Pasifika C++

AIDAN DELANEY, ALLEN B. DOWNEY, NARENDRA SISODIYA, TIRTHA P. CHATTERJEE

Contents

1 Introduction

1.1 Objective

In this chapter we motivate why we study computer programming.

1.2 Background

There are a lot of interesting things out there in the world. You could learn about magical realism in literature from Borges' writings through to Del Toro's excellent movie Pan's Labyrinth. Or maybe you're interested in how Einstein's theories of relativity led to Quantum Mechanics. Amidst all of these interesting topics why choose to study computer programming? I have multiple, and sometimes contradictory views, on why I like programming. It's essentially reductionist, to do it correctly requires great social skills and it represents about 50% of my daily universe. I'll explain each of these claims in turn.

Computer programming is reductionist. We study a problem and reduce the problem to it's bare bones. As an example, we reduce everything to numbers – we can't represent the depth and breadth how much you might love a pet, but we can represent it on a scale from 0 to 100! After reducing the problem, we then produce a computer program that solves the problem from very basic components. Each time you write a program it's like walking into a workshop containing a bunch of steel pipes and a furnace and being told that you have to build a motorbike. We generally look a problems from the bottom-up; how can we solve the problem with the small number of components that are available to us.

By contrast, the people who want us to write programs – employers, family, friends and wider society – look at things from a top-down perspective. They need to automate some systems in order to solve a real-world problem. Your employer might want a program to calculate their annual tax bill, and then automatically transfer the tax payment directly to the tax authorities[1]. The perspectives of

[1] This doesn't apply in Vanuatu as there is no income tax or corporation tax, but they are being introduced in the near future.

such stakeholders is almost the opposite of reductionist. It requires thinking about an entire system. Moreover solving real-world problems develops excellent communication skills in order to tease out the real issues the stakeholder wants to solve. Often stakeholders ask for a program to do X when they really want to solve problem Y. Programmers have their own language for describing such things. We call it the XY Problem.

So far we've considered a reductionist bottom-up reason to study programming. We've also briefly discussed how you also how programming can help you to develop great communication skills. My third reason for studying programming is an argument made by Simon Peyton Jones. He argues that in the 20th Century we introduced the natural sciences into schools – that is physics, chemistry and biology – so that all adults would have a good understanding of the physical world in which we live. In the 21st Century we live much of our day outside the physical world and inside the virtual world[2]. Many of our relationships are mediated by social networks – I have good friends who I've never seen. Other simple things in life are also digital, bank transfers can be organised through a web browser and avoid standing in line in a physical bank. There are growing trends where we use software to solve transport issues, negating the reasons to cough up $14k on a second-hand Toyota. The virtual world has such a huge impact on our real-world that it is useful to study programming so that we can understand the building blocks of the virtual world.

So there are three reasons to study programming; the first because it's a big box of lego from which you can build really interesting things, the second because it solves interesting (and not so interesting) real-world problems that require a lot of communication and the third reason is because it give you a good understanding of the virtual universe in which we spend much of our daily life.

1.3 What is Programming

If you're still reading you're either convinced by one of my three reasons for studying programming or you need the course credit given by passing a programming class. Hopefully it's the former as intrinsic motivation is more likely to drive you to success than extrinsic motivation. In both cases you know why you're studying programming, the question now becomes "what is programming?".

In one view, a program is an *unambiguous* list of instructions. Each time you follow the list of instructions, you will achieve the same result. The comparison is often made with cooking, where recipes are often shared as a list of instructions. Take for example the instructions for making a chicken lovo or umu. In the case of lovo you might get the instructions to:

1. Dig a pit,
2. Build a fire,

[2] If you don't believe me on this, then have a look around and see how many students think I can't see them texting under a desk in your next class.

3. Carefully place lava rocks over the fire,
4. Light the fire and let it burn down to embers,
5. Season the chicken,
6. Wrap the chicken in banana leaves,
7. Put the wrapped chicken on the hot stones,
8. Cover the food with earth
9. Wait 2 hours, remove chicken and eat!

To most people, the above instructions are enough to make a great dinner. However these instructions are ambiguous. They don't say how deep or wide to build the pit. They don't say how high the fire should be within the pit. How large should the lava rocks be? Or what do we season the chicken with? Moreover, if you've ever made a lovo or umu you know it's a lot of effort. Too much effort to cook a single chicken so maybe we should also place taro and fish alongside the chicken. The reason that the instructions can be ambiguous is because you, dear reader, are intelligent enough to fill in the blanks.

Computers, of course, are not intelligent. They cannot fill in any blanks in an explanation. So we need to present them with a list of instructions written in an unambiguous language. The process of producing unambiguous explanations is called *proof* by mathematicians, but we'll call it *programming*.

1.3.1 Formalisation

In order to remove ambiguity from the interpretation of sentences we're going to abandon the use of natural language. We won't try to write our instructions in English, French or Bislama. These are fantastic languages in which to express love or revolution. However, the very fact that these languages allow us to express and discuss poorly defined concepts is the very reason that they are unsuitable for providing unambiguous instructions to a machine. To program a computer we need a formal language, one in which each word and sentence has a well-defined meaning.

By writing instructions in a formal language we remove ambiguity from the instructions. It's also important to note that we also loose something in translation. A bunch of instructions that describe making a traditional lovo can be read and understood by any literate person. Lovo instructions written in an unambiguous language are likely to be very difficult to read. In order to illustrate this I need to introduce a problem that is much more straightforward than making a lovo. Let's consider the kids game *hangman*.

The game *hangman* is played by kids in schools all over the world. One student chooses a secret word and writes a number of dashes on a piece of paper. There is one dash for each letter of the secret word. The same person also draws a gallows[3]. A friend then guesses letters of the secret word. If they correctly guess a letter of the word then the letter is written on a dash in an appropriate position. If

[3] The name *hangman* and drawing a gallows are pretty awful, but then again most kids are awful!

they do not guess correctly then a bit of a stick man is drawn hung on the gallows. The friend wins the game if they correctly reveal all the letters of the word before the full stick man is drawn on the page.

Our previous paragraph is a reasonable explanation of the game. Again, like our lovo example, you're probably able to fill in the bits where I've explained it poorly. What if we had to write the instructions of the game in a way that they couldn't be misinterpreted? Legal people commonly write such instructions:

> The game of hangman, herein referred to as THE GAME, is a game for two parties referred to as THE PLAYER and THE JUDGE.
>
> (a) THE JUDGE will draw on paper a gallows and below the gallows a series of dashes where there is one dash for each letter of a secret word.
> - THE JUDGE will not prima-facia reveal the secret word to THE PLAYER
>
> (b) THE PLAYER will guess a letter of the secret word and
> - if the guessed letter is one or more letters in the secret word, then THE JUDGE writes the guessed letters on top of a dash that is at the same position in the series as the guessed letter in the secret word.
> - otherwise THE JUDGE will draw the next body part of a stick man on the gallows.
>
> (c) the body parts of a stick man are drawn in progression, on per turn, starting with a head, a body, a left leg, a right leg, a left arm and a right arm.
>
> (d) THE JUDGE wins the game if the stick man is drawn before the secret word is revealed.

The legal code for *hangman* is useful when two players need to argue about the implementation of a rule. We can still argue about the interpretation of some of the rules; should the judge draw the arm as an upper arm and lower arm? Can the player guess two letters at a time?[4] In our case we want our rule to be interpreted by a machine that can't argue about how to implement a rule. We know our machines have no intelligence. They can simply replace symbols with other symbols

Hopefully you now agree that our natural languages, even when restricted to legal language, are not precise enough to instruct a dumb computer. We need languages that are un-natural. We need formal languages that are suited to describing how to manipulate symbols and "move" symbols around. Such a language is going to be difficult to read because of its lack of expressiveness. It doesn't have the expressive *bandwidth* of a natural language.

In a more formal language designed for computation – a **programming language** – we have to write a lot of instructions to describe hangman. The instructions might look something like:

```
start program
run instructions to construct gallows
```

[4] A solicitor could write these rules in a different way. They could first formalise the definition of a turn and then describe the progress of the game as a series of turns. In any case the formalisation into legal language adds more complexity than the informal explanation.

```
run instructions to input secret word
repeat run instructions to take a turn until the game is over
end program

instructions to take a turn
ask the player to input a single character
read in a single character
if the single character is in the secret word
then run instructions to uncover characters
otherwise run instructions to draw next stroke
end instructions to take a turn
```

I've omitted a *lot* of detail above in order to make it readable. I've also kept my example formal language as close to English as possible. In the C++ programming language the code to `start program` and `run instructions to construct gallows` might look like :

```
int main() {
  construct_gallows();
}
```

It's a lot more compact than my English-like language. It is also much more difficult to read. We want to learn C++, which I'll delve into below, but first we'll look at what computers can do in order to better motivate why we want to learn such a precise language[5].

1.4 What can Computers Do?

It's obvious that a computer can't cook a lovo. So why should we bother devising instructions for them in complicated looking formal languages? Simply put, computers are insanely quick at performing mathematical operations. So if you can formalise the solution to your problem in terms of mathematical operations then a computer can solve it quickly. As an example, there are around 3000 staff at the University of the South Pacific. To calculate the monthly pay packet for a staff member might take a clerical assistant about a minute or two, which is roughly a week of work in total. A computer can perform the same functions in less than a second. Moreover, if the instructions given to the computer are correct, then the computer will not make a mistake on any pay packet. From an organisational point of view this allows us to free staff from boring repetitive work. We can then use those staff for tasks that humans are better suited to, such as teaching or research.

The example of calculating pay packets demonstrates how completing mathematical operations quickly is of use. Some other uses of computers are less clear. Consider the last movie you watched on a

[5] Assuming you're not already motivated by the huge salaries that programmers command! Or in the immortal words of MC Solaar "Du cash-money… Une voiture rouge. Donne-moi tout ça sinon faut qu'tu bouges".

computer. The movie is stored in a file. The file is a sequence of numbers that, roughly speaking, can be interpreted as 25 pictures per second. Each picture states the colour of pixels on the screen. The act of watching the movie requires a program that interprets the data file and plays the 25 pictures per second on your screen. Again, we're only using mathematical operations to calculate colour values and moving symbols around inside the memory of the computer.

You might not consider the fuel injector of a boat engine to be a job best handled by a computer. It is the case though that all modern engines use a computer, referred to as the *engine control unit*. The job of the engine control unit is often to control the timing of firing each cylinder in the engine. In many cases the control unit can read a stream of numbers provided to it from the engine exhaust and adjust the engine to run within pre-determined environmental limits. One practical outcome of this approach is a reduction in the amount of fuel used by an engine. This is something that is simply impractical to achieve without a computer in control.

So we know that computers are useful for movie night and driving to the movies. You can find other applications of computers in medical devices and – obviously – in mobile phones and running the entire Internet! We know that computers can do a lot, leading to the interesting question asking whether there are things that computers can't do?

1.5 What Computers Can't Do

Computers can be used to solve many problems. Are there problems that computers can't solve? The short answer is "yes", but a longer answer is more interesting. The idea of a computer was defined in 1936 by the mathematician Alan Turing. His idea has three very practical consequences:

1. If it can't be formalised then it can't be computed,
2. There are some formalised problems that can't be computed,
3. There are a large number of practical problems that can be computed but take too long to compute.

The first is a fairly obvious consequence of the previous discussion on formalisation. Humans can't seem to agree on a definition of *love* or even of what good music is. As these ideas can't be formalised in our formal language, then they can't be computed. The second point is not obvious and was the central point of Turing's definition of computing. There are problems that can be formalised but cannot be computed on a computer. One of these is the measure of minimum amount of information necessary to construct an original information source. You've probably used an approximation to this measurement. The `.zip` file format stores information compressed by an algorithm that approximates this problem. We can't give you the very smallest `.zip` file that represents your original information but we can give you a good guess at it.

The third consequence is again non-obvious but you use it every single day. Almost all encryption

on the web, that is HTTPS traffic, is based on a problem that can be computed but simply takes too long to reverse. Another problem, called the travelling salesperson problem (TSP), also illustrates how some computable problems take too long to compute. The TSP asks us to compute the quickest way to visit all USP campuses. USP has a on campus in Suva, Lautoka and Labasa in Fiji, it takes 3 hours to travel between Suva and Lautoka, 16 hours to travel between Suva and Labasa and 26 hours to travel between Lautoka and Labasa, this is visualised in figure {@fig:tsp}.

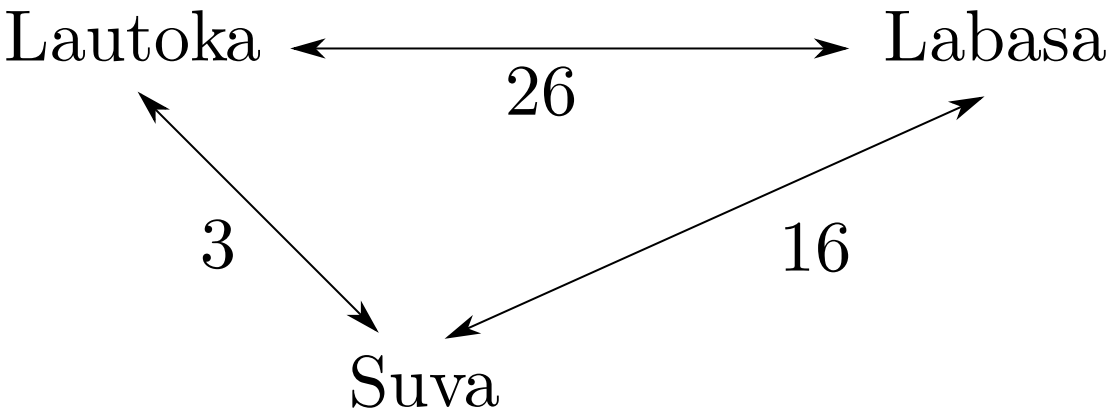


Figure 1: The travelling salesperson problem

I've turned the statement of the problem into a graph. This makes it a bit easier to focus on the actual problem, not on details such as exactly how we get from Suva to Lautoka. Given this kind of graph the TSP asks what is the shortest path in the graph that passes through all campuses? We can start in Suva and end in Lautoka or we can start in Lebasa and end in Suva. In this graph there are 3 combinations[6] of starting and ending points:

1. Suva to Lautoka (via Lebasa),
2. Suva to Lebasa (via Lautoka), and
3. Lautoka to Lebasa (via Suva).

For 3 campuses it's easy to work out which is the shortest route through all. You can probably do it in your head. As we in more campuses, say Port Vila in Vanuatu and Apia in Samoa, we find that the number of combinations is huge. Computer Scientists[7] have worked out that for 5 campuses there are $5^2 \times 2^5$ routes to calculate, that's 800 routes to calculate in total. If we make an assumption that it takes 1 micro second to calculate a route then, for 5 campuses, it takes less than a second to find the cost of all routes. That's great and much quicker than a human could do it.

Keep in mind I said that this problem takes a long time to compute. It appears that so-far I've not been telling the truth. Suppose then that we add in all the USP campuses and the costs of travelling (in

[6] You'll note that if this graph had a quick way of going from Suva to Lautoka but a slow way of getting from Lautoka to Suva, then there would be 6 combinations. We're dealing with the *undirected* case here.

[7] All Computer Scientists are awesome people. They're also very intelligent and witty individuals.

time) between each campus. There is one campus in each of 11 countries and then 3 campuses in Fiji, a grand total of 14. Just for fun we can also add in the University of Auckland as I have some research colleagues there and Université de la Nouvelle-Calédonie because that covers all the teams that play in the Suva 7's Rugby tournament. For 16 campuses of various universities there are a lot of routes to calculate:

$$16^2 \times 2^{16} = 16777216.$$

If we make the same assumption that a route costs 1 micro second to calculate then it will take 16777 seconds, or 279 minutes or 4.65 hours to calculate the correct answer. Again, if we transpose the problem to the EU, which (currently) contains 28 countries, we find that the TSP between each of the capital cities in the EU produces 210453397504 combinations or 6.6 *years* of computation. As the problem grows in size, the time it takes to solve the problem grows by an unmanageable amount. This is a direct consequence of Turing's definition of computation. Interestingly, there is an unproved hypothesis that says that if you can think of any other way of making a physical machine that can do computing, then your machine will only ever compute exactly what Turing described. This is the Church-Turing Thesis.

So we now have reasons to study Computer Science and we have some insight into why we have to learn a formal language in order to write computer programs. In this book we use C++ as our formal language. There are other computer programming languages, so why do we learn this particular one?

1.6 Why C++

C++[8] is a powerful programming language which has been used in industry for over 30 years. Many of the software applications that you use day-to-day are written in C++. These include your web-browser [9] and your word-processor [10].

The C++ language is a **standard**. This means that the great and the good of C++ programming get together regularly to improve the language. They produce a document that describes the latest version of C++. The current version of this document is C++17. It's the version of the standard that was produced in 2017. We expect the next version of the standard to be agreed in 2020 and it will be called C++20. I take the view in this book that we only work with modern C++ and in the ways that modern C++ programmers should work. This means that I assume we're working with, at least, C++14.

Like many programming languages, C++ is written in text files, normally with a `.cpp` file extension. These files simply contain plain-text and you can edit them with any editor that allows you to open

[8] Not to be confused with the Nerd Core rapper MC plus +.
[9] Each of Firefox, Chrome and Edge are largely written in C++.
[10] Both Libreoffice Writer and Microsoft Word are written in C++.

text (note: this excludes word processors – word processors don't edit plain text, they're much more complex than that in order to look like you're editing an A4 size page.) The C++ files are then **compiled** into a program. The understandable C++ is translated into an non-understandable sequence of 1's and 0's i.e binary machine code. A **compiler** translates from C++ to machine code.

With C++ we can choose to use a number of compilers from different vendors. Because of the C++ standard document we have a high guarantee that code that compiles using one compiler will operate in the same way it does when compiled with another compiler. The main C++ compilers are:

GCC The GNU Compiler Collection contains a very high-performance C++ compiler that runs on many operating systems and on many computer architectures. The Linux kernel and LibreOffice prefer to use GCC as they have to run on a variety of platforms. GCC is an open-source project and is free of cost to download and redistribute.

LLVM The LLVM Compiler Suite also contains a C++ compiler. Google's Chrome browser prefers this compiler and it's the default C++ compiler on Apple systems. LLVM is also open-source and free of cost.

Visual C++ Microsoft have their own C++ compiler that only runs on their Microsoft Windows operating system.

Intel, HP and IBM also have C++ compilers for various systems. Often these are very specialised high-performance computing machines. High-performance engineers love C++! Games programmers, particularly on consoles, also tend to prefer C++. But like everything in computing this is subject to constant change.

We now know that a compiler is a program that reads a high-level program and translates it all at once, before executing any of the commands. Often you compile the program as a separate step, and then execute the compiled code later. In this case, the high-level program is called the **source code**, and the translated program is called the **object code** or the **executable**.

As an example, suppose you write a program in C++. You might use a text editor to write the program (a text editor is a simple word processor). When the program is finished, you might save it in a file named `program.cpp`, where "program" is an arbitrary name you make up, and the suffix `.cpp` is a convention that indicates that the file contains C++ source code.

Then, depending on what your programming environment is like, you might leave the text editor and run the compiler. The compiler would read your source code, translate it, and create a new file named `program.o` to contain the object code, or `program.exe` to contain the executable. (Aside: on Microsoft Windows executable files have the `.exe` extension, on Unix (on x86) the files do not have to have a certain extension, but contain what's known as a *magic number*, the characters `.`, `E`, `L'` and`F'` as the first four characters in the file.)

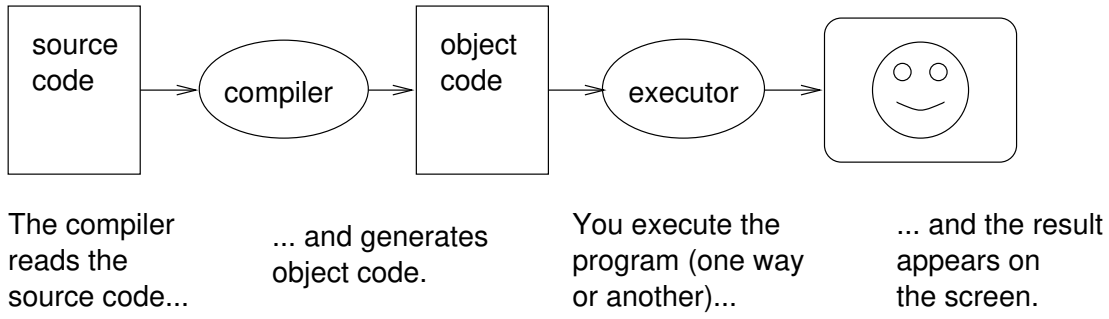


Figure 2: The compilation process

The next step is to run the program, which requires some kind of executor. The role of the executor is to load the program (copy it from disk into memory) and make the computer start executing the program.

Although this process may seem complicated, the good news is that in most programming environments (sometimes called development environments), these steps are automated for you. Usually you will only have to write a program and type a single command to compile and run it. On the other hand, it is useful to know what the steps are that are happening in the background, so that if something goes wrong you can figure out what it is.

1.7 What we can do now

We now have an understanding of the need for computation and we understand the limits of computation. We write software as a sequence of instructions in a formal language which is compiled into executable code.

2 Hello World

2.1 Objective

In this chapter we motivate why we use a poor user interface – the console. We also write our first program.

2.2 The Console

It's traditional to begin programming by writing a program that prints "Hello, world" on the screen. We will do this in the most simple way that we possibly can. However, this requires us to use a pretty ancient way of interacting with a computer. You might imagine that opening a window onscreen and

writing “Hello, world” in that window requires a reasonable amount of code. So we’ll avoid using windows and mouse input. We’re going straight back to the kinds of interfaces everyone used in the 1980’s and that are primarily only used server-side in this decade. In order to use this interface, which we’ll call the *console*, I’ll explain some of the features and limitations.

The console is a boring window. You can write text to it, one character at a time and one line at a time. It is a text-only interface. On Microsoft Windows a console is provided by PowerShell on Linux and other Unix-like systems the console is provided by a terminal emulator as seen in figure {@fig:console}. The console can trace it’s roots to around 1964, so you can’t expect it to work like the nice interface that you find on your phone or your desktop computer. The best analogy I’ve found is to think of it like text messaging[11] your computer and the computer responding with a reply text message. The interface feels clunky and error-prone to modern computer users. We are only using the console because of the simplicity of writing programs that use it.

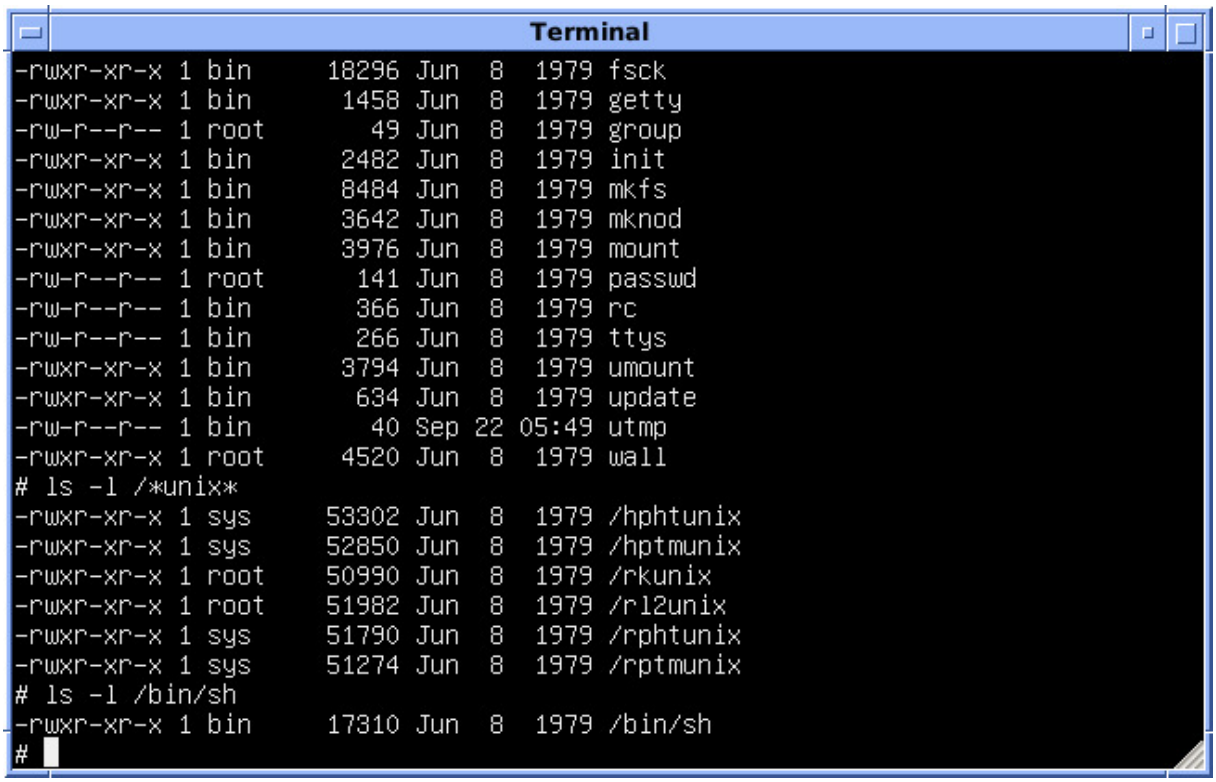


Figure 3: Text-only console (image from Wikipedia)

[11] By text messaging I mean Short Message Service (SMS)

2.3 First Program

Most of our first program will seem to be magic. But all of it is understandable if you take time. Just remember that in C++ each non-whitespace character is important. Whitespace characters are tabs, spaces and new-lines. Let's have a look:

```
#include <iostream>

int main() {
  std::cout << "Hello, world" << std::endl;
  return 0;
}
```

The first line `#include <iostream>` is a direction to bring functionality for writing to the console into our program. The `iostream` functionality defines `std::cout` and `std::endl` and the weird `<<` thing (called the *ostream operator*). We'll return to understanding exactly what these mean a little later. First I'd like to introduce `main`.

All C++ programs start in a *function* called `main`. This is something defined by the C++ standard. It could have been called `start` or `begin_here`, but it wasn't. For traditional reasons the starting point of a C++ program is called `main`. It's not called `Main`, which is different from `main`. Almost all programming languages, including C++, are case-sensitive. So the function name `Main` is not the function named `main` because the initial letters differ in capitalisation. I'll repeat that each non-whitespace character in a C++ program is important.

The `main` function starts at the opening curly brace `{` and ends at the closing curly brace `}`. In programming we often use brackets, so we have to refer to them correctly. Our program uses

parentheses `(` is a left parenthesis and `)` is a right parenthesis. You might commonly call these brackets, but programmers use the technical names for them.

braces we've already seen `{` and `}` referred to as braces or curly braces.

angular brackets the `<` and `>` characters are left and right angular brackets respectively.

In addition to the various types of brackets our program contains colons, `:`, and semi-colons, `;`. It's easy to confuse `:` and `;`.

Now that we've gotten all that out of the way, we can examine the *body* of the `main` function (the two lines between the curly braces). The first of these two lines is the most interesting. The line

```
std::cout << "Hello, world" << std::endl;
```

says a lot, it directs – using the ostream operator `<<` – the string of text, "Hello, world", followed by an end of line character, `std::endl` to the console. The console is represented by `std::cout`. We may

think of `std::cout` being the **st**andar**d** **c**onsole **out**put. Keep in mind that "Hello, world" is a **string**. We will look at string, int and some other **types** in the later section **`int`** and Other Types.

The << operator makes it easy(ish) to send more interesting information to `std::cout`:

```
std::cout << "Hello, world " << "it is today" << std::endl;
```

or

```
std::cout << "Ireland: " <<  23 << " Fiji: " << 20 << std::endl;
```

we can output `string`s and numbers[12].

There's a strange looking character, ␣, in the above string. This ␣ character is often used in programming manuals to indicate the presence of a space. The ␣ character draws your attention to the whitespace that we might otherwise ignore.

Finally, out first program contains the line **`return`** `0`. All C++ programs must **`return`** a number at the end of their `main` code. If the program has had no errors, it must return `0` to the operating system. As a computer user you never see this number. As a computer programmer this return number provides a lot of information to me. Let's see an example on the Unix terminal.

```
# Print out the files in this directory
$ ls
...
# Print out the return code
$ echo $?
0
# Make a mistake with the ‘ls‘ command
$ ls moo
ls: cannot access 'moo': No such file or directory
# Print out the return code again
$ echo $?
2
```

In the above example we can see that when we run the `ls` program correctly it returns `0` to the operating system. We can see this `0` on a Unix by asking the command interpreter to `echo $?`. When we run `ls` and it generates an error we can see that it returns `2` to the operating system. As a programmer I can look up this return code in a manual to further diagnose why the problem occurred.

Once we have a **`return`** `0` as the last line of `main` we can mostly ignore this detail. I've gone into an explanation of these details because it demonstrates one of the many differences between an advanced

[12] Which also reminds Fijians of the last scoreline between Ireland and Fiji.

computer user and a programmer. As a programmer you have to have a deeper understanding of the operating system on which you're running. You will have to appreciate entire computer systems at a deeper level.

2.3.1 A Stylistic Note on Whitespace

Programming style is vitally important. Our first program could also be written as follows:

```
#include<iostream>
int main(){std::cout<<"Hello, world"<<std::endl; return 0;}
```

Both programs do the same thing. However, one is readable by a human and the other is very difficult to read. There are some common styles for writing a program and I suggest you be consistent in your application of a style. I tend to write C++ where I indent code using 4 spaces (no tabs). If you use spaces to indent your code, then do not mix in tab characters!

I use newlines between statements, so the `std::cout` above and the **`return`** statement should appear on separate lines.

2.4 What we can now do

At the end of this chapter we can write a long `main` function which generates lots of output to the console.

```
#include <iostream>

int main() {
  std::cout << "Hello";
  std::cout << " this is Nuku'alofa calling." << std::endl;
}
```

3 `int` and Other Types

3.1 Objective

In this chapter we introduce the ideas of types, values and variables which are fundamental to all programming languages.

3.2 Simple Types

At a fundamental level in computers all information is represented as sequences of 1's and 0's. Take, for example, the sequence 01000001. If we interpret that sequence as a whole number then it represents the number 65. However, if we interpret the same sequence as a printable character, then it represents the character "A". Also, there are operations that we can perform on whole numbers, such as multiplication, that don't make sense to perform on characters. So, types will allow us to organise our data and to ensure that we only perform operations that make sense on that data. We have seen two types, `int` and `string`. I'll explain these now and add in some other basic types.

In C++ we have a type that represents whole numbers. That type is `int`, short for integer. It represents positive and negative whole numbers. So $-256, 0$ and 1024 are examples of `int`s. I can create a storage space to hold an `int` inside a function:

```
int main() {
  int x = 42;
  return 0;
}
```

In the above code, we create a new storage space called `x`. The storage space can only hold whole numbers. In this case I have assigned the value 42 to the storage space called `x`. I can also change the number in the storage space.

```
int main() {
  int x = 42;
  x = -1;
  return 0;
}
```

The first statement in the `main` function `int x = 42;` does exactly what we've previous seen. The second line, `x = -1` overwrites the value 42 in storage space `x` with the value -1. When we first mention `x` we have to tell the computer that we want `x` to be an `int`. On subsequent uses we don't have to tell the computer the type. The C++ compiler keeps track of that kind of information for us. Let's use the same pattern to construct storage spaces for other types.

I can store a character:

```
int main() {
  char c = 'A';
  return 0;
}
```

In this code we create a storage space, called `c`, to store a **`char`**. Like **`int`** is a shortened form of integer, the word **`char`** is shortened from character. I've also had to put the "A" character in single quotes. This is part of the C++ language, characters are single letters and must be surrounded by single quotes. Recall from our introduction that C++ is a formal language. One consequence of this is that we are restricted in how we write things.

There are two other interesting types, the first is `string` and the second is **`float`**. A **`float`** represents a number containing a decimal point. There are technical limitations to what **`float`**s can represent, so you can't treat them as precisely equivalent to decimals that you know and love from maths. For all the calculations we do in this book an **`int`** or a **`float`** is sufficient.

Let's write a more complex program that uses `string` and **`float`**. In this program each `string` of characters is surrounded by double quote marks. Moreover, the storage places in this program have more descriptive names than "c" or "x" used above. It is good to use descriptive names for your storage boxes.

```
#include <iostream>
#include <string>

int main() {
  std::string name = "Hiro Protagonist";
  float height = 183.5;

  std::cout << name << " is " << height << "cm tall." << std::endl;
  return 0;
}
```

The cool thing about the above code is that it uses functionality to print the values stored in `name` and `height` to the console. You'll also notice that the type of **`float`** is just **`float`** but the type of a string is `std::string`. In C++ **`int`**, **`char`** and **`float`** are part of the core language but `std::string` comes from the standard library: hence the `std::` prefix on the type name.

3.3 Type Errors

Types are an incredibly useful feature of programming languages. By enforcing type checking the compiler can let the programmer know that they've made an error in writing their code. As an example, it makes no sense to try and store a `string` in a storage box that should contain an **`int`**:

```
int main() {
  int age = "seventeen";
  return 0;
```

```
}
```

This error generates the rather cryptic output from a compiler:

```
type_errors.cpp: In function 'int main()':
type_errors.cpp:2:13: error: invalid conversion from 'const char*' to '
    int' [-fpermissive]
    int age = "seventeen";
              ^~~~~~~~~~~
```

Type errors are frustrating when you're learning to program. But they're less frustrating than speaking to an irate customer who just had your program fail because of a type error! It is useful to deliberately introduce some errors into a working program so that you can get used to the way compilers report errors.

3.4 Storage Boxes

Thus far I've used the term *storage box* to describe how we store values of a given type. We need to develop a more precise terminology in order to describe exactly what is going on here. The term we use is **variable**. In the following code we **declare** a variable of type `int`.

```
int main() {
  int my_variable;
  return 0;
}
```

We can, as we have seen, declare a variable and *initialise* it at the same time:

```
int main() {
  int my_variable = 42;
  return 0;
}
```

Or we can declare it in one statement and initialise it in another:

```
int main() {
  int my_variable;
  my_variable = 42;
  return 0;
}
```

If we try to use a variable before we have declared it, then we are in violation of the C++ language. When we violate the language our compiler generates an error.

3.5 What we can now do

We can now create variables of different types and output them to the console:

```
#include <iostream>

int main() {
  std::string university_name = "University of the South Pacific";
  int founding_year = 1968;
  int age = 2018 - founding_year;

  std::cout << university_name << " was founded in " << founding_year
     << std::endl;
  std::cout << "It is " << age << " years old." << std::endl;
  return 0;
}
```

We can create different instances of the program for other Universities or companies.

4 Functions

4.1 Objective

In the last chapter we introduced types. We now show how to pass instances of types around a program.

4.2 Simple Functions

We have seen an example of a function. The `main` function is the starting point of every application. We can write other functions too. These are vitally important to help us organise the structure of our application. I'm going to write a stupid function just to demonstate the concept. All functions have a name, and this function is called `zero`. We can call our `zero` function from another function, say `main`, and it will **`return`** the value `0`.

```
int zero() {
  return 0;
}
```

```
4
5  int main() {
6    int z = zero();
7    return 0;
8  }
```

So the above progam achieves nothing meaningful except that it demonstrates a simple function. It doesn't produce any visible output, but it does run. If we start the execution of the program from `main` we find that it creates a variable called `z` of type `int`. The variable `z` is assigned a value that is returned from the call to the function named `zero`.

The signature for the function called `zero` is given by `int zero()`. The signature provides a lot of information about the function. The `int` part of the signature tells us that `zero` returns a value of type `int`. The parentheses tells us that when we call the function `zero` we don't have to pass it any further information.

We can write a more useful function that multiplies a value by itself. It's normal to call this function `square`! If we want `square` to multiply any `int` by itself then we have to write the function so that it can be *passed* a copy of an `int`:

```
1  int square(int x) {
2    return x*x;
3  }
```

The signature for `square` is different from `zero`. The signature `int square(int x)` tells us that the function is called `square` and it returns an `int`. The interesting bit of `square` is within the parentheses. The `int x` tells us that we must pass a copy of an `int` into the function `square`. Inside the function `square` the passed in `int` will be called `x`. This allows us to multiply `x` by itself: `x*x`. The value of `x*x` is then returned from the function.

We can call `square` from `main` in the same way that we called `zero` from `main`:

```
1  int main() {
2    int s = square(7);
3    return 0;
4  }
```

4.3 Signature before use

Recall that we have to declare a variable before we use it:

```
int main() {
  int x = 0;
  x = 1;
  return 0;
}
```

as

```
int main() {
  x = 1;
  int x = 0;
  return 0;
}
```

generates a compiler error. We similarly must define a function before we use it. Hence, in the above examples I've defined either `int zero()` or `int square(int x)` before they are used in `main()`.

4.4 Multiple Parameters

A function can take more than one parameter passed to it. So for example, I could write a function that calculates the multiplication of two integers.

```
int mult(int x, int y) {
  int r = x * y;
  return r;
}

int main() {
  mult(8, 9);
  return 0;
}
```

Again, it's not a very interesting exampe, but it illustrates our point clearly. Our function is called `mult`, it returns an `int` i.e. a whole number to the calling code. The parameter list for `mult` states that it takes a copy of two `int` values. We call the first passed value `x` and the second passed value `y` within the function `mult`.

Functions can take any number of parameters, but it is unusual to pass in more than four values. If you find yourself writing functions that take seven or eight parameters you might think about restruturing your code.

We can write functions that take multiple parameters, but a function can only ever return a single value. Consider the equation for calculating the roots of a quadratic equation, a secondary-school favourite!

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

We can turn this into two functions `positive_root` and `negative_root`. I'll present the `positive_root` version here;

```
#include <cmath>

float positive_root(int a, int b, int c) {
  float x = (-b + sqrt((b * b) - (4*a*c)))/(2*a);
  return x;
}
```

In this presentation I use the `sqrt` function from the `cmath` library. The `sqrt` function returns the square root of a value. We can't write a ffunction that returns two **`float`**s (in a future chapter we can return a list of **`float`**s).

4.5 Testing Functions

Choose three values you know will work in the function, and choose three values you know will not work in the function. In many simple cases there won't be values that don't work with the function; in these cases we use our types to help us. An **`int`** ranges from $-2,147,483,648$ to $2,147,483,647$, so it's a good idea to check the minumum value, 0 and the maximum value.

Choosing the min and max values to test comes from the observation that programmers make errors in boundary cases. What happens if we call `square(2147483)`? The use of zero as a test value comes from experience. Again, programmers often make mistakes with zero values. In the case of strings we might use `""`, a string containing no characters as an example input that is allowed but we may not expect.[13]

If your function accepts an **`int`** parameter then you're allowing any possible integer input. If you accept a `string` as a parameter, then you're allowing every possible string. There exists a big list of naughty strings whcih can break some more advanced applications. The list is available at https://github.com/minimaxir/big-list-of-naughty-strings and you may wish to examine it. Do note, the list contains swear words and words of an innapropriate nature for testing purposes.

[13] If you're a maths person, the set of strings in programming form a Monoid under concatenation where the empty string is the identity value.

4.5.1 Test Harness

If you're choosing three valid test values and three invalid test values it is useful to think about how to put them into your application. Here's a straightforward pattern to follow:

1. Given a function with name `foo`.
2. Write another function called `test_foo`
    - Use `assert` to validate your test cases.
3. Run all test functions at the start of `main`.

So suppose we have our `square` function above we might also have:

```
#include <cassert>

void test_square() {
  int test_value1 = -2;
  int test_value2 = 3;
  int test_value3 = 100;

  int minimum_int = -2147483648;
  int maximum_int = 2147483647;
  int zero = 0;

  assert(4 == square(test_value1));
  assert(9 == square(test_value2));
  assert(10000 == square(test_value3));

  assert(0 == square(minimum_int));
  assert(1 == square(maximum_int));
  assert(0 == square(zero));
}

int main() {
  // Tests
  test_square();

  // Main Code
  int x = square(2);
  std::cout << x << std::endl;
}
```

4.6 `void` return

Functions return values. This makes it possible to test them. It's also possible to write a function that doesn't return a value. For example, if we wanted to write a function that only printed out some information, there would be no need to return a value. Such a function has a **`void`** return type:

```
#include <iostream>

int i_return_seven() {
  return 7;
}

void i_return_nothing() {
  std::cout << "You called a function" << std::endl;
}

int main() {
  i_return_seven();
  i_return_nothing();
}
```

The example above has two simple functions, one called `i_return_seven` which, unsurprisingly, returns the value `7`. The function `i_return_nothing` has a **`void`** return type. So it does not return any value but it has some **side-effects** such that it prints some information to the console. A function that has a **`void`** return type, does not need a **`return`** statement at the end.

4.7 What we can now do

We can now break our program down into different functions:

```
#include <iostream>

int calculate_age(int founding_year) {
  return 2018 - founding_year;
}

void print_information(string university_name, int founding_year) {
  int age = calculate_age(founding_year);
  std::cout << university_name << " was founded in " << founding_year
     << std::endl;
  std::cout << "It is " << age << " years old." << std::endl;
}
```

```
12
13 int main() {
14   int founding_year = 1968;
15   print_information("USP", founding_year);
16   return 0;
17 }
```

Using functions allows is to write smaller blocks of code. These are *easier to test*!

By breaking your code down into small functions (let's say 10 lines maximum as a rule-of-thumb) you *will* write higher-quality code.

5 Input and Output

5.1 Objective

We look at prompting users for input and processing that input.

5.2 Output

We have already seen how to output to the console. This was introduced in our "Hello, world!" program. Output to the console directs `string`s and **`int`**s to `cout`. This is useful to, for example, print current exchange rates to the screen:

```
1  #include <iostream>
2
3  void print_fjd() {
4    std::cout << "FJD$" << 1 << " is EUR" << 0.4076 << std::endl;
5  }
6
7  void print_vatu() {
8    std::cout << 1 << "VT is EUR" << 0.0076 << std::endl;
9  }
10
11 void print_sat() {
12   std::cout << "SAT$" << 1 << " is EUR" << 0.33143 << std::endl;
13 }
14
15 int main() {
16   print_fjd();
```

```
  print_vatu();
  print_sat();
  return 0;
}
```

We could also have structured this program as:

```
#include <iostream>

void print_with_pre_symbol(
     std::string symbol, float eur_value ) {
  std::cout << symbol << 1;
  std::cout << " is EUR" << eur_value << std::endl;
}

void print_with_post_symbol(
     std::string symbol , float eur_value ) {
  std::cout << 1 << symbol;
  std::cout << " is EUR" << eur_value << std::endl;
}

int main() {
  print_with_pre_symbol("FJD$", 0.4076);
  print_with_post_symbol("VT", 0.0076);
  print_with_pre_symbol("SAT$", 0.33143);
  return 0;
}
```

In fact there are an infinite number of ways of writing the same program[14]. The first version of our currency printing function is arguably easier to read. The second version is arguably easier to extend. As a rule-of-thumb I prefer ease of reading over ease of extension. The logic here is that you *will* have to read your own code it's only a possibility that you might have to extend it. In any case, if we need to extend our functions it's easy to rewrite them then.

We can also print how many Euro we might get for FJD$100:

```
#include <iostream>

int main() {
  float fjd_value = 100;
  float eur_value = fjd_value * 0.4076;
```

[14]This is one of the reasons that grumpy professors suspect plagiarism when two students hand up code with the same structure having only different variable and function names!

```
6
7     std::cout << "FJD$" << fjd_value << " is worth EUR" << eur_value <<
          std::endl;
8     return 0;
9   }
```

But every time we want to find out how many hard earned FJD we need to convert into holiday money we'd have to modify the source code. This makes the program *very* difficult to use. Surely there must be some way of taking that value in from the user?

5.3 Input

We've done console output and you've seen how it's not like the nice UIs you find on your phone. Console input is going to be similarly clunky. The only advantage is that it is easier to write the code. The major disadvantage is that this simply isn't how we write programs intended for normal users to use. It follows that you are probably not used to this mode of input and your mental model of a computer doesn't fit with console input. Let's just stick with it for the moment though.

To read in an integer from the console – which is *almost always* connected to a keyboard – we use the `istream` operator denoted >> i.e. the opposite direction of the `ostream` operator. Like `cout`, the console input exists in the `std` namespace so its full name is `std::cin`.

```
1  #include <iostream>
2
3  int main() {
4    int fjd_value;
5    std::cout << "How many FJD to convert to EUR? ";
6    std::cin >> fjd_value;
7    std::cout << "You have chosen to convert FJD$"  << fjd_value << std::
         endl;
8    return 0;
9  }
```

If we wanted to read in a `string` rather than an **`int`** then we would >> into a `string` shaped hole:

```
1  #include <iostream>
2
3  int main() {
4    std::string currency;
5    std::cout << "What is your favourite currency? ";
6    std::cin >> currency
7    std::cout << "Your favourite currency is " << currency << std::endl;
```

That seems straightforward! To read in an `int` we `std::cin >>` into an `int` variable. To do the same for `string` we `std::cin >>` into a `string` variable. What happens though if you type the character "a" as an input for an `int`?

It's useful to write yourself a suite of input functions, such as:

```
int get_int() {
  int x;
  std::cin >> x;
  return x;
}
```

Again, we apply the rule-of-thumb to write small functions. This will serve us very well in future chapters.

5.4 Testing Input

Here be dragons! Any time you ask a user to do something, they **will** do something you have not previously considered. Remember they're a creative bunch out there. We can't yet write code that tests the `get_int()` function from above. But recall another rule-of-thumb about having three values you expect to work and three values you don't expect to work. You can document these values in a comment!

```
/** Gets an 'int' from cin.
 *
 *  | Expected Input    | Result         |
 *  |-------------------|----------------|
 *  |   -1              | -1 is returned |
 *  |   0               |  0 is returned |
 *  |   1               |  1 is returned |
 *  |   'x'             |  unknown       |
 *  | "three"           |  unknown       |
 *  |  "" (empty input) |  unknown       |
 */
int get_int() {
  int x;
  std::cin >> x;
  return x;
}
```

This is good documentation for the poor programmer who has to come along later and use your code. This poor programmer is often you! Just a few weeks later after you've already forgotten the details of

how the function works. Your documentation can help yourself. I will often put a `FIXME` beside the inputs with unknown results. This encourages me to go back later and ensure that the code only ever returns valid `int`s.

We've uncovered a huge gap in our knowledge. Though we can now take in some input, we don't know how to check if the input is valid or not. It turns out that checking conditions is a useful thing to be able to do. We will study conditionals in the next chapter.

5.5 What we can now do

We can now read input values from the console.

```
1
2 int read_int() {
3   int val;
4   std::cin >> val;
5   std::cout << "You input the value " << val << std::endl;
6 }
```

6 Branching Computation

6.1 Objective

We want to introduce multiple decision points into our computations.

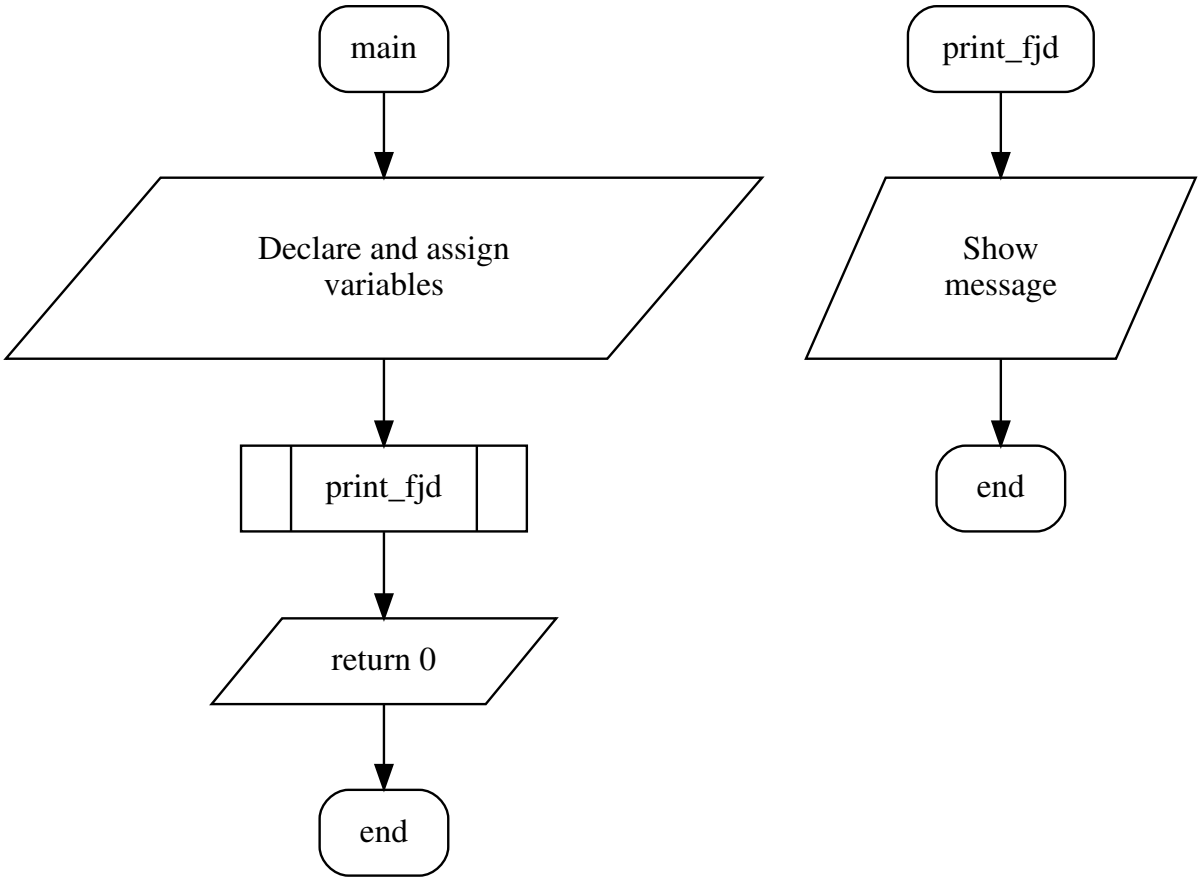
6.2 Static Structure

So far we have only written straight line code. This is code that starts, executes each line sequentially, and ends. Take a cut down version of the example from our previous chapter:

```
1 #include <iostream>
2
3 void print_fjd(int fjd_value) {
4   std::cout << "FJD$" << fjd_value << " is EUR" << (fjd_value * 0.4076)
        << std::endl;
5 }
6
7 int main() {
8   int fjd_amount = 100;
```

```
9     print_fjd(fjd_amount);
10    return 0;
11 }
```

This code can be represented as two **flowcharts**.



{@fig:straight-line-code}

In figure @fig:straight-line-code we see a representation of the `main` function. The `main` function starts and then immediately calls the `print_fjd` function passing in the value `100`. In a flowchart the fact that `print_fjd` is a function call is depicted using the rectangle with double lined walls on the left and right side. The `print_fjd` function itself has only one block of executable code, represented by the parallelogram. In my flowchart I've summarised the operation of the code in this block rather than go into the detail about `std::cout`.

Flowcharts are an excellent tool for starting with a problem statement and breaking it down into smaller functions. They provide a *static* view of the structure of software. The static view tends to ignore the values of variables and values passed to functions – it only shows the structure of the software. Sometimes we need a more *dynamic* view of what is happening taking account of the values

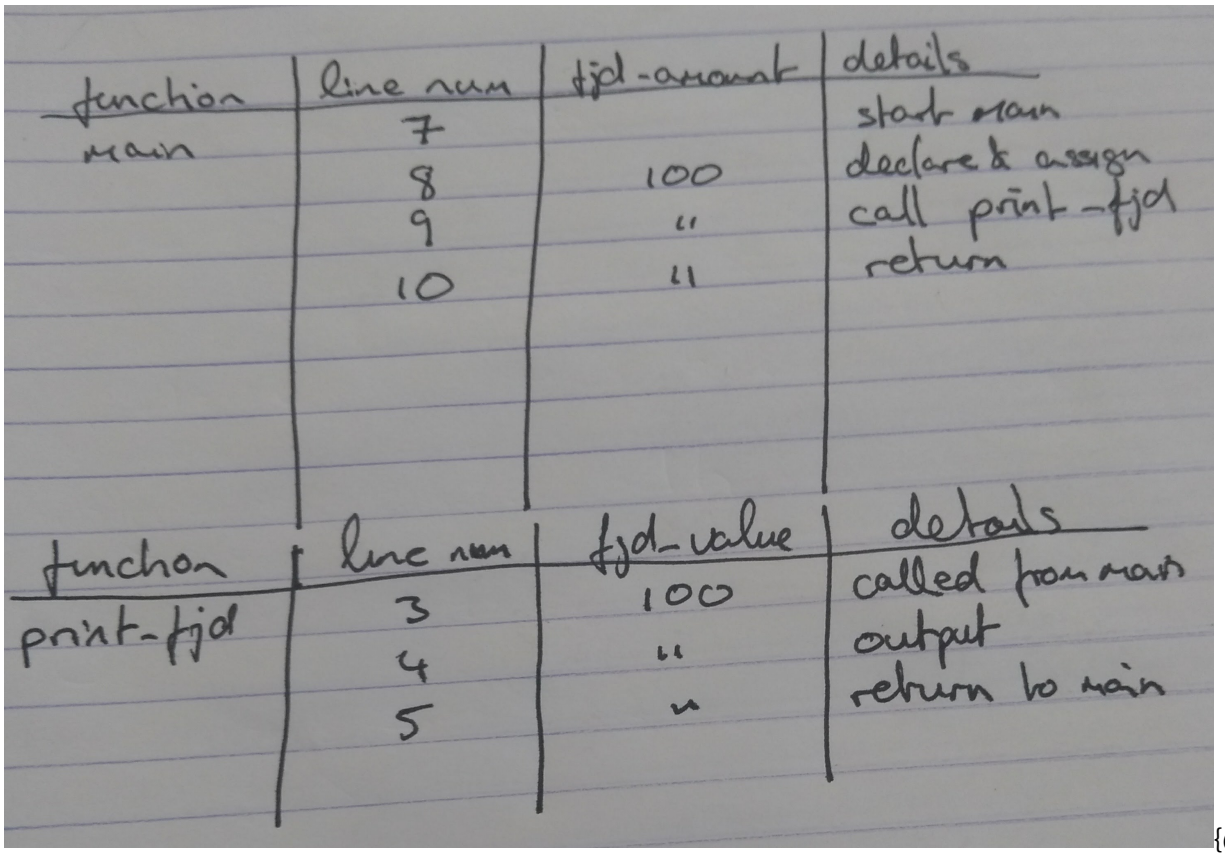
of variables. To see this we trace the execution of the software.

6.3 Tracing Execution

Tracing the execution of software is the process of stepping through each line of code in a concrete example. One useful way to trace the execution of code is to start with a table drawn on paper:

function	line number	variable name	details

For our simple program above a full trace through the program starts on line 7. I've traced throgh the execution of this program on paper, as seen in figure @fig:program-execution.



{@fig:program-execution}

In figure @fig:program-execution I started on line 7 and one column is called `fjd_amount` as a variable of that name appears in the `main` function. On the next line of chode where `fjd_amount` is declared and initialised we fill in the value of the variable as `100`. Interestingly, the next line, line 9, calls the `print_fjd` function. In order to trace this we draw a new grid to keep track of the value of variables in

that function.

This all feels a bit overkill for such simple code. The value of flowcharts and tracing will be seen when we introduce conditional execution.

6.4 Conditional execution

In order to write useful programs, we almost always need the ability to check certain conditions and change the behaviour of the program accordingly. Conditional statements give us this ability. The simplest form is the **if** statement:

```
if (x > 0) {
  cout << "x is positive" << endl;
}
```

% The expression in parentheses is called the condition. If it is true, then the statements in brackets get executed. If the condition is not true, nothing happens.

The condition can contain any of the comparison operators:

```
x == y  // x equals y
x != y  // x is not equal to y
x > y   // x is greater than y
x < y   // x is less than y
x >= y  // x is greater than or equal to y
x <= y  // x is less than or equal to y
```

Although these operations are probably familiar to you, the syntax C++ uses is a little different from mathematical symbols like $=$, $\neq$ and $\leq$. A common error is to use a single = instead of a double ==. Remember that = is the assignment operator, and == is a comparison operator. Also, there is no such thing as =< or=>‘.

The two sides of a condition operator have to be the same type. You can only compare **int** to **int** and **float** to **float**. Unfortunately, at this point you can’t compare strings in this manner! There is a way to compare strings, but we won’t get to it for a couple of chapters.

6.5 Alternative execution

A second form of conditional execution is alternative execution, in which there are two possibilities, and the condition determines which one gets executed. The syntax looks like:

```
  if (x%2 == 0) {
    cout << "x is even" << endl;
  } else {
    cout << "x is odd" << endl;
  }
```

If the remainder when `x` is divided by 2 is zero, then we know that `x` is even, and this code displays a message to that effect. If the condition is false, the second set of statements is executed. Since the condition must be true or false, exactly one of the alternatives will be executed.

As an aside, if you think you might want to check the parity (evenness or oddness) of numbers often, you might want to "wrap" this code up in a function, as follows:

```
void printParity (int x) {
  if (x%2 == 0) {
    cout << "x is even" << endl;
  } else {
    cout << "x is odd" << endl;
  }
}
```

Now you have a function named `printParity` that will display an appropriate message for any integer you care to provide. In `main` you would call this function as follows:

```
  printParity (17);
```

Always remember that when you `call` a function, you do not have to declare the types of the arguments you provide. C++ can figure out what type they are. You should resist the temptation to write things like:

```
  int number = 17;
  printParity (int number);          // WRONG!!!
```

Sometimes you want to check for a number of related conditions and choose one of several actions. One way to do this is by **chaining** a series of **`if`**s and **`else`**s:

```
  if (x > 0) {
    cout << "x is positive" << endl;
  } else if (x < 0) {
    cout << "x is negative" << endl;
  } else {
    cout << "x is zero" << endl;
```

```
  }
```

These chains can be as long as you want, although they can be difficult to read if they get out of hand. One way to make them easier to read is to use standard indentation, as demonstrated in these examples. If you keep all the statements and squiggly-braces lined up, you are less likely to make syntax errors and you can find them more quickly if you do.

In addition to chaining, you can also nest one conditional within another. We could have written the previous example as:

```
if (x == 0) {
  cout << "x is zero" << endl;
} else {
  if (x > 0) {
    cout << "x is positive" << endl;
  } else {
    cout << "x is negative" << endl;
  }
}
```

There is now an outer conditional that contains two branches. The first branch contains a simple output statement, but the second branch contains another `if` statement, which has two branches of its own. Fortunately, those two branches are both output statements, although they could have been conditional statements as well.

Notice again that indentation helps make the structure apparent, but nevertheless, nested conditionals get difficult to read very quickly. In general, it is a good idea to avoid them when you can.

On the other hand, this kind of **nested structure** is common, and we will see it again, so you better get used to it.

6.6 What we can now do

We can now branch our computations:

```
#include <iostream>

int read_int() {
  int val;
  std::cin >> val;
  return val;
}
```

```
8
9  int main() {
10   int fjd_amount;
11
12   std::cout << "Please enter an amount in FJD for conversion to EUR: ";
13   fjd_amount = read_int();
14
15   if(fjd_amount < 0) {
16     std::cout << "You have entered an invalid amount" << std::endl;
17   } else {
18     std::cout << "FJD$" << fjd_amount << " is worth EUR" << (fjd_amount
           * 0.4076) << std::endl;
19   }
20   return 0;
21 }
```

7 Iteration

7.1 Objective

In this chapter we introduce loops which allow the repition of blocks of instructions.

7.2 Multiple assignment

I haven't said much about it, but it is legal in C++ to make more than one assignment to the same variable. The effect of the second assignment is to replace the old value of the variable with a new value.

```
1   int fred = 5;
2   cout << fred;
3   fred = 7;
4   cout << fred;
```

The output of this program is `57`, because the first time we print `fred` his value is 5, and the second time his value is 7.

This kind of **multiple assignment** is the reason I described variables as a *container* for values. When you assign a value to a variable, you change the contents of the container, as shown in the figure:

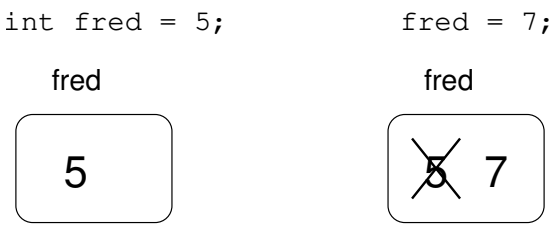


Figure 4: Assigning values to a variable

When there are multiple assignments to a variable, it is especially important to distinguish between an assignment statement and a statement of equality. Because C++ uses the = symbol for assignment, it is tempting to interpret a statement like `a = b` as a statement of equality. It is not!

First of all, equality is commutative, and assignment is not. For example, in mathematics if `a = 7` then `7 = a`. But in C++ the statement `a = 7;` is legal, and `7 = a;` is not.

Furthermore, in mathematics, a statement of equality is true for all time. If `a = b` now, then `a` will always equal `b`. In C++, an assignment statement can make two variables equal, but they don't have to stay that way!

```
1   int a = 5;
2   int b = a;      // a and b are now equal
3   a = 3;          // a and b are no longer equal
```

The third line changes the value of `a` but it does not change the value of `b`, and so they are no longer equal. In many programming languages an alternate symbol is used for assignment, such as `<-` or `:=`, in order to avoid confusion.

Although multiple assignment is frequently useful, you should use it with caution. If the values of variables are changing constantly in different parts of the program, it can make the code difficult to read and debug.

7.3 Iteration

One of the things computers are often used for is the automation of repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly.

The two features we are going to look at are the **`while`** statement and the **`for`** statement.

7.3.1 The `while` statement

Using a **`while`** statement, we can write a `countdown` program:

```
void countdown (int n) {
  while (n > 0) {
    cout << n << endl;
    n = n-1;
  }
  cout << "Blastoff!" << endl;
}
```

You can almost read a **while** statement as if it were English. What this means is, “While n is greater than zero, continue displaying the value of n and then reducing the value of n by 1. When you get to zero, output the word ‘Blastoff!’”

More formally, the flow of execution for a **while** statement is as follows:

- Evaluate the condition in parentheses, yielding **true** or **false**.
- If the condition is false, exit the **while** statement and continue execution at the next statement.
- If the condition is true, execute each of the statements between the squiggly-braces, and then go back to step 1.

This type of flow is called a **loop** because the third step loops back around to the top. Notice that if the condition is false the first time through the loop, the statements inside the loop are never executed. The statements inside the loop are called the **body** of the loop.

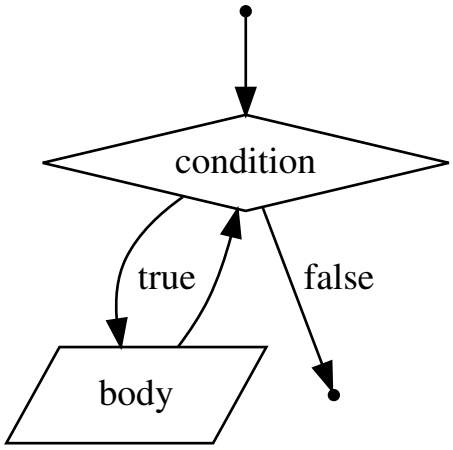


Figure 5: Structure of a loop

The body of the loop should change the value of one or more variables so that, eventually, the condition becomes false and the loop terminates. Otherwise the loop will repeat forever, which is called an infinite loop. An endless source of amusement for computer scientists is the observation that the directions on shampoo, “Lather, rinse, repeat,” are an infinite loop.

In the case of `countdown`, we can prove that the loop will terminate because we know that the value of `n` is finite, and we can see that the value of `n` gets smaller each time through the loop (each **iteration**), so eventually we have to get to zero. In other cases it is not so easy to tell:

```
void sequence (int n) {
  while (n != 1) {
    std::cout << n << std::endl;
    if (n%2 == 0) {       // n is even
      n = n / 2;
    } else {              // n is odd
      n = n*3 + 1;
    }
  }
}
```

The condition for this loop is `n != 1`, so the loop will continue until `n` is 1, which will make the condition false.

At each iteration, the program outputs the value of `n` and then checks whether it is even or odd. If it is even, the value of `n` is divided by two. If it is odd, the value is replaced by `3n+1`. For example, if the starting value (the argument passed to `sequence`) is 3, the resulting sequence is `3, 10, 5, 16, 8, 4, 2, 1`.

Since `n` sometimes increases and sometimes decreases, there is no obvious proof that `n` will ever reach 1, or that the program will terminate. For some particular values of `n`, we can prove termination. For example, if the starting value is a power of two, then the value of `n` will be even every time through the loop, until we get to 1. The previous example ends with such a sequence, starting with 16.

Particular values aside, the interesting question is whether we can prove that this program terminates for **all** values of n. So far, no one has been able to prove it **or** disprove it!

One of the things loops are good for is generating tabular data. For example, before computers were readily available, people had to calculate logarithms, sines and cosines, and other common mathematical functions by hand. To make that easier, there were books containing long tables where you could find the values of various functions. Creating these tables was slow and boring, and the result tended to be full of errors.

When computers appeared on the scene, one of the initial reactions was, “This is great! We can use the computers to generate the tables, so there will be no errors.” That turned out to be true (mostly), but shortsighted. Soon thereafter computers and calculators were so pervasive that the tables became obsolete.

Well, almost. It turns out that for some operations, computers use tables of values to get an approximate answer, and then perform computations to improve the approximation. In some cases, there have

been errors in the underlying tables, most famously in the table the original Intel Pentium used to perform floating-point division.

Although a “log table” is not as useful as it once was, it still makes a good example of iteration. The following program outputs a sequence of values in the left column and their logarithms in the right column:

```
double x = 1.0;
while (x < 10.0) {
  std::cout << x << "\t" << log(x) << std::endl;
  x = x + 1.0;
}
```

The sequence `\t` represents a tab character. These sequences can be included anywhere in a string, although in these examples the sequence is the whole string.

A tab character causes the cursor to shift to the right until it reaches one of the tab stops, which are normally every eight characters. As we will see in a minute, tabs are useful for making columns of text line up.

The output of this program is

```
1       0
2       0.693147
3       1.09861
4       1.38629
5       1.60944
6       1.79176
7       1.94591
8       2.07944
9       2.19722
```

If these values seem odd, remember that the `log` function uses base e. Since powers of two are so important in computer science, we often want to find logarithms with respect to base 2. To do that, we can use the following formula:

$$\log_2 x = \frac{log_e x}{log_e 2}$$

Changing the output statement to

```
std::cout << x << "\t" << log(x) / log(2.0) << std::endl;
```

yields

```
1  1       0
2  2       1
3  3       1.58496
4  4       2
5  5       2.32193
6  6       2.58496
7  7       2.80735
8  8       3
9  9       3.16993
```

We can see that 1, 2, 4 and 8 are powers of two, because their logarithms base 2 are round numbers. If we wanted to find the logarithms of other powers of two, we could modify the program like this:

```
double x = 1.0;
while (x < 100.0) {
  std::cout << x << "\t" << log(x) / log(2.0) << std::endl;
  x = x * 2.0;
}
```

Now instead of adding something to `x` each time through the loop, which yields an arithmetic sequence, we multiply `x` by something, yielding a *geometric sequence*. The result is:

```
1  1       0
2  2       1
3  4       2
4  8       3
5  16      4
6  32      5
7  64      6
```

Because we are using tab characters between the columns, the position of the second column does not depend on the number of digits in the first column.

Log tables may not be useful any more, but for computer scientists, knowing the powers of two is! As an exercise, modify this program so that it outputs the powers of two up to 65536 (that's 2^{16}). Print it out and memorize it.

7.4 `for` loops

The loops we have written so far have a number of elements in common. All of them start by initializing a variable; they have a test, or condition, that depends on that variable; and inside the loop they do

something to that variable, like increment it.

This type of loop is so common that there is an alternate loop statement, called **for**, that expresses it more concisely. The general syntax looks like this:

```
for (INITIALIZER; CONDITION; INCREMENTOR) {
  BODY
}
```

This statement is exactly equivalent to

```
INITIALIZER;
while (CONDITION) {
  BODY
  INCREMENTOR
}
```

except that it is more concise and, since it puts all the loop-related statements in one place, it is easier to read. For example:

```
for (int i = 0; i < 4; i++) {
  std::cout << count[i] << std::endl;
}
```

is equivalent to

```
int i = 0;
while (i < 4) {
  std::cout << count[i] << std::endl;
  i++;
}
```

We prefer for loops where the bounds are known, or when we iterate over a structure (see #lists). A **while** loop is preferred when we're waiting for some kind of event to happen.

7.5 What we can now do

We can now repeat instructions.

```
#include <iostream>

int read_int() {
  int val;
```

```
  std::cin >> val;
  return val;
}

int main() {
  int fjd_amount;

  std::cout << "Please enter an amount in FJD for conversion to EUR: ";
  fjd_amount = read_int();

  while(fjd_amount < 0) {
    std::cout << "You have entered an amount less than zero, please
        retry: " << std::endl;
    fjd_amount = read_int();
  }

  std::cout << "FJD$" << fjd_amount << " is worth EUR" << (fjd_amount *
      0.4076) << std::endl;
  return 0;
}
```

8 Lists

8.1 Objective

In this chapter we introduce one type of list. The importance of lists in computer science cannot be understated.

8.2 A `vector`

A **vector** is a list of values where each value is identified by a number (called an index). The nice thing about vectors is that they can be made up of any type of element, including basic types like **`int`**s and **`float`**s.

The `vector` type is defined in the C++ Standard Template Library (STL). In order to use it, you have to include the header file `vector`:

```
#include <vector>
```

You can create a vector the same way you create other variable types:

```
std::vector<int> count;
std::vector<float> floatVector;
```

The type that makes up the vector appears in angle brackets < and >. The first line creates a vector of integers named `count`; the second creates a vector of **`float`**s. Although these statements are legal, they are not very useful because they create vectors that have no elements (their size is zero). It is more common to specify the size of the vector in parentheses:

```
std::vector<int> count (4);
```

The syntax here is a little odd; it looks like a combination of a variable declarations and a function call. In fact, that's exactly what it is. The function we are invoking is an `vector` constructor. A **constructor** is a special function that creates new objects and initializes their instance variables. In this case, the constructor takes a single argument, which is the size of the new vector.

The following figure shows how vectors are represented in state diagrams:

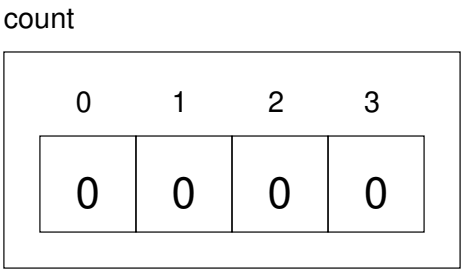


Figure 6: Visualising a `vector`

The large numbers inside the boxes are the **elements** of the vector. The small numbers outside the boxes are the indices used to identify each box. When you allocate a new vector, the elements are not initialized. They could contain any values.

There is another constructor for `vector`s that takes two parameters; the second is a "fill value," the value that will be assigned to each of the elements.

```
std::vector<int> count (4, 0);
```

This statement creates a vector of four elements and initializes all of them to zero.

8.3 Accessing elements

The `[]` operator reads and writes the elements of a vector in much the same way it accesses the characters in an `string`. As with `string`s, the indices start at zero, so `count[0]` refers to the "zeroeth" element of the vector, and `count[1]` refers to the "oneth" element. You can use the `[]` operator anywhere in an expression:

```
count[0] = 7;
count[1] = count[0] * 2;
count[2]++;
count[3] -= 60;
```

All of these are legal assignment statements. Here is the effect of this code fragment:

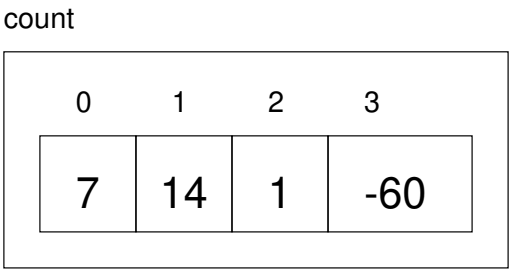


Figure 7: An example vector

Since elements of this vector are numbered from 0 to 3, there is no element with the index 4. It is a common error to go beyond the bounds of a vector, which causes a run-time error. The program outputs an error message like "Illegal vector index", and then quits.

You can use any expression as an index, as long as it has type **`int`**. One of the most common ways to index a vector is with a loop variable. For example:

```
int i = 0;
while (i < 4) {
  std::cout << count[i] << std::endl;
  i++;
}
```

This **`while`** loop counts from 0 to 4; when the loop variable `i` is 4, the condition fails and the loop terminates. Thus, the body of the loop is only executed when `i` is 0, 1, 2 and 3.

Each time through the loop we use `i` as an index into the vector, outputting the `i`th element. This type of vector traversal is very common.

8.4 Better Iteration

The C++11 standard added some syntax that allows more straightforward `for` loops to be written over vectors. An example of a C++11 for loop is the following:

```
for(auto c: count) {
  std::cout << c << std::endl;
}
```

Here we see that

- the syntax of a `for` loop has been simplified, and
- the use of the `auto` keyword to deduce the type of `c`.

The advantage of this newer notation is that we cannot overstep the boundary of the `count` vector. Suppose `count` contains 5 elements. Using the older notation we could write

```
  for (int i = 0; i < 6; i++) {
    std::cout << count[i] << std::endl;
  }
```

which is incorrect as it tries to access `count[5]` which is not an element `count`. The new notation protects us programmers from making such common errors (yes really! you'd be surprised how often these off-by-one errors are made).

8.5 Copying vectors

There is one more constructor for `vector`s, which is called a copy constructor because it takes one `vector` as an argument and creates a new vector that is the same size, with the same elements.

```
  vector<int> copy (count);
```

Although this syntax is legal, it is almost never used for `vector`s because there is a better alternative:

```
  vector<int> copy = count;
```

The = operator works on `vector`s in pretty much the way you would expect.

8.6 Vector size

There are a few functions you can invoke on an `vector`. One of them is very useful, though: `size()`. Not surprisingly, it returns the size of the vector (the number of elements).

It is a good idea to use this value as the upper bound of a loop, rather than a constant. That way, if the size of the vector changes, you won't have to go through the program changing all the loops; they will work correctly for any size vector.

```
for (int i = 0; i < count.size(); i++) {
  cout << count[i] << endl;
}
```

Though you should use the {#better itreration} example above to iterate over a vector.

The last time the body of the loop gets executed, the value of `i` is `count.size()- 1`, which is the index of the last element. When `i` is equal to `count.size()`, the condition fails and the body is not executed, which is a good thing, since it would cause a run-time error.

8.7 Vector functions

The best feature of a vector is its resizeability A vector, once declared, can be resized from anywhere within the program. Suppose we have a situation where we input numbers from the user and store them in a vector till he inputs `-1`, and then display them. In such a case, we do not know the size of the vector beforehand. So we need wish add new values to the end of a vector as the user inputs them. We can use then vector function `push\_back()` for that purpose.

```
#include<iostream>
#include<vector>

using namespace std;
int main()
{
  vector<int> values;
  int c,i,len;
  cin >> c;

  while(c != -1) {
    values.push_back(c);
    cin >> c;
  }

  for(auto i: values) {
    cout << i << endl;
  }
}
```

8.8 What we can now do

We can now create a list of values and perform operations on the list:

```
#include <iostream>
#include <vector>

int main() {
  std::vector<int> fjd_amounts;
  std::vector<int> eur_amounts;

  fjd_amounts.push_back(0);
  fjd_amounts.push_back(1);
  fjd_amounts.push_back(2);
  fjd_amounts.push_back(4);
  fjd_amounts.push_back(8);

  for(int fjd: fjd_amounts) {
    eur_amounts.push_back(fjd * 0.4076);
  }

  std::cout << eur_amounts << std::endl;
  return 0;
}
```

9 Random numbers

9.1 Objective

For many practical applications we need to use some randomness. We look at generating random numbers in C++.

9.2 Determinsm

Most computer programs do the same thing every time they are executed, so they are said to be **deterministic**. Usually, determinism is a good thing, since we expect the same calculation to yield the same result. For some applications, though, we would like the computer to be unpredictable. Games are an obvious example.

Making a program truly **nondeterministic** turns out to be not so easy, but there are ways to make it at least seem nondeterministic. One of them is to generate {pseudorandom} numbers and use them to determine the outcome of the program. Pseudorandom numbers are not truly random in the mathematical sense, but for our purposes, they will do.

C++ provides a function called `random` that generates pseudorandom numbers. It is declared in the header file `cstdlib`, which contains a variety of "standard library" functions, hence the name.

The return value from `random` is an integer between `0` and `RAND_MAX`, where `RAND_MAX` is a large number (about 2 billion on my computer) also defined in the header file. Each time you call `random` you get a different randomly-generated number. To see a sample, run this loop:

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main ()
{
  for (int i = 0; i < 4; i++) {
    int x = random ();
    cout << x << endl;
  }
  return 0;
}
```

On my machine I got the following output:

```
1804289383
846930886
1681692777
1714636915
```

You will probably get something similar, but different, on yours.

Of course, we don't always want to work with gigantic integers. More often we want to generate integers between 0 and some upper bound. A simple way to do that is with the modulus operator. For example:

```
  int x = random ();
  int y = x % upperBound;
```

Since `y` is the remainder when `x` is divided by `upperBound`, the only possible values for `y` are between 0 and `upperBound - 1`, including both end points. Keep in mind, though, that `y` will never be equal to `upperBound`.

It is also frequently useful to generate random floating-point values. A common way to do that is by dividing by `RAND_MAX`. For example:

```
int x = random ();
double y = double(x) / RAND_MAX;
```

This code sets `y` to a random value between 0.0 and 1.0, including both end points. As an exercise, you might want to think about how to generate a random floating-point value in a given range; for example, between 100.0 and 200.0.

9.3 Statistics

The numbers generated by `random` are supposed to be distributed uniformly. That means that each value in the range should be equally likely. If we count the number of times each value appears, it should be roughly the same for all values, provided that we generate a large number of values.

In the next few sections, we will write programs that generate a sequence of random numbers and check whether this property holds true.

9.3.1 Vector of random numbers

The first step is to generate a large number of random values and store them in a vector. By "large number," of course, I mean 20. It's always a good idea to start with a manageable number, to help with debugging, and then increase it later.

The following function takes a single argument, the size of the vector. It allocates a new vector of **`int`**s, and fills it with random values between 0 and `upperBound-1`.

```
vector<int> randomVector (int n, int upperBound) {
  vector<int> vec (n);
  for (int i = 0; i<vec.size(); i++) {
    vec[i] = random () % upperBound;
  }
  return vec;
}
```

The return type is `vector<int>`, which means that this function returns a vector of integers. To test this function, it is convenient to have a function that outputs the contents of a vector.

```
void printVector (const vector<int>& vec) {
  for (int i: vec) {
```

```
    cout << i << " ";
  }
}
```

Notice that it is legal to pass `vector`s by reference. In fact it is quite common, since it makes it unnecessary to copy the vector. Since `printVector` does not modify the vector, we declare the parameter **`const`**.

The following code generates a vector and outputs it:

```
int numValues = 20;
int upperBound = 10;
vector<int> vector = randomVector (numValues, upperBound);
printVector (vector);
```

On my machine the output is

```
3 6 7 5 3 5 6 2 9 1 2 7 0 9 3 6 0 6 2 6
```

which is pretty random-looking. Your results may differ.

If these numbers are really random, we expect each digit to appear the same number of times—twice each. In fact, the number 6 appears five times, and the numbers 4 and 8 never appear at all.

Do these results mean the values are not really uniform? It's hard to tell. With so few values, the chances are slim that we would get exactly what we expect. But as the number of values increases, the outcome should be more predictable.

To test this theory, we'll write some programs that count the number of times each value appears, and then see what happens when we increase `numValues`.

9.3.2 Counting

A good approach to problems like this is to think of simple functions that are easy to write, and that might turn out to be useful. Then you can combine them into a solution. This approach is sometimes called **bottom-up design**. Of course, it is not easy to know ahead of time which functions are likely to be useful, but as you gain experience you will have a better idea.

Also, it is not always obvious what sort of things are easy to write, but a good approach is to look for subproblems that fit a pattern you have seen before.

Back in Section {#loopcount} we looked at a loop that traversed a string and counted the number of times a given letter appeared. You can think of this program as an example of a pattern called "traverse and count." The elements of this pattern are:

- A set or container that can be traversed, like a string or a vector.
- A test that you can apply to each element in the container.
- A counter that keeps track of how many elements pass the test.

In this case, I have a function in mind called howMany that counts the number of elements in a vector that equal a given value. The parameters are the vector and the integer value we are looking for. The return value is the number of times the value appears.

```
int howMany (const vector<int>& vec, int value) {
  int count = 0;
  for (int i: vec) {
    if (i == value) {
      count++;
    }
  }
  return count;
}
```

9.3.3 Checking the other values

howMany only counts the occurrences of a particular value, and we are interested in seeing how many times each value appears. We can solve that problem with a loop:

```
  int numValues = 20;
  int upperBound = 10;
  vector<int> vector = randomVector (numValues, upperBound);

  cout << "value\thowMany";

  for (int i = 0; i<upperBound; i++) {
    cout << i << '\t' << howMany (vector, i) << endl;
  }
```

Notice that it is legal to declare a variable inside a **for** statement. This syntax is sometimes convenient, but you should be aware that a variable declared inside a loop only exists inside the loop. If you try to refer to i later, you will get a compiler error.

This code uses the loop variable as an argument to howMany, in order to check each value between 0 and 9, in order. The result is:

```
value   howMany
0       2
```

```
3  1        1
4  2        3
5  3        3
6  4        0
7  5        2
8  6        5
9  7        2
10 8        0
11 9        2
```

Again, it is hard to tell if the digits are really appearing equally often. If we increase `numValues` to 100,000 we get the following:

```
1  value    howMany
2  0        10130
3  1        10072
4  2        9990
5  3        9842
6  4        10174
7  5        9930
8  6        10059
9  7        9954
10 8        9891
11 9        9958
```

In each case, the number of appearances is within about 1% of the expected value (10,000), so we conclude that the random numbers are probably uniform.

9.3.4 A histogram

It is often useful to take the data from the previous tables and store them for later access, rather than just print them. What we need is a way to store 10 integers. We could create 10 integer variables with names like `howManyOnes`, `howManyTwos`, etc. But that would require a lot of typing, and it would be a real pain later if we decided to change the range of values.

A better solution is to use a vector with size 10. That way we can create all ten storage locations at once and we can access them using indices, rather than ten different names. Here's how:

```
1   int numValues = 100000;
2   int upperBound = 10;
3   vector<int> vector = randomVector (numValues, upperBound);
4   vector<int> histogram (upperBound);
```

```
5
6   for (int i = 0; i<upperBound; i++) {
7     int count = howMany (vector, i);
8     histogram[i] = count;
9   }
```

I called the vector `histogram` because that's a statistical term for a vector of numbers that counts the number of appearances of a range of values.

The tricky thing here is that I am using the loop variable in two different ways. First, it is an argument to {howMany}, specifying which value I am interested in. Second, it is an index into the histogram, specifying which location I should store the result in.

9.3.5 A single-pass solution

Although this code works, it is not as efficient as it could be. Every time it calls `howMany`, it traverses the entire vector. In this example we have to traverse the vector ten times!

It would be better to make a single pass through the vector. For each value in the vector we could find the corresponding counter and increment it. In other words, we can use the value from the vector as an index into the histogram. Here's what that looks like:

```
1   vector<int> histogram (upperBound, 0);
2
3   for (int i = 0; i<numValues; i++) {
4     int index = vector[i];
5     histogram[index]++;
6   }
```

The first line initializes the elements of the histogram to zeroes. That way, when we use the increment operator (++) inside the loop, we know we are starting from zero. Forgetting to initialize counters is a common error.

As an exercise, encapsulate this code in a function called {histogram} that takes a vector and the range of values in the vector (in this case 0 through 10), and that returns a histogram of the values in the vector.

9.4 Random seeds

If you have run the code in this chapter a few times, you might have noticed that you are getting the same *random* values every time. That's not very random!

One of the properties of pseudorandom number generators is that if they start from the same place they will generate the same sequence of values. The starting place is called a **seed**; by default, C++ uses the same seed every time you run the program.

While you are debugging, it is often helpful to see the same sequence over and over. That way, when you make a change to the program you can compare the output before and after the change.

If you want to choose a different seed for the random number generator, you can use the `srand` function. It takes a single argument, which is an integer between 0 and `RAND\_MAX`.

9.5 What we can do now

We can now create random numbers:

```
#include <iostream>
#include <cstdlib>

int dice_roll() {
  int number = (random() % 6) + 1;
  return number;
}

int main() {
  std::cout << "You roll a " << dice_roll() << std::endl;
  return 0;
}
```

10 Provenance

This book is based on "How to think like a computer scientist" which was originally developed in 1999 by Allen B. Downey. Allen is one

of those amazingly productive people who shares under free-content licences. Because of his generosity, others contributed to the book over many years. These include Narendra Sisodiya, Tirtha P. Chatterjee and Aidan Delaney. This book is a derivative work of “How to think...” and was written to focus on motivation and examples that were of significance to students in South Pacific countries.

10.1 Licence

As this book is a derivative work of Doweny’s original, it must retain the same licencing terms. The terms of this licence, the GNU General Public Licence, were developed to ensure that you - the reader - have the same distribution rights as me - the author. With this book you may

- make copies and share them with your friends,
- change the format of the book - if you received it as a web page you’re allowed to transform it into an ebook,
- make changes to the book once you provide those changes, under the same rights, to anyone who receives the book from you,
- you are allowed to profit from selling copies of the book, but again, you have to pass on the same rights to your customers.

The text of the licence is quoted below

10.2 GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

10.2.1 Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free

software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Lesser General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

10.2.2 TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it,

under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt

otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.
6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other

circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and “any later version”, you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Soft-

ware Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

10.2.3 NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS