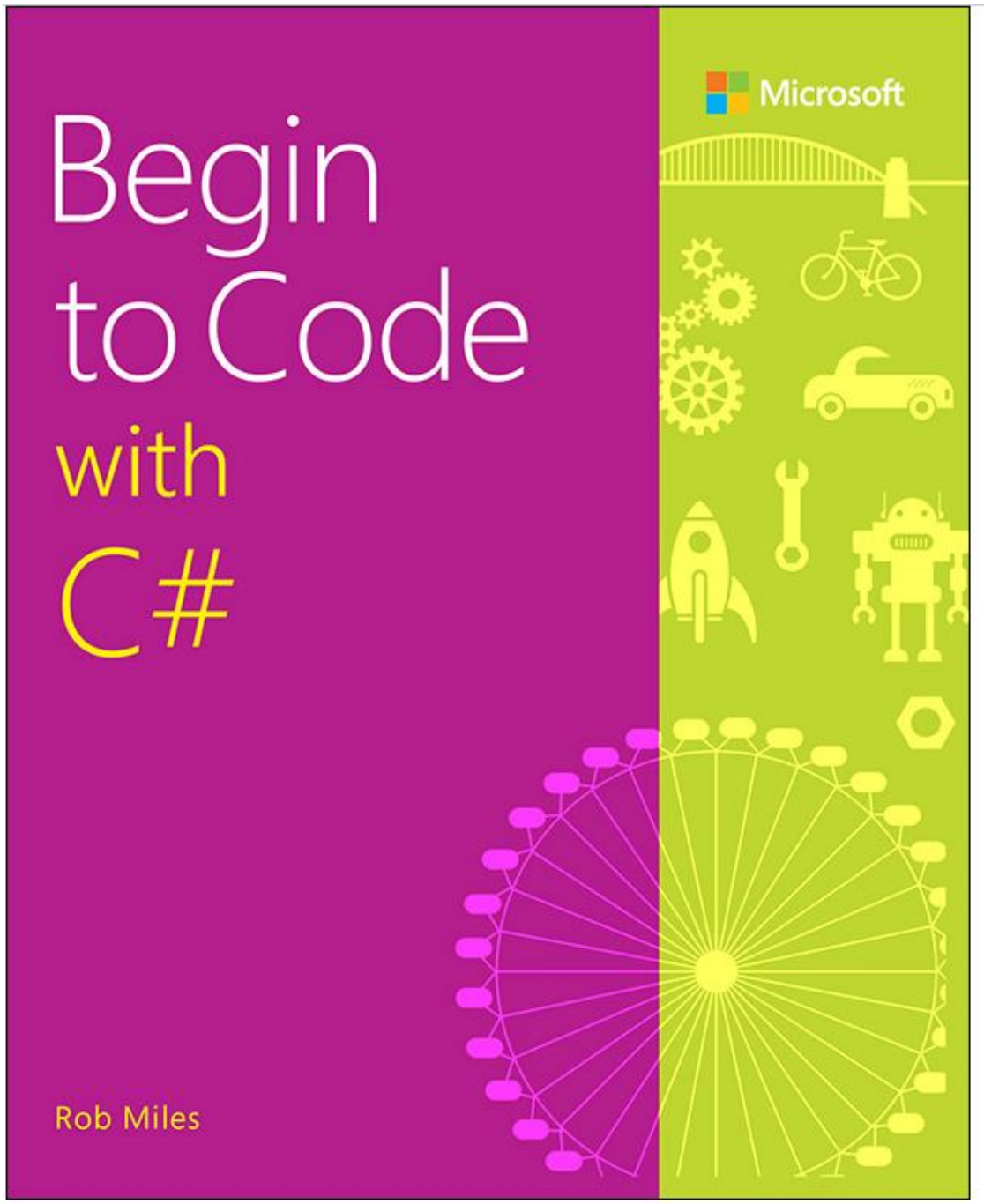
C# Programming Yellow Book

Rob Miles

“Cheese” Edition 8.2 November 2016

Shameless Plug:

The C# Yellow book is a great way to learn how to program by just reading. But if you want to do some coding while you are learning I can strongly recommend “Begin to Code with C#”.



This is a beautifully printed, all colour, programming text which will get you writing Universal Windows Applications using a specially written library of easy to use supporting functions, called Snaps. It is published by Microsoft Press and available from all the usual outlets, including Amazon.

ISBN-13: 978-1509301157
ISBN-10: 1509301151

Email: rob@robmiles.com Blog: www.robmiles.com Twitter @RobMiles

If you find a mistake in the text please report the error to foundamistake@robmiles.com and I will take a look.

Edition 8.1
Thursday, 10 November 2016
The cheese was delicious.

Introduction

Welcome

Welcome to the Wonderful World of Rob Miles™. This is a world of bad jokes, puns, and programming. In this book I'm going to give you a smattering of the C# programming language. If you have programmed before I'd be grateful if you'd still read the text. It is worth it just for the jokes and you may actually learn something.

If you have not programmed before, do not worry. Programming is not rocket science it is, well, programming. The bad news about learning to program is that you get hit with a lot of ideas and concepts at around the same time when you start, and this can be confusing. The keys to learning programming are:

Practice – do a lot of programming and force yourself to think about things from a problem solving point of view

Study – look at programs written by other people. You can learn a lot from studying code which other folk have created. Figuring out how somebody else did the job is a great starting point for your solution. And remember that in many cases there is no *best* solution, just ones which are better in a particular context, i.e. the fastest, the smallest, the easiest to use etc.

Persistence – writing programs is hard work. And you have to work hard at it. The principle reason why most folks don't make it as programmers is that they give up. Not because they are stupid. However, don't get too persistent. If you haven't solved a programming problem in 30 minutes you should call time out and seek help. Or at least walk away from the problem and come back to it. Staying up all night trying to sort out a problem is not a good plan. It just makes you all irritable in the morning. We will cover what to do when it all goes wrong later in section 5.9.

Reading the notes

These notes are written to be read straight through, and then referred to afterwards. They contain a number of *Programming Points.* These are based on real programming experience and are to be taken seriously. There are also bits written in a *Posh Font***.** These are really important, should be learnt by heart and probably set to music.

If you have any comments on how the notes can be made even better (although I of course consider this highly unlikely) then feel free to get in touch.

Above all, enjoy programming.

Rob Miles

Getting a copy of the notes and code samples

Printed copies of these notes are made freely available to Computer Science students at the University of Hull. The website for the book is at http://www.csharpcourse.com where you can also find the Powerpoint slide decks and laboratory exercises for a C# course based on this text. You can also find the C# code samples here too.

If you want make a single printed copy of the text for private use, that's fine by me, but if you to print larger numbers I'd appreciate you getting in touch first.

You can obtain a Kindle ebook version from Amazon.

1 Computers and Programs

In this chapter you are going to find out what a computer is and get an understanding of the way that a computer program tells the computer what to do. You will discover what you should do when starting to write a program, to ensure that you achieve a "happy ending" for you and your customer. Finally, you will take a look at programming in general and the C# language in particular.

1.1 Computers

Before we consider programming, we are going to consider computers. This is an important thing to do, because it sets the context in which all the issues of programming itself are placed.

1.1.1 An Introduction to Computers

Qn: Why does a bee hum?
Ans: Because it doesn't know the words!

One way of describing a computer is as an electric box which hums. This, while technically correct, can lead to significant amounts of confusion, particularly amongst those who then try to program a fridge. A better way is to describe it as:

A device which processes information according to instructions it has been given.

This general definition rules out fridges but is not exhaustive. However, for our purposes it will do. The instructions you give to the computer are often called a program. The business of using a computer is often called programming. This is **not** what most people do with computers. Most people do not write programs. They use programs written by other people. We must therefore make a distinction between users and programmers. A user has a job which he or she finds easier to do on a computer running the appropriate program. A programmer has a masochistic desire to tinker with the innards of the machine. One of the golden rules is that you never write your own program if there is already one available, i.e. a keen desire to process words with a computer should not result in you writing a word processor!

However, because you will often want to do things with computers which have not been done before, and further because there are people willing to pay you to do it, we are going to learn how to program as well as use a computer.

Before we can look at the fun packed business of programming though it is worth looking at some computer terminology:

1.1.2 Hardware and Software

If you ever buy a computer, you are not just getting a box which hums. The box, to be useful, must also have sufficient built-in intelligence to understand simple commands to do things. At this point we must draw a distinction between the software of a computer system and the hardware.

Hardware is the physical side of the system. Essentially if you can kick it, and it stops working when immersed in a bucket of water, it is hardware. Hardware is the impressive pile of lights and switches in the corner that the salesman sold you.

Software is what makes the machine tick. If a computer has a soul, it keeps it in its software. Software uses the physical ability of the hardware, which can run programs, to do something useful. It is called software because it has no physical existence and it

is comparatively easy to change. Software is the voice which says "Computer Running" in a Star Trek film.

Windows 10 is an example of an operating system. It gives computer programs a platform on which they can execute.

All computers are sold with some software. Without it they would just be a novel and highly expensive heating system. The software which comes with a computer is often called its Operating System. The Operating System makes the machine usable. It looks after all the information held on the computer and provides lots of commands to allow you to manage things. It also lets you run programs, ones you have written and ones from other people. You will have to learn to talk to an operating system so that you can create your C# programs and get them to go.

1.1.3 Data and Information

People use the words data and information interchangeably. They seem to think that one means the other. I regard data and information as two different things:

Data is the collection of ons and offs which computers store and manipulate.

Information is the interpretation of the data by people to mean something. Strictly speaking computers process data, humans work on information. As an example, the computer could hold the following bit pattern in memory somewhere:

```
11111111 11111111 11111111 00000000
```

You could regard this as meaning:

"you are 256 pounds overdrawn at the bank"

or

"you are 256 feet below the surface of the ground"

or

"eight of the thirty two light switches are off"

The transition from data to information is usually made when the human reads the output. So why am I being so pedantic? (pedantic means being fussy about something being exactly right) Because it is vital to remember that a computer does not "know" what the data it is processing actually means. As far as the computer is concerned data is just patterns of bits, it is the user who gives meaning to these patterns. Remember this when you get a bank statement which says that you have `£8,388,608` in your account!

Data Processing

Computers are data processors. Information is fed into them; they do something with it, and then generate further information. A computer program tells the computer what to do with the information coming in. A computer works on data in the same way that a sausage machine works on meat, something is put in one end, some processing is performed, and something comes out of the other end:

This makes a computer a very good "mistake amplifier", as well as a useful thing to blame.....

A program is unaware of the data it is processing in the same way that a sausage machine is unaware of what meat is. Put a bicycle into a sausage machine and it will try to make sausages out of it. Put invalid data into a computer and it will do equally useless things. It is only us people who actually give meaning to the data (see above). As far as the computer is concerned data is just stuff coming in which has to be manipulated in some way.

A computer program is just a sequence of instructions which tell a computer what to do with the data coming in and what form the data sent out will have.

Note that the data processing side of computers, which you might think is entirely reading and writing numbers, is much more than that, examples of typical data processing applications are:

`Digital Watch`: A micro-computer in your watch is taking pulses from a crystal and requests from buttons, processing this data and producing a display which tells you the time.

`Car`: A micro-computer in the engine is taking information from sensors telling it the current engine speed, road speed, oxygen content of the air, setting of the accelerator etc and producing voltages out which control the setting of the carburettor, timing of the spark etc, to optimise the performance of the engine.

`CD Player`: A computer is taking a signal from the disk and converting it into the sound that you want to hear. At the same time, it is keeping the laser head precisely positioned and also monitoring all the buttons in case you want to select another part of the disk.

`Games Console`: A computer is taking instructions from the controllers and using them to manage the artificial world that it is creating for the person playing the game.

Note that some of these data processing applications are merely applying technology to existing devices to improve the way they work. However, the CD player and games console could not be made to work without built-in data processing ability.

Most reasonably complex devices contain data processing components to optimise their performance and some exist only because we can build in intelligence. It is into this world that we, as software writers are moving. It is important to think of the business of data processing as much more than working out the company payroll, reading in numbers and printing out results. These are the traditional uses of computers.

Note that this "raises the stakes" in that the consequences of software failing could be very damaging.

As software engineers it is inevitable that a great deal of our time will be spent fitting data processing components into other devices to drive them. You will not press a switch to make something work, you will press a switch to tell a computer to make it work. These embedded systems will make computer users of everybody, and we will have to make sure that they are not even aware that there is a computer in there!

You should also remember that seemingly innocuous programs can have life threatening possibilities. For example, a doctor may use a spread sheet to calculate doses of drugs for patients. In this case a defect in the program could result in illness or even death (note that I don't think that doctors actually do this – but you never know..)

Programmer's Point: At the bottom there is always hardware

It is important that you remember your programs are actually executed by a piece of hardware which has physical limitations. You must make sure that the code you write will actually fit in the target machine and operate at a reasonable speed. The power and capacity of modern computers makes this less of an issue than in the past, but you should still be aware of these aspects. I will mention them when appropriate.

1.2 Programs and Programming

I tell people I am a "Software Engineer".

Programming is a black art. It is the kind of thing that you grudgingly admit to doing at night with the blinds drawn and nobody watching. Tell people that you program computers and you will get one of the following responses:

1. A blank stare.
2. "That's interesting", followed by a long description of the double glazing that they have just had fitted.
3. Asked to solve every computer problem that they have ever had, and ever will have.
4. A look which indicates that you can't be a very good one as they all drive Ferraris and tap into the Bank of England at will.

Programming is defined by most people as earning huge sums of money doing something which nobody can understand.

Programming is defined by me as deriving and expressing a solution to a given problem in a form which a computer system can understand and execute.

One or two things fall out of this definition:

- You need to be able to solve the problem yourself before you can write a program to do it.
- The computer has to be made to understand what you are trying to tell it to do.

1.2.1 What is a Programmer?

And remember just how much plumbers earn….

I like to think of a programmer as a bit like a plumber! A plumber will arrive at a job with a big bag of tools and spare parts. Having looked at it for a while, tut tutting, he will open his bag and produce various tools and parts, fit them all together and solve your problem. Programming is just like this. You are given a problem to solve. You have at your disposal a big bag of tricks, in this case a programming language. You look at the problem for a while and work out how to solve it and then fit the bits of the language together to solve the problem you have got. The art of programming is knowing which bits you need to take out of your bag of tricks to solve each part of the problem.

From Problem to Program

*Programming is **not** about mathematics, it is about organization and structure.*

The art of taking a problem and breaking it down into a set of instructions you can give a computer is the interesting part of programming. Unfortunately it is also the most difficult part of programming as well. If you think that learning to program is simply a matter of learning a programming language you are very wrong. In fact if you think that programming is simply a matter of coming up with a program which solves a problem you are equally wrong!

There are many things you must consider when writing a program; not all of them are directly related to the problem in hand. I am going to start on the basis that you are writing your programs for a customer. He or she has a problem and would like you to write a program to solve it. We shall assume that the customer knows even less about computers than we do!

Initially we are not even going to talk about the programming language, type of computer or anything like that; we are simply going to make sure that we know what the customer wants.

Solving the Wrong Problem

Coming up with a perfect solution to a problem the customer has not got is something which happens surprisingly often in the real world. Many software projects have failed because the problem that they solved was the wrong one. The developers of the system quite simply did not find out what was required, but instead created what they thought was required. The customers assumed that, since the developers had stopped asking them questions, the right thing was being built, and only at the final handover was the awful truth revealed. It is therefore very important that a programmer holds off making something until they know exactly what is required.

The worst thing you can say to a customer is "I can do that". Instead you should think "Is that what the customer wants?"

This is a kind of self-discipline. Programmers pride themselves on their ability to come up with solutions, so as soon as they are given a problem they immediately start thinking of ways to solve it, this is almost a reflex action. What you should do is think "Do I really understand what the problem is?" Before you solve a problem you should make sure that you have a watertight definition of what the problem is, which both you and the customer agree on.

In the real world such a definition is sometimes called a Functional Design Specification or FDS. This tells you exactly what the customer wants. Both you and the

customer sign it, and the bottom line is that if you provide a system which behaves according to the design specification the customer must pay you. Once you have got your design specification, then you can think about ways of solving the problem. You might think that this is not necessary if you are writing a program for yourself; there is no customer to satisfy. **This is not true**. Writing some form of specification forces you to think about your problem at a very detailed level. It also forces you to think about what your system is not going to do and sets the expectations of the customer right at the start.

Programmer's Point: The specification must always be there

I have written many programs for money. I would **never** write a program without getting a solid specification first. This is true even (or perhaps especially) if I do a job for a friend.

Modern development techniques put the customer right at the heart of the development, and involve them in the design process. These work on the basis that it is very hard (and actually not that useful) to get a definitive specification at the start of a project. You as a developer don't really know much about the customer's business and they don't know the limitations and possibilities of the technology. With this in mind it is a good idea to make a series of versions of the solution and discuss each with the customer before moving on to the next one. This is called prototyping.

1.2.2 A Simple Problem

Consider the scenario; you are sitting in your favourite chair in the pub contemplating the universe when you are interrupted in your reverie by a friend of yours who sells double glazing for a living. He knows you are a programmer of sorts and would like your help in solving a problem which he has:

He has just started making his own window units and is looking for a program which will do the costing of the materials for him. He wants to just enter the dimensions of the window and then get a print out of the cost to make the window, in terms of the amount of wood and glass required.

"This looks like a nice little earner" you think, and once you have agreed to a price you start work. The first thing you need to do is find out exactly what the customer wants you to do...

Specifying the Problem

When considering how to write the specification of a system there are three important things:

- What information flows into the system.
- What flows out of the system.
- What the system does with the information.

There are lots of ways of representing this information in the form of diagrams, for now we will stick with written text when specifying each of the stages:

Information going in

In the case of our immortal double glazing problem we can describe the information as:

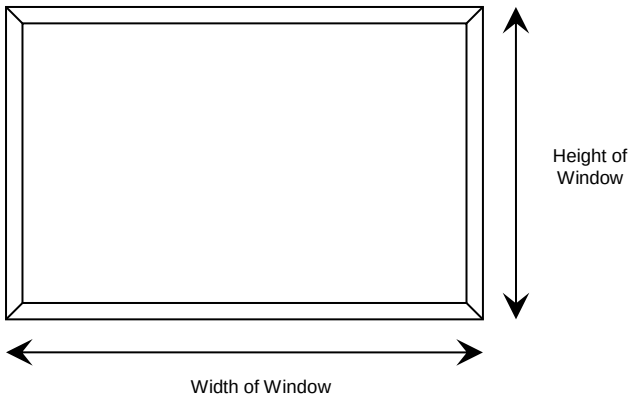
- The width of a window.
- The height of the window.

Information coming out

The information that our customer wants to see is:

- the area of glass required for the window
- the length of wood required to build a frame.

You can see what we need if you take a look at the diagram below:



The area of the glass is the width multiplied by the height. To make the frame we will need two pieces of wood the width of the window, and two pieces of wood the height of the window.

Programmer's Point: metadata is important

Information about information is called *metadata*. The word meta in this situation implies a "stepping back" from the problem to allow it to be considered in a broader context. In the case of our window program the metadata will tell us more about the values that are being used and produced, specifically the units in which the information is expressed and the valid range of values that the information may have. For any quantity that you represent in a program that you write you must have at least this level of metadata .

What the program actually does

The program can derive the two values according to the following equations:

```
glass area = width of window * height of window
wood length = (width of window + height of window) * 2
```

Putting in more detail

We now have a fairly good understanding of what our program is going to do for us. Being sensible and far thinking people we do not stop here, we now have to worry about how our program will decide when the information coming in is actually valid.

This must be done in conjunction with the customer, he or she must understand that if information is given which fits within the range specified, your program will regard the data as valid and act accordingly.

In the case of the above we could therefore expand the definition of data coming in as:

- The width of the window, in metres and being a value between 0.5 Metres and 3.5 metres inclusive.
- The height of the window, in metres and being a value between 0.5 metres and 2.0 metres inclusive.

Note that we have also added units to our description, this is very important - perhaps our customer buys wood from a supplier who sells by the foot, in which case our output description should read:

- The area of glass required for the window, in square metres. Remember that we are selling double glazing, so two panes will be required.
- The length of wood required for the frame, given in feet using the conversion factor of 3.25 feet per metre.

*Note that both you and the customer **must** understand the document!*

Having written this all up in a form that both you and the customer can understand, we must then both sign the completed specification, and work can commence.

Proving it Works

In a real world you would now create a test which will allow you to prove that the program works, you could for example say:

"If I give the above program the inputs 2 metres high and 1 metre wide the program should tell me I need 4 square metres of glass and 19.5 feet of wood."

The test procedure which is designed for a proper project should test out all possible states within the program, including the all-important error conditions. In a large system the person writing the program may have to create a test harness which is fitted around the program and will allow it to be tested. Both the customer and the supplier should agree on the number and type of the tests to be performed and then sign a document describing these.

Testing is a very important part of program development. There is even one development technique where you write the tests *before* you write the actual program that does the job. This is actually a good idea, and one we will explore later. In terms of code production, you can expect to write as much code to test your solution as is in the solution itself. Remember this when you are working out how much work is involved in a particular job.

Getting Paid

Better yet, set up a phased payment system so that you get some money as the system is developed.

At this point the supplier knows that if a system is created which will pass all the tests the customer will have no option but to pay for the work! Note also that because the design and test procedures have been frozen, there is no ambiguity which can lead to the customer requesting changes to the work although of course this can still happen!

The good news for the developer is that if changes are requested these can be viewed in the context of additional work, for which they can expect to be paid.

Customer Involvement

Note also in a "proper" system the customer will expect to be consulted as to how the program will interact with the user, sometimes even down to the colour of the letters on the display! Remember that one of the most dangerous things that a programmer can think is "This is what he wants"! The precise interaction with the user - what the program does when an error is encountered, how the information is presented etc., is something which the customer is guaranteed to have strong opinions about. Ideally all this information should be put into the specification, which should include layouts of the screens and details of which keys should be pressed at each stage. Quite often prototypes will be used to get an idea of how the program should look and feel.

Fact: If you expect to derive the specification as the project goes on either you will fail to do the job, or you will end up performing five times the work!

If this seems that you are getting the customer to help you write the program then you are exactly right! Your customer may have expected you to take the description of the problem and go into your back room - to emerge later with the perfect solution to the problem. This is not going to happen. What will happen is that you will come up with something which is about 60% right. The customer will tell you which bits look OK and which bits need to be changed. You then go back into your back room, muttering under your breath, and emerge with another system to be approved. Again, Rob's law says that 60% of the wrong 40% will now be OK, so you accept changes for the last little bit and again retreat to your keyboard....

The customer thinks that this is great, reminiscent of a posh tailor who produces the perfect fit after numerous alterations. All the customer does is look at something, suggests changes and then wait for the next version to find something wrong with. They will get a bit upset when the delivery deadline goes by without a finished product appearing but they can always cheer themselves up again by suing you.

Actually, we have come full circle here, because I did mention earlier that prototyping is a good way to build a system when you are not clear on the initial specification. However, if you are going to use prototypes it is a good thing to plan for this from the

start rather than ending up doing extra work because your initial understanding of the problem was wrong.

Fact: More implementations fail because of inadequate specification than for any other reason!

If your insistence on a cast iron specification forces the customer to think about exactly what the system is supposed to do and how it will work, all to the better. The customer may well say "But I am paying you to be the computer expert, I know nothing about these machines". This is no excuse. Explain the benefits of "Right First Time" technology and if that doesn't work produce a revolver and force the issue!

Again, if I could underline in red I would: All the above apply if you are writing the program for yourself. You are your own worst customer!

You may think that I am labouring a point here; the kind of simple systems we are going to create as we learn to program are going to be so trivial that the above techniques are far too long winded. You are wrong. One very good reason for doing this kind of thing is that it gets most of the program written for you - often with the help of the customer. When we start with our double glazing program we now know that we have to:

```
read in the width
verify the value
read in the height
verify the value
calculate width times height times 2 and print it
calculate ( width + height ) * 2 * 3.25 and print it
```

The programming portion of the job is now simply converting the above description into a language which can be used in a computer.....

Programmer's Point: Good programmers are good communicators

The art of talking to a customer and finding out what he/she wants is just that, an art. If you want to call yourself a proper programmer you will have to learn how to do this. One of the first things you must do is break down the idea of "I am writing a program for you" and replace it with "We are creating a solution to a problem". You do not work for your customers, you work with them. This is very important, particularly when you might have to do things like trade with the customer on features or price.

1.3 Programming Languages

Once we know what the program should do (specification), and how we are going to determine whether it has worked or not (test) we now need to express our program in a form that the computer can work with.

You might ask the question "Why do we need programming languages, why can we not use something like English?" There are two answers to this one:

1. Computers are too stupid to understand English.
2. English would make a lousy programming language.

Please note that this does not imply that tape worms would make good programmers!

To take the first point. We cannot make very clever computers at the moment. Computers are made clever by putting software into them, and there are limits to the size of program that we can create and the speed at which it can talk to us. At the moment, by using the most advanced software and hardware, we can make computers which are about as clever as a tape worm. Tape worms do not speak very good English; therefore we cannot make a computer which can understand English. The best we can do is to get a computer to make sense of a very limited language which we use to tell it what to do.

Time Flies like an Arrow. Fruit Flies like a Banana!

To take the second point. English as a language is packed full of ambiguities. It is very hard to express something in an unambiguous way using English. If you do not believe me, ask any lawyer!

Programming languages get around both of these problems. They are simple enough to be made sense of by computer programs and they reduce ambiguity.

Programmer's Point: The language is not that important

There are a great many programming languages around, during your career you will have to learn more than just one. C# is a great language to start programming in, but do not think that it is the only language you will ever learn.

1.4 C#

There are literally hundreds of programming languages around; you will need to know at least 3!

We are going to learn a language called C# (pronounced C sharp). If you ever make the mistake of calling the language C hash you will show your ignorance straight away! C# is a very flexible and powerful programming language with an interesting history. It was developed by Microsoft Corporation for a variety of reasons, some technical, some political and others marketing.

C# bears a strong resemblance to the C++ and Java programming languages, having borrowed (or improved) features provided by these languages. The origins of both Java and C++ can be traced back to a language called C, which is a highly dangerous and entertaining language which was invented in the early 1970s. C is famous as the language the UNIX operating system was written in, and was specially designed for this.

1.4.1 Dangerous C

I referred to C as a *dangerous* language. So what do I mean by that? Consider the chain saw. If I, Rob Miles, want to use a chain saw I will hire one from a shop. As I am not an experienced chain saw user I would expect it to come with lots of built in safety features such as guards and automatic cut outs. These will make me much safer with the thing but will probably limit the usefulness of the tool, i.e. because of all the safety stuff I might not be able to cut down certain kinds of tree. If I was a real lumberjack I would go out and buy a professional chain saw which has no safety features whatsoever but can be used to cut down most anything. If I make a mistake with the professional tool I could quite easily lose my leg, something the amateur machine would not let happen.

In programming terms what this means is that C lacks some safety features provided by other programming languages. This makes the language much more flexible.

However, if I do something stupid C will not stop me, so I have a much greater chance of crashing the computer with a C program than I do with a safer language.

Programmer's Point: Computers are always stupid

I reckon that you should always work on the basis that any computer will tolerate no errors on your part and anything that you do which is stupid will always cause a disaster! This concentrates the mind wonderfully.

1.4.2 Safe C#

The C# language attempts to get the best of both worlds in this respect. A C# program can contain *managed* or *unmanaged* parts. The managed code is fussed over by the system which runs it. This makes sure that it is hard (but probably not impossible) to crash your computer running managed code. However, all this fussing comes at a price, causing your programs to run more slowly.

To get the maximum possible performance, and enable direct access to parts of the underlying computer system, you can mark your programs as unmanaged. An unmanaged program goes faster, but if it crashes it is capable of taking the computer

with it. Switching to unmanaged mode is analogous to removing the guard from your new chainsaw because it gets in the way.

C# is a great language to start learning with as the managed parts will make it easier for you to understand what has happened when your programs go wrong.

1.4.3 C# and Objects

The C# language is *object oriented*. Objects are an organisational mechanism which let you break your program down into sensible chunks, each of which is in charge of part of the overall system. Object Oriented Design makes large projects much easier to design, test and extend. It also lets you create programs which can have a high degree of reliability and stability.

I am very keen on object oriented programming, but I am not going to tell you much about it just yet. This is not because I don't know much about it (honest) but because I believe that there are some very fundamental programming issues which need to be addressed before we make use of objects in our programs.

The use of objects is as much about design as programming, and we have to know how to program before we can design larger systems.

1.4.4 Making C# Run

You actually write the program using some form of text editor - which may be part of the compiling and running system.

C# is a *compiled* programming language. The computer cannot understand the language directly, so a program called a *compiler* converts the C# text into the low level instructions which are much simpler. These low level instructions are in turn converted into the actual commands to drive the hardware which runs your program.

We will look in more detail at this aspect of how C# programs work a little later, for now the thing to remember is that you need to show your wonderful C# program to the compiler before you get to actually run it.

A compiler is a very large program which knows how to decide if your program is legal. The first thing it does is check for errors in the way that you have used the language itself. Only if no errors are found by the compiler will it produce any output.

The compiler will also flag up *warnings* which occur when it notices that you have done something which is not technically illegal, but may indicate that you have made a mistake somewhere. An example of a warning situation is where you create something but don't use it for anything. The compiler would tell you about this, in case you had forgotten to add a bit of your program.

The C# language is supplied with a whole bunch of other stuff (to use a technical term) which lets C# programs do things like read text from the keyboard, print on the screen, set up network connections and the like. These extra features are available to your C# program but you must explicitly ask for them. They are then located automatically when your program runs. Later on we will look at how you can break a program of your own down into a number of different chunks (perhaps so several different programmers can work on it).

1.4.5 Creating C# Programs

Microsoft has made a tool called Visual Studio, which is a great place to write programs. It comprises the compiler, along with an integrated editor, and debugger. It is provided in a number of versions with different feature sets. There is a free version, called Visual Studio Express edition, which is a great place to get started. Another free resource is the Microsoft .NET Framework. This provides a bunch of command line tools, i.e. things that you type to the command prompt, which can be used to compile and run C# programs. How you create and run your programs is up to you.