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.

A game engine seems like a funny choice to explain regular computer programming, but Unity does a nice job letting you watch the program run; and moving 3D colored blocks around makes some nice examples.

C# isn't especially easy or pretty, but it's in common use. And it's a typical language – many of the commands are exactly the same same as in C++ or Java or Python ....

The draw-back is that Unity3D is a little trickier to set-up than the traditional "scroll words on a black screen." But it's a free download, and is popular so there are lots and lots of places to look for help. And this is partly written for people who are already using it.

Things like this used to be common, but you don't see them as much. An Intro to Computer Programming book covers the basic rules and ideas, examples, ways to combine them, and common programming tricks. The only point is to have them make sense so you're comfortable using them to build whatever you need.

They seem funny since they skip about 90% of all the little rules most languages have. But once you know the basics, you can look the rest up.

1.1 Notes

Since the programs really run, there are a certain amount of nit-picky `C#` rules. But that's fine. Every language has it's own set, and they're all different but the same.

About half of the examples are about stuff you can do in a 3D game engine

(Unity3D), but that's only because they make good examples. This skips a lot of Unity3D stuff.

C# has parts that are weird and overly complicated, but so does almost every other language. There are some things I just wave my hands at and say they just go there. That's normal.

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 problem is you can't always tell something is a reference manual.

The Unity3D on-Line Scripting Reference is one of these - it lists all the add-on Unity commands. It's well written, if you know `C#`. And like all reference manuals, the examples are only for the oddball stuff, not the things every programmer knows.

The on-line C# docs on the microsoft site are *mostly* a reference manual. They have some introductory examples, but only a tiny fraction of what you'd need. Each new section explains the concept a little, but is mostly is a list of every possible option (more on this later.)

A semi-popular type of book is "Learning language X for programmers." These are good since you only need to learn programming once, but most programmers will need to learn many new languages. They explain the new things pretty well, with lots of real examples. But, obviously, they don't cover the things every programmer knows. For the basics, they're also just a reference manual.

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 about it, a book that has you make a game is the same as showing you a completed game and then explaining all the parts.

If you think you only learn-by-doing, don't sell yourself short. Everyone learns with some practice and some reading the rules. Some people prefer to start with one or the other. After messing around and getting a feel for things, try reading the rules and they might make more sense than you thought they would.

Actual books about learning basic programming have difficulties. Suppose you want to learn C#. You go to a bookstore and see three books: "Learning computer programming," "Learning C#" and "Learning C# with 3.0 features and Linq." Obviously the last book seems better. If you're an author, that's your title.

But programming languages aren't like updated versions of of an App – they're more like a tools in a garage where you buy more every year. After three years you may have replaced some of the originals with better versions, but not many. Most of the newest tools are oddball special-purpose stuff. That last `C#` book essentially has to skim through screwdrivers and hammers, spending lots of time on that non-ferrous low-torque belt sander that works in outer-space.

Object Oriented Programming books have the same problem. There's pressure to include that, but it's a way to organize large programs. In an intro book it just makes the examples much longer and harder to read.

For books used by schools, you'd think it would be different, but there are trends in educational books, too. There was pressure to include Object Oriented early. Java's not a great language for teaching, but there was pressure to switch to it (schools are starting to switch away, after years of Java.) Having an on-line component, with a password in the book, is popular. Maybe pre-made power-point slides. Instructors don't aways pick their book – there's lead time for the book store so sometimes it was the out-going instructor. And reading them all is a lot of work.

In my time teaching I saw a lot of books. I couldn't figure out why they all had a quote in front of every chapter, with large margin notes on the history of computers. Sure some, but all of them?

The upper-level Com Sci textbooks are still pretty good. It was just the freshman computer programming ones that got weirder and weirder.

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.

My biggest problem when I was teaching was convincing students that programming was basically understandable. There are only a few real concepts, obvious once you know what problems they solve; many simple algorithms are essentially the way humans would do it; the syntax is mostly consistent and as simple as possible to get the job done.

It's like the big moment in math is when you realize FOIL isn't a rule – it's a reminder. If you know distribution, `(a+b)(c+d)` is `(a+b)c+(a+b)d` which become first, inner, outer, last pairs.

Everything is going to be some memorization and some cleverness; but students early on decide if a subject is mainly one or the other. For programming, you want to them to decide it's mostly clever use of logical rules.

I've found that pages and class time equals importance. If a textbook has 4 pages, for reference, explaining every type, that's a 4-page vote for "programming is mostly about memorization." The 3+ pages it takes to explain switch statements cancels out the previous 3 about using nested `if`'s.

It's really easy to write a page about the actual topic, then wander off into several pages about some obscure special-case rule, since it's *a rule*. But I'm not going to be the first person in the history of the world to combine how to program with a good reference manual. I cut a lot. Sometimes it takes a lot longer than it should have.

The other thing I've noticed is that students confuse different types of rules. 2 coding tricks and 5 suggestions about style mush into 7 rules about style.

I started college during the structured programming movement; when everyone coded like they were thinking in assembly, which they were, and writing an unreadable program meant you were a genius. In my first programming class, using Pascal, 25% of the homework grade was for style – using enough comments, meaningful variable names and not having functions longer than a printed page.

I understand that. They needed to counter the cowboy coding culture, and early languages needed a style imposed to be readable. But now I feel students are likely to follow style cues on their own and adapt "make it easy to read" as an obvious goal.

So that's what I don't want to do. As far as what to do, people learn by using things in different situations, in combination with other things, and by forgetting and relearning them a few times. There's no rule about how `if`'s work inside of a loop, but it seems different and it's something you want to cover.

And, I hate chess analogies, but books about chess have zero actual chess

rules, and it's fine. All the common ways we like to combine things to get stuff done in a program should in a coding book.

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]` can also be an L-value.
- Arrays of structs, structs with array fields, arrays of nested structs and so on are great. They're good for going left-to-right and doing what the type allows. They're full of examples 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 has a non-obvious advantage for later code examples. Suppose you want an example printing a name and a number. You have to set up two variables and wave your hands about pretending they get filled in. Maybe you explain how this is part of a larger program.

With functions that's just `void story(string name, int count)`. Clearly a larger program will use this, with whatever values it needs. Instead of writing "now suppose name="phil" and count=4", you can write `story("phil",4);`. It's shorter, clearer, and you're getting free practice with parameter use.

Even better, once you have returns you can stop saying "and now `n` is the final result." Or the even worse `print("largest is "+n);`. Writing `return n;` is shorter and clearly indicates you're done.

Writing snippet code examples as functions is just great. It's not even teaching bad habits. If your students automatically start a coding problem by writing the function heading for it, like you did in class, that's good.

Plus, sure, cover functions early so you can split it up. Use functions with parameters for while, then later learn about return values (I have those chapters side-by-side here, but I'm not happy about it.)

I moved `if`'s as the first real topic since `if`'s do things. In a class, where I'm there every day facing possible wrath, I can start you on so-far useless functions with the promise it will pay off in a few weeks. In a book someone would just pick up, I'd better get to interesting examples quickly.

Loops are so far back because of Unity's Update loop. It lets us write loop-like things and use loop thinking. And I wanted to abuse the loop chapter to write loop-using functions.

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.

Pointer math is an optional cool example: `*(p+2)` is further showing that `p` is an address, so clearly +2 is address walking, then we dereference. Comparing to `*p+2` is a great way to emphasize how `*` is an operator.

Then, once we've used pointers for a bit, we can move on to `new` and the heap.

We used to hold a traditional graduating Senior lunch, to get their thoughts on the program. By that time some of my former `C++` freshmen were there. Four years ago they wanted Java as the first language, for the jobs potential. By graduation they'd finally taken a Java course, had jobs lined up coding Java. But they told us we needed to keep `C++` as the first language. It taught them pointers.

Reference types are a mix of member variables, pointers, and the heap, all at once. They're a halfway step to using pointers, but they're not halfway to understanding them. For example, the first thing you learn is that `Cat c=new Cat();` declares a single Cat variable, which functions can change just because. Then you learn `Cat c2;` is a reference, usually to some pre-made, not by you, Cat. You have to un-learn that stuff later: `Cat c` is a local pointer to a heap Cat, and the rest follows. This is what my former students were talking about.

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.

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.

It took me a while to realize `C#`'s List class is almost the same as `Vector` (but why oh why is there no pop-back? Why is `O(n) L.Remove(0)` so easy to use?) I'm a little embarrassed I didn't see it sooner. I didn't realize ArrayList was the old Java-syle list of Objects, while List is the `C++` template-style, even

though I knew `C#` has a kitchen-sink approach to features.

Anyway, this version finally covers the array-wrapper `List` class first, then 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 build a case that arrays are a rarely-needed type of List.

1.4 Legal-ish

Copyright Owen Reynolds 2016, 2018. Permission is given to print, copy or otherwise distribute however you think might be helpful to you.

Unity3D is a trademark of Unity Technologies, used here without permission.