Plagiarism in a Programming Context

Contents

1 Foreword
2 What is Plagiarism?
3 But Code Reuse is Good, Right?
4 Programming is a Craft
5 Crafts Require Practice
6 Coding Practice is a Solitary Pursuit
7 Plagiarism isn't Just Copying
8 The Reality of the Internet
9 In Summary (or TL;DR)

Foreword

Students first encountering a programming course often have some confusion or misunderstandings about what plagiarism means in the context of programming, and why it is unacceptable. The following is an attempt to address each of these misunderstandings.

What is Plagiarism?

Let's start with a definition. According to Merriam-Webster (http://www.merriam-webster.com/dictionary/plagiarism), plagiarism is:

the act of using another person's words or ideas without giving credit to that person: the act of plagiarizing something

That definition is sufficiently general to apply to plagiarism of any kind. Since we are concerned specifically with plagiarism of programming code here, consider the following specialization of the definition:

Plagiarism with respect to programming code is the act of using another person's implementation of an algorithm without giving credit to that person.

The most obvious way to plagiarize programming code is to directly copy it. Unfortunately, computers make this a trivial task. It is up to your own ethics as a programmer to avoid the temptation to copy code.

But Code Reuse is Good, Right?

One of the guiding Principles of good software design is to “Reuse, Reuse, Reuse”. A programmer should never spend time re-developing a tool that already exists in the same form. A single correct tool should be built for every unique task, then those tools should be re-used whenever that task is encountered. The caveat here is that someone has to build the tool the first time. A second caveat is that you cannot expect to correctly use a tool you don’t understand. And in programming, the most sure way to understand how a piece of software works it to write it yourself (if only once, for the exercise). In fact, learning to program is more like learning a craft such as cooking than like learning a liberal art such as history or reading (although there is certainly a language aspect as well). This brings us to the next point:

Programming is a Craft

Computer Science concerns itself with the fundamental capabilities of computing machines and the expression of algorithms in such a way that they can be executed correctly by those machines. The day-to-day business of how we actually express those algorithms to build useful things is the focus of the study of computer programming. Programming itself is more closely akin to a craft than to a science. Computer science studies the things that are possible and then it is up to programmers to make those things a reality. In the same way, physical science taught us about electricity and now electricians and electrical engineers can make useful things with that knowledge.

Programming is a necessary craft for computer scientists — in order to design experiments, and test hypotheses, a computer scientist must be able to interact with the computing machine s/he is studying. Programming provides the set of tools that make this possible.

Programming is a desirable craft for other disciplines — modern advances in mathematics, physics, chemistry, and biology have come as a result of the application of computational power to those areas. This is only possible because programmers applied their tools to those problems. This means that scholars from almost every discipline would be better off if they learned the craft of programming. The tools provided by programming create a “force multiplier” effect — you can do so much more with the help of a computer (through programming) than you can without it.

Crafts Require Practice

To make this point, consider programming alongside another common craft: Woodworking. There are plenty of books available on the subject. The Complete Manual of Woodworking by Albert Jackson, David Day, and Simon Jennings is a good place to start. It contains chapters ranging from “Wood: The Raw Material” to “Joinery” and “Wood Carving”. In Chapter 2, “Designing in Wood”, the authors include a section “Principles of Chair Construction”. The section, like the rest of the book, is fully illustrated. It discusses how the seat angle should relate to the angle of the chair back, and how this is affected by the human body and how it in turn affects the posture (and comfort) of the person sitting in the chair. Suggested measurements are given, and a discussion of methods of joinery involved in different styles of chairs is presented. Even the order of construction is clearly stated.

Would you expect an average person — most likely not having a background in woodworking — to be able to successfully create a quality chair after reading this book and passing a test over the terms and concepts? Probably not. What is missing here is practice. In order to learn the craft of woodworking, one must practice it by building things. Early projects are likely to fail, but over time the woodworker will become more and more proficient through a combination of trial-and-error and repetition.

Just like you would not commission a “book educated” woodworker with no practical experience to build a fine set of furnishings for a fancy dining hall, you would not hire a “programmer” with no actual programming experience to build a payroll application for your company. The results in either case would be disastrous, and the money spent would have been wasted. Programming simply cannot be learned from a book alone. As with any craft, practice is a necessity. Trial and error, struggle, and overcoming difficulty eventually lead to experience (and maybe enlightenment). The basic things get easier, and learning new concepts becomes faster because of this prior experience. The quality of the programmer’s work will improve over time. The “novice” craftsman eventually becomes skilled, and maybe progresses to become a “master” of the craft.

Coding Practice is a Solitary Pursuit

Just as with the woodworking example, programming requires every skill to be practiced in order to achieve mastery. You cannot allow someone else to build the tables while you build the chairs, and then honestly claim that you understand everything that went into building the dining suite. Likewise, a programmer must learn to develop programs from “first principles” by practicing those skills. Programming is a unique craft in that we first build the tools that we use later to build even more powerful tools, and then eventually use those to build the final product. You cannot expect to jump directly to the final product without any understanding of the parts (and tools) used to create it. Of course, you will eventually get to collaborate with a team on a large project, but this only works when every team member trusts that every other team member deeply understands the way each part will interact to
Plagiarizing in programming amounts to going to the local furniture store and buying the dining room suite, then trying to claim it was an original design of your own. In some cases, you might get by with this approach. But when your customer comes back later and wants a unique piece of furniture the likes of which have never been seen before — well, then you will be exposed as a fraud. The craft of programming involves constantly solving new and novel problems by combining the tools and building blocks developed along the way in new and different ways. It requires an intricate understanding of what is happening in every piece of the code in order to create something truly new and different. There is no shortcut to the mastery required for this; it comes only through practice and struggle.

Plagiarism isn't Just Copying

It was stated earlier that the most obvious form of plagiarism is direct copying of code. This is not the only way to end up in violation of the Academic Misconduct Policy in the Plagiarism section. Plagiarism broadly encompasses several related ethical violations that relate to producing code that isn't original. The set of related ethical violations that fall into the "plagiarism" category with respect to grading in a programming course are:

- directly copying code
- even if some changes are made to it
- re-typing someone else's implementation of an algorithm
- even if you are typing the programs independently, but working together on the contents
- re-using someone else's code from a previous semester
- even if you make some changes to it
- using code from the Internet
- even if you type it yourself, not just a copy/paste
- allowing someone else to write your program for you

To sum it up succinctly, it is a violation of the Plagiarism and Academic Misconduct Policy if you turn in code that was not completely produced by your own industry as a result of your own ingenuity.

The Reality of the Internet

The Internet is a modern programmer's most valuable source of reference material; you are encouraged to make use of it as such. A danger posed by this is that you may uncover a solution to the exact problem you are trying to solve (or a nearly identical solution). This leaves you with a serious ethical dilemma. If you use the solution you just discovered, you are in clear violation of the Academic Misconduct Policy. But you had good ethical intentions with respect to searching for the information that led you here. What should you do?

The answer is that you should absolutely not use the code that you have discovered and try to claim it as your own. You have two possible choices:

1. Close the page immediately and (later) produce your own independent solution without consulting it again.
2. Produce your own solution (inspired by, or perhaps making use of the one you have seen) and cite the source of the original version.

The first choice may seem like the "best option" at first — but there is a problem. If the code you saw was reasonably short, there is a good chance it "imprinted" on you when you read through it. It is impossible to "unsee" the solution once you have seen it. There may be no way — even after closing the page — to produce a solution that would not be the same as the one you saw. If this is the case, then option 2 is your only ethical choice. If the code was long enough that you think you can close the page and then produce an original solution, a recommendation would be to leave the room for a while (go get a coffee or something) then come back and see if you can write a solution to the problem without consulting the page again. If you can, you are probably ethically in the clear...

The second choice — cite the source — is the way a situation like this should most often be handled in industry. If the source was a good solution to the problem, and was free of any licensing encumbrance, then you are probably free to use it with just a citation. You should always cite the source of the code, not just for ethical reasons, but also so that you can return to it later for reference in case something needs to be fixed or changed.

In the context of a programming course, the second option (citing the source) is still not ideal, since you have not actually produced the solution yourself (and you didn't gain the experience that comes with struggle). As such, you probably won't receive "full credit" for a solution that wasn't original, but at least you are not in violation of any ethics policies.

In Summary (or TL;DR)

Learning to program is akin to learning a craft like cooking, pottery, or woodworking. You cannot successfully master the necessary skills without significant time spent practicing those skills on your own. Through struggle, trial-and-error, and overcoming difficulty, experience and mastery are gained. Any attempt to "take a shortcut" by using someone else's code, whether by blatantly copying or by re-producing manually is not only dishonest (and a violation of the Academic Misconduct Policy), it is also a waste of time. Your time is too valuable to waste; spend it practicing and improving, not plagiarizing.