FOREWORD

This book is the product of me (Jim) trying something new in our introductory Computer Science course at the University of Manitoba. Through the process of trying to see how I can teach programming using Processing, I started to panic as none of my examples worked, and ended up writing copious notes to help myself work through the problems. The first draft of those notes were used in the first term we taught programming using Processing. Following, there has been a major overhaul for content and exercises, to the version you see now. Finally, I am ready to toss this online, in hopes it may be useful to others.

Acknowledgments

I really need to thank the University of Manitoba Computer Science Department for this book. While I wrote the text here, many of the strategies, techniques, and explanation methods are drawn from the ethos kicking around our department on how we teach programming. This is as much my own teaching style as what I have learned from all the great teachers around here. In particular, I need to thank Andrea Bunt, who was brave enough to try a Processing assignment with me, John Anderson who gave the red light to do a whole course this way, and John Bate, who helped immensely with the material and many of the exercises: in fact, some are straight-up taken from John's labs in prior terms.

History

2016-may-31 – Version 1.0 live!

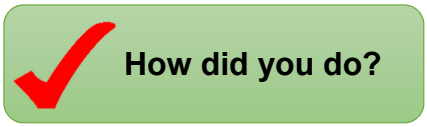
(page intentionally left blank)

UNIT 1. INTRODUCTION

Each unit will start with the three sections below: **Summary**, **Learning Objectives**, and **How to Proceed**. The **Summary** will just provide an overview of the activities and topics covered, and the **How to Proceed** will be a simple description of how to approach the material. The **Learning Objectives** are particularly important as they describe the skills you will have by the end; once finishing the unit, return to this section to be sure that you feel comfortable with all the items listed.

1.1 Text Book Annotations and Aids

This course has the following aids to help you out:

	[1]Learning Objectives at the beginning of each Unit. Check yourself against these to be sure you are learning the required skills.
	[2]Important sections that you should pay special attention to.
	[3]Academic sections for interest only, which are not testable.
	[4]Sections with exercises to check your understanding. These sections are your primary way to develop your skills and practice – work hard at them.
	At the end of a unit, go back and review the learning objectives

[1] https://commons.wikimedia.org/wiki/File:Target.svg
[2] https://commons.wikimedia.org/wiki/File:Important-1.svg
[3] https://commons.wikimedia.org/wiki/File:Mortarboard.svg
[4] https://commons.wikimedia.org/wiki/File:Red_check.svg

Summary

In this unit you will assess whether or not this course is for you, and learn briefly about the many things that you can do with programming.

Learning Objectives

After finishing this unit, you will be able to…

- Explain to a layperson what computer programming is
- Calculate how many operations a computer can perform each second based on its specifications

How to Proceed

- Read the unit content.
- Have a web browser open so that you can do the searches recommended.
- Do the sets of exercises in the **Check your Understanding** section.
- Re-check the **Learning Objectives** once done.

1.2 Introduction

Programming is awesome! That's all you need to know. Seriously, though, why bother with learning programming? Look around you, computers are everywhere. The obvious ones are your laptop, tablet, and phone – and game system if you're into that. But don't forget about your car (if it's newer than 1980), and even your home appliances (if they're newer than mine, at least). Learning computer programming gives you some insight into how and why these things work. And these days, there are great new tools (toys!) that let you build and test things yourself. Do an internet search for Arduino, Raspberry Pi, or Phidgets, and you'll find large communities of home-cooked cool gadgets by do-it-yourselfers with limited programming background. Don't forget immersive VR, which is emerging as the next cool thing – these are computers, too! And you can use programing to be creative and make your own software with these. So, with basic programming, you'll have the foundation to start hacking away at your own computer programs (games! business tools! art projects!), or even get into hobbyist hardware hacking and inventing. Who knows, maybe you'll go into Computer Science and keep learning more.

1.3 Do I belong in this course?

Probably (we need to get our numbers up!). More practically speaking, this is a true introduction course. No programming background or experience is required or expected. You may see people around you who have taken prior courses, but they are ahead of the curve and the course is not aimed at them. So what background do you need? Just basic computer skills – typing, using new software, using the web, etc. Installing new software is helpful, too.

If you do have extensive programming experience, you can still learn a great deal in this course. Many prior students have commented on the additional depth they have learned and dusty corners cleaned out through this (and, ahem, the easy A+?) – You may be surprised at what you will learn. However, there are ways of challenging the course and skipping ahead. Talk to your undergraduate advisor about this.

1.4 What is Programming?

So what is this course about? It is about problem solving, it is about solving puzzles with a given set of tools and constraints. Computers are really capable, but they are also really stupid and can't do much on their own. You need to learn specific tools to wrangle the computer into doing what you want it to do. This ends up being a nice set of puzzles, and as such, many people who enjoy puzzles also enjoy programming. That's right, I said *enjoy* – programming can be so enjoyable, in fact, that there is an immense community called the "open source community" where people, in their spare time, create computer software for fun and give it away for free. So, the course is basically a) learn tools that computers understand b) practice by solving puzzles to make the computer do work for you.

Imagine that you are a woodworker – wood is a very versatile and capable material, but you need to learn the techniques and tools (saws, hammers, screws, nails, etc.) to make that wood into a nice new deck. Programming is similar – the possibilities are endless, but you need to master the tools first, and figure out how to use them to make your dreams reality! In this course you will learn these basic tools that will stick with you no matter where you encounter computer programming or in what language.

You will learn abstract thinking and problem solving with respect to how computers work. This will not be a course in learning how to use software like MS Word or Excel, you will not learn how to make webpages, or make a blog, or how to use the internet. We will do cool stuff, however, and you will be more comfortable with understanding how computers work.

Earlier I said that computers are really stupid. So why do we use them? The reason is simple: they are really (really) fast. A typical computer these days can do about 2 Gigahertz and has 4 to 8 cores – translation:

- 2 Gigahertz means 2,000,000,000 operations per second
- 8 cores means it can do 8 calculations in parallel
  - 16,000,000,000 operations per second. (16 billion)

That's a really big number that is basically unfathomable to us mere mortals. But! If we can harness that power, we can put it to work, and we can do a lot of work with 16 billion operations (like watch cute kitten videos on the internet).

For Information Only (not testable): Throughout this course, there will occasionally be information given that is purely for academic purposes. Those interested can sit and think about these items, but they are beyond the scope of the course and are not testable. These sections are marked with a graduation hat to the left. For example, the above Gigahertz example assumes that the computer can do one complete operation every time the clock ticks. This is not true – many operations take multiple ticks. Further, when you have multi-core machines (the above example has 8 cores), it is not as simple as the 8 cores working in parallel. It actually takes work, overhead, to coordinate all the cores and enable them to share resources such as memory or hard disks. It turns out that a great deal of factors determine how much work your computer can get done, not just the clock speed or cores.

Coming back to computers being stupid – if you tell your computer to smack its head against the wall forever, it will do it, just as you tell it to (and probably do it 16 billion times each second). As a programmer, your job is to give clear instructions to the computer so that it does exactly what you want it to. Unfortunately, if there is even one mistake in your programming, the computer will follow that mistake blindly and your program will not work like you expect it to. It has no common sense, and cannot fill in the blanks like a person can.

computers do exactly what you tell them to do, not what you want them to do.

Notice the green exclamation mark to the left? Throughout this course, you will see this mark to signify that there is something really important here. A definition, core concept, or requirement for assignments.

This highlighted statement may seem a little strange right now, but by the time you finish your first assignment, you will clearly understand what this means. As a programmer, your job is to use the tools of programming to give clear instructions to a computer. You need to translate what you want the computer to do into the computer's language.

Given a job to do, like drawing a diagram, you provide the recipe for the computer to follow to do the work. This is kind of like cooking – a perfect recipe will enable you to reproduce a favorite dish or dessert. Similarly, sheet music is a program for a musician to reproduce a piece of music. Unfortunately, in both these cases you give some credit to the cook or musician – with computers, you need to spell out everything and leave nothing to common sense, style, or interpretation.

1.5 Computer Language

Different computers speak different languages. Also, the real on-the-chip language that computers need is really confusing and hard to understand – even for computer experts. Basically it is a bunch of 1s and 0s (called *binary*), or a short-hand called *assembly language*, which makes you micro-manage every single computer operation. Assembly is also different across computers, making it hard to remember. These are called *low-level* languages because they are very close to how the machine works.

Higher-level languages try to make things more accessible to people by moving further away from the hardware requirements (good news for us puny humans). Some common high-level languages that you may have heard of are C, C++, C#, Java, Python, PHP, etc. In this class, we will be using a language called Processing, which is effectively Java with all the crap removed (err, I mean, simplified for learning?). The other cool thing about Processing is that we will be doing graphics from the beginning!

There are a lot of similarities across computer languages. Processing (and Java) are "C-Like" languages, in that they resemble the classic programming language "C". Many modern languages do. In fact, you will find that most languages are so similar that, once you finish this course, you can pick up many new languages just by reading a quick primer. The main computer language tools you will learn here exist

in very similar forms in almost all languages. As such, you are not really learning the Processing language, but rather, learning fundamental computer tools that exist in the vast majority of programming languages.

Check your Understanding

1.6 Check your Understanding: Exercises

In this course, exercises will be posted at the end of the unit. These exercises will not be marked or assessed in any way, but are primarily for you to think about the things in the unit, and to provide structure for you to work on what you learned. Of course, feel free to email your instructor regarding any of the exercises. These exercises do not have solutions posted for learning purposes. If you need help, engage the forums, work in teams, or ask your instructor.

The exercises in this section are primarily about looking around the internet to learn a little and consider what programming really is all about. In later chapters the exercises become much more complex and can be quite difficult.

Exercises in this course are marked as having a difficulty level: bronze, silver, or gold. You should use these to gauge your success in the course and expected grade. For example, if you want to pass the course you need to be able to do all the bronze exercises without difficulty. If you want a B in the course, you need to be finishing the silver exercises. If you want an A or A+, then doing many of the gold exercises is required. If you are struggling with the silver exercises, then be prepared that a B may be difficult to attain.[5]

Exercise 1. Go to your favorite computer store website, and take a quick look at the computers for sale. Find a machine that you may buy (laptop or desktop, your price range).

a. How fast is the machine in GHz (gigahertz)? You may have to click on a “details” or “specs” link, but the information will be available.
b. How many “cores” does the machine have? That is, how many operations can it do in parallel? This is often specified as “dual” or “quad” core. This may be harder to find.
c. Assuming that the computer can do one whole calculation every time the clock ticks, calculate how many operations that computer can do

[5] The medal images are public domain: https://commons.wikimedia.org/wiki/File:Plakette-bronzefarbenmitZackenamRand.png, https://commons.wikimedia.org/wiki/File:Plakette-goldfarbenmitZackenamRand.png, https://commons.wikimedia.org/wiki/File:Plakette-silberfarbenmitZackenamRand.png

per second, which is the GHz (billion cycles per second) multiplied by the cores.

d. If the computer could look at one person every operation, how many people could it look at in one second? How many people are there in the world?
e. If the computer could look at one star per operation, how long would it take to look at all the stars in the milky way?

Exercise 2. A lot of people are interested in learning to program. Do an internet search for learning how to program and take a look at some of the resources. If there is so much available, why take a course? The answer is usually multi-faceted: a) on-line tutorials can be poorly written and, as such, overwhelming. b) you may need a lot of specialized software and it can be hard to get started. c) on-line tutorials often teach how to do something specific (which you may not want to do), while a course teaches fundamentals that set you up for further learning. d) other reasons, even for those with experience?

Exercise 3. Why bother learning to program? Do a search for this: why learn to program? and look at some of the resources. There are many news articles, essays, and opinion pieces by experienced people that talk about some of the reasons why learning to program can be important for anyone.

a. Assuming you do not plan to be a computer science professional, list three reasons that learning to program may be useful for you.

Exercise 4. Search on the internet for the “open source” movement.

a. Why do people create software for free?
b. Find an example of an open source “operating system” that can be used instead of Windows or MacOS
c. What is the world’s most widely used web server software. Is it open source?

How did you do?

Learning Objectives

How did you do? Go back to the beginning of the unit and check how you measure up to the learning objectives.

(page intentionally left blank)