How Will You Practice Virtue Without Skill?: Preparing Students to be Virtuous Computer Programmers

Victor Norman
Assistant Professor
Computer Science Department
Calvin College
July 2015

Introduction

The teaching of computer programming is often thought to be simply the teaching of a useful skill, that of learning the syntax and semantics of a programming language, and learning to solve problems by decomposing them into programmable units. And, while computer programming certainly does require a great deal of skill, is learning a new skill all that we should require of our students?

At my Christian liberal arts college, we are encouraged to teach three things: knowledge, skills, and virtues. [1] The challenge of teaching virtues in the computer science classroom intrigues me. How do I encourage the development of virtues while teaching the skill of computer programming? If I encourage virtues while teaching computer programming, what effect do these virtues have on the computer code that our students produce? And, why is it important to teach these virtues, besides the fact that my college says I should?

In the context of the care of God's creation, Wendell Berry, in his essay, "The Gift of Good Land" [2], speaks about the virtue of charity, and how it relates to learned skills:

> Charity is a theological virtue and is prompted, no doubt, by a theological emotion, but it is also a practical virtue because it must be practiced. The requirements of this complex charity cannot be fulfilled by smiling in abstract beneficence on our neighbors and on the scenery. It must come to acts, which must come from skills. Real charity calls for the study of agriculture, soil husbandry, engineering, architecture, mining, manufacturing, transportation, the making of monuments and pictures, songs and stories. It calls not just for skills but for the study and criticism of skills, because in all of them a choice must be made: they can be used either charitably or uncharitably. How can you love your neighbor if you don't know how to build or mend a fence, how to keep your filth out of his water supply and your poison out of his air; or if you do not produce anything and so have nothing to offer, or do not take care of yourself and so become a burden? How can you be a neighbor without

> applying principle—without bringing virtue to a practical issue? How will you practice virtue without skill?
>
> The ability to be good is not the ability to do nothing. It is not negative or passive. It is the ability to do something well—to do good work for good reasons. In order to be good you have to know how—and this knowing is vast, complex, humble and humbling; it is of the mind and of the hands, of neither alone.

Although Berry speaks here about care of God's creation, his words apply equally well to the craft of computer programming. Berry emphasizes that the application of a virtue such as charity is essential. One could translate Berry's statements to apply to computer programming this way:

> Real charity calls for the study of software design, software maintenance, creation of charitable code and documentation, and thorough testing. It calls not just for skills but for the study and criticism of skills, because in all of them a choice must be made: they can be used either charitably or uncharitably. How can you love your neighbor if you don't know how to build or mend software, how to document it, and how to build testability into it? How can you be a neighbor without applying principle – without bringing virtue to a practical issue? How will you practice virtue without skill?

Notice that Berry's comments require a person to always be thinking about others. I have found that if I am going to emphasize in the classroom the development and application of virtues to the practice of computer programming, then the first step is to convince students that they need to consider others when they are developing code. Students often think that the code they write need only produce the correct results (i.e., have the correct function), and be readable by them. And, truth be told, code written for early assignments in an introductory computer programming course probably will only be read by them (and the grader). However, I emphasize that the student must learn the habit of writing code for others to use, as is done in the "real world" outside the classroom. For example, the code they write must be readable and testable, well documented, and able to be easily modified.

While Berry emphasizes that a practitioner must learn a skill in order to be able to practice virtue (that is, skill comes before virtue), I believe that students can learn both the skill and the practice of virtue at the same time. Some of the virtues such as hospitality and humility can be emphasized in early computer science courses; others are better emphasized in advance courses, where projects are larger and more freedom can be given to students to make more coding decisions.

The remainder of this paper will investigate the relationship between virtues and the act (and art) of computer programming, and how a teacher can encourage the practice of these virtues in programming assignments. The virtues I will investigate

are **hospitality, humility, integrity, honesty, creativity, stewardship, and diligence**.

Note that I do not concern myself with the relationship between these virtues and the choice of what kinds of programs to write, how those programs present themselves (the GUI or command-line), or the process used to create the code (working in teams, code reviews, etc.). Instead, I look at the relationship between these virtues and the actual computer code the students write. So, for example, we want to investigate how the practice of hospitality, creativity, or integrity affects the code that a student writes, or how the practice of defensive programming demonstrates humility. We will also investigate how one teaches (or at least encourages) students to develop virtuous coding practices.

We first begin by investigating the practice of hospitality and how it could affect the writing of computer code.

Hospitable Coding

Most people, when expecting guests, try to make their home hospitable by cleaning it, lighting it properly, and making it comfortable. They prepare food and drink, and perhaps, entertainment. In short, they make the space welcoming. I teach my students that hospitable code gives the reader of the code this same sense of being welcomed, a sense of warmth, and a confidence that the code was created with care. Hospitable code welcomes the reader to come in and be comfortable, to enjoy the cleanliness of the code, to feel at home, and to see that the space has been carefully prepared with guests in mind.[3]

Hospitality is also an attribute of our Triune God. Cornelius Plantinga, Jr., in *Engaging God's World* [4, p. 20], writes,

> At the center of the universe, self-giving love is the dynamic currency of the trinitarian life of God. The persons within God exalt each other, commune with each other, defer to one another. Each person, so to speak, makes room for the other two. I know it sounds a little strange, but we might almost say that the persons within God show each other divine *hospitality.*

How does one do this with computer code? A programmer has many choices to make when writing code, many of which affect the readability of the code. Let me give three examples of choices the programmer has which can affect the hospitality of the code.

- Clear and descriptive variable, function, module, and class names. A line of code as simple as

  ```
  a = input("Enter grade division: ")
  ```

can be improved and be made more hospitable to the reader by instead being written:

```
minAGrade = input("Enter grade division: ")
```

When a reader encounters the first line, the reader may not immediately understand the purpose of the code. This may make the reader anxious that he is already losing understanding of this code. However, if the reader instead encounters the second line, he can intuit immediately that the code uses this variable to indicate the minimum score that is an A grade. Thus, the reader is left more confident that he understands the code so far.

- Proper in-code documentation (i.e., comments). The proper amount of documentation in the code explains to the reader any tricky or non-obvious steps or structures in the code. Again, the proper use of documentation can have a profound positive effect on the reader of the code. Conversely, a person who is not programming with other people in mind has little motivation to add good documentation to his code.
- Consistent indentation. All modern programming languages contain control structures that cause the code to execute code conditionally or repeatedly. The coding structures can become nested within one another. For example, here is a piece of code that is not indented:

```
foreach elem in aList {
if (elem.score < FAILING_GRADE) {
fails.append(elem)
aList.remove(elem)
}}
```

Understanding this code is difficult. However, if I indent the code consistently and carefully, it becomes much easier to see that the code removes all elements from `aList` have scores less than FAILING_GRADE, and adds them to `fails`.

```
foreach elem in aList {
   if (elem.score < FAILING_GRADE) {
       fails.append(elem)
       aList.remove(elem)
   }
}
```

If the programmer does not indent the code consistently, then the reader will find it difficult to determine how control flows through the code. This difficulty can undermine the reader's confidence in understanding the code's logic.

Coding with Humility

Humility is a "just estimation of our powers as finite and fallen knowers." [1]. Thus, for a student learning to program, humility means, first, that the student should have a proper judgment of his ability as a programmer, and second, that he should recognize his own limitations and fallenness.

Three ways in which humility can be demonstrated in computer code are 1) defensive programming, 2) code that has been thoroughly tested by the author, and 3) code that has an appropriate level of complexity. All these properties of code can be taught in the classroom.

Defensive programming includes code that checks inputs to a program or subprogram to make sure they conform to assumptions made by the code. For example, if you have written a function to take a list of numbers and return the average, defensive code means the code in the function should:

- Check to see if the list that is passed in contains only numbers (integers and/or floats) and not some other data types, like strings or booleans, which cannot be averaged.
- Check to see if the list is empty, in which case one cannot compute an average.

The following code demonstrates defensive programming.

```
def average(aList):
   // Check for empty list and to make sure all values
   // in the list are types that can be averaged.
   if aList.isEmpty():
       throw ValueError("Empty list")
   foreach elem in aList:
       if not isInstance(elem, (int, float):
           throw TypeError("Bad value type in list")
   total = 0
   foreach elem in aList:
       total += elem
   return total / aList.len()
```

In my classroom, when I introduce this concept of defensive programming, students often ask me why this extra code is necessary. Their opposition comes from thinking that they will be the only one writing code to call their function and so they will just take care not to call their function with bad values. This is a teachable moment where one can emphasize that they shouldn't overestimate their own ability to get

things right every time. In fact, the term "defensive programming" exists because seasoned professionals have learned (the hard way) that they should not trust themselves to not make these kinds of mistakes.

A second result of having the virtue of humility while programming is writing unit tests to exercise your code and find errors in it. Again, students often question why this is necessary, believing (often incorrectly) that the code they write will just be correct the first time, for all cases. However, when you teach them to write unit tests, they quickly learn that often they have not thought about all the cases that their code should handle. Again, this is a way for students to learn humility while learning to program.

A third result of humility is that students will write code that has appropriate complexity. Often one can "show off" in code by writing code that is more complex in structure but shorter than an alternative, more obvious solution. Sometimes this code is described as being "cute" – interesting to study and understand, but not obviously correct. Humility discourages one from writing code this way. A programmer with humility acknowledges that she may not understand the problem fully at first, and she needs to write code that handles the problem and its many cases in a systematic and clear way.

As an example, consider this code snippet taken from "The C Programming Language", an early introductory programming book. [5, p. 106]

```
/* copy t to s; pointer version 3 */
void strcpy(char *s, char *t)
{
     while (*s++ = *t++)
          ;
}
```

While this code correctly does a string copy from one array to another, and is short and efficient, it is not obviously correct. In one line the code employs pointer dereferencing, postfix increments, assignment, and an implied comparison to the end-of-string character `'\0'`. (Note that some may argue that this code is the most efficient code one could write. However, modern compilers do such good code optimization that longer, more readable code, employing fewer tricks and shortcuts, will still be compiled down to code that matches the above code in efficiency.)

Teaching Integrity in Code

I mentioned above that code written with hospitality demonstrates code that has integrity, being well designed on the inside as well as on the surface. But, code that demonstrates the virtue of integrity is more than that. Coding with integrity means creating code that is complete, sound in construction, has internal consistency, and

has honest naming and documentation. One can teach this virtue of integrity by asking students to code in the following ways.

First, and similar to the naming of variables as discussed above, students should use function names that honestly represent the functionality that the function code implements. A function definition is the creation and naming of an abstraction. Multiple lines of code can now be referred to by a single name – a name that can serve to either enhance the readability and understandability of the code, or can mislead the reader or obfuscate the behavior of the code. In labs or homework assignments, students sometimes use function names that are overly short or too generic, or simply do not represent the functionality of the function (sometimes because students have changed their minds about what the function should implement). All these behaviors should be discouraged.

Second, any output produced by a program, textually or graphically, should be backed by code that honestly computes the values being outputted. Students are sometimes tempted to write code that simply prints out the expected values for the assignment, instead of "honestly" computing the values and then printing them out. Similarly, some code may "pretend" to simulate some process and display the simulation graphically, when actually the code just moves the graphical elements around the screen to look like the values are being computed.

As an example, a student chose a project to simulate a Newton's cradle, a device that demonstrates the transfer for energy and momentum between swinging balls. However, the student's code actually just computed how to display circles on the screen, so that they appear like the balls in a Newton's cradle. The code did not compute how the energy is transferred between the balls. This project did not demonstrate honesty and integrity in its implementation.

Third, students should honestly document known limitations in the implementation of a function or of a program as a whole. It is always tempting when doing a project to be graded by the professor (or grader) to not document known problems or limitations, and hope that the professor does not notice. (This is also a strong temptation for professionals working in large teams on large software projects.) However, instructors should encourage students to adopt the practice of documenting known problems, perhaps by giving credit for doing so.

Fourth, students should write code with consistent quality throughout. In advanced programming projects consisting of many hundreds of lines of code, a student will often start strong, producing code that is well documented and well tested. However, as time goes on and the project gets larger and larger, code quality may fall off. Thus, the consistency is lost and integrity of the code lowered.

Similarly, students should be consistent in their naming of identifiers and coding style throughout their program. Many companies have a coding style document and expect employees to write code that conforms to it. Similarly, students should be

given a coding style document and should be expected to follow it, consistently and thoroughly in all code that they write for the class. This is a practice that will serve them well when they get into the workforce.

Creativity in Design and Implementation of Code

The process of writing computer code is inherently a creative endeavor, as the programmer has enormous freedom in how to solve a given problem and implement the solution. The act of creating echoes God's behavior during creation of the world. The first characteristic of God revealed in the Bible is creativity, revealed in God creating a complex, interdependent, and beautiful world. Computer programming can approximate this activity. As Fred Brooks says [6, p. 7],

> The programmer, like the poet, works only slightly removed from pure thought-stuff. He builds his castles in the air, from air, creating by exertion of the imagination. Few media of creation are so flexible, so easy to polish and rework, so readily capable of realizing grand conceptual structures.

In computer programming, the breadth of creative options is vast. I have already mentioned some of the options: choices for variable and function names, choices for code indentation, choices for how many or few comments to add to the code, and choices for the complexity of code to solve a problem. Other choices include:

- How to design data structures that accurately, robustly, and efficiently store collections of data.
- How to design code modules (classes or structures) and what functionality each module should implement and expose.
- Which algorithm to use to solve the problem, and how to implement it.
- How to design a graphical user interface.
- How and where to insert debugging statements in the code, to help testing.
- How and when to use parallelism to solve a problem more efficiently.

In [7], I argue that computer programming has many similarities to the creation of classical fine art – painting, sculpting, or composing music, for example. Just as a painter sits before an empty canvas ready to express her thoughts and emotions, a programmer starts with an empty screen, ready to express herself through solving a problem by writing code. A painter's ability to express herself depends on her experience with her tools – her brushes, colors, techniques, and especially experience of how to translate her ideas into a painting. Similarly, a computer programmer relies on her knowledge (and choice) of the programming language, her editor, compiler, debugger, and most importantly, her ability to translate her ideas into code, in a clean and clear way.

How does one teach creativity in computer programming – or in a fine art, such as painting or sculpting? And, how does a computer program demonstrate creativity in its code?

Perhaps a more basic question is, "Do we, as educators, want to encourage our computer science students to be creative?" In computer programming, as I stated above, a programmer can be tempted to write code that is "cute": code that solves a problem in an unusual way but which is perhaps difficult to understand. While this may be a creative solution, it is not typically a behavior I encourage. This is especially true when I am teaching beginners to program for the first time. I, as an educator, want to encourage the students to learn and excel at the fundamentals, before I encourage them to be too creative with their solutions. In Berry's essay, quoted above, note that he encourages the application of virtue to one's calling, but only *after* the basic skills have been mastered.

That being said, once students mature in their programming abilities, I do want to allow them to experiment more with creative solutions to problems. Creativity can be encouraged in:

- Design of data structures and class hierarchies. Creative application of a well-known "design pattern" can lead to a solution that is more robust, easier to test and maintain, isolates future changes, uses less memory, and allows more efficient algorithms.
- Design of algorithms. Creative algorithms can improve the efficiency and scalability of a program.
- Design of useful libraries and tools. Useful and creative implementation of libraries, such as logging or debugging libraries, can make programming much easier and productive.

Creative designs are often evident by viewing code. Well-designed, creative class definitions, collections, data structures, and object hierarchies become clear when one sees the simple, elegant code that uses them. Creativity is clearly evident in code when one sees a difficult problem solved elegantly and quickly through a creative algorithm.

Another question is, "How do we encourage students to be creative in their programming?" Research on creativity reveals a few factors that encourage creativity [8, 9]:

- Do not overly specify programming assignments. Do not tell students how functions, modules, classes, or files should be named. Allow students to be creative in those details.
- Give students permission to fail. Creativity is squelched when students are afraid of taking risks and possibly making mistakes. When they are freed from this burden, they are given room to experiment, which can lead to creative solutions. Of course, we don't really want our students to fail, but we

can create a safe environment in which the experience of struggling for a creative solution is rewarded more strongly than the importance of getting a correct answer.

- Impose artificial constraints on a problem. Sometimes artificial constraints on a problem force students to think of creative solutions. As an example, an assignment could ask students to limit all blocks of code to a certain number of lines. This would force students to modularize and create functional abstractions, leading perhaps to more creative and elegant implementations.
- Encourage our students to be bold, have courage, self-confidence, and playfulness. Creativity blossoms out of these characteristics.

Teaching Stewardship in Design and Code

Scripture shows that God entrusted humans with the responsibility to care for, tend to, develop, delight in, and learn from God's good creation. Our relationship with the creation should not be one of reckless use and abuse. The idea that we are given the responsibility to care for, develop, and lovingly use God's creation is termed *stewardship.*

How can the virtue of stewardship manifest itself in the writing of computer code? Adams [10] links stewardship directly to the amount of electricity a computer uses. A computer uses varying amount of electricity based on the behavior of the programs it runs. The behavior of a program can affect the computational power required to execute it through two means: the efficiency of the algorithms implemented in the program, and the efficiency of the data structures used in the program.

Stewardship can be demonstrated in a computer program by the efficiency of the algorithms the program uses. A poorly designed or implemented program can use significantly more processing power than a well designed and implemented program. For example, many sorting algorithms exist, and some have been demonstrated to be significantly less efficient than others. All the sorting algorithms produce the same result – sorted data – but the less efficient ones require significantly more processing power, and thus use more electricity and are less stewardly.

The amount of memory a program uses can also affect its efficiency, and thus the amount of electricity the program requires to run. Students sometimes store data inefficiently. For example, I have seen students store decimal or binary integer values as strings of characters – a very wasteful and inefficient way to store the data. Students will sometimes also store one piece of information in duplicate fields in a data structure or object.

When data is stored inefficiently, the program using it may have to run code to process the data each time it is accessed, thus making the program less efficient.

Also, if the program stores many large data structures in memory and the data is stored non-optimally, the operating system may have to swap out pages of data to disk. Swapping is a very time consuming and inefficient process and can cause a program to have poor responsiveness and execute significantly more slowly. Swapping also requires frequent hard-disk reads and writes, which are expensive and consume significant electricity. Thus, a programmer can demonstrate stewardship by writing code having careful design of data storage.

Finally, students sometimes allocate copies of objects frivolously. Some modern programming languages include automatic garbage collection, thus freeing the programmer from having to free object memory space explicitly. However, excessive allocation of objects will cause the garbage collection thread to execute more often, using processor time, and thus wasting electricity.

Teaching Diligence in Design and Coding

The virtue of diligence is not difficult to define or understand: it is the habit of pursuing some goal with great persistence, stamina, and tenacity. All the other virtues we have investigated above are enhanced by diligence. Diligence, when applied to hospitality, humility, stewardship, and so on, will cause all of these virtues to be more evident in the computer program.

Diligence can be encouraged in the classroom by

- Encouraging students, whether they be introductory or advanced students, to stick with a programming assignment until it is done completely and tested thoroughly. Many students struggle with being diligent in a project right through to the very end when they should be concentrating hard on thorough testing of their code.
- Ask students to spend more time planning a project before beginning to implement it. Often students (and practitioners) begin implementing a solution to a problem before taking the time to define and understand it up front. Encouraging diligence up front can cause shorter implementation times later.
- Allow and encourage students who do not get a good grade on a programming assignment to rework and resubmit the assignment. This will encourage the development of a habit of diligence.

Conclusions

As a Christian educator, I want to teach my students that all aspects of their lives need to be done for God's glory and in obedience to His will: to love Him and to love others. While students may think that the skill of computer programming is "just a skill," the learning of that skill, as Wendell Berry explains, is a stepping-stone for

effectively loving others. Assuming that students want to live virtuous lives, it is useful for an educator to explicitly identify in the classroom how certain virtues can have a real effect on the code the students write and how certain best practices demonstrate certain virtues. Students practicing the virtues of stewardship, hospitality, humility, integrity, honesty, creativity, stewardship, and diligence can change the way they design, write, and test code, resulting in code that more effectively demonstrates love for God and others. Additionally, they will be better prepared to "practice their virtue" through their skill.

[1] Calvin College, "An Engagement with God's World: the Core Curriculum of Calvin College," Apr-2006. [Online]. Available: http://www.calvin.edu/admin/provost/documents/core-curriculum.pdf. Accessed: 04-Aug-2014.

[2] W. Berry, "The Gift of Good Land by Wendell Berry — Flourish." [Online]. Available: http://flourishonline.org/2011/04/wendell-berry-gift-of-good-land/. Accessed: 25-Apr-2013.

[3] V. T. Norman, "Teaching How to Write Hospitable Computer Code," *Dyn. Link J.*, no. 3, pp. 10–11, 2011.

[4] C. Plantinga Jr., *Engaging God's World*. William B. Eerdmans Publishing, 2002.

[5] B. W. Kernighan, *The C Programming Language, 2nd Edition*, 2 edition. Englewood Cliffs, N.J: Prentice Hall, 1988.

[6] F. P. Brooks, *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*, 2nd ed. Addison-Wesley Professional, 1995.

[7] V. T. Norman, "Beauty and Computer Programming," *ACM Inroads Mag.*, vol. 3, no. 1, pp. 46–48, Mar. 2012.

[8] L. Klatt, "Interview with Lew Klatt, Professor of English, Calvin College," Jul-2014.

[9] R. S. Root-Bernstein, *Sparks of Genius: The Thirteen Thinking Tools of the World's Most Creative People*, Early edition. Mariner Books, 2001.

[10] J. Adams, "Carbon Footprints and Computing: Efficiency as Stewardship of God's Creation," *Dyn. Link J.*, no. 3, 2011.