IRONCODE: Think-Twice, Code-Once Programming

Mark W. Bailey
Computer Science Department
Hamilton College
mbailey@hamilton.edu

ABSTRACT
To become proficient programmers, novices must develop the skills of writing, reading, debugging, and testing code. We believe that learning to write short pieces of code correctly the first time helps strengthen all of these skills. In this paper, we describe a type of exercise, called IRONCODE, that helps develop the code-once skill. We describe the exercise, the programming environment, its implementation, and our experiences using IRONCODE in a second semester programming class.

Categories and Subject Descriptors
K.3.2 [**Computers and Education**]: Computer and Information Science Education—Computer science education; D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools*;

General Terms
Design, Reliability, Experimentation.

Keywords
Correct Code, Program Reading, Program Writing

1 INTRODUCTION
An important skill that beginning programmers must develop and master is the skill of writing correct code *the first time*. Unfortunately, the tremendous increases in processor speeds has shortened the edit-compile-run cycle to the point that students often rely too heavily on testing. Frequently, students choose to "try" a code snippet — or hack — over careful craftsmanship of a piece of code. In this paper, we present an exercise we use in a second semester programming course to develop this skill that we call "think-twice, code-once programming."

The ability to construct carefully a few lines of code is critical in software development. In teaching programming, we often focus on clean and elegant solutions at the expense of emphasizing the importance of "correct" solutions. This skill is used in not only writing code, but also reading and modifying code. How often do we have to "debug" a piece of code by examination? Such examination requires the same skills that writing "bullet-proof" code does: the ability to understand the interactions of two or more lines of code. The exercise we use to develop "bullet-proof" coding skills we call IRONCODE (in retrospect, we should have called it KEVLARCODE!).

2 RELATED WORK
For many years, instructors have used online environments for programming labs. Bowles used drills and programming quizzes to test programming skills [4]. More recently though, online examinations have been used in both closed and open environments [5, 7, 8, 12]. We often evaluate student progress in courses using tests or drills [2, 1, 3, 9, 10]. In addition, online quizzes have been used in introductory programming courses [11]. Main and Savitch provide software with their text that will automatically test methods for correctness [6]. However, to our knowledge, there has never been any experience with online code correctness exercises reported in the literature.

3 SOLUTIONS IN TEN LINES OR FEWER
In order to develop strong code-once programming skills, we hone a student's skill using small programming problems. Each IRONCODE problem asks the student to write a subprogram. Each problem is one that might have been encountered in an introductory programming course. An important characteristic of IRONCODE problems is that a common solution can be written in ten or fewer lines.

We take many of the early IRONCODE problems from the standard **C** library. These include: string copy (strcpy), string comparison (strcmp), string length (strlen), and character search (similar to strtok). These are good starters because students have used these functions (or a variant) in their programming and may already be familiar with their implementation. Additional problems are taken from classic introductory problems: sum the integers from *M* to *N*, determine if a number is prime, sum of squares, *etc*. Finally, as the semester progresses and student skills improve, we introduce problems relevant to current data structures course material: find the minimum in a stack, determine the depth of a tree, *etc*. Often, these are problems whose solutions have already been presented in class and have been coded recently by the students. These advanced problems not only further develop a student's IRONCODE skills but help reinforce current concepts in the course.

4 IRONCODE LAB ENVIRONMENT

We introduce IRONCODE in data structures, our second semester programming course. Our course introduces students to C++ after a semester of programming in Java. Our course has a dedicated three-hour weekly lab, though we've used IRONCODE during lecture-only formats as well. Since we lecture in the lab, our enrollments are limited to the lab capacity of 26 students.

Students gather in a closed laboratory to solve the weekly IRONCODE problem. Students work individually on online problems. Each IRONCODE problem asks the student to implement a **C**++ function. In a space provided on a web page, the student types their solution to the problem. When the student is satisfied with their solution, he submits it for evaluation. Figure 1 shows a snapshot of the IRONCODE programming environment.

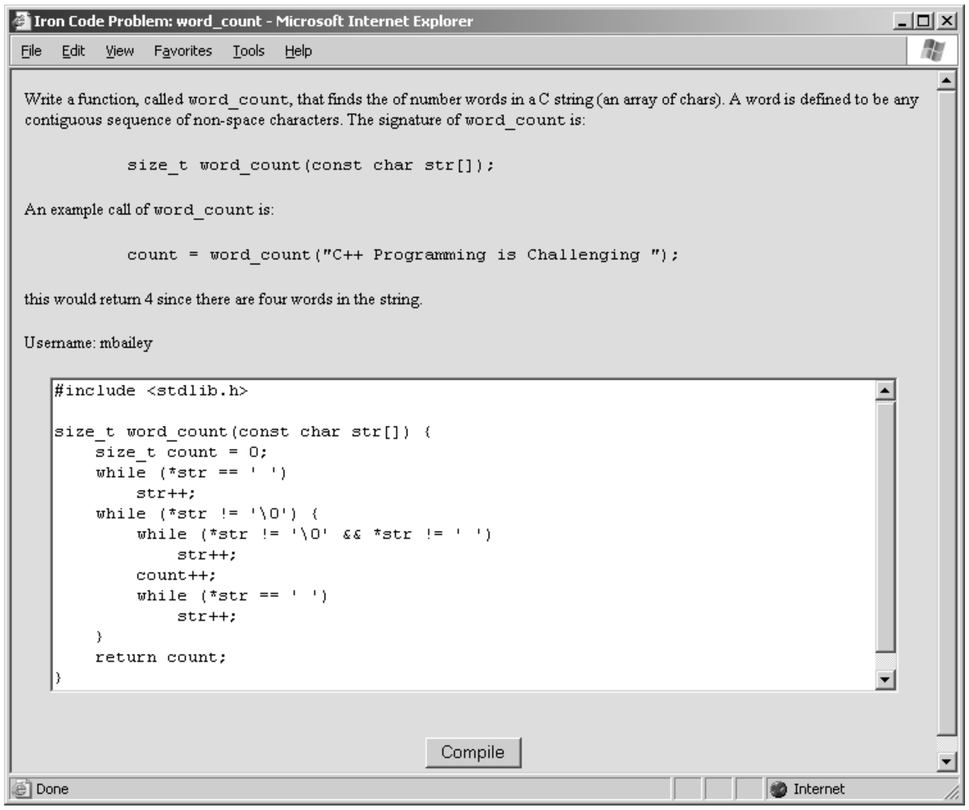
During evaluation, the student's solution is compiled and linked with a test harness. If the solution fails to produce an executable program, the student is notified and asked to make appropriate modifications. However, if the solution produces an executable program, the solution undergoes a series of tests to evaluate its correctness. The solution either passes (the test harness failed to identify a problem with the solution) or fails. At this point, the student completes the exercise by receiving the results. Most importantly, if the student's solution fails, the student may not modify the program to receive credit.

The preceding description will undoubtedly seem unreasonably harsh to many readers. However, it doesn't tell the whole story. Students receive instructor support throughout the exercise. At any time, a student may ask the instructor questions. In particular, questions similar to the following are both allowed and encouraged:

- What result should the function return if it is given an empty string?
- What is the question asking?
- I don't know where to begin. Could I solve the problem by searching through the array?
- When is the increment expression of a `for` loop executed?
- What does this compilation error mean? How do I fix it?

These questions help the students understand the nature of the problem they are trying to solve and deal with the simple technology aspects of the exercise. We encourage significant interaction throughout the exercise. There are, however, questions that we deem inappropriate for the instructor to answer. These include:

- Why doesn't my program work?
- What does my solution return if the parameter is an empty string?


Iron Code Problem: word_count - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Compile
Done
Internet

Write a function, called word_count, that finds the of number words in a C string (an array of chars). A word is defined to be any contiguous sequence of non-space characters. The signature of word_count is:

```
size_t word_count(const char str[]);
```

An example call of word_count is:

```
count = word_count("C++ Programming is Challenging ");
```

this would return 4 since there are four words in the string.

Username: mbailey

```
#include <stdlib.h>

size_t word_count(const char str[]) {
    size_t count = 0;
    while (*str == ' ')
        str++;
    while (*str != '\0') {
        while (*str != '\0' && *str != ' ')
            str++;
        count++;
        while (*str == ' ')
            str++;
    }
    return count;
}
```

Figure 1. The IRONCODE programming environment.

- Should I place this if statement inside or outside the loop?
- What's the value of variable *x* at this location?

Each of these questions focuses on the interaction of a sequence of two or more lines of code. Since the purpose of the exercise is to develop exactly these skills, we gently encourage them to answer the question themselves. Often, however, it is possible to guide the student to a different, more appropriate, question that will help them with their solution.

5 IMPLEMENTATION

In order to implement IRONCODE, we needed an environment that allows students to edit and compile programs, but does not allow them to run or test them. Since we were aware of no such system, we constructed a custom-built system for the purpose.

The IRONCODE environment is built using CGI (common gateway interface), a **C**++ compiler, custom CGI scripts, and testing and support code for the IRONCODE problems. The IRONCODE front page presents the "open" problems. The student selects the appropriate problem and is presented with an IRONCODE coding page. This page always contains an English description of the problem, a function signature (or **C**++ prototype if you prefer), and an HTML text area for entering code.

The student progresses through the system by entering a proposed solution to the problem. When complete, the student presses the "submit" button (this action is verified before proceeding). Upon submission, the content of the text area is uploaded to the web server.

5.1 Results of Submitting a Solution

When a solution is submitted, it is compiled, run, and tested to evaluate its conformance to the problem's specification. The mechanics for this process will be discussed in the next section. During the evaluation process, any of the following outcomes may occur:

- The program doesn't compile—this is the simplest and most common outcome. The student's source has a compilation error. The errors need to be displayed to the student so he can fix the problem.
- The program doesn't link—a linking error will occur if the student failed to provide the requested function, or if the student failed to provide an auxiliary function that he calls. Again, the errors need to be displayed for the student.
- The program runs correctly—in this case, the run of the program is displayed to the student. This always involves displaying the different instances of calls to the function.
- The program runs, incorrectly—in this case, the student has failed to correctly solve the problem. The run of the program is displayed. Each call to their subprogram is displayed, with a correct result and the student's result shown. Incorrect results are clearly marked for examination. In addition to the run, the version of the function that the student was tested against is shown. We do this for two purposes. Usually, the student wants to see a working version so he can learn from his mistakes. Frequently, this demonstrates a far easier solution than the student proposed. We feel it is important for student to be exposed to superb solutions to problems. Finally, in the event that the solution is not "superb" (the solution has an error), this gives the student "appeal" material. Fortunately, this hasn't happened (yet).
- The program runs, but terminates prematurely. This can occur for many reasons. Any number of exceptions—most notably dereferencing a null pointer—can cause a program to terminate. Again, the program has not executed correctly, so the student doesn't receive credit (or another chance) for a correct solution. This outcome is equivalent to the program running incorrectly.
- The program runs, but doesn't terminate (in a timely manner)—one problem with small, execute-once solutions is that infinite loops are common. To overcome this problem, solutions are required to execute in a timely manner (a couple of seconds). After a predetermined length of time (set on a per-problem basis), the program is terminated. This outcome is equivalent to the program running incorrectly.
- Any of the above, but the program has already been graded—after a program has been graded (the program has executed at least once), the student can continue to work on the problem. However, he will not get credit. This option was added after many students indicated that they wanted to polish their solutions (for no credit!).

5.2 Mechanics of Building a Test Program

In order to evaluate a student's solution, it must be tested. Since IRONCODE solutions always consist of one or more **C**++ functions, we can repeatedly call the student's function to determine correctness. Operationally determining correctness requires that for each call to the student's function, the correct result must be known. This can be accomplished in at least two ways:

- For each set of parameters, the correct result(s) may be recorded. This can be handled using an array of records that contains the parameters and the corresponding results.
- For each set of parameters, the correct result(s) may be computed. This requires a correct solution be available in the test harness.

It seems that either approach would work, but we felt that providing a correct solution would be less error-prone than providing correct results, so we opted for the correct solution approach.

Such a test harness will have a solution similar in structure to the code in Figure 2. This approach will work, in theory, but fails for even the simplest problems. For example, this approach limits problems to those that return a single result as a return value. Functions that return results through parameters cannot be included in this test harness. In addition, functions such as **C**'s string copy (strcpy) destructively modify one of the parameters causing subsequent use of the same parameters to produce different return results (consider strcat for example). Further, incorrect execution of one function can impact the subsequent execution of other functions if the stack is corrupted. All of these issues therefore require a more general solution than presented previously.

In our first implementation, we simply skirted the problems of developing a "general" test harness by writing custom test harnesses for each problem. At first this sounds tedious, but often test harnesses can be adopted or adapted from previous IRONCODE problems. For example, the kernel of the test harness for reversing an array of characters is shown in Figure 3.

```
<for each student solution>
    if (student_soln(<parms>) !=
        correct_soln(<parms>))
        <signal incorrect solution>;
<signal correct solution>;
```

Figure 2. A possible structure of a test harness.

```
for (i = 0; tests[i]; i++) {
    strcpy(solution, tests[i]);
    strcpy(theirs, tests[i]);
    solution_reverse(solution);
    start_alarm(2);
    reverse(theirs);
    stop_alarm();
    if (strcmp(solution, theirs)) {
        errors++;
    }
}
```

Figure 3. Test harness for reversing a string.

This illustrates the kinds of contortions that one must go through to test such a simple problem. First, notice that the correct solution is called prior to calling the student's solution. This prevents the student solution from corrupting the stack of the correct solution (for the current iteration of the loop at least!). Second, in the case of string reversal, the single parameter is destructively modified. Thus, a copy of the parameter must be made (strcpy) before calling the first solution. A copy of the result must also be made, or at least it must be placed in a different location. Finally, a type-specific comparison operation (strcmp) must be performed to determine if the result is correct. This technique proves to be acceptable for small numbers of simple IRONCODE problems, but mounting a more ambitious program requires a more robust system.

A second-generation IRONCODE test harness system is in development. This system uses a complex collection of templated **C++** classes to provide a general solution to this problem. In this system, a separate class is required for each set of problems with differing arity (that is, functions taking four parameters use a different templated class than functions that take three). In addition to these classes, we have constructed a set of operators for copying and comparison of common types. The system can model in, out, and in/out parameters and provides a mechanism for registering instances of tests to be run. Figure 4 shows the complete test harness for the reverse example.

6 CONCLUSIONS

We began the IRONCODE initiative because we found that students were exiting our first programming course with a firm understanding of programming through experimentation, but with deficiencies in their understanding of fundamental programming language semantics. For example, it was common for students to misunderstand the intricacies of the for loop. We have found that since beginning the IRONCODE project, students have responded to the close instructor interaction that IRONCODE promotes. We have also found the environment helps the students quickly identify and strengthen their weaknesses.

We have been experimenting with IRONCODE for several semesters. For the most part, the experiment seems to be a great success. Since our courses generally have fewer than 20 students each semester, student performance metrics are not statistically significant enough to evaluate IRONCODE as a pedagogical tool. Instead, we conclude with our observations of how IRONCODE is working in our course.

When students first experience IRONCODE, they often feel quite intimidated and anxious (we have tried to find ways to alleviate this but to date have been unsuccessful). They feel this way even though IRONCODE is not used to determine student grades. However, after a few exercises, most students rise to the challenge of

```
int main() {
    VoidOneParm<char *> TestRun(reverse_solution, reverse, "reverse");

    char *tests[] = {
        "bob",
        "bo",
        "a couple of words",
        "a couple",
        "BOB",
        "Next one is empty!",
        "",
    };

    for (size_t i = 0; i < sizeof(tests) / sizeof(char *); i++) {
        TestRun.addTest(tests[i]);
    }

    return TestRun.performTests();
}
```

Figure 4. A templated test harness for reverse.

IronCode. The success rate for IronCode typically hovers around 50%. This seems to be sufficiently high to encourage students but sufficiently low to challenge them to improve their IronCode skills. By the end of the course, students are more confident about their programming-in-the-small skills and look forward to future IronCode challenges.

REFERENCES

1. J. M. Adams and B. L. Kurtz. A state-of-the-art CS undergraduate lab. In *Proceedings of the SEI Conference on Software Engineering Education*, pages 85–94. Springer-Verlag, 1990.
2. H. P. B. Kurtz. Developing programming quizzes to support instruction in abstract data types. *SIGCSE Bulletin*, 21(1):66–70, February 1989.
3. R. Bennett and J. Wadkins. Interactive performance assessment in computer science: the advanced placement computer science (APCS) practice system. *Journal of Education Computing Research*, 12(4):636–678, 1995.
4. K. Bowles. A CS1 course based on stand-alone microcomputers. *SIGCSE Bulletin*, 9(1):125–127, February 1978.
5. M. Joy and M. Luck. Effective electronic marking for on-line assessment. In *Proceedings of the 6th Annual Conference on the Teaching of Computing and the 3rd Annual Conference on Integrating Technology into Computer Science Education*, pages 134–138. ACM Press, 1998.
6. M. Main and W. Savitch. *Data Structures and Other Objects using C++*. Addison-Wesley, 2nd edition, 2001.
7. D. V. Mason and D. M. Woit. Integrating technology into computer science examinations. In *Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education*, pages 140–144. ACM Press, 1998.
8. M. D. Medley. On-line finals for CS1 and CS2. In *Proceedings of the 6th Annual Conference on the Teaching of Computing and the 3rd Annual Conference on Integrating Technology into Computer Science Education*, pages 178–180. ACM Press, 1998.
9. R. Sanford and P. Nagsue. Selftest, a versatile menu-driven PC tutorial simulates test-taking. *Computers in Education Journal*, pages 58–69, 1992 1992.
10. A. Walworth and R. Herrick. The use of computers for educational and testing purposes. In *Proceedings of Frontiers in Education. 21st Annual Conference. Engineering Education in a New World Order*, pages 510–514, 1991.
11. D. Woit and D. Mason. Enhancing student learning through on-line quizzes. In *Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education*, pages 367–371. ACM Press, 2000.
12. D. M. Woit and D. V. Mason. Lessons from on-line programming examinations. In *Proceedings of the 6th Annual Conference on the Teaching of Computing and the 3rd Annual Conference on Integrating Technology into Computer Science Education*, pages 257–259. ACM Press, 1998.