COMP 161 - Lecture Notes - 01 - Programming

January 8, 2016

This course is not your first look at programming. In these introductory notes we look at how this course differs from your experience in COMP160 and how it builds off of your experiences programming in BSL Racket with DrRacket. We also get a brief overview of tools we'll use to craft our programs in this course.

What's a Program?

The title of this course is *Introduction to Programming*. You're probably thinking, "we just had a semester's worth of programming, so what's this?" Well, yes, you know enough about programming to write code to solve pretty much any problem you encounter. However, you don't know enough to stay out of trouble nor do you have practice transferring your programming knowledge from one language to another. In this course we transfer knowledge from Racket to C++ and then expand on that knowledge with new computing fundamentals.

The word **program** itself implies a plan or a schedule. To see the plan in a Racket program you had to think like the Racket Virtual Machine[1] and decide the order in which actual computation was enacted. What if instead, you the programmer set the plan? From this we arrive at the perspective[2] of programming in which we issue imperative statements to a computer to executed in the order we specify. Groups of statements can be written as procedures. Thus, we call this style of programming *Imperative, Procedural programming*.

[1] or "DrRacket"

[2] or paradigm

Imperative, procedural programming will force you to think much more explicitly about the machine that is running your code than you did in the Functional style used in COMP160. So, one of the major goals for this class is to understand this new programming paradigm and begin comparing and contrasting it to the functional style. In doing so, you can better understand them both. Let's be clear. You are by no means starting over. We can and will leverage everything we learned about computation and programming in COMP160. We just need to revisit the fundamental ideas and programming methodologies we learned about programs and computation to fit within our new paradigm and our new language. Before we can do that, we need to find a new set of tools for replace DrRacket.

Programming: Science, Engineering, and Craftsmanship

Programming is part science, part engineering, and part art. You're beginning to understand the science, but its time we think about the art and engineering and take a broader view of the craft of programming. Plumbers, carpenters, and mechanics all have a standard set of tools they use and shared set of goals for the things they do. They all know what good craftsmanship[3] looks like and how to evaluate the quality of their work and the work of others. The same is generally true of programming. In order to program you need a specific set of tools. When you're programming, you have a specific set of goals you're trying to achieve in order to produce a high quality program. It are these tools and these goals that you need to start thinking about and integrating into your program design process. We should, of course, begin with the goals as they are the end to which programming tools are a means.

[3] craftsperonship

The Problem

In general, we can say that a program begins well before the first line of code is written. It begins when someone, *identifies a problem* and decides to *solve the problem using computing technology*[4]. Now, sometimes the problems are as straight forward as, "design a game to make me lots of money," and sometime they're less clear like, "How do we enhance the quality of life of people suffering from dementia?" Once someone decides they're going to use computation to model and solve a problem, they must identify the *platforms* on which they'll launch their solution.

[4] we're just thinking software, but hardware and hybrid systems are options as well

The Platform

When we say platform we generally are thinking about the hardware and operating system layers on which our software runs. However, these days software itself is a viable platform. Some possible platforms are:

- Smartphones: Apple, Android, Windows, Blackberry, etc.
- Wearable, embedded systems: Google Glass, smart watch, etc.
- PC Computer GUI or CLI: Windows, Mac, Linux, etc.
- Laptop Computer GUI or CLI: Windows, Mac, Linux, etc.
- Tablet: Android, Apple, Windows, etc.
- Gaming Console: XBox, Playstation, Steam Box, etc.
- Web Browsers: Chrome, Firefox, etc.
- Virtual Machine: Java, Racket, CLR (Windows .net architecture), etc.

Writing apps for multiple platforms is more possible today than it has been in the past, but on the other hand, there are arguably more viable platforms than in the past, so things aren't that much easier on that front. In this course, we'll develop on one platform: the Linux Command line. More specifically, we'll write code to run on the department's Ubuntu[5] Linux[6] server.

[5] `http://www.ubuntu.com`

[6] `http://en.wikipedia.org/wiki/Linux`

For most of you, this is likely to be a new computing environment on two fronts. First, you've probably never seen or maybe even heard of Linux. Even amongst those of you that know about Linux, it's likely you've never really worked at a Command Line Interface (CLI). The command line interface uses no windows, no mouse clicks, just text-based commands. Working at the CLI is roughly equivalent to working at the interactions window of DrRacket. You type commands at a prompt, the computer executes them, and usually prints[7] some results back at the prompt. The CLI pre-dates Graphical User Interfaces[8], but is still widely used in many computing environments. It is also usually hiding in the background when GUIs are installed and as such is a viable choice for many projects. In short, the CLI is still alive and kicking and you'll gain a lot from knowing how to work with it. The first thing we'll do in this class is be sure you're able to work and survive at the CLI[9].

[7] DrRacket always prints. The CLI does not.

[8] GUI

[9] survive = basic working knowledge not ninja CLI hacker skills.

The Criteria of Quality

So we have a problem to solve and a platform on which we'll deploy our computational solution. Or, we have an end in mind and need to start thinking about a means to achieve that end. However, before we start hacking away at some code we should reflect on what it takes to write good code[10]. A craftsman tries to produce quality work and knows how to judge their work for quality. Programs are written in order to be[11]:

[10] If your goal is to write bad code, you're in the wrong place

[11] In order of importance!

1. Correct
2. Simple
3. Efficient

Thus the quality of a program[12] can be evaluated on these criteria[13].

[12] and its programmer

[13] We could be more detailed, but odds are if you meet these criteria, any other more detailed criteria will be covered as well. Alternatively, you should learn to work around these simple goals and only after you've gain experience in this realm, explore more involved criteria of quality programs.

If a program does not correctly carry out its intended task, solve its specified problem, then it's not very good. Until the program works correctly, then we don't even need to consider any other measure of quality. This sounds simple enough, but program correctness is a very tricky thing. First off, correct might be subjective and it can be difficult to clearly identify what perfect correctness will be. Furthermore, when we do know what constitutes correctness, it is nearly impossible to guarantee absolute correctness. As such, good programs more often than not exist in a state of *mostly correct*. This is clear from the fact that even the best software needs to fix bugs and update itself. If absolute correctness is unachievable, what then is a programmer to do? First and foremost, programmers need to identify the level of correctness a user can expect and try to guarantee at least that much. This means clearly *identifying and documenting the specifications for the program in such a way that a correctness benchmark can be set and tested*. We must also plan for the unexpected bug to occur and develop code that is easily maintained over the life-time of the program. This means developing code that is not just machine readable but human readable.

Programs*should be simply and elegantly written* so that when bugs appear, you can easily return to the code to fix them[14]. Program code is more often read by human beings than it is computers, and as a written document should be judged on those standards. Your code should have structure and style and be easily read by other programmers. As you can imagine, simplicity is subjective and can be hard to evaluate. None the less, there are well accepted styles of programming out there upon which we can choose to evaluate our programs[15]. Simplicity is not only a boon to correctness, but a boon to business. Well designed and simply written code is often easy to extend. New features are easier to add to programs that exhibit well established metrics of good design[16]. Simple code is also often faster as it cuts out unnecessary program logic and avoid repetition, and when its not fast enough, its simplicity makes it easier to reason about and thereby easier to optimize. And this brings us to goal number three, efficiency.

[14] Keep in mind here, we're not really talking about the small programs you've written so far. Yes, even those projects from last semester are small by program standards. We're talking about millions of lines of code.

[15] choose a normal when no universally accepted normal can be found

[16] the kinds of things the HtDP design recipe gives you

Sometimes when you focus on writing correct and simple code, you get a program that performs as well as it needs to on the target system. On the other hand, your program may often run too slowly or use up too much memory. Put anther way, it may make inefficient use of the computation resources provided to it by the computer. When this is the case[17] we must optimize the resource usage of our program. This often means trying to make it run fast, i.e. make better use of the CPU cycles, or use less memory. Of the three programming goals we'll be looking at, this is the newest one for you to think about. We'll need to learn how programmers talk about efficiency and how we measure the efficiency of our code. Eventually, you'll learn how to build efficiency concerns into your initial designs and specifications. This often boils down to knowing and deploying efficient *algorithms* from the start rather than optimizing inefficient code after the start.

[17] and only when this is the case!

Putting this all together. We have a problem, an idea of how to model and solve that problem computationally, and criteria for evaluating the computational solution. Now we need to choose the right tools for the job[18].

[18] Platform often dictates or restricts your choice of tools. At which point, you have a new problem... building the tool or platform you want!

The Tools

DrRacket is a one-stop shop for all your programming tools needs. Such programs are called *Integrated Development Environments*[19]. You will not be using an IDE in this course[20]. Instead we'll look at an established set of industry tools and learn the basics of making them work together. This deconstructed view of the programming tool chain will hopefully give you a better appreciation for the tools that are out there and the different systems tucked away inside IDEs like DrRacket. All the tools you'll be using see wide use today and are viable options of program development.

[19] IDE

[20] we'll come back to IDEs in COMP220 and COMP210

The Essentials

At a minimum you need a **programming language**[21], **text editor to write the code** and either an **interpreter** to execute the code or a **compiler** to build an executable program for your platform. You know what a programming language is and that we'll be using C++ in this class, so we'll focus on the other two tools.

[21] yes. languages are tools

Text editors do what the name implies, they edit text. They do not process words. The difference is that text editors don't really get into presentation details and most importantly do not encode the text in anything other than a plain text encoding[22] Windows Notepad is a text editor, but not particularly well suited for programming. Good text editors for programming are programmed with information about the language you're using and provide help and cues to ease the task of writing code. DrRacket's definitions window color coded text, helped match parenthesis, fixed indentation to meet Racket style, and much more. These are the types of things we need our text editor to do. For this class, you'll learn to work with GNU[23] *Emacs*[24]. Other options you might explore include[25]:

[22] Probably ASCII. Possibly Unicode.

[23] `http://www.gnu.org/`

[24] `http://www.gnu.org/software/emacs/`

[25] some of these are platform dependent

- Vim[26]
- Sublime Text[27]
- Notepad++[28]
- Atom [29]

[26] `http://www.vim.org/`

[27] `http://www.sublimetext.com/`

[28] `http://notepad-plus-plus.org/`

[29] `https://atom.io/`

Feel free to explore these and other options, but Emacs is the only supported text editor for this course[30].

[30] Don't expect answers to non-Emacs questions

Interpreters are installed on the platform and can read and execute code on a line-by-line basis. They run the program as they read the code. On the other hand, a compiler translates the code to another format, typically a fully executable file[31]. These days it's not uncommon to see a combination of the two. A just-in-time[32] compiler will interpret some code but compile performance critical code for faster execution. Racket uses a JIT system. In this class we'll use a traditional compiler system, namely the GNU GCC[33] compiler *g++*. Other notable C++ compilers are:

[31] or something that's interpreted

[32] JIT

[33] `http://gcc.gnu.org/`

- LLVM and clang [34]
- Visual Studio and CLR [35]

[34] `http://clang.llvm.org/`

[35] `http://msdn.microsoft.com/en-us/vstudio/hh386302`

Programs quickly grow to involve multiple files. The CLI compiler we'll be using is pretty good at helping your compile all those files quickly with a minimum number of commands. However, it is more common to use a special program building tool to manage the complexity of the compilation process. The build tool *make*[36] is a very flexible tool and widely used to manage the problem of making programs. Make is itself a special purpose programming language and interpreter. To use make we write a small file called *Makefile* that the command make then reads and interprets. So, unless you like to enter ten commands when one will do, make is really awesome.

[36] `http://www.gnu.org/software/make/manual/make.html`

Tools for Correctness

A language, an editor, and a compiler[37] will get you to a working program. Now we need to address the correctness of that program. Good languages often provide you with language features specifically designed to help you write correct code[38]. But, practiced programmers also make use of several tools for helping reach their correctness goals. The most common are:

[37] and build management system

[38] assertions and exceptions are two examples

- compilers
- debuggers
- memory checkers
- code testing frameworks

Compilers are the first line of defense. Basic compilers will catch deviations from the language grammar[39]. Unlike your professors, the computer does not[40] infer your intentions from your code. In addition to guaranteeing the grammatical correctness of your code, a good compiler will also warn you when you do something that might lead to problems. We'll also see that compilers can effectively

[39] syntax errors

[40] and should not

annotate our code such that other tools can more effectively analyze it. In particular, g++ can add special flags to the finished product that enable debuggers and profilers to give us better reports about our program's execution behavior.

Grammatical correctness is a pretty weak level of correctness. Every programmer has written a program that compiles and runs but produces incorrect results[41] or crashes at run-time[42]. A *debugger* allows programmers to step through program execution one step[43] at a time while keeping an eye on program data. DrRacket had a stepper that allowed for this. For stepping through and debugging our C++ programs, we'll explore the GNU debugger **gdb**[44].

[41] logic errors

[42] run-time errors

[43] or programmer specified skips

[44] `https://www.gnu.org/software/gdb/`

Run-time errors can often be the result of running afoul of the allowed usage of the computer's memory system. To correct these mistakes, we use programs that observe the memory usage patterns of our program and generate detailed reports of where something goes awry. The standard tool for this in Linux, the tool we're going to use, is *memcheck*[45]. The memcheck tool is a part of the *Valgrind*[46] family of code analysis instruments.

[45] `http://valgrind.org/docs/manual/mc-manual.html`

[46] `http://valgrind.org/`

Where all the previously discussed tools were programs in their own right, testing frameworks are just libraries of code written to more easily enable standard program testing regimes. In COMP160 you learned to do *unit-tests* and we'll continue to use them in this course. These tests look at individual units of the program and test for expected functionality and behavior. In this class we'll make use of a C++ unit testing framework call *gtest*[47] developed by Google.

[47] `https://code.google.com/p/googletest/wiki/Documentation`

Tools for Simplicity

There's a general lack of tangible tools for simplicity[48] checking really. The best tool to check for sufficient simplicity is your fellow programmer. Different programming communities often agree upon what good, simply written coding style looks like. These stylistic guidelines ensure that code looks and reads consistently within the community and is thereby simple to the members of the community. So, one of the best things you can do is have your code peer-reviewed[49] for its style[50]. We'll adopt some basic style guides for this class and will go over them as we learn C++. In the meantime, you should take a look at what professional style guidelines are like. Google has their C++ style guide published on the web along with style guides for other languages they use[51]. There are programs called *linters* that scan for suspicious looking code, but we won't play with them in this class. Google has an in-development linter/style checking tool called cpplint. You might check it out, it's with their style guides.

[48] in the sense that we're using this word

[49] We need a writing center for code!

[50] Don't have your peers write and debug your code for you. That's called academic dishonesty!

[51] `https://code.google.com/p/google-styleguide/`

Tools for Efficiency

Once the code works and looks good, it's often time to try and speed it up or lower its memory footprint. In most cases, there are no tools that can make your code more efficient for you. You'll need to do your own optimization. Smart programmers involve real data about program performance in their optimization process[52], so many of our tools are used to gather data that allows us to make informed decisions about optimizing our code.

[52] Check out: `https://www.facebook.com/notes/facebook-engineering/the-mature-optimization-handbook/10151784131623920`

- Compilers
- Mathematical analysis
- Memory system profilers
- CPU profilers

There is one tool that will auto-magically make[53] your code faster: the compiler. Modern compilers can carry out basic to sophisticated transformations on common code patterns in order to improve code performance. This is wonderful as we generally just need to focus on big picture logic and not low-level optimization details. However, this process effectively re-writes your code, making it harder to debug so compiler optimizations are often something we don't introduce until we're confident that our program is correct enough. Compiler's can only do so much for you though. If you're code is inherently inefficient, then it's not going to fix that for you. This means we need to make good, efficient coding choices before we even turn the compiler loose.

[53] or attempt to make

Using a standard form of mathematical analysis[54], we can guarantee[55] the worst case behavior of our code under some fairly reasonable assumptions. Once we know we've make sound algorithmic decisions and have acceptable upper-bounds on program efficiency, then we must delve down into reality as our assumptions are reasonable but simplify some key details. To see what happens to our code on hardware we must run it and use a *profiler* to gather performance metrics about its execution. For the types of programs we're looking at, we need to know how efficiently our program makes use of the CPU and of memory system.

[54] Asymptotic analysis or "Big-Oh"

[55] as in mathematically prove!

Valgrind provides us with a CPU profiler called *callgrind*[56]. This profiler attempts to count how often each procedure is called and where in the code it's called. From this we can begin to understand what code is running most often and where we can get the most bang for our optimization buck[57]. It should be noted that to profile the CPU we often count the things that are executed and not how

[56] `http://valgrind.org/docs/manual/cl-manual.html`

[57] Make the common case fast

long they take to execute. We'll come back to this. For now, you should think about why that might be a good idea.

Valgrind also provides us with a memory system profiler called *cachegrind*[58] and another called massif[59]. These tools let us look at different parts of the memory system[60] and determine how often we're using them and if we're using them efficiently.

[58] `http://valgrind.org/docs/manual/cg-manual.html`

[59] `http://valgrind.org/docs/manual/ms-manual.html`

[60] the cache and heap respectively

Other Tools

The final, commonly used tool, that we're likely to play with is a Version Control System[61] called *git*[62]. You hopefully have picked up on the possibility that real programs have long life spans. You write some code then fix somethings and optimize others. In professional settings in particular, it is important to keep track of code that results in the last, stable piece of software you developed. Version control systems effectively let you take snapshots of your code and then do things like jump back to a previous snapshot or merge new code with an existing snapshot. They also enable easy off-sight backup in case the computer you're working on goes kaput and you lose your code[63].

[61] VCS

[62] `http://git-scm.com/`

[63] checkout `http://github.com`

Tool Wrap-up

That's a lot of tools. We'll just barely scratch the surface of what most of them can do. Our goal is not to master these tools but to recognize that they're there and how they are used to develop better programs. If you come at this from the other direction, their very existence and functionality sheds light on what matters to practiced programs. Programmers encounter problems with developing good[64] code and these are the programs they developed to solve their problems. So even if the types of programs we write in this and other classes don't really need these tools, we can rest assured that some day we'll bump in to the exact kinds of problems these tools were developed to solve. With that in mind, let's revisit the tools we'll be putting to use in this class:

[64] correct, simple, and efficient

1. Platform: Linux CLI
2. Language: C++
3. Text Editor: Emacs
4. Compiler: g++
5. Build Automation: make
6. Unit Testing Framework: gtest

7. Debugger: gdb
8. Memory Checker: valgrind memcheck
9. Efficiency Analysis: mathematics. asymptotic analysis.
10. CPU Profiler: valgrind callgrind
11. Memory Cache Profiler: valgrind cachegrind
12. Stack and Heap Profiler: valgrind massif
13. VCS: git

Finally, one of our best tools in our tool box is our fellow programmer. Programming is, more often than not, something done by a community of like minded individuals setting out to solve some problems. Feedback from your programming community can really help improve your code and make you a better programmer.