Chapter 1

Introduction

This is about basic computer programming. It starts from nothing and covers a little more than one semester of a Programming-I class. It uses C# in the Unity3D game engine, but it isn't about them any more than it needs to be.

A game engine seems like a funny choice. It's more difficult to get started than the traditional "scroll lines against a black screen". But the 3D environment lets us write some nice examples with moving blocks and colors. It also provides a nice way to watch our variables as we run.

C# isn't an especially good first language, but it's fine. The basics are the same as every other language – much of it is exactly the same. And it's not the worst thing to learn a language in common use.

1.1 Notes

Since the programs need to run, there will be certain amount of nit-picky `C#` rules. That's fine – every language has them, so we may as well get a feel now. I'll try to be clear about the parts everyone uses vs. the odd bits that everyone does a little differently.

About half of the examples are moving blocks around and other game tricks. But only because they make good examples. This absolutely isn't going over 3D game skills. It even leaves out some easy 3D tricks which aren't good for examples.

Whenever you teach anything, you lie a lot. In first grade math you learn you can't have 3 minus 5 since you don't know negative numbers yet. Then second grade says you can't divide 9 by 4, since you haven't learned fractions yet.

I'm going to do the same thing. Instead of telling you the Ultimate Rule for something, I'll start with a good-enough-for-now rule, then fill it out later. At

the end that seems to work better.

Many of the examples that do something useful are the same way. You'd probably do that exact thing in a different way. But you won't know that yet, and the exact trick is still good, except as part of something larger and more complicated that would make a terrible example.

Examples that do nothing useful are also good – you have to pay close attention to every step of the rules. I'll try to label these as teaching/useless/silly so you don't waste time trying to figure out what they do, which is nothing.

1.2 Computer Languages, Books and Teaching

This is just for fun, if you wonder about all the various books on programming. These are just my opinions.

1.2.1 Computer languages

This whole thing is based on all computer programming sharing the same ideas, no matter which language you use. Learn the ideas and you easily figure out any language. Here's some background why I claim that:

Computers don't understand *any* programming languages. They only run a set of very simple instructions. The way programming languages work is by translating your code into real computer instructions.

We can write programs directly in computer code (called assembly code.) It's a huge pain, and takes forever, and we invented programming languages so we didn't have to. But all programming languages are shortcuts for the same simple computer commands.

Another thing is that programming languages get revised, and borrow from each other. We started writing them in the 1950's, tried a lot of things, and over the decades figured out the best concepts, rules and even the symbols that seemed the best.

As new things were invented, not that many, we did the same thing. Older languages often added them once all the kinks were worked out. Modern programming languages were created by taking things from the common pool of "programming ideas."

This makes it sound like there should be only one programming language. There are lots of them because there are lots of trade-offs:

Languages good at quickly writing small programs aren't good for huge ones. Some nice features make the program run slower; sometimes a lot slower. Adding too many shortcuts and features makes it hard to read. Some languages

can run with very little memory by cutting to a bare minimum of features. Some great features are error-prone for new users – "safe" languages cut those.

Then, some people just like different sets of optional features, or like commands spelled a certain way.

1.2.2 Books and Teaching

There are maybe three kinds of Computer programming books: basic computer programming, language reference manuals, and Project books.

Reference manuals are great once you know how to code. Obviously they don't tell you anything about how to program, and only have examples for the weird special case stuff. The Unity3D online Scripting Reference is like this. It's incomprehensible if you aren't a programmer who knows game stuff, too. But if you are, it's just fine.

The problem is you can't always tell something is a reference manual. The microsoft online C# docs have some introductory examples – just enough to fool you, but not nearly enough to teach you programming. The site's main purpose is to be a reference manual.

Many books about a particular language are written for programmers wanting to learn it. They have a teaching style, but only about new features in that one particular language. The book will mention all of the basics, but leave out what every programmer already knows. These books are just great, but can be especially frustrating if you don't realize they're not written for beginners.

Project books are things like making a single 3D game, learning the programming as you go. Those are fun, give you a nice overview, and might inspire you to learn more. They seem better since you finish with a free game. But I don't think they work.

A problem is there's too much to cover. For a game you need to learn the engine, 3D models, art, sound, particle systems ... and coding. Any of those could fill a book. Another problem is the best examples for each coding topic come from all over. And a real game is too complicated - you get lost in the explanation of all the things to make even a simple program to move the player.

If you think you only learn-by-doing, don't sell yourself short. Everyone learns with some practice, some reading, and some explanation. In a textbook it's fine to jump straight to the examples, try to run a few things, and then go back to read the explanation.

Books about learning basic programming have their own difficulties – you need to write what buyers think they want. When someone goes to a bookstore to learn C#, they might see "Learning computer programming", "Learning C#" and "Learning C# with 3.0 features". The first one teaches you how to use loops. The second shows the 4 different kinds of C# loops. The third also

shows the new special-case loops over arrays. Each one is more of a reference manual, and less about how to write programs.

Everyone knows that the manual for the XK-20 nail-gun won't let you make a birdhouse. You need to know basic carpentry for where the nails go and how many. But not many people know that about programming.

Object Oriented Programming books have a similar problem. "Learning Object Oriented C#" sounds pretty good. But OOP is an advanced trick – a way to organize large programs. In an intro book all it does is make the examples longer and harder to read.

For books used by schools, you'd think it would be different, but there's also pressure to teach OOP and the latest hot language. It's also fine for it to be more of a reference manual – the part about how to program will be done in class. And the book selection process for an intro class can be a bit messy. Often you're using whatever book it had before, and only teaching it for a year.

Upper-level Com Sci textbooks tend to be pretty good. The instructor knows the area well, cares about it, and is very interested in choosing the right book.

1.3 Pedagogy

This is a standard part of programming books where you explain to other teachers why you arranged things the way you did.

When I was teaching I gradually realized that the most important thing was convincing students that programming is understandable. They really, really want to memorize things. My job was to convince them they could understand the basic idea of `if`'s, solve lots of problems with them, and the rest was just details.

All I had to do was ruthlessly strip away everything else. Not quite, but I have to keep reminding myself that I'm not writing a manual and they're not going to be reading production code immediately after taking my class. We don't need special cases, alternate syntaxes, or shortcuts.

I learned to avoid lists. Seeing all of the Reserved Words is fun, but I'm trying to show them they *don't* need to memorize things. The same with the traditional 2-page list of every data type. I love `long long int`'s as much as the next guy – but why do we show them a big list of things we're not going to use?

I like to put some of the optional stuff in its own chapter. Not so much to cover it, but to show how a working language fleshes things out. But even then, keep it short. One line with a foreach loop, mentioning how it's a shortcut for an index loop, is plenty. That's enough for an interested student to look it up.

A guideline for me is that pages and space equals importance. Switch statements are individually so long and have so many rules that they take 3 pages. If you spent 2 pages on boolean logic, switch statements automatically seem more important. If most examples also include one bonus rule, the net effect is that

there sure are a lot of rules to remember.

I've noticed that students confuse actual rules with style rules and defensive-coding rules. We think we're giving them 2 or 3 rules in 3 different categories, but it blurs into 8 general purpose rules. Pretty soon they think wrong camel-casing caused a logic error. The simplest way to fix this is to only give real rules.

I think style can be shown by example. The later, instead of "the body of an `if` statement is indented" you have the non-rule rule "the body didn't need to be indented, but it looks nice that way."

Showing proper programming is just too horrendously distracting. This is where every example checks for valid input and uses try-catch blocks.

The thing is, I understand the impulse to hammer away at style from the start. I started college during the structured programming movement. General contempt for style, or readable programs at all, was a glaring problem. You didn't tell true genius programmers how to do their jobs. Job security meant writing code that only you could figure out. And so on. I took the class that was a reaction to that, where 25% of the homework grade was for style.

But that old culture is mostly gone. Python's geek culture is proud of "pythonic" code, which is all about style. Modern languages are more readable, editors push style on you. If anything, style is now probably over-emphasized.

My favorite tricks:

- int, double, string are the best group of types. 3 is the perfect number. String is exotic; int vs. double shows how a type is what we say it is. string/int/double casting is enough to get the idea, and sort of interesting.
- Use strings to teach indexing. People are so much better at looking through letters 0 through 4 of `"zebra"` than the list {`5,8,4,12,7`}. Use strings to teach everything you can about index loops and off-ends. Then move onto formal arrays.

  `C++` is a little better for this, since `w[i]='x'` is legal.
- Arrays of structs, structs with array fields, arrays of nested structs and so on are great. They're good for going left-to-right through each dot and seeing what the current type allows. They're an opportunity for all sorts of interesting nested loops. They're an excuse to write more small classes with a member function or two. Then they're just a nice example of making a data structure.

Covering functions early is great since all of your examples afterwards can be functions. If you want to do something with a name and a number you normally have to declare them and wave your hands about them somehow getting filled. You write things like "now suppose w is `cow` and n is 8".

With functions it's shorter and more clear to write `void story(string name, int count)` at the top. That literally means that they somehow get

filled. `story("cow",8)` is the best way to say "pretend they are ...". Even better, once you have returns you can stop saying "and now `n` is the final result." Just write `return n;`.

Functions are the best way to set up and bracket code snippets.

I'd normally have functions as the first thing. In a class I can talk students through how function don't accomplish anything, but they'll be very useful and will be on the test. In a book it seems important quickly do something useful, so I moved `if`'s as the first thing.

Loops are so far back because of Unity's Update loop. It lets us write loop-like things and use loop thinking very early on.

Reference Types make teaching pointers and the heap a big pain. Here's how I covered pointers in `C++`: first pointers to normal vars, like `int* p1=&n;`. Having everything explicit is really an advantage: `int*` holds an address and `&n` creates an address. There aren't any secret steps, and at this point students understand about types having to match.

Then, once we've used pointers for a bit, we can move on to `new` and the heap. This is the best way to learn pointers, which includes knowing how to use Reference Types.

Learning Reference types first is a mess. In C++ terms you're learning structs, pointers, and the heap, all at once. The way we first hand-wave `Cat c=new Cat();` just makes it worst. The best way I could think of was to avoid classes at first. Cover structs. That gets practice with member variables and dots. Unity examples use lots of `Vector3` and `Color` structs, so that helps. Then, much later, cover pointers and `new` together when you introduce classes.

All of the old textbooks had triangle-printing nested loops. I thought they were self-indulgent and show-offy. Then I realized they were great nested-loop examples, with indexing practice and immediate visual feedback. if you're off-by-one, the picture is off-by-one. In that section, the checkerboard example is original. The rest are from every "Learn BASIC" book written in the 80's.

It's a shame Unity's debug window breaks up the output, but you can still see the star patterns.

I think I inherited my first `C++` class using `Vector` instead of arrays. Or maybe I was smart enough to do it myself. Either way, an array container class is the way to go. Easier to use but still all the fun of indexing. Much later, cover arrays. Students now know indexing so you're just covering the fixed-size rule and the work-arounds for it. Here I've got the the array-wrapper `List` class first, with arrays much later.

Sadly, I don't think the Unity3D API uses a single List – it's all arrays. That makes sense for a game engine. But it's not helping building a case that you should primarily use Lists over arrays.

1.4 Legal-ish