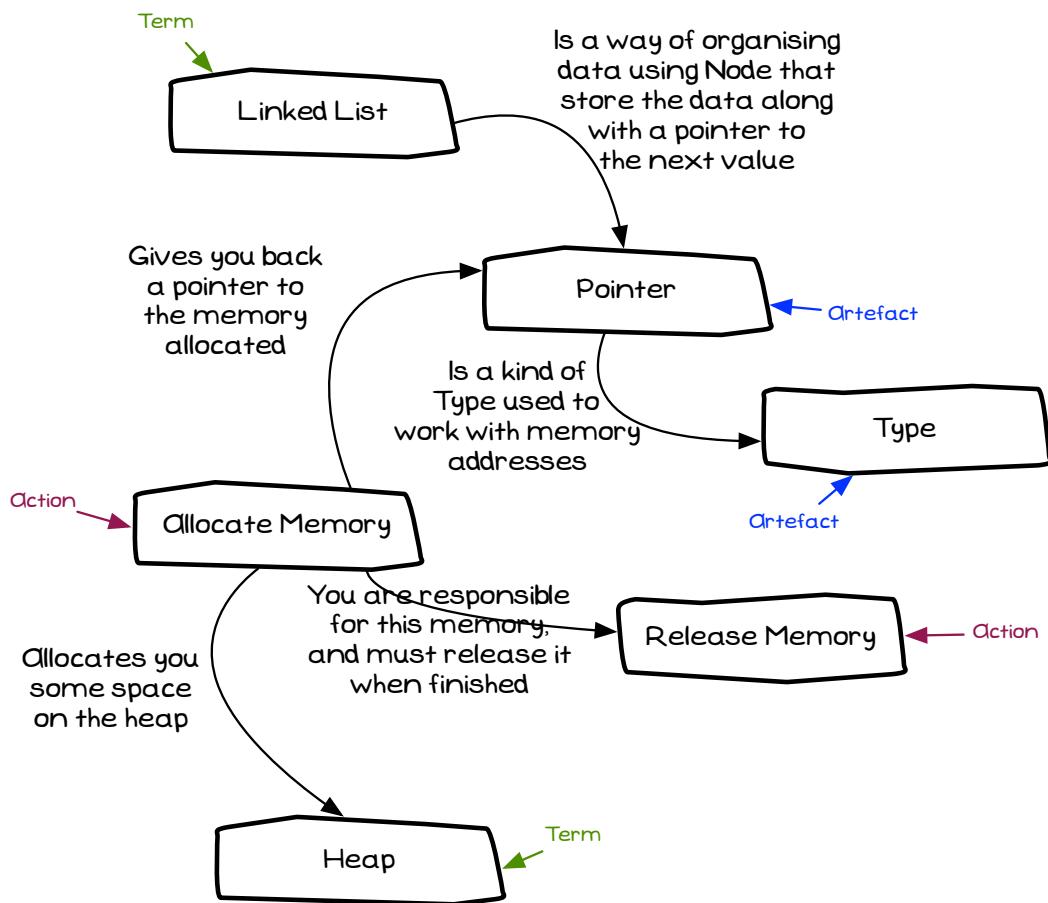


Figure 6.20: Memory management focuses on allocating memory, releasing this allocation, pointers, and the heap

Note

- **Heap** - an area of memory you can be allocated to store values.
- **Allocate Memory** - gives you ownership of a piece of the heap's memory.
- **Release Memory** - once you own the memory it is yours until it is released. If you forget to release it, it cannot be used by others.
- **Pointers** - are values that point to locations in memory. They store the address of the area of memory they refer to, and are needed to give you access to the heap.

6.2 Using Dynamic Memory Allocation

Dynamic memory allocation makes it possible for you to allocate additional space for your program to use from the [Heap](#). We are going to look at two different examples of how to use dynamic memory allocation, both of which extend the Small DB program created in Chapter [5](#). The first version will use a dynamically allocated array to allow the program to store a variable number of rows. Whereas, the second example will dynamically store each row, and link them together in memory.

6.2.1 Designing Small DB 2

Table [6.1](#) contains the extended description of the Small DB program that will be explored in this chapter. The main change is that in Chapter [5](#) the program only read in a fixed number of values. In this version the program will be able to respond to the user wanting to add or delete data stored in the program. In effect the program will store a *variable* number of elements, with elements being added and removed by the user.

Program Description	
Name	<i>Small DB 2</i>
Description	<p>Stores a number of values for the user. These values can be text (up to 7 characters), a whole number, or a real number. Each value entered is stored in a row with a unique identifier that will be assigned by the system. The first value will be assigned the id 0, the second will be assigned the id 1, and so on.</p> <p>The program will show the user a menu, and allow them to add new data to the program, print the data in the program, delete data from the program, or quit.</p> <p>Add: will read a value from the user and store it within the program. This data will be allocated a sequential id.</p> <p>Print: will print all of the values from the program, along with their types.</p> <p>Delete: this will ask the user to enter the id of the data they want to delete, and then search for this data in the program. If a row exists with this id, it is removed from the program.</p> <p>Quit: terminates the program. The menu will be repeatedly shown to the user until they decide to quit.</p>

Table 6.1: Description of the Small DB program.

As before, the process to design and implement this program will follow a number of steps:

1. **Analyse** the problem (understand it and associated tasks).
2. **Design** the solution, its artefacts and control flow.
3. **Implement** the design in code.
4. **Test** the solution, compiling and running it to check it works as required.

6.2.2 The Analysis Phase: Understanding Small DB 2

The first task in any software development is to understand, as fully as you can, the requirements of the program, and the associated tasks and data. As this is an extension of the Small DB program from Chapter 5 most of that analysis has already been done. A number of tasks and data types were identified in the process of [Understanding Small DB](#), and [Choosing Artefacts for Small DB](#).

To successfully implement these new features you must first think about what is required of the new software. Then you can move on to think about how you can achieve these goals in your code.

As identified above, the new goal is to allow the user to enter a variable number of values. They need to be able to add and remove data as they wish. To achieve this a menu will need to be added that allows them to choose the action they want to perform next. From the menu the user will be able to choose to **add**, **print**, or **delete** data or to quit the program. These tasks give us hints about the kinds of functions and procedures we can add to achieve this.

As well as thinking about additional tasks, you should also see if there is additional data. Read over the description and see if you can identify any additional data that the user may need to either enter into the program, or get out of it.

At first glance there may not appear to be any new data needed by this program. After all it is still allowing the user to enter the same values that it did in Chapter 5. There is, however, one additional kind of data that is hinted at in the description. There will be data associated with the **menu**. The program is likely to need to work with values associated with the options the user will be selecting. This can be modelled in the code, making it a better reflect the the concepts associated with the program.

6.2.3 The Design Phase: Choosing artefacts and designing control flow

Once you have understood what is required, you need to set about designing the solution. This involves choosing the artefacts to create and use, and designing the control flow within the functions and procedures you create. This is all about making decisions, how will you structure this functionality? What control flow will enable this behaviour? The many decisions you make will define the overall design of the software.

This program already has an existing design, that needs to be extended. The data is described in the *Data Dictionary* in Table 5.2. An overview of the functions and procedure of the solution was shown in the structure chart in Figure 5.19. These are a good starting point, and for the most part will require no changes. The new additions will build on top of these.

Modelling the menu options

Modelling the data should always be high on your priority list, so the first task can be to think about how the menu options can be modelled. The user needs to be able to select from **add**, **print**, **delete**, and **quit**. At later stages there may be more options, but at this point that is the list.

If you think back to Chapter 5, there are only three kinds of type you can use to model the data associated with a program: **records**, **unions**, and **enumerations**. These are the tools that are available to you to model your data. A *record* has a number of values one for each field, a *union* has one value with the type based on the field it was stored in, a *enumeration* represents a list of options.

An **enumeration** is the obvious choice for modelling the list of options the user can choose from the menu. We can create a *Menu Option* type that has the values ADD_DATA, PRINT_DATA, DELETE_DATA, and QUIT.

Modelling the dynamic rows

One of the first processes that needs to be designed is the process to Add a Row to the data stored in the program. This can use the Read Row function that already exists to get the data from the user, so its focus is on allocating additional space for the row and determining the row's id.

A review of the code in the current Small DB program indicates that the row's index and its id were the same value, and Main was taking advantage of this. When Main called Read Row, it passed in i as the id for the new row. In effect, the value in i was being used as the index that looped over the elements of the array as it was being populated, and it was keeping track of the id value for the newly created rows. This is no longer going to be the case as the user can now add and delete rows. When they delete a row the index and the id will no longer linked, so one variable cannot track both values. A **Next Row Id** value needs to be kept somewhere.

The **Next Row Id** and the **Rows** data are associated. These values can be modelled as a single **Data Store record**, with each Data Store having fields for the **Next Row Id** and **Rows**. This will keep the relevant information together, and allow the code to work with these values as a group.

- Next Row Id will be an integer, it will be assigned the value 0 at the start and have its value incremented each time a row is added.
- Rows will be a dynamically allocated array of Row values.

C++

In C, another issue that needs to be resolved is the fact that there will now be a *variable* number of rows in the program. The number of rows will also need to be recorded, and maintained as rows are added and deleted. A **Row Count** value can also be added to the Data Store record.

```
typedef struct {
    int next_row_id;      // The id of the row that will be added next
    int row_count;        // The number of rows in the array
    row *rows;            // A pointer to the rows in memory
} data_store;
```

Pascal

Pascal has built in support for dynamic arrays. This means that it keeps track of the Row Count for you. The number of rows in a Data Store value can be determined by using the Length function with the Data Store's Rows field.

```
type DataStore = record
    NextRowId: Integer; // The id of the row that will be added next
    Rows: array of Row; // The dynamic array of rows
end;
```

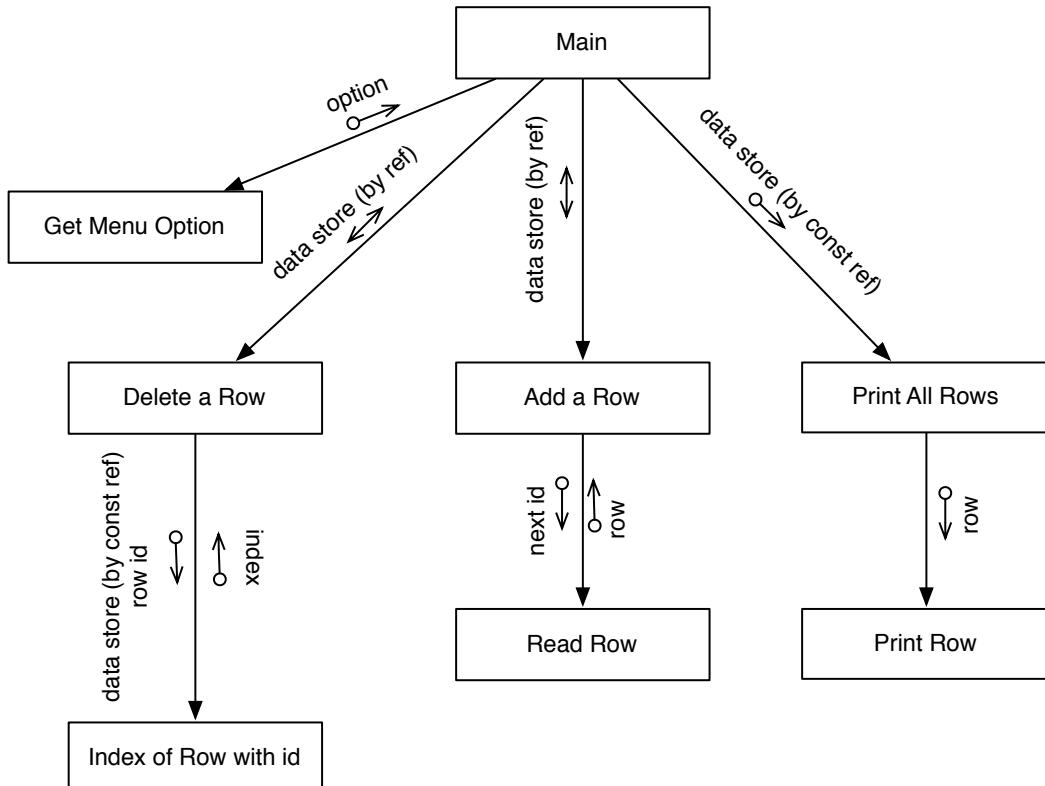
Table 6.2 shows the new data dictionary for Small DB 2. This includes the addition of the Menu Option enumeration and the Data Store record.

Designing Small DB 2's Structure

The next step is to choose the functions and procedures that need to be created or used to implement this program. Once again, this can build on the structure implemented in Small DB, as shown in Figure 5.19. This included the code needed to read a row from the user, and to output the row to the Terminal.

Small DB 2 needs to add some additional functionality to this program. The following list shows the tasks that need to be coded, and the functions or procedures that will code their

Data	Details	
Menu Option	An enumeration with the following options:	
ADD DATA	The user wants to add a row.	
PRINT DATA	The user wants to print all rows.	
DELETE DATA	The user wants to delete a row.	
QUIT	The user wants to quit.	
Row	The record/struct from Table 5.2 that stores the row's id, value, and kind.	
Data Kind	The enumeration from Table 5.2 representing the kind of value that can be stored in a row.	
Column Value	The union from Table 5.2 that stores either an integer, double, or text value.	
Data Store	A record to store row data with the following fields:	
Next Row Id	The id value of the next row to be added.	
Rows	A dynamically allocated array of Row values.	
Row Count	The number of rows in the Data Store. (C Only)	

Table 6.2: Data Dictionary for Small DB 2**Figure 6.21:** Structure chart for Small DB 2

behaviour. This is shown in the Structure Chart in Figure 6.21.

- Show the menu to the user, and get the option they want to perform (Get Menu Option function).
- Add a row to the data managed in the program (Add a Row procedure).
- Print all of the rows (Print All Rows procedure).
- Delete a row from the data managed by the program (Delete a Row procedure).

In Small DB the logic for Print all Rows was coded into Main. This logic can be moved from there into its own procedure. The control flow for the other functions and procedures will need to be designed.

Control flow for Get Menu Option

The code for Get Menu Option should be fairly simple. The basic actions it needs to perform will be:

1. Output the text showing the list of options to the Terminal.
2. Read a number from the user, making sure it is in the range of the list of options (1 to 4).
3. Return the value of the option selected as a Menu Option value.

The first part of this process will involve a sequence of output commands, displaying the different text to the Terminal. The code to read a number from the user will need a standard *validation* loop, repeatedly asking them to enter a number until they enter one between 1 and 4. The final part can use a [Case Statement](#) to return the correct result from the function.

Control flow for Add a Row

The process for adding a row will involve the following steps, as shown in Listing 6.2:

1. Record the next row id, and then increase the Next Row Id in the Data Store.
2. Increase the memory allocated to the Data Store's Rows (and the Row Count, in C), and store the Row read from the user into this newly allocated memory.

Pseudocode

```

Function: Add Row
-----
Parameters:
1: Db Data (Data Store, passed by reference)
-----
Locals:
- Row Id (Integer)
-----
Steps:
1: Assign Row Id, the value of Db Data's Next Row Id field
2: Increase the Db Data's Next Row Id field by 1
3: Increase memory allocated to Db Data's Rows by 1 Row (and Row Count in C)
4: Store in Db Data's new row, the result of calling Read Row (row_id)

```

Listing 6.2: Pseudocode for the Add a Row procedure



Control flow for Delete a Row

Deleting an element out of an array is a common task, but one that will require you to perform all of the hard work. Remember that arrays are *contiguous blocks of memory*, so technically

you cannot delete an element from the middle of this array. You can only remove the last element. There are three options for how this can be achieved, as shown in Figure 6.22. All of the options involve moving data, and then resizing the dynamically allocated array to have one fewer element than before. The different options involve copying different pieces of data to free the last slot so that its allocation can be released.

1. Only copy the last element over the element to be deleted. This is the fastest, but the order of the elements is not preserved. If this is important then you cannot use this approach.
2. The second option is to copy the value of each element in the array back over the previous element. This maintains the order of the elements, but takes more time as you have to copy all of the values after the element being deleted back one spot.
3. A faster option is to take advantage of the fact the array is stored as a contiguous block, and to perform a bulk move/copy of the memory past the element. In effect, you can copy all of the elements with one request and take advantage of the hardware which is optimised for this kind of task.

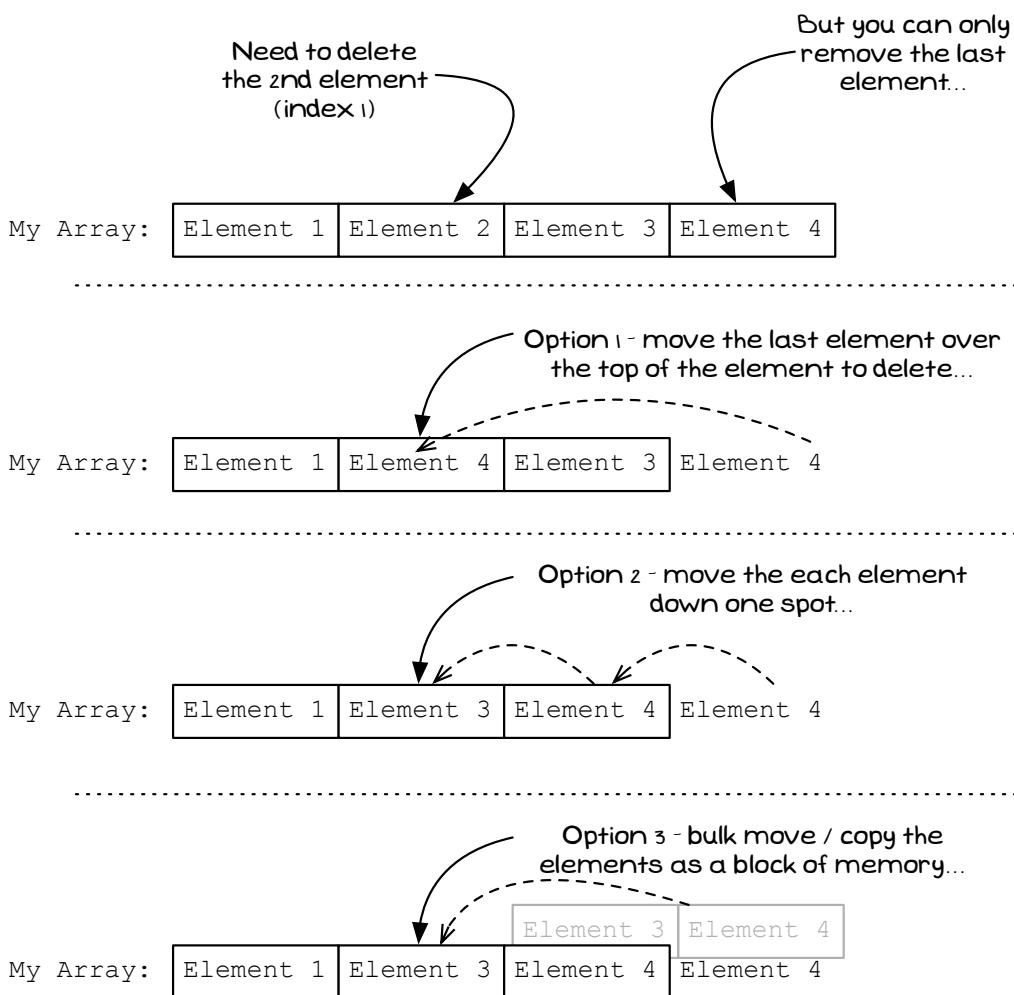


Figure 6.22: Options for deleting an element from an array

Figure 6.23 shows the flowchart of the process for deleting a node from the Rows array using Option 2. This option has been chosen as it helps demonstrate the use of the elements of the array, as well as working with the dynamic memory.

The implementation of this option also demonstrates a frequently seen pattern when working

with two consecutive elements of an array. In this case one element is being copied over another, but many algorithms will require you to work with two elements at a time. Each time through the loop, this code will access the i^{th} element and its neighbour, the $(i+1)^{th}$ element.² This means the loop needs to go from a starting element, to the **second last** element of the array. If i looped to the *last element* of the array then $i+1$ would attempt to access an element past the end of the array.

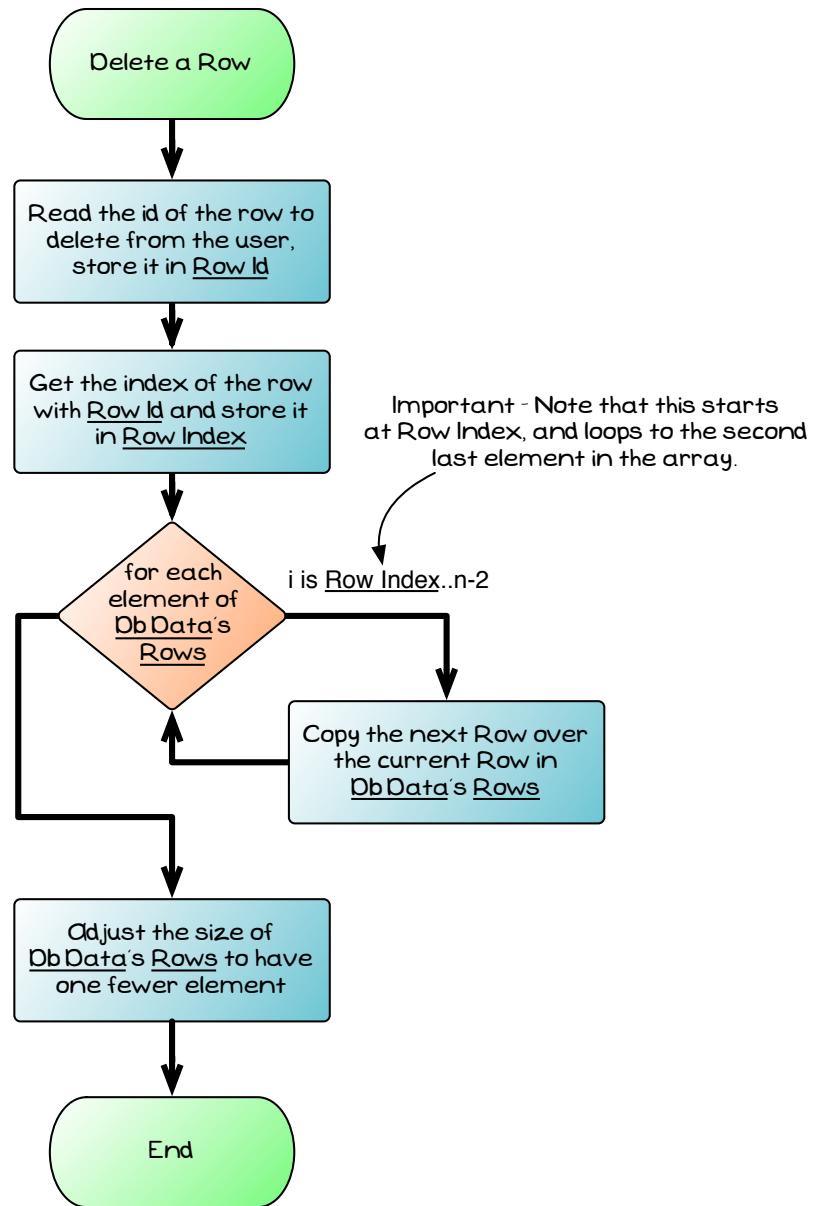


Figure 6.23: Flowchart describing the process for deleting a row with a given id

Control flow for Index of Row with Id

The Index of Row with Id function will search a Data Store's Rows for the index of the Row with the Id it is searching for. This is used by the Delete a Node procedure so that it can find the index of the row it needs to delete.

²An alternative version of this is to start 1 element past the first element you want to interact with, and use the i^{th} element and the previous element, the $(i-1)^{th}$ element.

This is implemented using a standard **search pattern**. This pattern involves looping over all of the elements in the array, and *for each* element checking ‘*Is this the element I am after?*’. When a match is found the search can end, as can the function. This can be implemented using the appropriate [Exit](#) statement for the language, with the function returning the index of the row it found, in this case. However, if the loop gets to the end of the array without finding a match, then there is no element in the array that matches the search and the function can return a value that indicates no match was found, in this case the index -1 is returned as this cannot be a valid index. The flowchart illustrating these steps is shown in Figure 6.24.

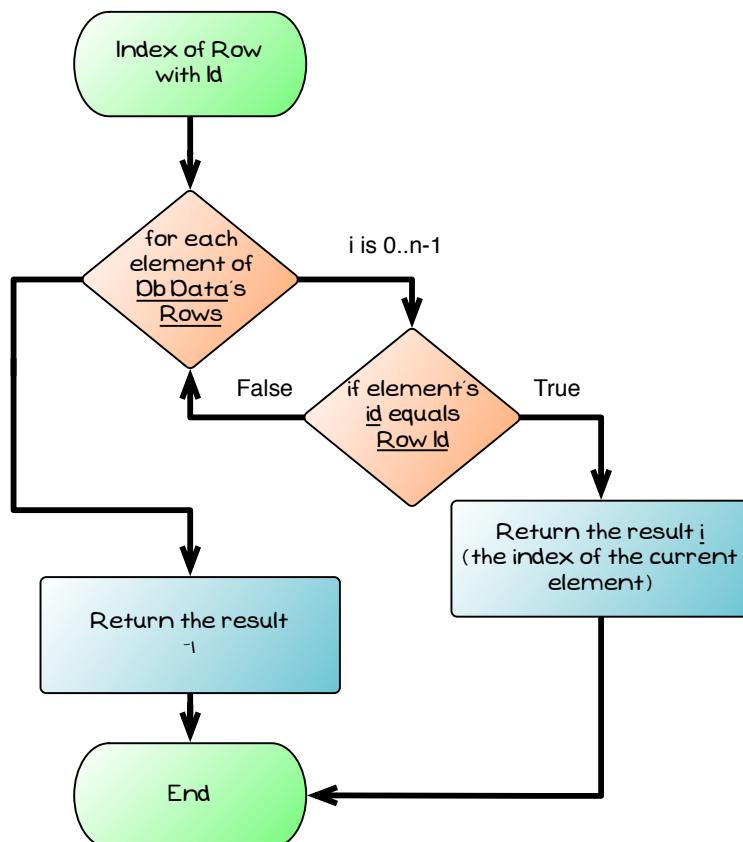


Figure 6.24: Flowchart describing the process for finding the index of a row with a given id

Control flow for Main

Main is the last remaining procedure. The implementation of this will require you to declare a Data Store variable that will be manipulated by the other procedures previously discussed. This will need to be initialised with no elements in its Rows, and other a Next Row Id of 0.

The control flow of Main will involve a [Post-Test Loop](#) that will repeat code until the user chooses to quit. Within the loop Main can get the option the user wants to perform by calling the Get Menu Option function, can then use a [Case Statement](#) to run either the Add a Row procedure, the Print all Rows procedure, or the Delete a Row procedure.

6.2.4 The Implementation Phase: Writing the code for Small DB 2

The following two sections, Section 6.3 Dynamic Memory Allocation in C and Section 6.4 Dynamic Memory Allocation in Pascal, contain a description of the syntax needed to code dynamic memory allocation in the C and Pascal programming languages. Use this information to write the code for Small DB 2, and other programs.

6.2.5 The Testing Phase: Compiling and running Small DB 2

Whenever you work with dynamic memory allocation, you need to spend a good proportion of your time checking that your solution is working. This is now an interactive program, so you can test multiple things each execution. The following are some of the aspects that you should test in this program:

- Test the basic functionality:
 - Can you add new rows?
 - Can you delete a row?
 - Can you print the rows?
 - Are you able to quit the program?
- Test for potential issues related to memory allocation:
 - Try deleting the first row, the last row, and a non-existent row.
 - Delete all of the rows.
 - Print when there are no rows.
 - Delete and then add, delete all rows then add, etc.

Think about the places where you may have made a mistake, and test to check that you can not cause the program to crash.

Note

There **is** a bug in the current design, as presented in this chapter. If you run these tests you should be able to find the bug, which has the potential to cause the program to crash. See if you can locate the bug, correct it, and implement the required fix. Like most bugs the fix only requires a small change, so think carefully about what is required.



6.2.6 Designing Linked Small DB 2

Arrays are only one way of dynamically allocating space for a program. In an array the elements are allocated in a contiguous block, each element next to the previous one. This structure is good when you want to access an element based on its position, but operations like delete and insert can be tricky as you need to move elements around to make or remove space.

An alternative to the array, is to dynamically allocate space for each value individually, and to use pointers to record their locations. The general structure is called a **Linked List**, and we can use this to manage the Rows in the linked version of the Small DB 2 program.

Figure 6.25 shows the difference between how the rows are stored in the previous array version of Small DB 2, and the linked version we will now explore. In the array version, each Row is stored next to the previous one in an array. With this version you can use an index value to access the middle elements of the array quickly, but it is slower to delete or insert elements.

The linked version of Small DB 2 is made up of nodes (the Rows) that each store the data and a *link* (pointer) to the next node in the list. In this version the Rows are not stored next to each other, so you cannot use an index to calculate the position of any one element. Instead you need to start at the first element, and work your way through the list one element at a time using the next pointer to move from the current node to the next node of the list. Inserting or deleting elements of the list can be much faster as the *links* can be adjusted to introduce new elements, or remove existing one from the list.

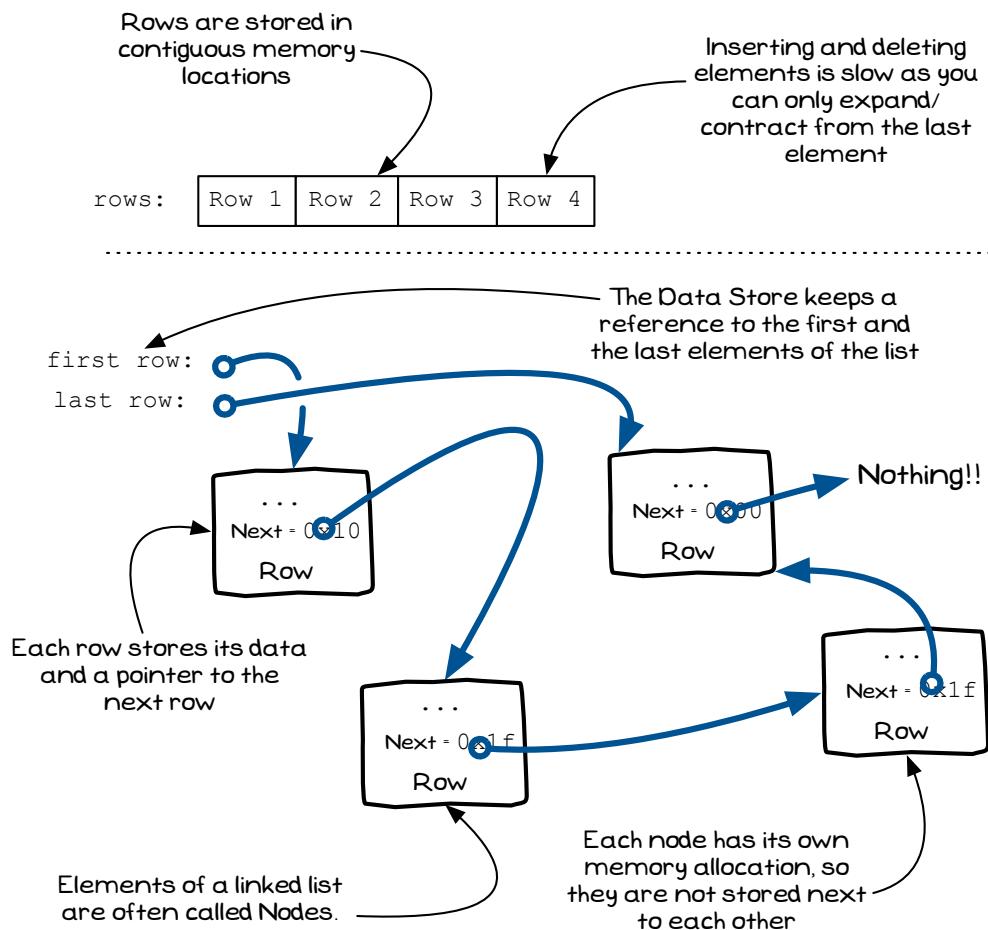


Figure 6.25: Illustration of memory layout for Array and Linked versions of Small DB 2

6.2.7 The Design Place: Designing Linked Small DB 2

The linked version of the Small DB 2 program is an alternative implementation for the same program developed in Section 6.2.1. This means that the information from the analysis phase is still valid, and can be used to inform what must be done in this program. Similarly, the testing strategies developed will also be valid so when the design and implementation are complete you can test it in the same way as the previous program.

The parts that do need to change will be the **design** and **implementation**. The design needs to structure the data differently, and this will impact on the structure of the code as well. The first step, therefore, is to design the new structure for the data and then to use this to determine the new structure for the code.

Modelling the links in data

Table 6.3 shows the new data dictionary for the linked version of the Small DB 2 program. Most of the changes relate to the Data Store record. This used to store all of the rows in a dynamically allocated array. Now, in the linked version, the Data Store includes two pointers: one pointing to the first Row, the other pointing to the last Row. Only the first of these two is actually required, but the pointer to the last Row will make some tasks easier.

The First Row pointer, points to the first Row value allocated on the heap. Tasks like Print all Rows will use this pointer to start looping through the Rows. The Last Row pointer exists to make it easier to add new Rows to the Data Store. New Rows are added to the end of the list, and the Last Row pointer means you can get to the end of the list without first having to loop through each node.

Data	Details	
Row	The record/struct from Table 5.2 that stores the row's id, value, and kind. This now also has an additional field to point to the location of the next Row	
	Next	A pointer to the next Row.
Data Kind	The enumeration from Table 5.2 representing the kind of value that can be stored in a row.	
Column Value	The union from Table 5.2 that stores either an integer, double, or text value.	
Menu Option	An enumeration from Table 6.2 with options for adding a row, printing all rows, deleting a row, or quitting the program.	
Data Store	A record to store row data with the following fields:	
	Next Row Id	The id value of the next row to be added.
	First Row	A pointer to the first Row in the Data Store.
	Last Row	A pointer to the last row in the Data Store.

Table 6.3: Data Dictionary for Linked version of Small DB 2

One tricky aspect of the linked Rows, is that the Row must include a pointer to a Row. In most cases the compiler requires that the thing you are using is declared before its use. Here that is not possible, the declaration of the row must include the point to the row which is still being declared. Each language caters for this in its own way, the tricks for doing this in C are shown in Figure 6.26, the Pascal version is shown in Figure 6.27.

C++

The following code shows how the new Data Store and Row structures can be implemented in C. The one trick here is that you must use the struct name of the Row when declaring the next field. This is because C has not yet encountered the type name for the Row, so you need to use its alternate name.

```
// The Row record/structure. Each row contains an id
// a kind, and some data (a Column Value).
typedef struct row_struct {
    int                  id;
    data_kind            kind;
    column_value         data;
    struct row_struct   *next; // The next Row in the list
} row;

// The data store is a dynamic linked list of rows, keeping track
// of the number of rows in the list, and the id for the next row
typedef struct {
    int      next_row_id;    // The id of the row that will be added next
    row     *first_row;      // A pointer to the first row
    row     *last_row;       // A pointer to the last row
} data_store;
```



Figure 6.26: C code for declaring the Row, with a next pointer

Pascal

The following code shows how the new Data Store and Row structures can be implemented in Pascal. The trick here is to declare an alias for the RowPtr (a pointer to a Row) before the Row declaration, but in the same type declaration part.

```
type
  // A Pointer to a row (Row must be in same type decl part)
  RowPtr = ^Row;

  // The Row record/structure. Each row contains an id
  // and some data (a Column Value).
  Row = record
    id: Integer;
    data: ColumnValue;
    next: RowPtr;           // The next Row in the list
  end;

  // The data store is a dynamic linked list of rows, keeping track
  // of the number of rows in the list, and the id for the next row
  DataStore = record
    nextRowID: Integer; // The id of the row that will be added next
    firstRow:    ^Row;   // A pointer to the first row
    lastRow:    ^Row;   // A pointer to the last row
  end;
```



Figure 6.27: Pascal code for declaring the Row, with a next pointer

Reviewing the structure for the linked version of Small DB 2

The overall structure for Small DB 2 will not change much, as the basic actions the program needs to complete remain the same. The only real change is that the Index of Row with ID function is no longer required, as indexes no longer serve as a means of accessing the Row values in the DataStore. The updated version of the Structure Chart is shown in Figure 6.28.

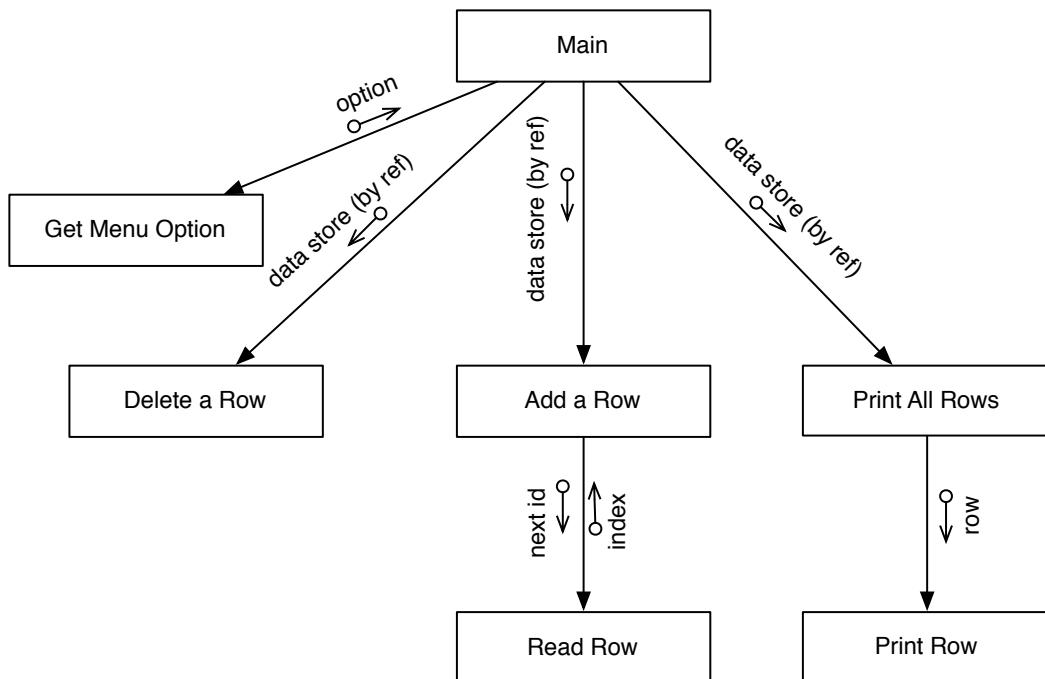


Figure 6.28: Structure chart for the Linked version of Small DB 2

Adding a Row to the linked version of Small DB 2

The first activity that can be examined in detail is the Add a Row procedure. This will need to create a new Row value, and have it added to the end of the list. To start with, let us have a look at how this should work conceptually.

Figure 6.29 shows an illustration of the add process for the first row. At the start the code has a single Data Store value, this will be Db Data from within the Main procedure. When the program starts the data store will need to have its First Row and Last Row fields initialised to point to *Nothing*. This indicates that there is no first row, or last row in the Data Store at this stage.

When a new Row is added the first step is to allocate space for it on the heap, and then to read the user's input into that space (using Read Row). As this Row will be added to the end of the Data Store, its Next field can be set to *Nothing* to indicate that there are no more Row values after this one.

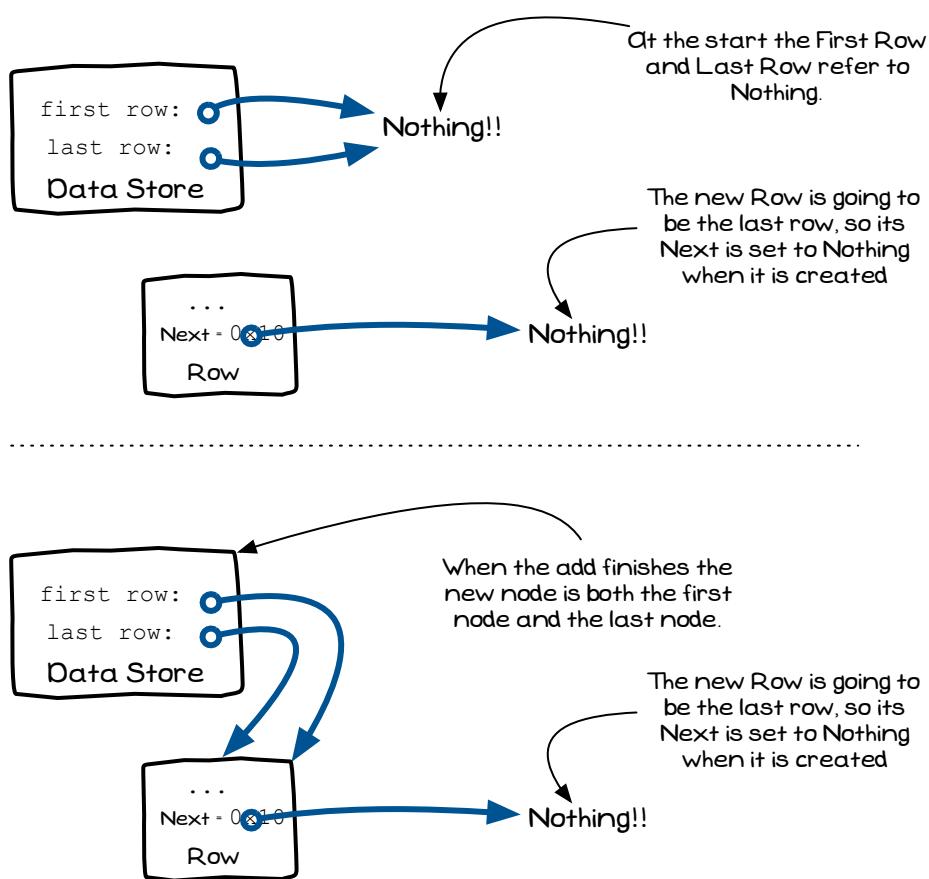


Figure 6.29: The first Row becomes the start and end of the List

The code for Add a Row now needs to link this new Row into those already in the Data Store. This is where it checks to see if there is a Row value currently referred to be the last row of the Data Store value. As this refers to *Nothing* at this point, the code takes a branch that sets the first row and the last row both to refer to the newly created Row value. At this point, the new Row is both the first and the last row in the Data Store.

Most the time when you add a new row it will not be the first one in the list. Typically, you will need to add the row to the existing rows in memory. This involves updating the current last row so that its next points to the newly created row, and then you can change the Data Store's last row to point to it as well. This is shown in Figure 6.30, with the whole pseudocode shown in Listing 6.3.

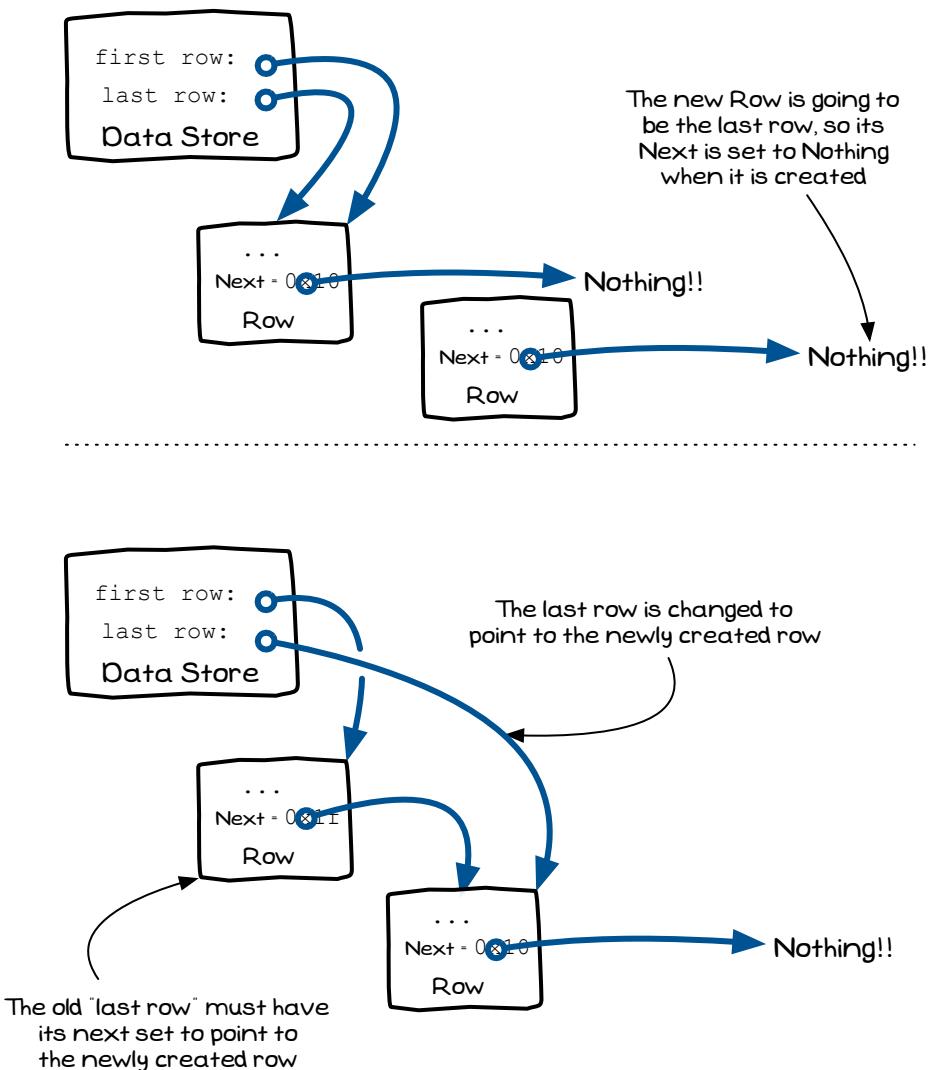


Figure 6.30: When a new row is added, it becomes the new last row, and the old last node points to it

Pseudocode

```
Function: Add Row
-----
Parameters:
1: Db Data (Data Store, passed by reference)
-----
Locals:
- Row Id (Integer)
- New Row (Row Pointer)
-----
Steps:
1: Assign Row Id, the value of Db Data's Next Row Id field
2: Increase the Db Data's Next Row Id field by 1
3: Allocate space on heap for a new row, assign the address to New Row
4: Into the memory pointed to by New Row, store the result of Read Row(row_id)
5: Assign to New Row's next, the value Nothing
6: If Db Data's Last Row is Nothing
    7: Store in Db Data's First Row, the pointer New Row
8: Else
    9: Store in Db Data's Last Row's Next, the pointer New Row
10: Store in Db Data's Last Row, the pointer New Row
```

Listing 6.3: Pseudocode for the Add a Row procedure in the linked version of Small DB 2

**Note**

- Step 3 of Listing 6.3 allocates space on the heap for the row data, and stores a pointer to it in New Row.
- Step 6 checks if this is the first Row in the Data Store.
- Step 9 reads Db Data's Last Row pointer, then follows it to the Row, so that it can store a value in that Row's Next field.



Printing all rows in the linked version of Small DB 2

The code in Add a Row will enable you to create a list of any length. Each row is linked in by changing the old last row to point to the newly created row, and then the last row of the Data Store is updated. This is all very nice, but now that you have the data within the program how do you use it?

As with an array, the think that you need to determine is how to perform some action *for each* node in the list. There is no index that you can use to access the Nodes, so the standard **For Loop** is not going to be of assistance. Instead, what you need to do is to iterate through the list by following the next pointers in each Row.

The standard pseudocode for looping through *each* node in a list is shown in Listing 6.1. The main idea is that you can have a current node that you are processing. Then, to get to the next node you follow current pointer and read the next field. The result is then a pointer to the next node in the list and can be stored as the current node. This process can then be repeated **while** current is not nothing. This process is shown in Figure 6.31.

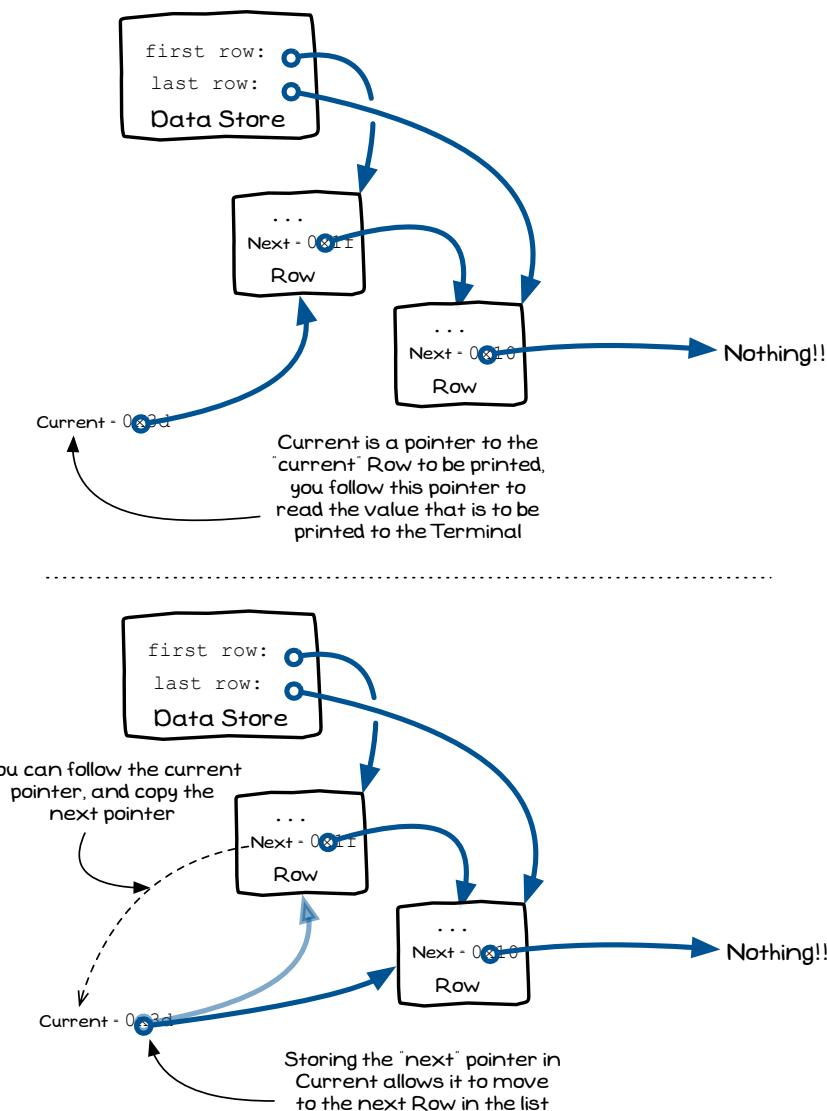


Figure 6.31: When a new row is added, it becomes the new last row, and the old last node points to it

Pseudocode

```

Procedure: Print All Rows
-----
Parameters:
1: List (Linked List)
-----
Locals:
- Current (pointer to a Node)
-----
Steps:
1: Assign Current, the pointer to List's first Node
2: While Current is not Nothing
3:   Call Print Row with the Row pointed to by Current
4:   Assign Current, Current's Next

```

**Listing 6.4:** Pseudocode for the Print all Rows procedure in the linked version of Small DB 2**C++**

```

void print_all_rows(const data_store *db_data)
{
    row *current;

    if (db_data == NULL) return;

    current = db_data->first_row;

    // While there is a current node
    while(current != NULL)
    {
        // Print the row to the Terminal
        print_row(*current);           // Follow the pointer, pass the value
        current = current->next;      // Move to next Row
    }
}

```

**Listing 6.5:** C code for Print All Rows procedure in linked version of Small DB 2**Pascal**

```

procedure PrintAllRows(const dbData: DataStore);
var
    current: ^Row;
begin
    current := dbData^.firstRow;

    // While there is a current node
    while current <> nil do
    begin
        // Print the row to the Terminal
        PrintRow(current^);           // Follow the pointer, pass the value
        current := current^.next;     // Move to next Row
    end;
end;

```

**Listing 6.6:** Pascal code for Print All Rows procedure in linked version of Small DB 2

Deleting a row in the linked version of Small DB 2

Deleting a row in the linked version of Small DB 2 will perform much faster than the equivalent array version, as it does not need to copy the array elements. Instead, the deletion of a Row is just a matter of adjusting the links so that the row is no longer included. Though, while it may be faster it will require more thinking and testing as it requires some careful work with pointers.

Figure 6.32 shows an illustration of the actions that need to be coded into the Delete a Row procedure. This code will need to locate the Row to delete, the previous Row, as well as the next Row. When these are located the delete code can change the previous row's next field to be a pointer to the Row that follows the row that is being deleted. With this done the Row has been removed but is still consuming memory. So the last step of this procedure will need to release that memory as it is no longer needed.

The code in Listing 6.7 shows the C code for this process, while the code in Listing 6.8 shows the matching Pascal code. One key thing to notice is the tracking of the prev Row pointer along with the current. Effectively this is a standard '*for each node*' loop, like that used in [Printing all rows in the linked version of Small DB 2](#). In this case, the prev pointer remembers the last value of current each time through the loop. This ensures that it will have a pointer to the previous Row when the desired Row is found.

Note

This illustration, and the matching code, only works for nodes in the middle of the list. To fully implement the required functionality you will need to add in a few additional branches that test for the following conditions:

- Was a matching Row found? If there is no match then current will point to *Nothing*, and reading current's next will cause the program to crash.
- If the Row is the first row in the list there will be no previous Row, so prev will point to *Nothing*. Setting prev's next will cause the program to crash.
- Similarly, if the Row is the last row in the list then next will be *Nothing*. This will not cause the program to crash, but will give you errors when you try to add a new node.

See if you can work out the required logic to address these issues, and implement the full version of this procedure.

Implementing and Testing the linked version of Small DB 2

This concludes the design for the linked version of Small DB 2. As you have done in the past you will need to work out how to code this using the syntax diagrams and examples from the following two sections: Section [6.3 Dynamic Memory Allocation in C](#) and Section [6.4 Dynamic Memory Allocation in Pascal](#).

During the implementation process you should also be testing your solution. With pointers, as with anything, it is best to write a little and then test it. Use the same testing strategy as discussed in [The Testing Phase: Compiling and running Small DB 2](#). This should help you locate any memory issues with this code.

Note

As this code requires more work with pointers you are likely to have more issues. You should expect your program to crash a few times as you work through these, so do not worry if this does happen. The best way to work through these issues is with a piece of paper and a pencil. Use this to draw up the different Row values and their pointers. Then work through the actions yourself on paper.

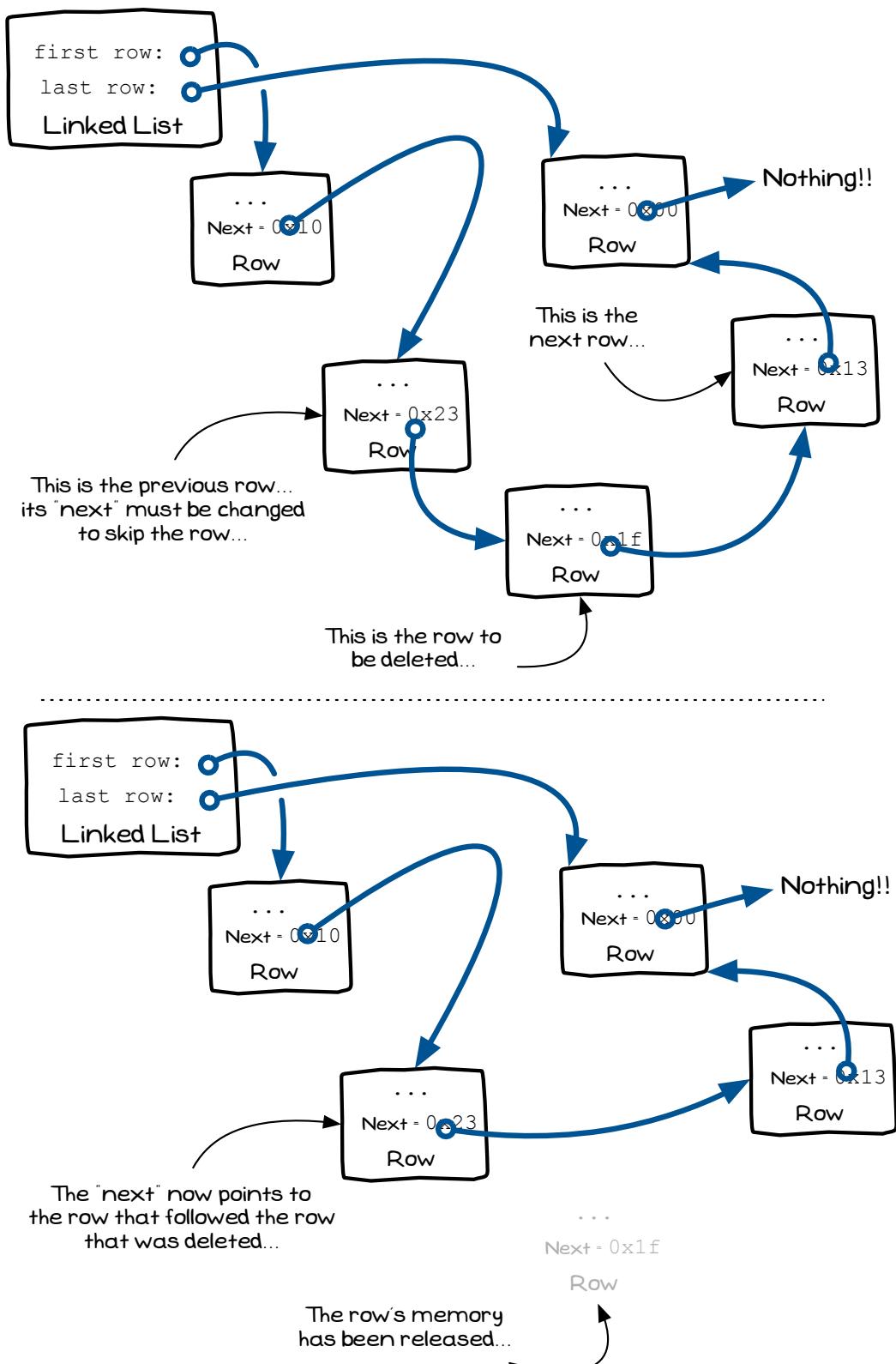


Figure 6.32: When a new row is added, it becomes the new last row, and the old last node points to it

C++

```

void delete_a_row(data_store *db_data)
{
    int row_id;
    row *current, *next, *prev;

    if (db_data == NULL) return;

    printf("Please enter id of row to delete: ");
    scanf("%d", &row_id);

    current = db_data->first_row; // Start searching for the row to delete
    prev = NULL; // There is no previous for the first row
    while(current != NULL && current->id != row_id)
    {
        prev = current;           // Old current is new prev
        current = current->next; // New current is current's next
    }

    next = current->next; // get the new "next" Row
    prev->next = next;   // skip the current Row
    free(current);       // Release the memory
}

```

Listing 6.7: C code for Delete a Row procedure in linked version of Small DB 2

Pascal

```

procedure DeleteARow(var dbData: DataStore);
var
    rowId: Integer;
    current, next, prev: ^Row;
begin
    Write('Please enter id of row to delete: ');
    ReadLn(rowId);

    current := dbData.FirstRow; // Start searching for the row to delete
    prev := nil; // There is no previous for the first row

    while (current <> nil) and (current^.id <> rowId) do
    begin
        prev := current;           // Old current is new prev
        current := current^.next; // New current is current's next
    end;

    next := current^.next; // get the new "next" Row
    prev^.next := next;   // skip the current Row
    Dispose(current);   // Release the memory
end;

```

**Listing 6.8:** Pascal code for Delete a Row procedure in linked version of Small DB 2

6.3 Dynamic Memory Allocation in C

6.3.1 Small DB 2, the dynamic array version in C

Section 6.2, [Using Dynamic Memory Allocation](#), introduced a version of the Small DB program with a dynamic array structure, as opposed to the fixed array structure used to manage the rows in Chapter 5. The C code for the altered functions and procedures is shown in Listing 6.9, the original version can be found in Listing 5.1.

```
/* Program: small-db-2.c - array version */
#include <stdio.h>
#include <strings.h>
#include <stdbool.h>
#include <stdlib.h>
#include <errno.h>
#include <limits.h>

// =====
// = Type declarations =
// =====

typedef union { /* same as original version */ } column_value;
typedef enum { /* same as original version */ } data_kind;
typedef struct { /* same as original version */ } row;

// The data store is a dynamic array of rows, keeping track
// of the number of rows in the array, and the id for the next row
typedef struct {
    int next_row_id;      // The id of the row that will be added next
    int row_count;        // The number of rows in the array
    row *rows;            // A pointer to the rows in memory
} data_store;

// The user can choose to add, delete, or print data or to quit
typedef enum {
    ADD_DATA,
    DELETE_DATA,
    PRINT_DATA,
    QUIT
} menu_option;

// =====
// = General Functions and Procedures =
// =====
void trim(char* text, int n) { /* same as original version */ }
bool is_integer(const char* text) { /* same as original version */ }
bool is_double(const char* text) { /* same as original version */ }
void clear_input() { /* same as original version */ }

// =====
// = Small DB Functions and Procedures =
// =====
row read_row(int next_id) { /* same as original version */ }
void print_row(row to_print) { /* same as original version */ }

// Show a menu to the user and get the option they select
```

```

menu_option get_menu_option()
{
    int input = 0;

    printf("=====\\n");
    printf("| Small DB |\\n");
    printf("=====\\n");
    printf(" 1: Add Data\\n");
    printf(" 2: Print Data\\n");
    printf(" 3: Delete Data\\n");
    printf(" 4: Quit\\n");
    printf("=====\\n");
    printf("Choose Option: ");

    while(scanf("%d", &input) != 1 || input < 1 || input > 4 )
    {
        clear_input();
        printf("Please enter a value between 1 and 4.\\n");
        printf("Choose Option: ");
    }
    // Ensure that input is clear after menu is read.
    clear_input();

    switch(input)
    {
        case 1: return ADD_DATA;
        case 2: return PRINT_DATA;
        case 3: return DELETE_DATA;
        case 4: return QUIT;
        default: return QUIT;
    }
}

// Add a row to the data store
void add_a_row(data_store *db_data)
{
    int row_id = 0;

    if (db_data == NULL) return;

    // Allocate the id
    row_id = db_data->next_row_id;
    db_data->next_row_id++;

    // Store the data - using realloc to allocate space for the rows
    db_data->row_count++;
    db_data->rows = (row *) realloc(db_data->rows, sizeof(row) * db_data->row_count);
    db_data->rows[db_data->row_count - 1] = read_row(row_id); // last idx = n - 1
}

// Get the array index of the row with the indicated ID
int idx_of_row_with_id(const data_store *db_data, int row_id)
{
    int i;

    if (db_data == NULL) return -1;

    // Loop through each element of the array
    for(i = 0; i < db_data->row_count; i++)
    {
        // is this the one we are after?
        if (db_data->rows[i].id == row_id)

```

```

    {
        return i; // return the index found.
    }
}

// Nothing found...
return -1;
}

// Delete a row from the data store
void delete_a_row(data_store *db_data)
{
    int row_id, i, row_index;

    printf("Please enter id of row to delete: ");
    scanf("%d", &row_id);

    // Get the index of the row (will test if db_data == NULL)
    row_index = idx_of_row_with_id(db_data, row_id);

    if (row_index >= 0) // a row was found to delete...
    {
        // copy all data past the row, back over the row to delete it
        for(i = row_index; i < db_data->row_count - 1; i++)
        {
            // copy data back one spot in the array (one element at a time)
            db_data->rows[i] = db_data->rows[i+1];
        }

        // change the row count, and resize the array
        db_data->row_count--;
        db_data->rows = realloc(db_data->rows, sizeof(row) * db_data->row_count);
    }
}

// Print all of the rows from the data store
void print_all_rows(const data_store *db_data)
{
    int i = 0;

    if (db_data == NULL) return;

    // For each row in the array
    for (i = 0; i < db_data->row_count; i++)
    {
        // Print the row to the Terminal
        print_row(db_data->rows[i]);
    }
}

// =====

```

```
// = Main =
// =====

// Entry point
int main()
{
    menu_option opt;
    data_store db_data = {0, 0, NULL};

    do
    {
        opt = get_menu_option();

        switch(opt)
        {
            case ADD_DATA:
                add_a_row(&db_data);
                break;
            case DELETE_DATA:
                delete_a_row(&db_data);
                break;
            case PRINT_DATA:
                print_all_rows(&db_data);
                break;
            case QUIT:
                printf("Bye.\n");
                break;
        }
    } while(opt != QUIT);
}
```

Listing 6.9: C code for the dynamic array version of Small DB, see Listing 5.1 for the original version of this program

Note

- This version of the Small DB program includes the ability to add, delete, and print rows from the data store.
- The data store includes a dynamic array that is managed using `realloc`.
- See Section 6.2 for a discussion of how this works.

6.3.2 Small DB 2, the linked version in C

Section 6.2, Using Dynamic Memory Allocation, introduced a version of the Small DB program with a linked structure, as opposed to the array structure used to manage the rows in Chapter 5. The C code for the altered functions and procedures is shown in Listing 6.10, the original version can be found in Listing 5.1.

```
/* Program: small-db-2.c */

/* Includes are the same as original version */

// =====
// = Type declarations =
// =====

typedef enum { /* same as array version */ } menu_option;
typedef union { /* same as original version */ } column_value;
typedef enum { /* same as original version */ } data_kind;

// The Row record/structure. Each row contains an id
// a kind, and some data (a Column Value).
typedef struct row_struct {
    int           id;
    data_kind     kind;
    column_value  data;
    struct row_struct *next; // Points to the next row
} row;

// The data store is a dynamic linked list of rows, keeping track
// of the number of rows in the list, and the id for the next row
typedef struct {
    int      next_row_id;    // The id of the row that will be added next
    row     *first_row;       // A pointer to the first row
    row     *last_row;        // A pointer to the last row
} data_store;

// =====
// = General Functions and Procedures =
// =====

void trim(char* text, int n) { /* same as original version */ }
bool is_integer(const char* text) { /* same as original version */ }
bool is_double(const char* text) { /* same as original version */ }
void clear_input() { /* same as original version */ }

// =====
// = Small DB Functions and Procedures =
// =====

// Read a row in from the user and return it. The next_id
// is the id number for the newly created row.
row read_row(int next_id)
{
    char line[16] = "", temp[2];
    row result = {0, UNK_VAL, {0}, NULL}; // need to initialise next to point to NULL

    /* The remainder of this function is the same as the original version */
}

void print_row(row to_print) { /* same as original version */ }
menu_option get_menu_option() { /* same as array version */ }
```

```

// Add a row to the data store
void add_a_row(data_store *db_data)
{
    int row_id = 0;
    row *new_row;

    if (db_data == NULL) return;

    // Allocate the id
    row_id = db_data->next_row_id;
    db_data->next_row_id++;

    // Allocate space on the heap for the new row
    new_row = (row *)malloc(sizeof(row));

    *new_row = read_row(row_id);
    new_row->next = NULL; // there is nothing after this row

    if (db_data->last_row == NULL)
    {
        // The data store must be empty, new row is
        // the start and the end.
        db_data->first_row = new_row;
    }
    else
    {
        // The row come after the last row, so change then
        // current last row's next
        db_data->last_row->next = new_row;
    }

    // The new row is the last row in the list
    db_data->last_row = new_row;
}

// Delete a row from the data store
void delete_a_row(data_store *db_data)
{
    int row_id;
    row *current, *next, *prev;

    if (db_data == NULL) return;

    printf("Please enter id of row to delete: ");
    scanf("%d", &row_id);

    current = db_data->first_row;
    prev = NULL; // There is no previous for the first row

    while(current != NULL && current->id != row_id)
    {
        prev = current;
        current = current->next;
    }

    if (current == NULL) return; // No row found

    next = current->next;

    if (prev == NULL)
    {
        // Deleting the first row, so change the start

```

```

        db_data->first_row = next;
    }
    else
    {
        // Skip the row that is about to be deleted
        prev->next = next;
    }

    if ( current == db_data->last_row )
    {
        // Last row was deleted, so update the last row of the data store
        db_data->last_row = prev;
    }

    // Now free the current row
    free(current);
}

// Print all of the rows from the data store
void print_all_rows(const data_store *db_data)
{
    row *current;

    if (db_data == NULL) return;

    current = db_data->first_row;           // current is the first row

    while(current != NULL)                 // While there is a current node
    {
        print_row(*current);              // Print the row to the Terminal
        current = current->next;
    }
}

// =====
// = Main =
// =====

// Entry point
int main()
{
    menu_option opt;
    data_store db_data = {0, NULL, NULL}; // id, first, last

    /* The remainder of this function is the same as the array version */
}

```

Listing 6.10: C code for the linked version of Small DB, see Listing 5.1 for the array version of this program

Note

- `print_all`, `delete_a_row`, and `add_a_row` are the only procedures that have changed significantly.
- Each of these is explained in more detail in Section 6.2.
- Each row has a pointer to the next row in the database, this will point to nothing in the last row.

6.3.3 C Variable Declaration (with pointers)

In C you can declare pointer variables. This includes [Local Variables](#), [Global Variables](#), and [Parameters](#).

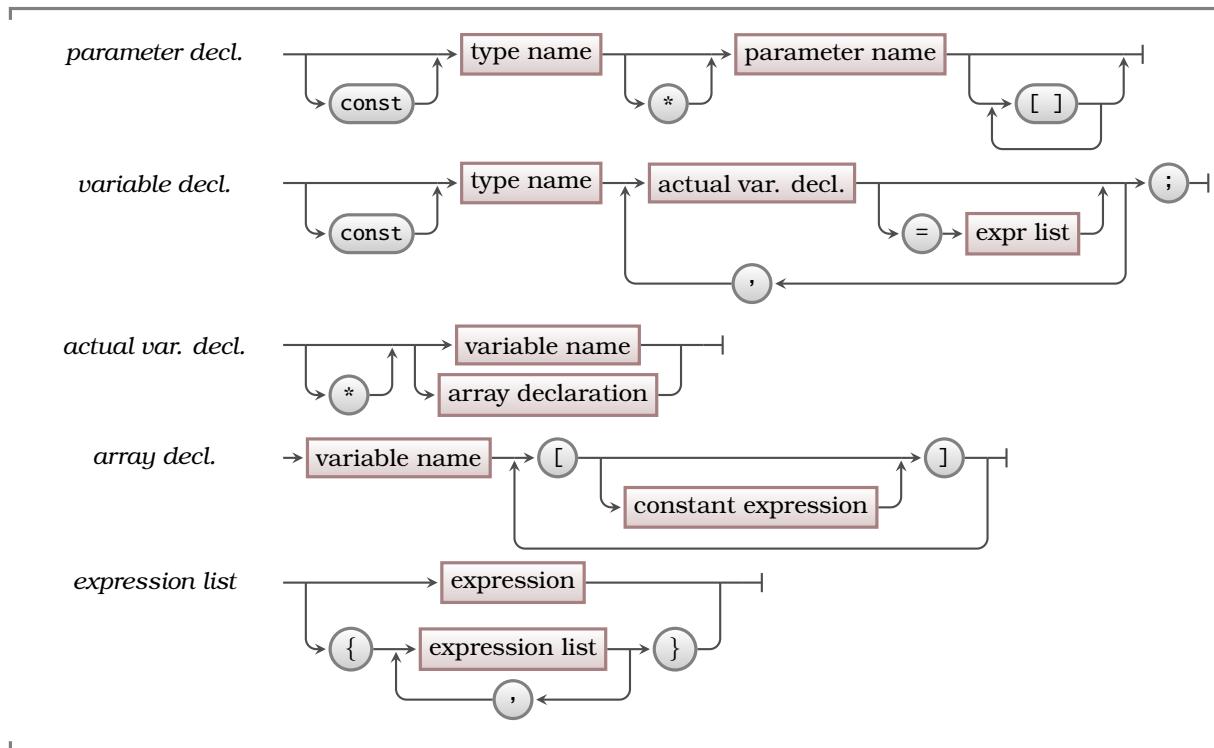


Figure 6.33: C++ Syntax for variable declarations with pointers

Note

- This code allows you to declare your own [Pointer](#) variables in C.
- For variable declarations, the main aspect to pay attention to is the *. This indicates that the variable is a pointer.
- See Listing 6.11 for example variable declarations.



Using pointers to emulate pass by reference in C

C does not have built in support for [Pass by Reference](#). Instead, in C you must pass a [Pointer](#) to the variable you want passed to the function or procedure. Listing 6.11 shows an examples of procedures that accept pointers variables.

Note

- A void pointer can be used to point to *any* value.
- Also see: Section 2.3.8 C Procedure Call (with pass by reference), and Section 2.3.7 C++ Reference Parameters.



C++

```
#include <stdio.h>

void read_data(const char *prompt, const char *format_tag, void *p)
{
    printf("%s", prompt);
    scanf(format_tag, p);
}

void print_intp(int *ptr)
{
    printf("%p -> %d\n", ptr, *ptr);
}

int main()
{
    int i, j;
    float x;
    int *p;
    int *ptrs[] = { &i, &j };

    read_data("Enter a value for i:", "%d", &i);
    read_data("Enter a value for j:", "%d", &j);
    read_data("Enter a value for x:", "%f", &x);

    p = &i;
    print_intp(p);
    print_intp(&i);
    print_intp(&j);
    print_intp(ptrs[1]);

    return 0;
}
```

Listing 6.11: C code with pointer variables

6.3.4 C Pointer Operators

C provides a number of pointer operators that allow you to get and use pointers.

Name	Operator	Example	Description
Address Of	&	&x	Gets a pointer to the variable/field etc.
Dereference	*	*ptr	Follow the pointer, and read the value it points to.
	->	ptr->field_name	Follow a pointer to a struct or union, and read a field value.
	[]	ptr[2]	Allows you to access a pointer as if it were an array.

Table 6.4: C Pointer Operators

You can get a pointer to a value using the ampersand operator (&). This operator lets you get the address of a variable, field, etc.

C++

```
#include <stdio.h>

typedef struct
{
    float x, y;
} point_2d;

int main()
{
    float my_x = 0.0;
    point_2d pt = {1.0, 2.0};
    float *f_ptr;
    point_2d *pt_ptr;

    f_ptr = &my_x;           // get pointer to my_x variable
    printf("%f\n", *f_ptr); // print the value pointer to by fptr

    f_ptr = &pt.x;
    printf("%f\n", *f_ptr); // print the value pointer to by fptr
    printf("%f,%f\n", f_ptr[0], f_ptr[1]); // prints pt.x, pt.y (bad practice)

    pt_ptr = &pt;
    // follow pointer, and get x and y fields from what it points to...
    printf("%f,%f\n", pt_ptr->x, pt_ptr->y);
    printf("%f,%f\n", (*pt_ptr).x, (*pt_ptr).y); // same as above
    return 0;
}
```

Listing 6.12: C code showing pointer operator usage

Note

- The address of operator gets a pointer to the value in the expression that follows it.
- Dereference means '*follow the pointer, and read what it points to*'.
- Use the asterisks (*) to dereference the pointer and get the value it points to.
- You can use -> to dereference a pointer to a structure value, and then to read one of the fields from within the structure.
- Listing 6.12 shows how you can get addresses of different variables, and how you can access the value pointed to using * and ->.

6.3.5 C Type Declarations (with pointers)

In C you can declare custom types that make use of pointers. This includes alias types, structs, and unions.

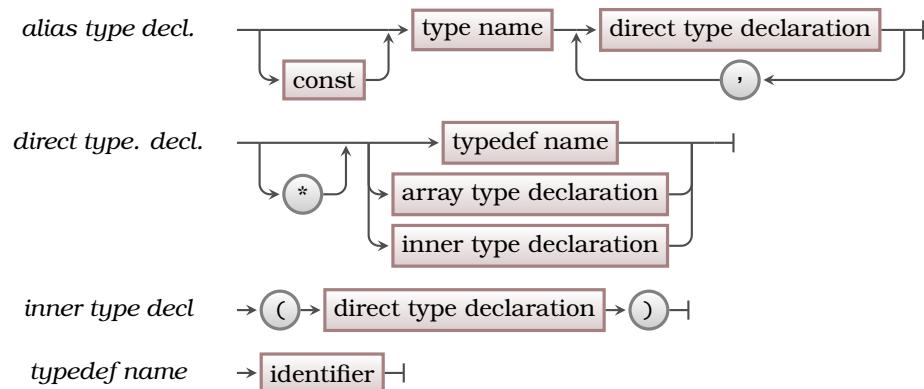


Figure 6.34: C++ Syntax for array and alias type declarations with pointers

- This is the C syntax to for custom alias type that include [Pointers](#).
 - The main difference is the inclusion of the * in the *direct type declaration*. This indicates that the custom type can alias pointer types. This would allow you to declare a type such as person.ptr that is a pointer to a person.
 - The *inner type declaration* allows you to have array of pointers, and pointers to arrays. In these cases you use the brackets to indicate if you want to declare a pointer to an array, or an array of pointers. See Listing 6.13 for an example of these type declarations.
 -



C++

```

/*program: alias-types-with-ptrs.c */

// The int_ptr type is a pointer to an integer
typedef int *int_ptr;

// The five_ints type is an array of five integers
typedef int five_ints[5];

// The ptr_to_array_of_5_ints is a pointer to an array of five ints
typedef int (*ptr_to_array_of_5_ints)[5];

// The five_int_ptrs is an array of five int pointers
typedef int *five_int_ptrs[5];

int main()
{
    int i;
    int x = 10;

    // x_ptr is an pointer to x
    int_ptr x_ptr = &x;

    // an array of five ints
    five_ints data = {0,1,2,3,4};

    // a pointer to the array of 5 ints
    ptr_to_array_of_5_ints data_ptr = &data;

    // an array of five int ptrs
    five_int_ptrs data2 = {&x, &data[0], &data[1], &data[2], &data[3]};

    printf("%d", x);

    for(i = 0; i < 5; i++)
    {
        printf("data[%d] = %d\n", i, data[i]);
        printf("data_ptr[%d] = %p -> %d\n", i, data_ptr[i], (*data_ptr)[i]);
        printf("data2[%d] = %p -> %d\n", i, data_ptr[i], *data2[i]);
        printf("=====\n");
    }

    return 0;
}

```

Listing 6.13: C code demonstrating type aliasing with pointers

C Structure Declarations (with pointer fields)

The fields of a structure may be a pointer.

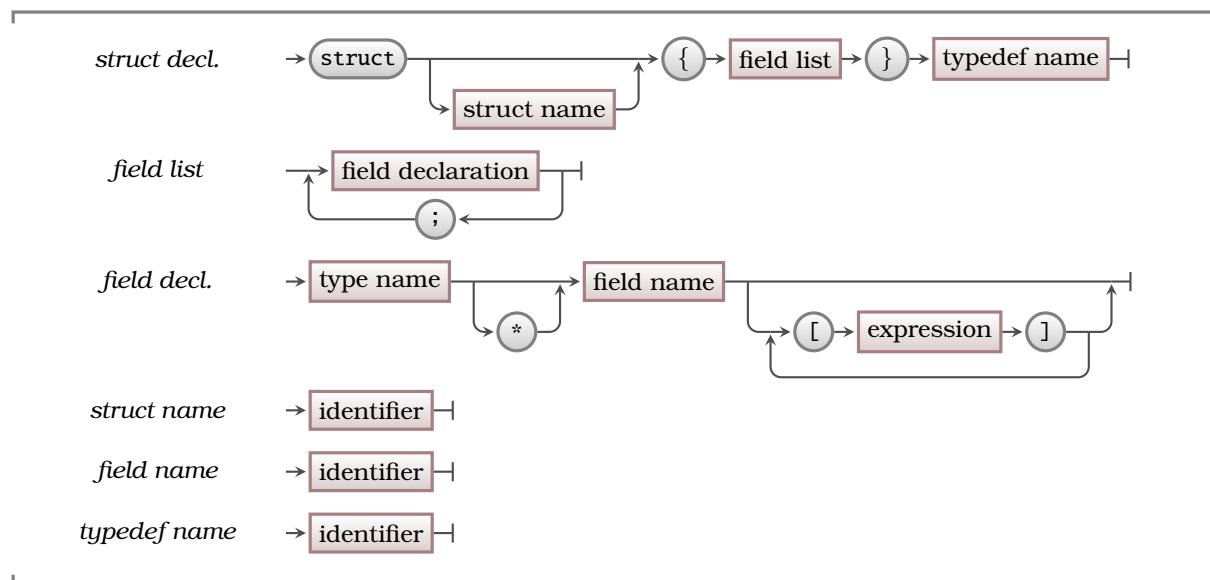


Figure 6.35: C++ Syntax for struct fields with pointers

C++

```

/* struct example */
#include <stdlib.h>

typedef struct node_struct
{
    int value; // node value
    struct node_struct *next; // pointer to next node
} node;

node *create_node(int val, node *next)
{
    node *result;
    result = (node *)malloc(sizeof(node));
    result->value = val;
    result->next = next;
    return result;
}

int main()
{
    node *current;
    current = create_node(0, NULL);
    current = create_node(1, current);
    return 0;
}
  
```

Listing 6.14: C code with a struct that contains a pointer

C Union Declaration (with pointer fields)

The fields of a union may be pointers.

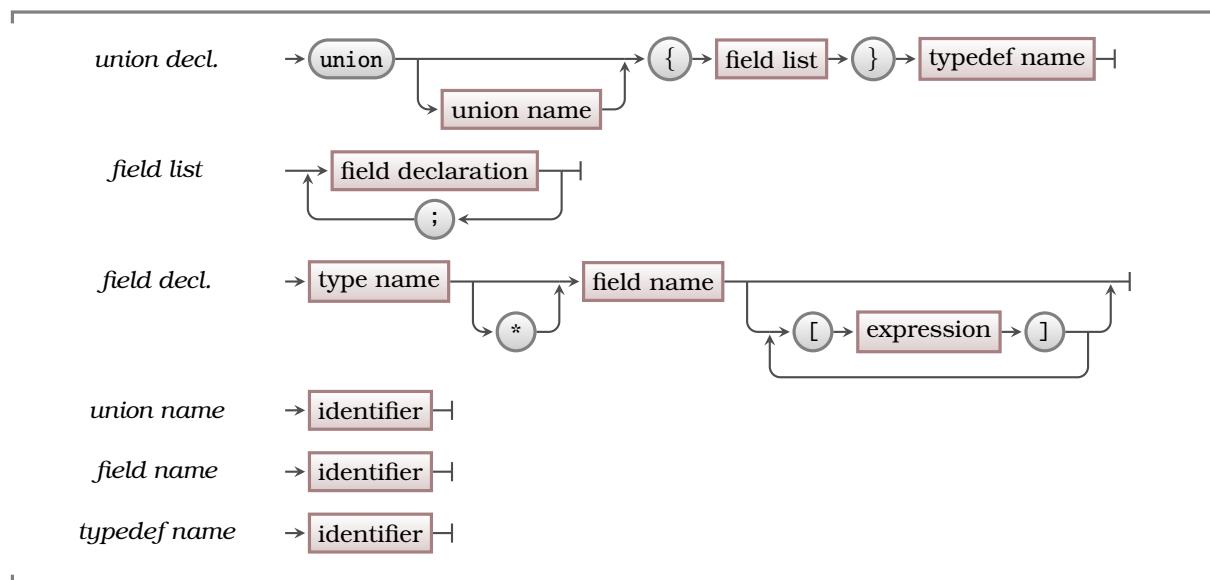


Figure 6.36: C++ Syntax for union fields with pointers

C++

```

/* program: union pointer test */

typedef union
{
    int      *int_ptr;      // stores either an int pointer
    float    *float_ptr;    // or a float pointer
    double   *dbl_ptr;     // or a double pointer
} value_ptr;

int main()
{
    value_ptr ptr, ptr1, ptr2;
    int i = 1.0;
    float f = 1.1;
    double d = 2.2;

    ptr.int_ptr = &i;        // Stores a pointer to an int
    ptr1.float_ptr = &f;     // Stores a pointer to a float
    ptr2.dbl_ptr = &d;      // Stores a pointer to a double

    return 0;
}
  
```

Listing 6.15: C code with a union that contains pointer fields

6.3.6 C Memory Allocation Functions

C includes a number of memory allocation functions: `malloc`, `calloc`, `realloc`, `free`.

`malloc`

`malloc` is the standard memory allocation function. You tell it how much space you want, and it allocates you that many bytes on the heap. This is a function, that returns a pointer to the space allocated.

Function Prototype	
<code>void *malloc(size_t size)</code>	
Returns	
<code>void *</code>	A pointer to the allocated space is returned.
Parameter	Description
<code>size</code>	The number of bytes to allocate on the heap.

Table 6.5: Details of the `malloc` function

C++

```
#include <stdlib.h>

int main()
{
    int *p, *p_arr;

    // get space for one integer from the heap
    p = (int *)malloc(sizeof(int));
    // get space for 5 integers from the heap (like an array)
    p_arr = (int *)malloc(sizeof(int) * 5);

    // free all space allocated
    free(p);
    p = NULL;
    free(p_arr);
    p_arr = NULL;

    return 0;
}
```

Listing 6.16: Example calls to `malloc`

Note

- `malloc` is used for *memory allocation*.
- You need to include **stdlib.h** to use `malloc`.
- `malloc` allows you to allocate space on the heap. It returns a pointer to this space.
- `malloc` returns a void pointer, you need to type cast this to the kind of pointer you want, for example `(int *)` casts it to an integer pointer.
- `malloc` returns `NULL` if it fails to allocate memory.

calloc

The difference between `calloc` and `malloc` is that `calloc` clears the memory allocation. When you call `calloc` you pass it a number and a size, and `calloc` returns you a pointer to a block of memory that is $number \times size$ bytes.

Function Prototype	
void *calloc(size_t num, size_t size)	
Returns	
void *	A pointer to the allocated space is returned.
Parameter	Description
num	The number of elements to allocate to the array.
size	The size of each element to be allocated on the heap.

Table 6.6: Details of the `calloc` function

C++

```
#include <stdlib.h>

int main()
{
    int *p, *p_arr;

    // get space for one integer from the heap - will be zeroed
    p = (int *)calloc(1, sizeof(int));
    // get space for 5 integers from the heap (like an array) - all zeroed
    p_arr = (int *)malloc(5, sizeof(int));

    // free all space allocated
    free(p);
    p = NULL;
    free(p_arr);
    p_arr = NULL;

    return 0;
}
```

Listing 6.17: Example calls to `calloc`

Note

- `calloc` is used for getting a *cleared memory allocation*.
- You need to include **stdlib.h** to use `calloc`.
- `calloc` performs a similar task to `malloc`, with the addition of clearing the space allocated.
- After calling `calloc` the memory you are allocated will have all of its bytes set to 0, whereas with `malloc` the memory retains whatever value was there previously.
- `calloc` returns `NULL` if it fails to allocate memory.

realloc

Like malloc and calloc, realloc allows you to allocate space from the heap. realloc allows you to allocate or change (*reallocate*) space on the heap.

Function Prototype	
void *realloc(void *ptr, size_t size)	
Returns	
Parameter	Description
ptr	The pointer to <i>realloc</i> ate space for on the heap.
size	The size of each element to be allocated on the heap.

Table 6.7: Details of the realloc function

C++

```
#include <stdlib.h>

int main()
{
    int *p = NULL;

    // get space for one integer from the heap
    p = (int *)realloc(p, sizeof(int));
    *p = 1; // give it a value...

    // reallocate space for 5 integers
    p = (int *)realloc(p, sizeof(int) * 5);

    // p[0] is still 1...
    // but there is now also space for p[1]...p[4]

    // free all space allocated
    free(p);
    p = NULL;

    return 0;
}
```

Listing 6.18: Example calls to realloc

Note

- realloc allows you to *reallocate memory* for a pointer.
- You need to include **stdlib.h** to use realloc.
- ptr must be a NULL, or a pointer to a memory block on the heap, i.e. space previously allocated with **malloc**, **calloc**, or **realloc**.
- realloc returns NULL if it fails to allocate memory.
- realloc may need to move the memory allocation, so you need to assign the result to a pointer as it may differ from the value passed to the ptr parameter.

free

When you allocate memory you are responsible for freeing that memory when you no longer require it. The free function allows you to do this.

Procedure Prototype	
void free(void *ptr)	
Parameter	Description
ptr	The pointer to the space to free on the heap.

Table 6.8: Details of the free function

Note

- free allows you to free the memory allocated to a pointer.
- You need to include **stdlib.h** to use free.
- ptr a pointer to a memory block on the heap, i.e. space previously allocated with **malloc**, **calloc**, or **realloc**.
- You can also pass ptr a NULL value, in which case nothing occurs.
- It is good practice to assign a NULL value to the pointer after freeing it.



6.4 Dynamic Memory Allocation in Pascal

6.4.1 Small DB 2, the dynamic array version in Pascal

Section 6.2, [Using Dynamic Memory Allocation](#), introduced a version of the Small DB program with a dynamic array structure, as opposed to the fixed array structure used to manage the rows in Chapter 5. The Pascal code for the altered functions and procedures is shown in Listing 6.19, the original version can be found in Listing 5.7.

```

program SmallDb;
uses SysUtils;

// =====
// = Type declarations =
// =====

type
  DataKind = {same as original version}
  ColumnValue = {same as original version}
  Row = {same as original version}

  // The data store is a dynamic linked list of rows, keeping track
  // of the number of rows in the list, and the id for the next row
  DataStore = record
    nextRowId: Integer;           // The id of the row that will be added next
    rows:      array of Row;      // A dynamic array of rows
  end;

  MenuOption = ( ADD_DATA, DELETE_DATA, PRINT_DATA, QUIT );

// =====
// = Small DB Functions and Procedures =
// =====

function ReadRow(nextId: Integer): Row; {same as original version}
procedure PrintRow(toPrint: row); {same as original version}

function GetMenuOption() : MenuOption;
var
  input: Integer = 0;
begin
  WriteLn('=====');
  WriteLn('| Small DB          |');
  WriteLn('=====');
  WriteLn(' 1: Add Data');
  WriteLn(' 2: Print Data');
  WriteLn(' 3: Delete Data');
  WriteLn(' 4: Quit');
  WriteLn('=====');

  Write('Choose Option: ');
  ReadLn(input);

  while (input < 1) or (input > 4) do
  begin
    WriteLn('Please enter a value between 1 and 4.');
    Write('Choose Option: ');
    ReadLn(input)
  end;

```

```

case input of
  1: result := ADD_DATA;
  2: result := PRINT_DATA;
  3: result := DELETE_DATA;
  4: result := QUIT;
  else result := QUIT;
end;
end;

procedure AddRow(var dbData: DataStore);
var
  rowId: Integer = 0;
begin
  // Allocate the id
  rowId := dbData.nextRowId;
  dbData.nextRowId += 1;

  // Store the data
  SetLength(dbData.rows, Length(dbData.rows) + 1);
  dbData.rows[High(dbData.rows)] := ReadRow(rowId);
end;

function IndexOfRowWithID(const dbData: DataStore; rowId: Integer) : Integer;
var
  i: Integer;
begin
  // Loop through each element of the array
  for i := Low(dbData.rows) to High(dbData.rows) do
  begin
    // is this the one we are after?
    if dbData.rows[i].id = rowId then
    begin
      result := i;    // return the index found.
      exit;          // exit the function
    end;
  end;

  // Nothing found...
  result := -1;
end;

procedure DeleteRow(var dbData: DataStore);
var
  rowId, i, rowIndex: Integer;
begin
  WriteLn('Please enter id of row to delete: ');
  ReadLn(rowId);

  // Get the index of the row
  rowIndex := IndexOfRowWithID(dbData, rowId);

  if rowIndex >= 0 then // a row was found to delete...
  begin
    // copy all data past the row, back over the row to delete it
    for i := rowIndex to High(dbData.rows) - 1 do
    begin
      // copy data back one spot in the array (one element at a time)
      dbData.rows[i] := dbData.rows[i+1];
    end;

    // resize the array
  end;
end;

```

```

        SetLength(dbData.rows, Length(dbData.rows) - 1);
    end;
end;

// Print all of the rows from the data store
procedure PrintAllRows(const dbData: DataStore);
var
    i: Integer = 0;
begin
    // For each row in the array
    for i := Low(dbData.rows) to High(dbData.rows) do
    begin
        // Print the row to the Terminal
        PrintRow(dbData.rows[i]);
    end;
end;

// =====
// = Main =
// =====

procedure Main();
var
    opt: MenuOption;
    dbData: DataStore;
begin
    dbData.nextRowId := 0;
    SetLength(dbData.rows, 0);

    repeat
        opt := GetMenuOption();

        case opt of
            ADD_DATA:      AddRow(dbData);
            DELETE_DATA:   DeleteRow(dbData);
            PRINT_DATA:    PrintAllRows(dbData);
            QUIT:          WriteLn('Bye.');
        end;
    until opt = QUIT;
end;

begin
    Main();
end.

```

Listing 6.19: Pascal code for the dynamic array version of Small DB, see Listing 5.7 for the original version of this program

Note

- This version of the Small DB program includes the ability to add, delete, and print rows from the data store.
- The data store includes a dynamic array.
- See Section 6.2 for a discussion of how this works.



6.4.2 Small DB 2, the linked version in Pascal

Section 6.2, [Using Dynamic Memory Allocation](#), introduced a version of the Small DB program with a linked structure, as opposed to the array structure used to manage the rows in Chapter 5. The Pascal code for the altered functions and procedures is shown in Listing 6.20, the original version can be found in Listing 5.7.

```

program SmallDb;
uses SysUtils;

type
  MenuOption = {same as original version}
  DataKind = {same as original version}
  ColumnValue = {same as original version}

  // A Pointer to a row (Row must be in same type decl part)
  RowPtr = ^Row;

  // The Row record/structure. Each row contains an id
  // and some data (a Column Value).
  Row = record
    id: Integer;
    data: ColumnValue;
    next: RowPtr;           // The next Row in the list
  end;

  // The data store is a dynamic linked list of rows, keeping track
  // of the number of rows in the list, and the id for the next row
  DataStore = record
    nextRowId: Integer;     // The id of the row that will be added next
    firstRow: RowPtr;       // A pointer to the first row
    lastRow: RowPtr;        // A pointer to the first row
  end;

  // Read a row in from the user and return it.
  function ReadRow(nextId: Integer): Row;
  var
    line: String = '';
  begin
    //store the id
    result.id := nextId;    // The nextId is the id number for the newly created row
    result.next := nil;      // Nothing after this row... at this point

    {Remainder is the same as original version...}
  end;

  procedure PrintRow(toPrint: row); {same as original version}
  function GetMenuOption() : MenuOption; {same as dynamic array version}

  procedure AddRow(var dbData: DataStore);
  var
    rowID: Integer = 0;
    newRow: RowPtr;
  begin
    // Allocate the id
    rowID := dbData.nextRowId;
    dbData.nextRowID += 1;

    // Allocate space on the heap for the new row
    New(newRow);
  end;

```

```

newRow^ := ReadRow(rowID);
newRow^.next := nil; // there is nothing after this row

if dbData.lastRow = nil then
begin
    // The data store must be empty, new row is
    // the start and the end.
    dbData.firstRow := newRow;
end
else
begin
    // The row come after the last row, so change then
    // current last row's next
    dbData.lastRow^.next := newRow;
end;

// The new row is the last row in the list
dbData.lastRow := newRow;
end;

procedure DeleteRow(var dbData: DataStore);
var
    rowId: Integer;
    current, next, prev: RowPtr;
begin
    WriteLn('Please enter id of row to delete: ');
    ReadLn(rowId);

    current := dbData.firstRow;
    prev := nil; // There is no previous for the first row

    while (current <> nil) and (current^.id <> rowId) do
    begin
        prev := current;           // previous, is now current
        current := current^.next; // current is... one after current
    end;

    if current = nil then exit; // No row found

    next := current^.next;      // the one after the node to delete

    if prev = nil then
    begin
        // Deleting the first row, so change the start
        dbData.firstRow := next;
    end
    else
    begin
        // Skip the row that is about to be deleted
        prev^.next := next; // the one before points to the one after
    end;

    if current = dbData.lastRow then
    begin
        // Last row was deleted, so update the last row of the data store
        dbData.lastRow := prev;
    end;

    // Now free the current row
    Dispose(current);
end;

```

```
// Print all of the rows from the data store
procedure PrintAllRows(const dbData: DataStore);
var
  current: RowPtr;
begin
  current := dbData.firstRow;           // current is the first row

  while current <> nil do            // While there is a current node
begin
  PrintRow(current^);                // Print the row to the Terminal
  current := current^.next;
end;
end;

// =====
// = Main =
// =====

// Entry point
procedure Main();
var
  opt: MenuOption;
  dbData: DataStore;
begin
  dbData.nextRowId := 0;
  dbData.firstRow := nil;
  dbData.lastRow := nil;

  {Remainder is the same as dynamic array version...}
end;

begin
  Main();
end.
```

Listing 6.20: Pascal code for the linked version of Small DB, see Listing 6.19 for the array version of this program

Note

- PrintAll, DeleteRow, and AddRow are the only procedures that have changed significantly.
- Each of these is explained in more detail in Section 6.2.
- Each row has a pointer to the next row in the database, this will point to nothing in the last row.
- The DataStore has a pointer to the first and last rows in the database.
- Adding and removing rows is done by changing the links between row values on the heap.



6.4.3 Pascal Variable Declaration (with pointers)

In Pascal you can declare pointer variables and types. Pointer variables can be used to declare **Local Variables** and **Global Variables**, but cannot be used as **Parameters**. To pass a pointer to a parameter you need to declare your own pointer type and use that.

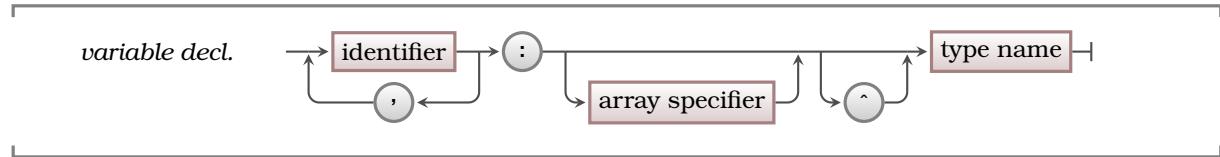


Figure 6.37: Pascal Syntax for variable declarations with pointers

Note

- This code allows you to declare your own **Pointer** variables in C.
- For variable declarations, the main aspect to pay attention to is the `^`: this indicates that the variable is a pointer.
- See Listing 6.21 for example variable declarations.
- The variables in Main could be declared as `^Integer` or as `IntPtr`. It is good practice to declare and use your own pointer types, so `IntPtr` is preferred.

Pascal

```

program PointerVariables;

type
  IntPtr = ^Integer;

procedure PrintIntPtr(ptr: IntPtr);
begin
  WriteLn(HexStr(ptr), '=>', ptr^);
end;

procedure Main();
var
  i, j: Integer;
  p: IntPtr;
  ptrs: array [0..1] of ^Integer;
begin
  ptrs[0] := @i;
  ptrs[1] := @j;

  Write('Enter values for i and j: ');
  ReadLn(i, j);

  p := @i;
  PrintIntPtr(p);
  PrintIntPtr(@i);
  PrintIntPtr(@j);
  PrintIntPtr(ptrs[1]);
end;

begin
  Main();
end.
  
```

Listing 6.21: Pascal code with pointer variables

6.4.4 Pascal Pointer Operators

Pascal provides a number of pointer operators that allow you to get and use pointers.

Name	Operator	Example	Description
Address Of	@	@x	Gets a pointer to the variable/field etc.
Dereference	^	ptr^	Follow the pointer, and read the value it points to.

Table 6.9: C Pointer Operators

You can get a pointer to a value using the *at* operator (@). This operator lets you get the address of a variable, field, etc.

Pascal

```

type
  Point2D = record
    x, y: Single;
  end;

  Point2DPtr = ^Point2D;

procedure Main();
var
  my_x: Single = 0.0;
  pt: Point2D = (x: 1.0; y: 2.0);
  floatPtr: ^Single;
  ptPtr: Point2DPtr;
begin
  floatPtr := @my_x;           // get pointer to my_x variable
  WriteLn(floatPtr^);         // print the value pointer to by floatPtr

  floatPtr := @pt.x;          // print the value pointer to by floatPtr

  ptPtr = @pt;                // follow pointer, and get x and y fields from what it points to...
  WriteLn(ptPtr^.x, ':', ptPtr^.y);
end;

begin
  Main();
end.

```

Listing 6.22: Pascal code showing pointer operator usage

Note

- The address of operator gets a pointer to the value in the expression that follows it.
- Dereference means '*follow the pointer, and read what it points to*'.
- Use the *caret* (^) to dereference the pointer and get the value it points to.
- Listing 6.22 shows how you can get addresses of different variables, and how you can access the value pointed to using ^.

6.4.5 Pascal Type Declarations (with pointers)

In Pascal you can declare custom types that make use of pointers.

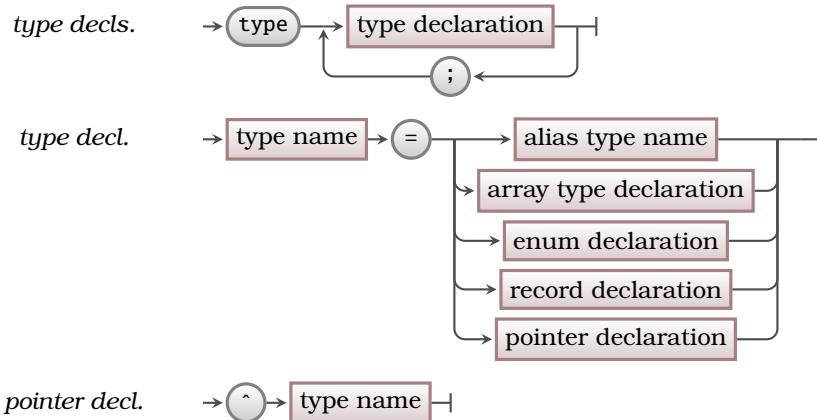


Figure 6.38: Pascal Syntax for array and alias type declarations with pointers

Pascal

```

type
  // The IntPtr type is a pointer to an integer
  IntPtr = ^Integer;

  // The FiveInts type is an array of five integers
  FiveInts = array [0..4] of Integer;

procedure Main();
var
  i: Integer;
  x: Integer = 10;
  // xPtr is a pointer to x
  xPtr: IntPtr;
  // an array of five ints
  data: FiveInts = (0,1,2,3,4);
begin
  xPtr := @x;
  WriteLn(x, ' = ', xPtr^);

  for i := 0 to 4 do
  begin
    printf('data[%d, %d] = %d', i, data[i]);
  end;
end;

begin
  Main();
end;
  
```

Listing 6.23: Pascal code demonstrating type aliasing with pointers

Pascal Structure Declarations (with pointer fields)

The fields of a structure may be pointers.

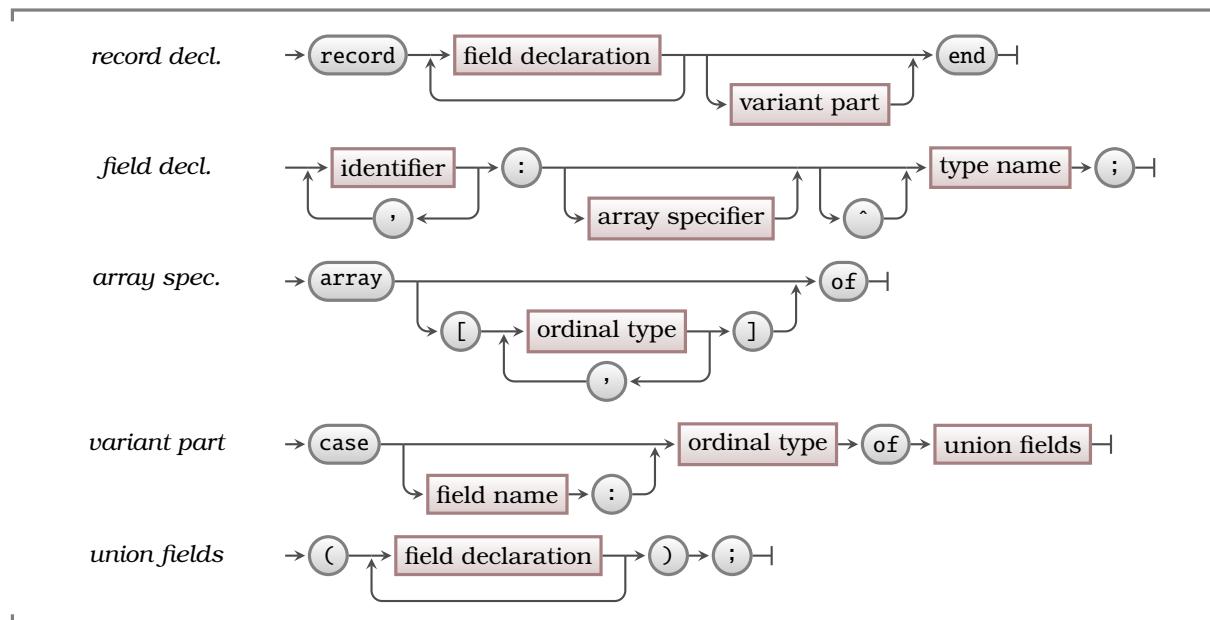


Figure 6.39: Pascal Syntax for struct fields with pointers

Pascal

```

type
  NodePtr = ^Node;
  Node = record
    value: Integer; // node value
    next: NodePtr; // pointer to next node
  end;

function CreateNode(val: Integer; next: NodePtr): NodePtr;
begin
  New(result);
  result^.value := val;
  result^.next := next;
end;

procedure Main();
var
  current: NodePtr;
begin
  current := CreateNode(0, nil);
  current := CreateNode(1, current);
end;

begin
  Main();
end.
  
```

Listing 6.24: Pascal code with a struct that contains a pointer

6.4.6 Pascal Memory Allocation Functions

Pascal includes a number of memory allocation functions: [New](#), [Dispose](#), and [Set Length](#).

New

In Pascal the New procedure allocates space for a pointer. The amount of memory allocated is based on the size of the type referred to by the pointer, for example an Integer pointer is allocated enough space to store one integer value.

Function Prototype	
procedure New(var ptr: Pointer);	
Parameter	Description
ptr	The pointer to allocate the space for. After the call this will point to the allocated memory.

Table 6.10: Details of the New procedure

Pascal

```
program NewExample;

procedure Main();
var
  ptr: ^Integer;
begin
  ptr := nil;

  // Allocate space on the heap...
  New(ptr);

  // Store a value on the heap (at locate allocated to ptr)
  ptr^ := 10;

  Writeln('Ptr is ', HexStr(ptr));
  Writeln('The value it points to is ', ptr^);

  Dispose(ptr);
  ptr := nil;
end;

begin
  Main();
end.
```

Listing 6.25: Example calls to New

Note

- New is used for *memory allocation*.
- New allows you to allocate space on the heap.



Dispose

When you allocate memory you are responsible for freeing that memory when you no longer require it. The Dispose procedure allows you to do this.

Procedure Prototype	
procedure Dispose(ptr : Pointer);	
Parameter	Description
ptr	The pointer to the space to free on the heap.

Table 6.11: Details of the Dispose procedure

Note

- Dispose allows you to free the memory allocated to a pointer.
- See Listing 6.26 for example code.
- ptr is a pointer to a memory block on the heap, i.e. space previously allocated with [New](#).
- You can also pass ptr a nil value, in which case nothing occurs.
- It is good practice to assign a nil value to the pointer after freeing it.



Set Length

Pascal includes support for dynamic arrays. These are arrays where the contents is stored on the heap, and can be dynamically resized during execution using the SetLength procedure.

Procedure Prototype	
procedure SetLength(arr : DynamicArray; len: Integer);	
Parameter	Description
arr	The pointer to the space to free on the heap.
len	The new length for the array arr, preserving any existing data up to the new length.

Table 6.12: Details of the SetLength procedure

Pascal

```
program DynamicArrayExample;

procedure Main();
var
  line: String;
  names: array of String;
  i: Integer;
begin
  SetLength(names, 0);

  WriteLn('Enter a list of names, ending with an empty value.');
  ReadLn(line);

  while Length(line) > 0 do
  begin
    SetLength(names, Length(names) + 1);      // Increase array size
    names[High(names)] := line;                // Set last name
    ReadLn(line);
  end;

  for i := Low(names) to High(names) do
  begin
    WriteLn(names[i]);
  end;
end;

begin
  Main();
end.
```

Listing 6.26: Example calls to SetLength

Note

- SetLength allows you to set the length of a dynamic array.
- High, Low, and Length determine the valid indexes and length of the array.
- Data for a dynamic array is allocated on the heap.
- Dynamic arrays are declared without specifying the indexes (just use array of type).

6.5 Exercises for Dynamic Memory Allocation

6.5.1 Concept Questions

Read over the concepts in this chapter and answer the following questions:

1. What is the difference between the heap and the stack?
2. Why would you want to allocate space on the heap?
3. How can you allocate space on the heap?
4. Why do you need to free the space you are allocated? Why do you not need to do this with values stored on the stack?
5. What is a pointer?
6. What can a pointer point to?
7. Why do you need pointers to make use of the heap?
8. Where can pointers be stored?
9. How can you get a pointer to an existing value?
10. What can you do with the pointer?
11. The pointer has a value, and points to a value. What is the value of the pointer? How is this different to the value it points to?
12. What are the different ways you can allocate memory? Describe each, and explain what they can be used for.
13. What additional issues are you likely to encounter when working with pointers? Explain each, and how you plan to handle these issues.

6.5.2 Code Reading Questions

Use what you have learnt to read and understand the following code samples, and answer the associated questions.

1. Read the code for the dynamic array version of Small DB 2 (for your language of choice) and do the following:
 - (a) Draw a picture of an empty data store that shows what it looks like in memory.
 - (b) Draw a new picture showing how the data store will appear after one row is added.
 - (c) Draw a new picture showing how the data store will appear after three rows have been added.
 - (d) Explain how the Add Row code is able to add rows to the data store.
 - (e) Explain how the Delete Row code is able to delete a row from the data store. Include a drawing that illustrates the process.
 - (f) Explain the steps you would need to perform to add an *Insert Row* option for the user.
2. Read the code for the linked version of Small DB 2 (for your language of choice) and do the following:
 - (a) Draw a picture of an empty data store that shows what it looks like in memory.
 - (b) Draw a new picture showing how the data store will appear after one row is added.
 - (c) Draw a new picture showing how the data store will appear after three rows have been added.
 - (d) Explain how the Add Row code is able to add rows to the data store.
 - (e) Explain how the Delete Row code is able to delete a row from the data store. Include a drawing that illustrates the process.
 - (f) Explain the steps you would need to perform to add an *Insert Row* option for the user.

6.5.3 Code Writing Questions: Applying what you have learnt

Apply what you have learnt to the following tasks.

1. Alter your address book program from Chapter 5, so that you can enter any number of contacts, and output their details to the Terminal. Use a small menu to allow the user to choose between adding a new contact, printing contacts, and quitting the program.
2. Revisit your statistics program from Chapter 4.
 - (a) Alter its implementation so that the user can enter a variable number of values. The program can start by asking how many values the user will enter, and sizing the array appropriately.
 - (b) Add a loop and menu to the program so that the user can add more values, display statistics, or quit.
3. Revisit your small-db program from Chapter 5.
 - (a) Alter its implementation to introduce a Data Store type that contains a dynamic number of row values.
 - (b) Introduce a menu with options to allow the user to add a row, delete a row, print all rows, and quit the program.

6.5.4 Extension Questions

If you want to further your knowledge in this area you can try to answer the following questions. The answers to these questions will require you to think harder, and possibly look at other sources of information.

1. Try implementing an alternate approach to deleting a row from the array version of the small db program.
2. Alter the Row type in your small db program to have a variable number of column values (rename).
3. Compare the dynamic array and linked versions of the Small DB 2 program. Discuss the relative advantages and disadvantages of each approach.
4. Test the speed difference between the dynamic array and linked versions of the Small DB 2 program for the following operations:
 - (a) Adding rows (test with adding 10, 100, 1000, and 10000 rows)
 - (b) Inserting rows (test with inserting 10, 100, 1000, and 10000 rows)
 - (c) Deleting rows (deleting 10, 100, 1000, and 10000 rows)

7

Input and Output

ou are progressing well. You have already mastered most of the basics of spell and potion craft, so now we can turn our attention to the creation of scrolls. These magical devices will allow you to capture the magical energies created in your spells, and store them to be retrieved at a later time. Gather your parchment and wand, now summon the energies for your spell and ...

Over the previous chapter you have learnt to create programs that manipulate data. So far this data has only existed within the program, with the values stored being lost when the program terminates. If you want to be able to maintain these values between executions you need to learn to save data to file. By saving the program's data to file you can then load it back in when the program is restarted.

This chapter will introduce the artefacts needed to save data from your program to file, and to load that data from file. Using this you will be able to persist data, making it available to future executions of the program.

When you have understood the material in this chapter you will be able to save and load data from text and binary files.

Contents

7.1 Input and Output Concepts	583
7.1.1 Persisting Data	583
7.1.2 Interacting with Files	584
7.1.3 File Formats	585
7.1.4 File Structures	586
7.1.5 Other Devices	587
7.2 Using Input and Output	588
7.2.1 Saving Data from Small DB	588
7.2.2 Loading Data for Small DB	590
7.2.3 New Structure for Small DB	591
7.2.4 Writing Code to Load and Save Data for Small DB	591
7.3 Input and Output in C	593
7.3.1 Implementing Small DB File IO in C	593
7.3.2 C File Type	596
7.3.3 C File Functions	597
7.4 Input and Output in Pascal	603
7.4.1 Implementing Small DB File IO in C	603
7.4.2 Pascal Text Type	605
7.4.3 Pascal File Procedures	606
7.5 Input and Output Exercises	610
7.5.1 Concept Questions	610
7.5.2 Code Writing Questions: Applying what you have learnt	610
7.5.3 Extension Questions	610

7.1 Input and Output Concepts

7.1.1 Persisting Data

When a program is running it uses variables and dynamically allocated memory to store the data it requires. These values exist within the memory allocated to the program when it was started. When the program ends its allocated memory is released, and the values stored within the program are lost. If values must be *remembered* between executions then this data must be stored outside of the program. To achieve this you can save data from the program into files that are stored on the computer's hard drive or solid state drive (SSD).

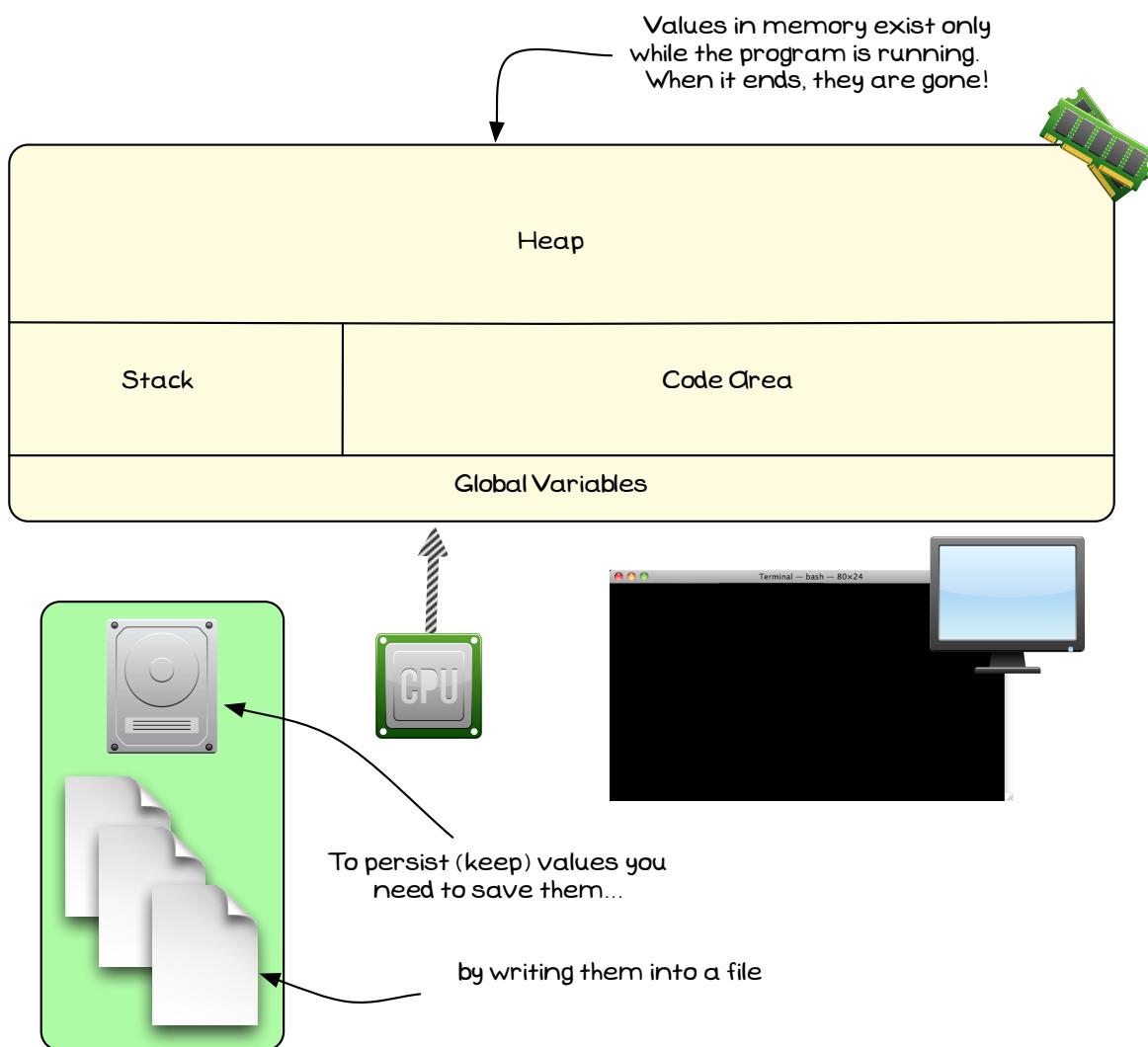


Figure 7.1: When a program ends its data is gone, unless you save it to file

Note

- Data stored within a program is lost when the program ends.
- To *remember* data between executions it must be saved to file.
- The files are stored in a persistent way on a hard drive, or solid state drive.
- Hard drives, and solid state drives, are as data storage devices to persist data needed by programs on the computer.

7.1.2 Interacting with Files

Programming languages offer a number of functions and procedures that are used to interact with files. These will allow you to save data to a file, and load data back from the file.

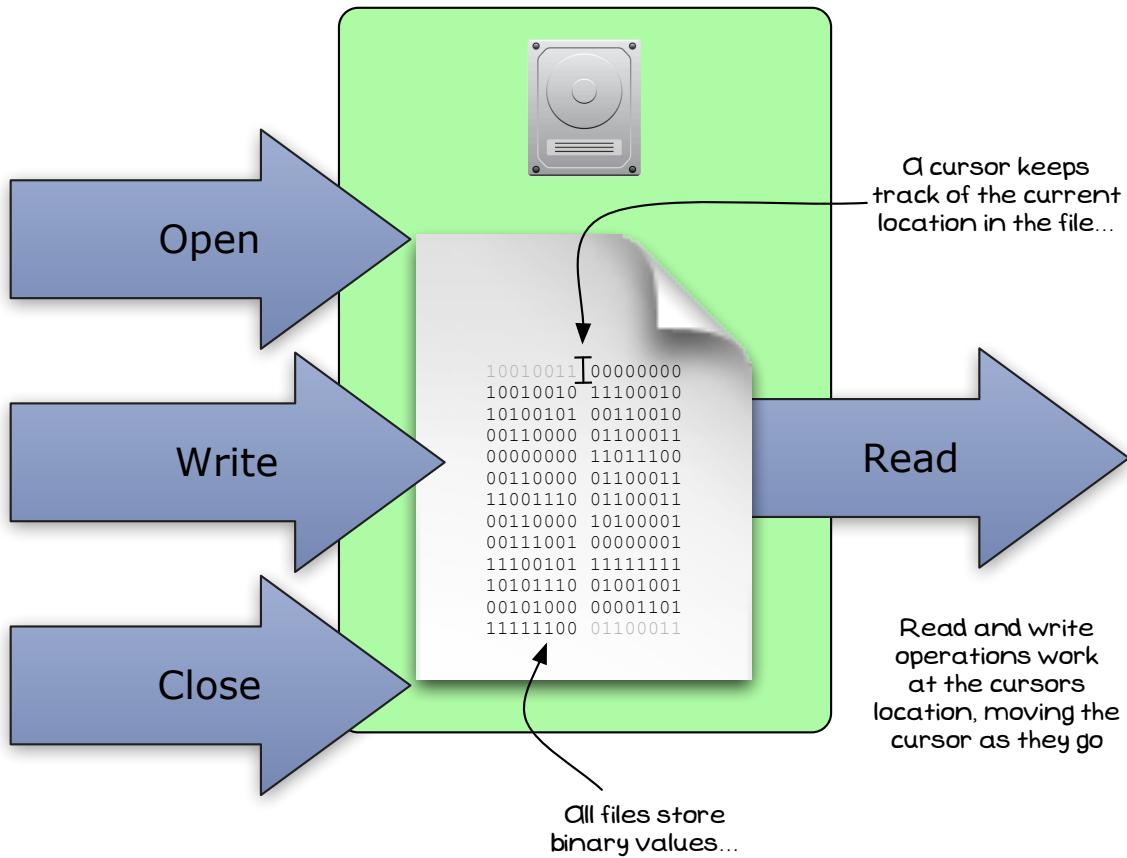


Figure 7.2: File operations include the ability to open, read, write, and close files

Note

- Programming languages will provide functions and procedures to:
 - **Open** a file so you can interact with its contents.
 - **Read** data from a file that has been opened to be read.
 - **Write** data to a file that has been opened to be written to.
 - **Close** a file that is currently open.
- Languages will also provide built in file types that are used with these functions and procedures.
- When you open a file you indicate if you want read and/or write access to its contents.
- Opening a file gives you access to a cursor that you can use to read data from the file, or write data to the file.
- Standard usage will be to open the file, read/write data to the file, and then close the file.

7.1.3 File Formats

There are two main file formats that you can work with in your program: binary files and text files. A text file stores its data as textual characters, whereas a binary file stores the values directly in the file.

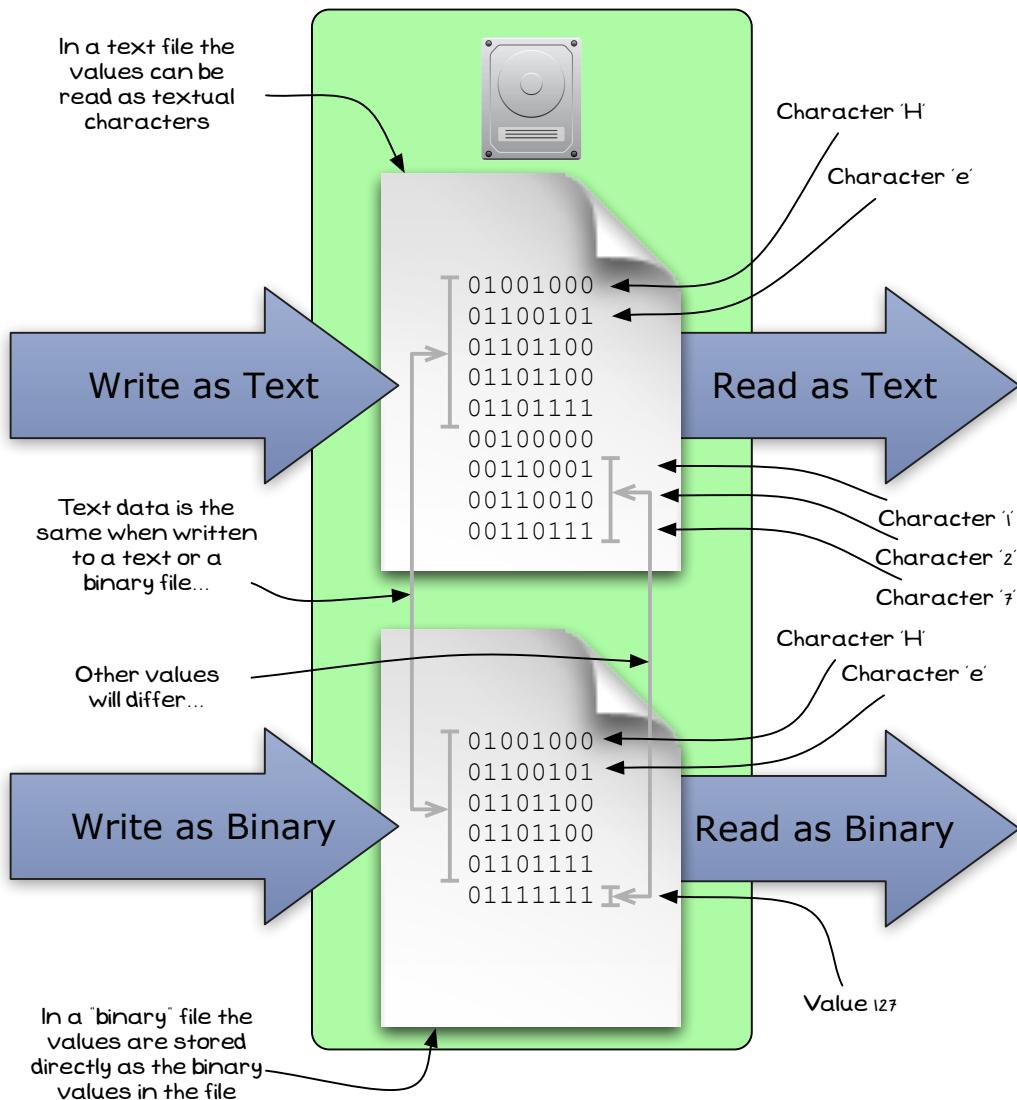


Figure 7.3: Files can store *textual* or *binary* data

Note

- Figure 7.3 shows a binary and text file used to save the text 'Hello' and the integer 127.
- In the *text* file they are stored as the characters 'H', 'e', 'l', 'l', 'o', and ' ', '1', '2', and '7' separated by a space.
- In the *binary* file the characters are stored in the same way (as they are text), but the integer is stored as the value 127 which if interpreted as text is a *delete* character.
- If you opened these files in a text editor you could read the values from the text file, but you would see a strange character instead of the number 127 in the binary file.

7.1.4 File Structures

When thinking about using files, the one aspect that you need to spend the most time on will be the organisation of the data in the file. You need to ensure that the data you save includes sufficient information that it can be read back into the program at a later stage. Some common strategies are to:

1. Write fixed size data blocks where possible.
2. Store meta data¹, such as the number or size of variable data blocks.
3. Alternatively, mark the end of variable sized or numbered data blocks with a *sentinel* values.

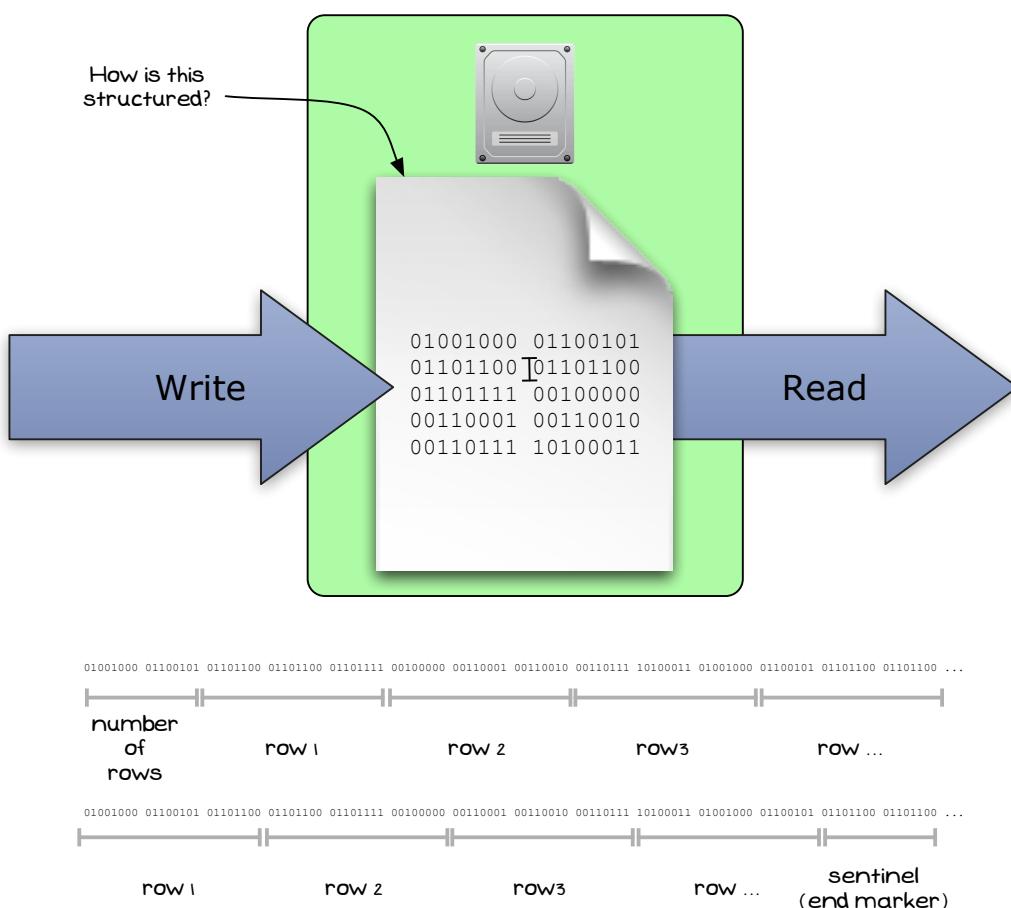


Figure 7.4: You need to structure data within the file to make it possible to read back successfully.

Note

- Figure 7.4 shows different ways of structuring a variable number of 'row' values within a file.
- The row values are fixed size blocks, or use their own strategy to manage variable length data.
- One version stores the number of elements in the data before storing the data itself.
- An alternate strategy is to store a sentinel value at the end of the variable length data.^a

^aThere is an end of file marker that can also be used for this purpose.

¹Meta data is data about your data.

7.1.5 Other Devices

The input/output operations are fairly standardised across different device types. Saving data to a file is very similar to writing it to the Terminal or to a network. The skills you learn with any one of these will be transferable to other devices.

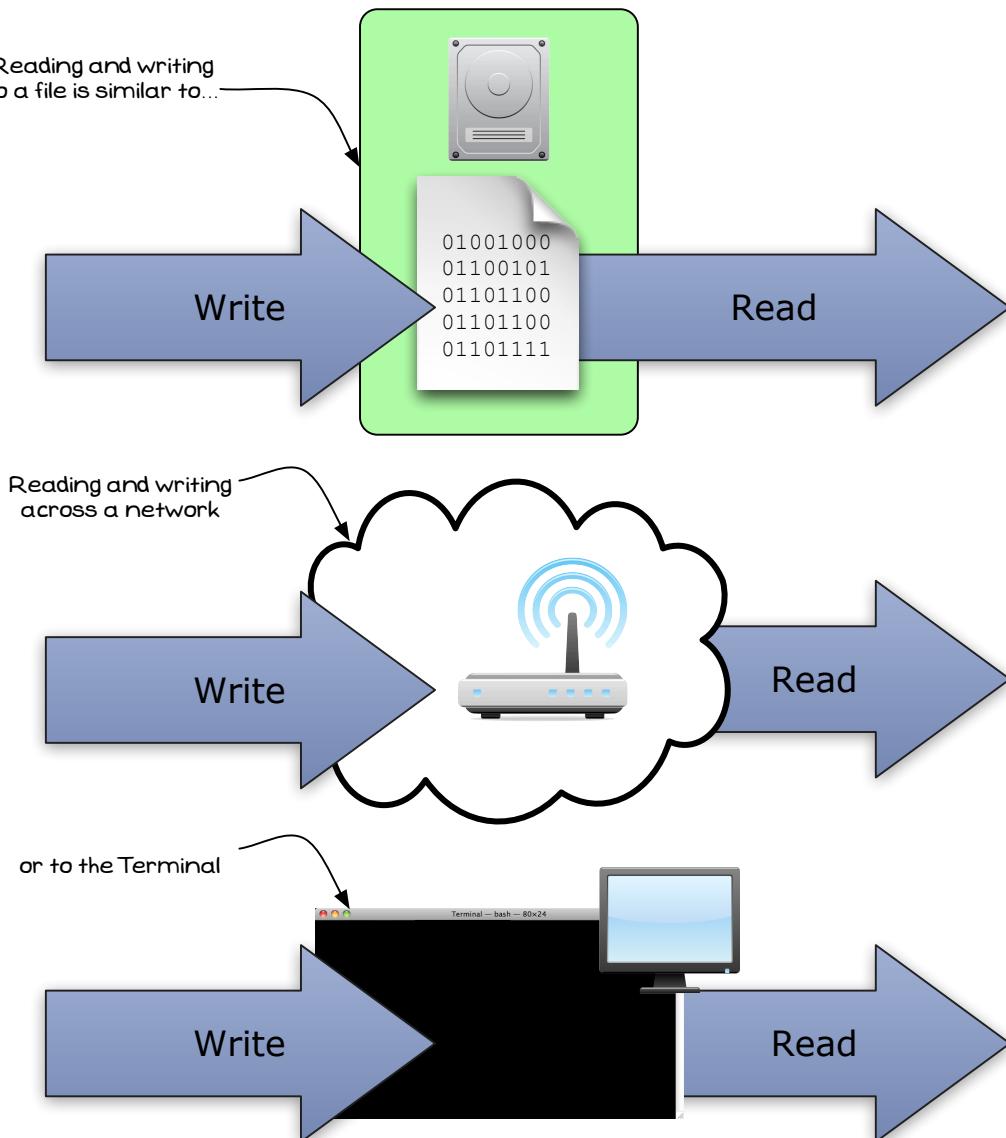


Figure 7.5: Writing to other devices also follows similar patterns

Note

- The tasks you need to do to read and write data are similar, regardless of the destination device.
- Reading and writing to file is similar to reading and writing from the Terminal.
- You can open connections to other machines, and read and write data across these connections. Look up details on sockets if you are interested in doing this.



7.2 Using Input and Output

Using File Input and Output it is now possible to load and save data. As an example we will examine how you can go about saving and loading data in the small db program.

7.2.1 Saving Data from Small DB

The Small DB² program allows the user to enter a number of *row* values, with each row having a single column that stores a data value (either an integer, text, or double value). At this stage the program only keeps its data while it is executing, once it ends the data is gone. The first step is therefore to save the data from the program into a file.

Row File Format

When thinking about saving data the first task is to try to determine how the data can be saved so that it can later be read back into the program. The following information can help us design the structure of the file saved from the program:

1. There are a variable number of rows.
2. Each row has a fixed size, when you know the kind of data it is storing.

In the array based version of the Small DB program the number of rows is stored in the data store. Saving this data to file can be achieved by saving the number of rows before storing the data from each row. This will mean that when the file is loaded the program can read the number of rows, and use this information to create enough space for these in memory before reading them from the file. Figure 7.6 shows an example of the file structure saved from the program.

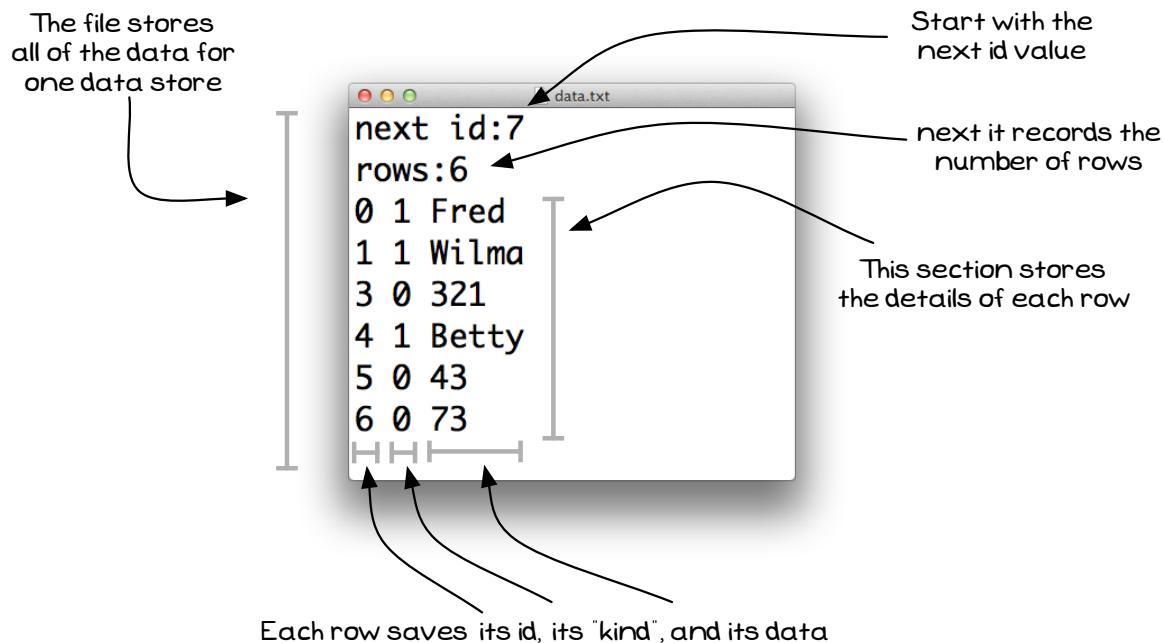


Figure 7.6: The structure of the Small DB data file

²In this chapter we will be saving data from the array based version of the Small DB program, though a similar approach would be taken to save the linked version.

Saving the Data Store

The pseudocode in Listing 7.1 shows the steps that can be followed to save the data from a data store into a file. Notice that the number of rows is being saved into the file, before the row data. This will make it easier to load the file back into memory.

Pseudocode

```

Procedure: Save
-----
Parameters:
1: Db Data (Data Store, passed by reference)
2: Filename (String)
-----
Locals:
- i    (Integer)
- out  (Text file)
-----
Steps:
1: Assign to out, the result of opening the file with Filename (for write)
2: If this failed, exit from this procedure and return to the caller
3: Write Db Data's Next Row Id to the out file
4: Write Db Data's Row Count to the out file
5: For each of the rows in Db Data
6:   Call Write Row to File, passing in Db Data's current row and the out file
7: Close the out file

```

Listing 7.1: Pseudocode for Save (for Small DB array version)

The Save procedure calls a Write Row to File procedure to store each row in the file. The pseudocode for this procedure is shown in Listing 7.2. This code saves the id and kind values to the file, and then uses a Case Statement to ensure it saved the correct value from the row's data.

Pseudocode

```

Procedure: Write Row to File
-----
Parameters:
1: To Save    (Row)
2: Out        (Text File)
-----
Steps:
1: Write the row's id and kind to the Out file
2: Based on the row's kind:
3:   if it is a INT VAL:
4:     Write the row's data's Int Val to the Out file
5:   if it is a DBL VAL:
6:     Write the row's data's Dbl Val to the Out file
7:   if it is a TXT VAL:
8:     Write the row's data's Txt Val to the Out file
9: Write a new line to the Out file

```

Listing 7.2: Pseudocode for the Write Row to File procedure (for Small DB array version)

7.2.2 Loading Data for Small DB

Once you can save data the logical next step will be to load that data back into the program. The file structure is set, so all that needs to be done is to determine the steps that need to be taken in order to load that data back into memory.

The pseudocode for loading the data store file and reading an individual row from file are shown in Listing 7.3 and Listing 7.4. Notice how these mirror the structure of the save procedures. The Load procedure open the file, and then reads the Next Row Id value and Row Count. The Row Count data is used to allocate space in memory for the data store's rows, and to determine how many row values to read from the file.

Pseudocode

```

Procedure: Load
-----
Parameters:
1: Db Data    (Data Store, passed by reference)
2: Filename   (String)
-----
Locals:
- i          (Integer)
- input      (Text file)
-----
Steps:
1: Assign to input, the result of opening the file with Filename (for read)
2: If this failed, exit from this procedure and return to the caller
3: Read Db Data's Next Row Id from the input file
4: Read Db Data's Row Count from the input file
5: Allocate space for Db Data's Rows, based on Db Data's Row Count
6: For each of the rows in Db Data
   Call Load Row, passing in the current row from Db Data and input file
7: Close the input file

```

Listing 7.3: Pseudocode for Load (for Small DB array version)

Pseudocode

```

Procedure: Read Row From File
-----
Parameters:
1: To Load    (Row)
2: Input       (Text File)
-----
Steps:
1: Read the row's id and kind from the Input file
2: Based on the row's kind:
   3: if it is a INT VAL:
      4: Read the row's data's Int Val from the Input file
   5: if it is a DBL VAL:
      6: Read the row's data's Dbl Val from the Input file
   7: if it is a TXT VAL:
      8: Read the row's data's Txt Val from the Input file

```

Listing 7.4: Pseudocode for the Read Row from File procedure (for Small DB array version)

7.2.3 New Structure for Small DB

Figure 7.7 shows the new structure chart for this version of the Small DB program. This shows the new Load and Save procedures. When the program starts Main will load the data from file, and then loop through performing the add, delete, and print actions as the user desires. When the user chooses to quit Main will save its Data Store back into the file. In this way the changes the user makes in the program will be persisted across executions.

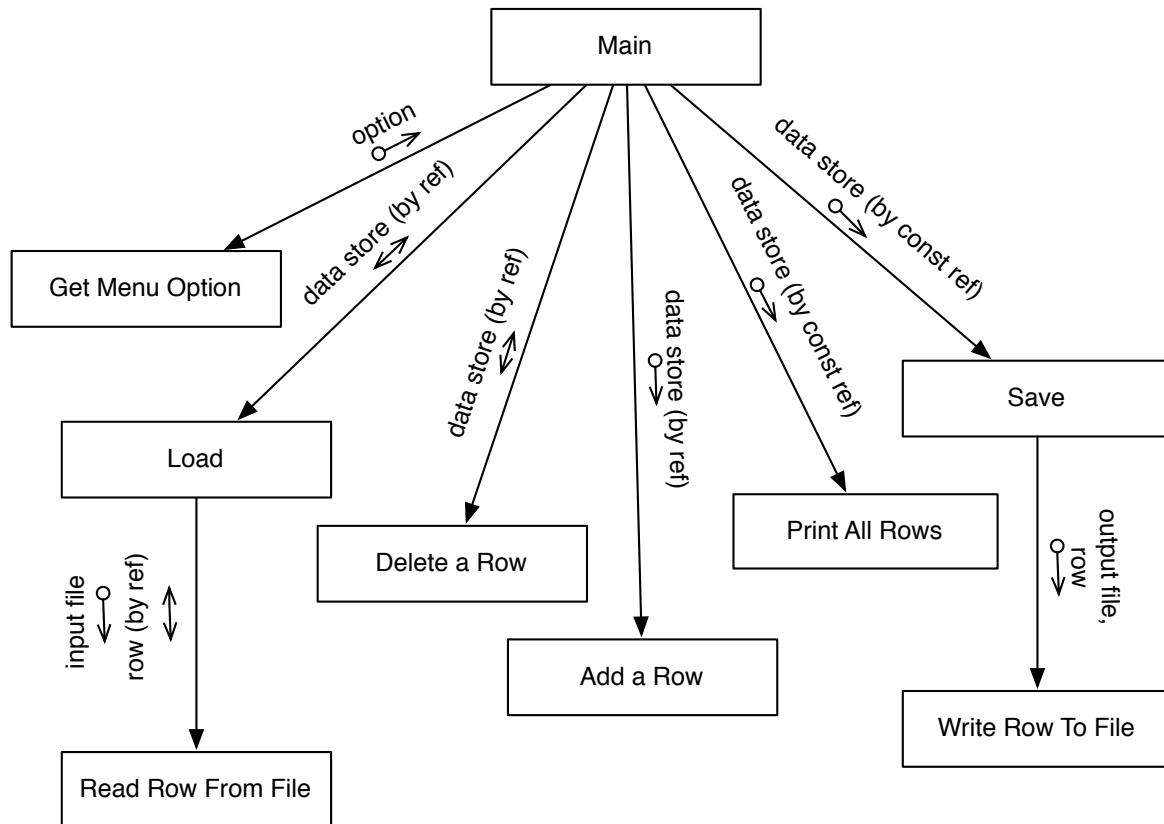


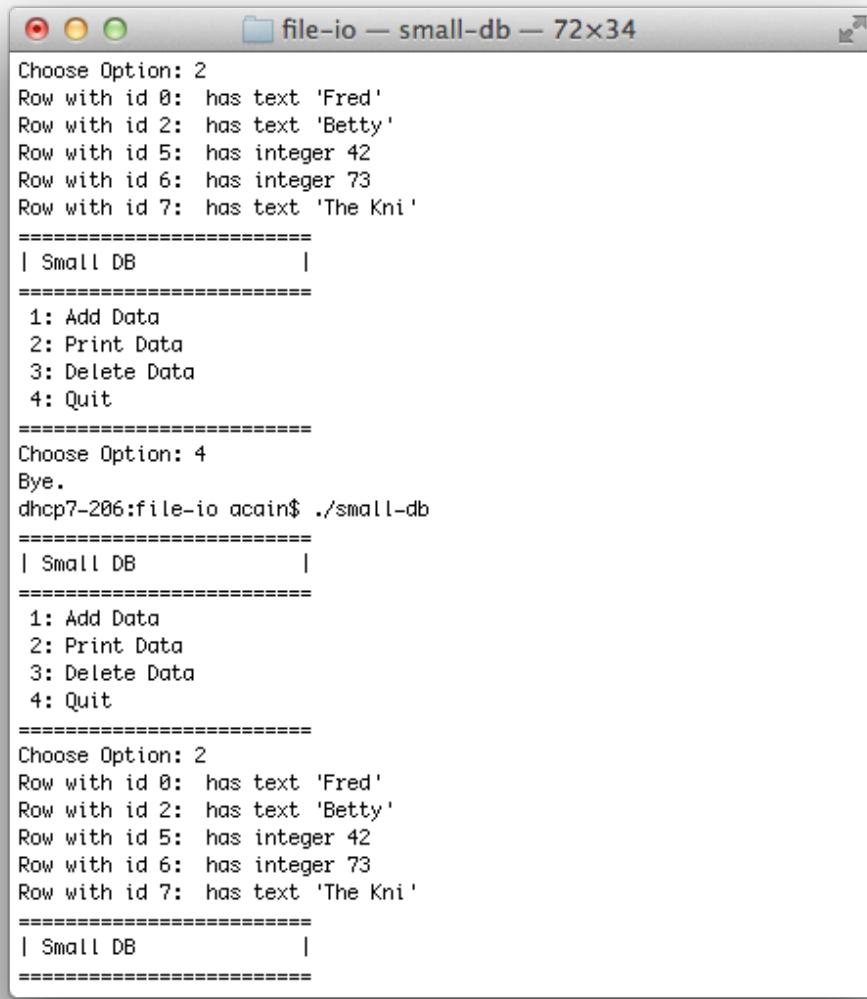
Figure 7.7: The structure chart showing the functions and procedures in Small DB

7.2.4 Writing Code to Load and Save Data for Small DB

Having completed the design for this version of the Small DB program, the next step is to convert these ideas into code. The pseudocode shown in this section communicate the logic that needs to be coded into the functions and procedures of the new version of the Small DB program. The following two sections, Section 7.3 Input and Output in C and Section 7.4 Input and Output in Pascal, contain a description of the tools needed to load and save data in the C and Pascal programming languages.

The good approach for implementing these additions will be to write the code to save the data to file first. Once you have this working you can run the program and check the file to see that the data you added was successfully saved. When this is working correctly you can move on to the code needed to read the values back from file.

Figure 7.8 shows the program running. Notice that that data remains the same even after quitting and running the program again.



```
Choose Option: 2
Row with id 0: has text 'Fred'
Row with id 2: has text 'Betty'
Row with id 5: has integer 42
Row with id 6: has integer 73
Row with id 7: has text 'The Kni'
=====
| Small DB           |
=====
1: Add Data
2: Print Data
3: Delete Data
4: Quit
=====
Choose Option: 4
Bye.
dhcp7-206:file-io acain$ ./small-db
=====
| Small DB           |
=====
1: Add Data
2: Print Data
3: Delete Data
4: Quit
=====
Choose Option: 2
Row with id 0: has text 'Fred'
Row with id 2: has text 'Betty'
Row with id 5: has integer 42
Row with id 6: has integer 73
Row with id 7: has text 'The Kni'
=====
```

Figure 7.8: Running Small DB program twice, notice the data persists between executions



```
next id:8
rows:5
0 1 Fred
2 1 Betty
5 0 42
6 0 73
7 1 The Kni
```

Figure 7.9: The contents of Small DB's data file, from Figure 7.8

7.3 Input and Output in C

7.3.1 Implementing Small DB File IO in C

Section 7.2 presented an altered version of the Small DB program from Chapter 6. The changes introduced four new procedures used to save the programs data to file, and to reload the data from file. The new code is presented in Listing 7.5.

```
// Previous code for small db remains unchanged...

// =====
// = Loading and Saving Data =
// =====

void read_row_from_file(row *to_load, FILE *input)
{
    // Load defaults in case loading fails...
    to_load->id = 0;
    to_load->kind = INT_VAL;
    to_load->data.int_val = 0;

    // Read in the id and the kind (as an integer)
    fscanf(input, " %d %d ", &to_load->id, (int*)&to_load->kind);

    // Branch based on the kind, and output the data
    switch (to_load->kind)
    {
        case INT_VAL:
            // Read in the integer from the file
            fscanf(input, "%d\n", &to_load->data.int_val);
            break;
        // Add double as an option
        case TXT_VAL:
            // Read in the text value from the file (upto 7 characters)
            fscanf(input, "%7[^\\n]\\n", to_load->data.txt_val); // no & as char array
            break;
        default:
            // Dont know what the value is... set it to 0
            to_load->data.int_val = 0;
    }
}

void load(data_store *db_data, const char *filename)
{
    FILE *input;

    input = fopen(filename, "r");
    if (input == NULL) return;

    int i = 0;

    if (db_data != NULL)
    {
        // Store 0 rows as a default, in case load fails
        db_data->row_count = 0;

        // Save the row count...
        fscanf(input, "next id:%d ", &(db_data->next_row_id));
        fscanf(input, "rows:%d ", &(db_data->row_count));
```

```

// Allocate space for rows...
db_data->rows = (row *) realloc(db_data->rows, sizeof(row) * db_data->row_count);
if (db_data->rows == NULL)
{
    fclose(input);
    return;
}

// For each row in the array
for (i = 0; i < db_data->row_count; i++)
{
    // read the row from the file, into the space allocated for this row
    read_row_from_file(&(db_data->rows[i]), input);
}
}

fclose(input);
return;
}

void save_row(row to_save, FILE *out)
{
    fprintf(out, "%d %d ", to_save.id, to_save.kind);

    // Branch based on the kind, and output the data
    switch (to_save.kind)
    {
        case INT_VAL:
            fprintf(out, "%d\n", to_save.data.int_val);
            break;
        // Add double as an option
        case TXT_VAL:
            fprintf(out, "%s\n", to_save.data.txt_val);
            break;
        default:
            fprintf(out, "\n"); //dont save unknown data
    }
}

void save(const data_store *db_data, const char *filename)
{
    FILE *out;
    int i;

    out = fopen(filename, "w");
    if (out == NULL) return;

    if (db_data != NULL)
    {
        // Save the row count...
        fprintf(out, "next id:%d\n", db_data->next_row_id);
        fprintf(out, "rows:%d\n", db_data->row_count);

        // For each row in the array
        for (i = 0; i < db_data->row_count; i++)
        {
            save_row(db_data->rows[i], out);
        }
    }

    fclose(out);
}

```

```
        return;
}

// =====
// = Main =
// =====

// Entry point
int main()
{
    menu_option opt;
    data_store db_data = {0, 0, NULL};

    load(&db_data, "data.txt");

    do
    {
        // Code as before ...
    } while(opt != QUIT);

    save(&db_data, "data.txt");

    return 0;
}
```

Listing 7.5: New C code for the Small DB program with file loading and saving

Note

- The FILE type is used to refer to files that have been opened for the application. See Section [7.3.2 C File Type](#).
-

7.3.2 C File Type

C includes a FILE type that is used to interact with files on the computer. This type includes all of the information that C needs to read and write data to a file.

C++ 

```
#include <stdio.h>

int main()
{
    char message[12] = "Hello World";
    char in_message[6] = "";
    int val = 127;
    int in_val = 0;

    FILE *out;
    FILE *in;

    out = fopen("test.txt", "w");
    if (out == NULL) return -1;
    fprintf(out, "%s\n%d", message, val); //Write the message, and a new line
    fclose(out);

    in = fopen("test.txt", "r");
    if (in == NULL) return -1;

    fscanf(in, "%5[^\\n]", in_message); // read first 5 characters of the message
    fscanf(in, "%*[\\n]"); // skip anything else on the line
    fscanf(in, "%*[\\n]"); // skip the new line
    fscanf(in, "%d", &in_val); // read the number

    fclose(in);

    printf("Read: %s & %d\\n", in_message, in_val);

    return 0;
}
```

Listing 7.6: Example use of the FILE type to read and write text data

Note 

- The FILE type is used to read and write data from a file.
- You open the file using `fopen`.
- After the file has been opened you must remember to close it using `fclose`.
- File based versions of `printf` and `scanf` allow you to read and write data from the file.
- The IO code in C always works with FILE pointers, as a result all of your FILE variables will be pointers.

7.3.3 C File Functions

There are a number of functions and procedures in the **stdio.h** header that will give you the ability to read and write data from files.

Opening a file

Before you can interact with a file the first step will be to open the file. This is done with the `fopen` function. This function will return a `FILE` pointer you can then use to interact with the file.

Function Prototype															
<code>FILE *fopen(const char *filename, const char *mode)</code>															
Returns															
Parameter	Description														
<code>filename</code>	The name of the file to open. This can include a relative or absolute path to the file.														
<code>mode</code>	Indicates the kind of operations that can be performed on the file. Mode should be one of the following: <table border="1"> <thead> <tr> <th>mode</th> <th>description</th> </tr> </thead> <tbody> <tr> <td>"r"</td> <td>Open the file for reading. The file must exist.</td> </tr> <tr> <td>"w"</td> <td>Open the file for writing. This will create a new file, or replace an existing file.</td> </tr> <tr> <td>"a"</td> <td>Open the file for appending data. This will append data to an existing file, or create a new file if it does not exist.</td> </tr> <tr> <td>"r+"</td> <td>Open the file for reading, and writing. The file must exist.</td> </tr> <tr> <td>"w+"</td> <td>Open the file for writing, and reading. This will create a new file, replacing any existing files.</td> </tr> <tr> <td>"a+"</td> <td>Open the file for appending, and reading. This ensures that all write operations are always performed at the end of the file.</td> </tr> </tbody> </table>	mode	description	"r"	Open the file for reading. The file must exist.	"w"	Open the file for writing. This will create a new file, or replace an existing file.	"a"	Open the file for appending data. This will append data to an existing file, or create a new file if it does not exist.	"r+"	Open the file for reading, and writing. The file must exist.	"w+"	Open the file for writing, and reading. This will create a new file, replacing any existing files.	"a+"	Open the file for appending, and reading. This ensures that all write operations are always performed at the end of the file.
mode	description														
"r"	Open the file for reading. The file must exist.														
"w"	Open the file for writing. This will create a new file, or replace an existing file.														
"a"	Open the file for appending data. This will append data to an existing file, or create a new file if it does not exist.														
"r+"	Open the file for reading, and writing. The file must exist.														
"w+"	Open the file for writing, and reading. This will create a new file, replacing any existing files.														
"a+"	Open the file for appending, and reading. This ensures that all write operations are always performed at the end of the file.														

Table 7.1: Details of the `fopen` function

Note

- Remember to check that `fopen` has returned you a valid pointer, if you get back `NULL` then it failed to open the file.
- For binary files you can append a `b` to the mode. For example, to open a binary file for reading you use the mode "`rb`", to open for writing and reading you use "`w+b`".

Closing a file

Once you have opened a file it is important that you also close it. The `fclose` function can be used to close an opened file.

Function Prototype	
<code>int fclose(FILE *stream)</code>	
Returns	
<code>int</code>	Returns 0 when it is successfully closed.
Parameter	Description
<code>stream</code>	The file to close.

Table 7.2: Details of the `fclose` function

Note

- Make sure to close all opened files.
- Remember to check all paths through your code.



Writing text data to file

You can use `fprintf` to write data to a `FILE` that has been opened with write capabilities. This works the same as `printf` and `sprintf`.

Function Prototype	
<code>int fprintf(FILE *destination, const char *format, ...)</code>	
Returns	
<code>int</code>	The number of characters written to the destination by <code>fprintf</code> .
Parameter	Description
<code>destination</code>	The <code>FILE</code> to write the output into.
<code>format</code>	The text that is to be written to the file. This text may contain format tags to include other values. This is the same as <code>printf</code> , see Figure ?? for the syntax of the format tag.
<code>...</code>	Optional values, must have at least as many values as format tags.

Table 7.3: Parameters that must be passed to `fprintf`

Note

- This will write the data as text to the file.
- The file must be opened with write permissions.



Reading text data from file

Reading text data from a file is similar to reading data from the Terminal or from a string. The `fscanf` function works in the same way as `printf` and `sprintf`, but writes its data to a text file.

Function Prototype	
<code>int fscanf(FILE *source, const char *format, ...)</code>	
Returns	
Parameter	Description
<code>int</code>	The number of values read by <code>fscanf</code> .
<code>source</code>	The file from which the input is read.
<code>format</code>	The format specifier describing what is to be read from the Terminal. This is the same as with <code>scanf</code> , see Table 2.4.
<code>...</code>	The variables into which the values will be read. There must be at least as many variables as format tags in the format specifier.

Table 7.4: Parameters that must be passed to `fscanf`

Note

- This will read text data from the file.
- The file must be opened with read permissions.

C++

```
// Program: text_io.h

#include <stdio.h>

int main()
{
    FILE *out, *in;
    int to_save = 20, to_load;

    out = fopen("text.dat", "w");
    if (out == NULL) return 0;
    fprintf(out, "%d", to_save);
    fclose(out);

    in = fopen("text.dat", "r");
    if (in == NULL) return 0;
    fscanf(in, "%d", &to_load);
    fclose(in);

    printf("Wrote %d, read %d\n", to_save, to_load);
    return 0;
}
```

Listing 7.7: Example code that demonstrates writing a value and reading it back from a text file

Writing binary data to file

The `fwrite` function allows you to write binary data to a file. This requires you to pass a pointer to your data, as well as the size and number of elements you want written.

Function Prototype	
<code>size_t fwrite(const void *ptr, size_t size, size_t count, FILE *destination)</code>	
Returns	
Parameter	Description
<code>ptr</code>	A pointer to the data to be saved to the file.
<code>size</code>	The size of each element to be saved.
<code>count</code>	The number of elements to be saved to the file.
<code>destination</code>	The <code>FILE</code> to write the output into.

Table 7.5: Parameters that must be passed to `fwrite`

Note

- The file must be opened with write permissions.
- This will write the binary data to file from the values pointed to by the `ptr` parameter.

C++

```
// program: write_binary.c

#include <stdio.h>

int main()
{
    FILE *out;

    int count = 3;
    double data[3] = { 73.98, 43.21, 3.1415 };

    out = fopen("data.bin", "wb");
    if (out == NULL) return 0;

    fwrite(&count, sizeof(int), 1, out); // write number of doubles
    fwrite(data, sizeof(double), 3, out); // write 3 doubles

    fclose(out);

    return 0;
}
```

Listing 7.8: Example code that writing an array of double values to a binary file.

Reading binary data from file

To read back binary data you need to use `fread`. This reads back a block of data from the file, and stores it in memory at a location indicated by a pointer.

Function Prototype	
<code>size_t fread(void *ptr, size_t size, size_t count, FILE *destination)</code>	
Returns	
Parameter	Description
<code>int</code>	The number of elements read from the destination by <code>fprintf</code> . If this does not equal the count parameter it indicates an error occurred reading the data from the file.
<code>ptr</code>	A pointer to the location to store the loaded data. This must be large enough to store the values loaded.
<code>size</code>	The size of each element to be loaded.
<code>count</code>	The number of elements to be loaded from the file.
<code>destination</code>	The <code>FILE</code> to read the data from.

Table 7.6: Parameters that must be passed to `fwrite`

Note

- The file must be opened with read permissions.
- You must ensure that `ptr` points to sufficient space to load the data into.

C++

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

int main()
{
    FILE *in;
    int i, count;
    double *data; // dynamic array

    in = fopen("data.bin", "rb");
    if (in == NULL) return 0;

    fread(&count, sizeof(int), 1, in); // read the count

    data = calloc(count, sizeof(double)); // allocate space for "count" doubles
    if (data == NULL) return 0;

    fread(data, sizeof(double), count, in); // read data from the file
    fclose(in);

    for ( i = 0 ; i < count; i++ )
    {
        printf("data[%d] = %lf\n", i, data[i]);
    }

    free(data); // Free the space allocated to the data "array"

    return 0;
}
```

Listing 7.9: Example code that reads an array of double values from a binary file.

7.4 Input and Output in Pascal

7.4.1 Implementing Small DB File IO in C

Section 7.2 presented an altered version of the Small DB program from Chapter 6. The changes introduced four new procedures used to save the programs data to file, and to reload the data from file. The new code is presented in Listing 7.5.

```
// Previous code for small db remains unchanged...

// =====
// = Loading and Saving Data =
// =====

procedure ReadRowFromFile(var toLoad: Row; var input: Text);
begin
    // Load defaults in case Loading fails...
    toLoad.id := 0;
    toLoad.data.kind := INT_VAL;
    toLoad.data.intValue := 0;

    // Read in the id and the kind (as an integer)
    ReadLn(input, toLoad.id, toLoad.data.kind);

    // Branch based on the kind, and output the data
    case toLoad.data.kind of
        INT_VAL: ReadLn(input, toLoad.data.intValue);
        // Add double as an option
        TXT_VAL: ReadLn(input, toLoad.data.txtVal); // no & as char array
        else
            // Dont know what the value is... set it to 0
            toLoad.data.intValue := 0;
    end;
end;

procedure Load(var dbData: DataStore; filename: String);
var
    input: Text;
    i, rowCount: Integer;
    nextIdBuff: String[8]; // A buffer used to skip "Next id:" from the file
    rowsBuff: String[5]; // A buffer used to skip "rows:"
begin
    Assign(input, filename);
    Reset(input);

    // Save the row count...
    ReadLn(input, nextIdBuff, dbData.nextRowId);
    ReadLn(input, rowsBuff, rowCount);

    // Allocate space for rows...
    SetLength(dbData.rows, rowCount);

    // For each row in the array
    for i := Low(dbData.rows) to High(dbData.rows) do
    begin
        // read the row from the file, into the space allocated for this row
        ReadRowFromFile(dbData.rows[i], input);
    end;

    Close(input);
```

CHAPTER 7. INPUT AND OUTPUT

```

end;

procedure SaveRow(const toSave: Row; var output: Text);
begin
  WriteLn(output, toSave.id, ' ', toSave.data.kind);

  // Branch based on the kind, and output the data
  case toSave.data.kind of
    INT_VAL: WriteLn(output, toSave.data.intValue);
    // Add double as an option
    TXT_VAL: WriteLn(output, toSave.data.txtVal);
    else
      WriteLn(output, ''); //don't save unknown data
  end;
end;

procedure Save(const dbData: DataStore; filename: String);
var
  output: Text;    //Text = tempfile
  i: Integer;
begin
  Assign(output, filename);
  Rewrite(output);

  // Save the row count...
  WriteLn(output, 'next id:', dbData.nextRowId);
  WriteLn(output, 'rows:', Length(dbData.rows));

  // For each row in the array
  for i := Low(dbData.rows) to High(dbData.rows) do
  begin
    SaveRow(dbData.rows[i], output);
  end;

  Close(output);
end;

procedure Main();
var
  opt: MenuOption;
  dbData: DataStore;
begin
  dbData.nextRowId := 0;
  SetLength(dbData.rows, 0);
  Load(dbData, 'data.db');

  {code as before...}

  Save(dbData, 'data.db');
end;

begin
  Main();
end.

```

Listing 7.10: New Pascal code for the Small DB program with file loading and saving

7.4.2 Pascal Text Type

Pascal includes File and Text types that are used to interact with files on the computer. The File type is used for binary files, the Text type is used for text files.

Pascal

```

procedure Main();
var
  message: String = 'Hello World';
  inMessage: String;
  val: Integer = 127;
  inVal: Integer;
  output, input: Text;
begin
  Assign(output, 'test.txt');
  Rewrite(output); // Change to write mode, deleting old contents
  WriteLn(output, message);
  WriteLn(output, val);
  Close(output);

  Assign(input, 'test.txt');
  Reset(input); // Reset to read from the start of the file

  ReadLn(input, inMessage); // read first 5 characters of the message
  ReadLn(input, inVal); // read the number

  Close(input);

  WriteLn('Read ', inMessage, ' & ', inVal);
end;

begin
  Main();
end.

```

Listing 7.11: Example use of the Text type to read and write text data



Note

- The Text type is used to read and write text data from a file.
- You open the file for reading using `Assign` then `Reset`, to write use `Assign` then `Rewrite`.
- After the file has been opened you must remember to close it using `Close`.
- File based versions of `WriteLn` and `ReadLn` allow you to read and write text data from the file.



7.4.3 Pascal File Procedures

There are a number of functions and procedures in Pascal that will give you the ability to read and write data from files.

Assigning a filename

Before you can interact with a file the first step will be to assign a filename to the Text variable.

Procedure Prototype	
procedure Assign(var fileVar: Text; filename: String)	
Parameter	Description
fileVar	The file variable to have its filename set
filename	The name of the file to open. This can include a relative or absolute path to the file.

Table 7.7: Details of the Assign procedure

Opening the file to read

The Reset procedure is used to reset the file cursor to the start of the file for reading.

Procedure Prototype	
procedure Reset(var fileVar: Text)	
Parameter	Description
fileVar	The file to reset, after this call you can read from this file.

Table 7.8: Details of the Reset procedure

Opening the file to write

You can call either Rewrite or Append to write to the file. Rewrite deletes the old file contents, append moves to the end of the file and adds new data there.

Procedure Prototype	
procedure Rewrite(var fileVar: Text)	
Parameter	Description
fileVar	The file to rewrite. After this call you can write data to this file, this will override the existing contents.

Table 7.9: Details of the Rewrite procedure

Append can also be used to open a file for write access. This will not overwrite existing data, keeping the cursor at the end of the existing file.

Procedure Prototype	
procedure Append(var fileVar: Text)	
Parameter	Description
fileVar	The file to append data to. After this call you can write data to this file and it will appear after the existing contents of the file.

Table 7.10: Details of the Rewrite procedure

Closing a file

Once you have opened a file it is important that you also close it. The Close procedure can be used to close an opened file.

Procedure Prototype	
procedure Close(var fileVar: Text)	
Parameter	Description
fileVar	The file to close.

Table 7.11: Details of the Close procedure

Note

- Make sure to close all opened files.
- Remember to check all paths through your code.

Writing text data to file

You can use WriteLn to write data to a Text file that has been opened with write capabilities.

Procedure Prototype	
<code>procedure WriteLn(destination: Text; ...)</code>	
Parameter	Description
destination	The Text file to write the output into.
...	The data to be written

Table 7.12: Parameters that must be passed to WriteLn

Note

- This will write the data as text to the file.
- The file must be opened with write permissions using Rewrite or Append



Reading text data from file

Reading text data from a file is similar to reading data from the Terminal or from a string.

Procedure Prototype	
<code>procedure ReadLn(source: Text; ...)</code>	
Returns	
int	The number of values read by fscanf.
Parameter	Description
source	The Text file from which the input is read.
...	The variables into which the values will be read.

Table 7.13: Parameters that must be passed to ReadLn

Note

- This will read text data from the file.
- The file must be opened with read permissions.



Pascal

```
program TextIO;

procedure Main();
var
    output, input: Text;
    toSave: Integer = 20;
    toLoad: Integer;
begin
    Assign(output, 'text.dat');
    Rewrite(output);

    WriteLn(output, toSave);
    Close(output);

    Assign(input, 'text.dat');
    Reset(input);

    ReadLn(input, toLoad);
    Close(input);

    WriteLn('Wrote ', toSave, ', read ', toLoad);
end;

begin
    Main();
end.
```

Listing 7.12: Example code that demonstrates writing a value and reading it back from a text file



7.5 Input and Output Exercises

7.5.1 Concept Questions

1. Why would you want to save data to file?
2. What are the basic operations that you can perform with files?
3. What is the difference between a text file and a binary file? Explain.
4. What challenges to saving variable sized data to file raise? How can these be addressed?
5. How is reading and writing to files similar to working with the Terminal?

7.5.2 Code Writing Questions: Applying what you have learnt

1. Write two programs that save three integer values to file, one that saves it to a text file the other to a binary file. Execute the two programs and compare the files created.
2. Write two programs that read three integer values from file, one that reads them from a text file the other from a binary file. Once the data is loaded into memory have the program output them to the terminal.
3. Use the code from this chapter to implement saving data for the Small DB program.
4. Revisit your statistics programs and have it save the values the user enters to file when the programs ends. Once this is working add the code to load the data from the file when the programs starts.

7.5.3 Extension Questions

1. Explore the other file IO functions and procedures offered by the language you are using. See if you can work out how to move the cursor within the file (seek to a new location). Use this to move back and forth within a file reading individual values in response to user input.