UNIVERSITY OF CALIFORNIA
RIVERSIDE

A Methodology for Teaching from Student Errors in Computer Science Education

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Adam Thomas Koehler

June 2020

The Dissertation of Adam Thomas Koehler is approved:

Committee Chairperson

University of California, Riverside

ACKNOWLEDGEMENTS

I would like to thank Dr. Vahid for his guidance in the first years of my research, and for allowing me access to the zyBooks tools to create exercises and studies.

To my lab mates, I thank you for the support in this joined experience of graduate school. Special thanks to Justin Gyllen and Kevin Rawson for their guidance in lab interactions and help in crafting my dissertation. Lastly, to Amir Darvishzadeh, thank you for the encouragement as we both proposed, wrote, defended, and finished.

Three individuals were pivotal in my Ph.D. First, Dr. Brylow, my Marquette University advisor, thank you for the support and opening this pathway. Next, Dr. Stahovich, my UCR advisor, thank you for everything. As an advisor, he allowed my independence while also providing guidance. Lastly, Dr. Pilegard, asking her to be on my committee was the best decision of my Ph.D. She provided feedback and help when I needed it most. Her genuine interest in my work created positive reinforcement for my research. She may have saved my dissertation, and for that, I am endlessly grateful.

Finally, I would like to thank my family for everything they have done during all my academic pursuits. And to Ralph and Billy, the countless hours of Hangouts chatter provided everything from mindless distraction to positive encouragement, thank you.

Chapter 2 is a reprint of the material published by the American Society of Engineering Education and presented at ASEE 2016. I was the sole author of the paper published in June 2016. The material in Chapter 3 is based on research presented as a poster at SIGCSE 2019 in February 2019. The material in Chapter 4 is based on research presented as a poster at SIGCSE 2018 in February 2018.

ABSTRACT OF THE DISSERTATION

A Methodology for Teaching from Student Errors
in Computer Science Education

by

Adam Thomas Koehler

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, June 2020
Dr. Thomas F. Stahovich, Chairperson

In education, many assessments boil down to getting the correct solution or necessary result to receive credit. This end goal mentality, in turn, influences how educators transfer knowledge to students. For example, some educators may present or walk through completed solutions. However, continually displaying and using worked-out solutions to teach can quickly become an obstruction to learning. In computer science, a significant amount of learning occurs while fixing the errors that litter the pathway from a blank page to a working solution. This dissertation establishes a methodology for teaching from student errors in computer science.

The first part of the dissertation establishes how we developed over fifty lightweight exercises to integrate into a ten-week course without content replacement. Using past research in computer science, education theory, and cognitive load theory, we developed and refined a standard exercise structure that incorporates student submissions containing erroneous code, past student solutions presented during student-instructor interactions, and instructor feedback. Collectively, our core exercises are known as "What's Wrong With My Code" exercises.

Next, we evaluated the "What's Wrong With My Code" exercises in three distinct ways. First, we performed a study to assess student improvement when using our exercises in place of current course activities (e.g., CodeLab). Second, we analyzed the differences in student error encounters for an entire term by comparing error counts in prior course offerings to offerings with our exercises integrated into the weekly course workload. Lastly, we evaluated student self-efficacy improvements over an entire term by comparing offerings with and without our exercises. In each of the studies, our exercises proved beneficial with increased student performance (with effect sizes of 0.56 and 0.42), increased self-efficacy (p-value < 0.05), and diminished student error encounter rates.

Finally, we used our methodology to implement additional exercises to demonstrate a pathway for use beyond common errors. Specifically, we developed exercises to teach programming style in an introductory C++ computer science course. We evaluated the style exercises alongside data from seven years of submissions, which spanned four different instructional methods of teaching programming style. Our research concludes that students showed increased use of proper programming style before receiving any assessment feedback in academic terms that utilized our exercises. Additionally, we discovered that using an automatic assessment tool with an assigned style grade significantly improves the use of proper programming style.

This dissertation creates a methodology for teaching from student errors in any computer science course, utilizes the methodology to provide multiple implementations and use case examples for an introductory computer science course in C++, and suggests concrete changes for computer science course instructors.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1: Introduction

In education, many assessments boil down to getting the correct solution or necessary result to receive credit. This end goal mentality, in turn, influences how educators transfer knowledge to students. For example, some educators may present or walk through completed solutions. However, continually displaying and using worked-out solutions to teach can quickly become an obstruction to learning. In computer science, a significant amount of learning occurs while fixing the errors that litter the pathway from a blank page to a working solution.

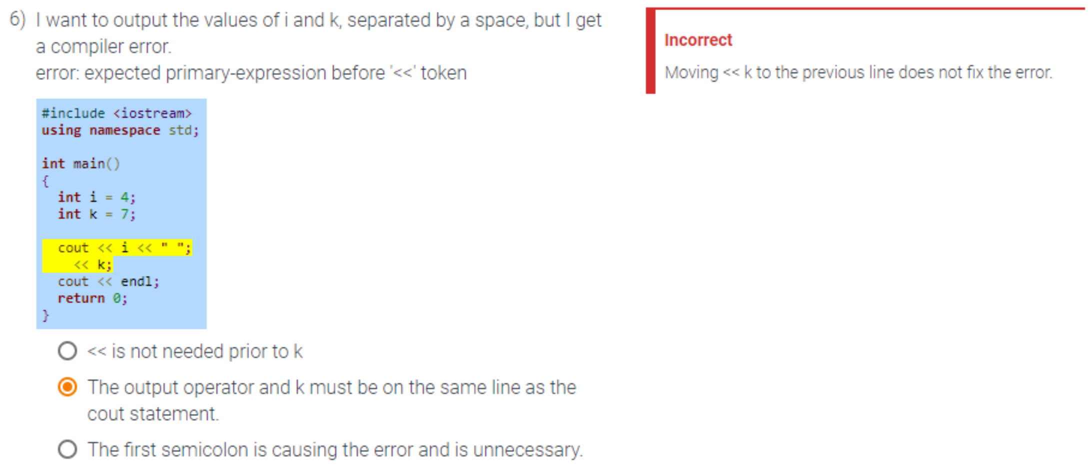
Motivation & Background

Growing class sizes, as well as dwindling resources, have led instructors to pursue different tactics, methods, and implementations in scalable and effective teaching habits. New tools and online environments help recreate standard teaching pedagogy, such as worked-out examples, but they also create an opportunity to introduce alternatives to complement long-standing teaching practices. In the era of growing class sizes, automated assessment of programming exercises helps bridge the scalability gap. However, the open-ended nature of programming assignments can lead to (1) misguided automatic feedback, (2) a disconnection between an errant student solution and proper advice, (3) a complete lack of advice due to the student not understanding the presented question, or (4) knowledge gaps due to students never encountering a problem/feedback pairing. Altering worked-out examples into broken examples for students to fix is a novel way to supplement the overuse of worked-out examples in current teaching methods.

Incorporating teaching from errors into courses is an active pursuit in computer science education (Denny et al., 2012; Du Boulay, 1986; Ginat & Shmalo, 2013; Mathis, 1974; Murphy et al., 2010) and many other fields. However, many computer science explorations have limited impact because the research requires altering an entire course or removing content to add the newly developed content of the researchers. Additionally, a common stance is to offer an entire course on debugging, but students who may take the course are well into their computer science studies. Other fields offer potential implementation pathways for using broken examples or examples with missing pieces. In Chemistry, Barke (2015) incorporated incorrect examples to teach students to balance chemical equations. Rogers et al. (2000) utilize errant examples in a teaching model to correct student misconceptions in electrochemistry. At the grade school level, Durkin and Rittle-Johnson (2012) use correct and incorrect worked-out examples to teach decimal mathematics, and Adams et al. (2014) extend the research to introducing a web-based tutoring system using incorrectly completed examples. In the medical field, Kopp et al. (2008) use misleading or broken examples to help students build diagnostic knowledge. We build on the research both from computer science education and from other disciplines to create building blocks for our methodology.

Cognitive Load Theory

The APA Dictionary of Psychology defines cognitive load as the number of mental resources required to complete a task (VandenBos, 2007). Cognitive load has three distinct types, intrinsic, extraneous, and germane. Intrinsic cognitive load is the load of the actual content and correlates with the amount of effort output by the learner during information consumption. Extraneous cognitive load relates to the presentation style of the material and the effort placed into parsing or interpreting the presentation. Lastly, germane cognitive load is the load necessary to store the knowledge into memory. Cognitive load theory ties cognitive load to problem-solving, a key element of instruction and learning (Sweller, 1988). Chandler and Sweller (1991) use cognitive load theory to guide the design of pedagogy by reducing the complexity and presentation of examples and exercises. Following up on Sweller's work, Renkl (2010) shows that the current use of worked-out examples does not take advantage of prior research in cognitive load theory. Our work builds upon research from Chandler, Sweller, and Renkl to redefine what a worked-out example should look like within an exercise.



6) I want to output the values of i and k, separated by a space, but I get a compiler error.
error: expected primary-expression before '<<' token

```
#include <iostream>
using namespace std;

int main()
{
  int i = 4;
  int k = 7;

  cout << i << " ";
    << k;
  cout << endl;
  return 0;
}
```

- ○ << is not needed prior to k
- [✓ The output operator and k must be on the same line as the cout statement.]
- ○ The first semicolon is causing the error and is unnecessary.

Incorrect

Moving << k to the previous line does not fix the error.

Figure 1: Example "What's Wrong With My Code" exercise.

Developing Exercises: Building on Prior Research

Traditionally, instruction from examples teaches from the correct way of programming by presenting a completely worked-out solution with proper syntax. However, students are not always able to infer the important principles, such as fixing a commonly occurring syntax mishap, from an already correct presentation. In our exercises, we teach from the point of an error. For example, instead of showing a student a proper “cout” statement, we present a broken code attempting to use “cout” and our exercise asks the student to choose a proper fix for the problem. Durkin and Rittle-Johnson (2012) concluded that mixing in exercises with errant solutions alongside normal practice problems and worked-out examples increases student understanding.

Our core group of exercises (Appendix D) adapts cognitive load reduction strategies developed in prior research. Figure 1 contains an example of our exercises. To reduce extraneous cognitive load in exercise presentation, we separate each problem into three distinct parts, 1) the student’s description of the problem combined with an example with erroneous code, 2) several student-produced potential solutions to the described problem, and 3) the instructor feedback area that outlines why a chosen solution is correct or incorrect.

In the first area, we use student descriptions to describe the problem. Presenting a problem from a student’s perspective lowers the intrinsic cognitive load of the problem description. Similarly, we lower the intrinsic and extrinsic cognitive load of the program code by using prior student submissions to represent the problem and stripping the code of an extraneous syntax. According to Große and Rekl (2004), highlighting can reduce cognitive load. In our exercises, we highlight the specific errant line or lines of code to draw the student's attention to problematic lines.

In the second area, we reduce cognitive load by using a multiple-choice layout and presenting potential solutions from a student's perspective. Presenting student-oriented answers in a multiple-choice format builds upon prior research about encouraging student self-explanation (Chi, 1994) and research reducing the cognitive load of self-reflection by providing options to game players (Johnson and Mayer, 2010).

Finally, in the instructor feedback area, we use clear and concise statements at the student's knowledge level to provide feedback on correctness based on the selected multiple-choice answer. Additionally, we color code the feedback. We use red for incorrect responses and green for correct responses. Coloring the feedback helps students quickly pick up on whether the chosen answer is right or wrong.

The design of our exercises aims to reduce the intrinsic and extraneous cognitive load of each exercise. This facilitates the student's understanding of the problem and enables the student to commit the knowledge to memory without experiencing cognitive overload.

Impact

This dissertation makes several significant contributions to computer science education.

1. We create a methodology for incorporating exercises containing erroneous code within computer science education.

2. We implement over fifty exercises for use within introductory computer science courses in C++.
3. We show that utilizing exercises containing erroneous code can increase student performance on short term assessments.
4. We show that utilizing exercises containing erroneous code can reduce the number of errors students encounter.
5. We show that student self-efficacy increases when using exercises containing erroneous code.
6. We utilize our methodology to create exercises to teach programming style within an introductory computer science course.
7. We show that automatically assessing style with immediate feedback is an effective means for getting a student to use proper programming style.
8. We show that our exercises increase student use of proper programming style with and without automatic style assessment.
9. We provide practical implementations and directions for using our methodology to computer science instructors.

Research Inquiries and Hypotheses

We collected our data within the Introduction to Computer Science for Science, Mathematics, and Engineering I at the University of California, Riverside (CS 010 in UCR's catalog). CS 010 is the first computer science course taken by computer science majors at UCR. Colloquially the first computer science course for majors is referred to as "CS 1" because the catalog designation varies by university.

Research Inquiry 1) How does the introduction of exercises containing erroneous code affect student performance?

> **Hypothesis 1a)** We hypothesize that completing exercises containing erroneous code will increase student performance on quizzes in comparison to normal lightweight programming activities.
>
> **Hypothesis 1b)** We hypothesize that the introduction of weekly exercises containing erroneous code will reduce the number of compiler errors directly related to the newly introduced exercises.

Research Inquiry 2) How does the use of exercises containing erroneous code across an entire term affect student self-efficacy?

> **Hypothesis 2a)** We hypothesize that student self-efficacy will increase across the entire term with the addition of weekly exercises containing erroneous code.

Research Inquiry 3) How effective are current strategies at teaching programming style in introductory computer science courses? How can we introduce exercises to take advantage of previously discovered benefits of exercises that contain erroneous code? What effect does the use of exercises containing erroneous style examples have on the student's use of proper programming style for assignments?

> **Hypothesis 3a)** We hypothesize that students will use proper programming style more often when they receive a grade for proper programming style.
>
> **Hypothesis 3b)** We hypothesize that students with better programming style will perform better in the course.

Hypothesis 3c) We hypothesize that the introduction of style exercises that use erroneous code will increase the student's ability to employ proper programming style.

Summary

This dissertation consists of three primary studies. In Chapters 2, 3, and 4, we present our studies in a format similar to the one used by the American Society for Engineering Education. One of our studies (Chapter 2) has already been published as a paper (Koehler, 2016; doi: doi.org/10.18260/p.27196) and is presented with some edits. The other two studies (Chapter 3, Chapter 4) were published as posters (doi: doi.org/10.1145/3287324.3293720, doi: doi.org/10.1145/3159450.3162330) and will be published as full papers. The three studies address the following connected lines of inquiry:

A. *How does the introduction of exercises containing erroneous code affect student performance?*

B. *How does the use of exercises containing erroneous code affect student self-efficacy when used for an entire term?*

C. *How can we introduce alternate exercises to take advantage of previously discovered benefits of exercises that contain erroneous code?*

Chapter 2: What's Wrong With My Code (WWWMC)

Abstract

Student-instructor interaction and passage of knowledge is often optimal in a one-on-one setting[1]. Individual interactions between a student and an instructor form a distinct pathway for the passage of knowledge, including information about topic-specific misconceptions. Unfortunately, these interactions can be time-consuming. Automated assessment and directed educational tools attempt to address the time concerns by presenting the student with immediate feedback but can often be lacking in other ways.

Our paper presents a teaching instrument, "What's Wrong With My Code," that allows us to capture these one-on-one interactions. We provide our design methods used to create the "What's Wrong With My Code" problem sets and analyze the results of studies that utilized the problem sets within a teaching environment.

Introduction

Growing class sizes, as well as dwindling resources, have led instructors to pursue different tactics, methods, and implementations for scalable and effective instruction. The goal is to adapt specific teaching methodologies to accommodate resource and personnel constraints with rising per-class student enrollments. The apprenticeship method is one such teaching strategy that requires adaptation. With this method, a single student learns through direct instructor feedback across various examples through one-on-one

[1] The material in this chapter was previously published at 2016 ASEE Annual Conference & Exposition (Koehler, 2016). This chapter is an edited version of that publication. The material was edited to achieve consistency with the rest of this document and to provide clarification of some details. Also, an appendix was added to present additional details (Appendix A).

interactions. One-on-one interactions help facilitate an excellent teaching environment, and instructors use one-one-one interactions to teach students about programming misconceptions and errors in introductory programming courses. The repetitive nature of a substantial portion of these interactions makes them a prime candidate for improving scalability through automation.

Automated assessment of programming exercises can bridge the scalability gap. However, the open-ended nature of programming assignments can lead to (1) misguided automatic feedback, (2) a disconnection between an errant student solution and proper advice, (3) a complete lack of advice due to the student not understanding the presented question, or (4) knowledge gaps due to students never encountering a problem/feedback pairing. “What's Wrong With My Code” (WWWMC) attempts to emulate the apprenticeship environment and solve these problems by using a much more stringent and guided experience while still maintaining scalability.

Background

As class sizes grow, we must support the instructor-student relationship with growing technologies that allow interaction outside the typical lecture environment. Automated tools are now capable of grading homework and tutoring students on specific topics (Farrel et al., 1984; Vanlehn et al., 2005; Wood, 1996). In recent years a few of note include Web-CAT (Edwards & Perez-Quinones, 2008), Marmoset (Spacco et al., 2006), and Codelab (Arnow & Barshay, 1999). One of the goals of automated assessment is to replace both human-generated grading and feedback with automated testing suites, thus reducing the workload of instructors with growing class sizes. As an example, the

Codelab environment allows automated assessment through grading of a multitude of exercises as well as feedback on errant solutions to guide the student to a correct submission (Arnow & Barshay, 1999). However, automated assessment utilities can have drawbacks based on how general or how specific the tool implementation and use is. Autograders often reduce their ability to provide specific tutoring and feedback as a compromise to create an implementation for a broader set of programming submissions. Therefore, applying and scaling appropriate feedback becomes the most challenging part of scaling automated assessments.

As resources continue to change in computer science education, many old techniques utilized by instructors to distribute knowledge have moved to an online environment. Systems such as Codelab (Arnow & Barshay, 1999) have established online presence as tutoring software for computer science education. Additionally, researchers like Edgcomb, Vahid, and Wood have shown that increasing the interaction level of the tools utilized within computer science education can improve students' scores (Edgcomb & Vahid, 2014; Wood, 1996). For example, Edgcomb and Vahid showed that using interactive web content, such as animations, within an online textbook is more effective than simply migrating a book to a static online version (Edgcomb & Vahid, 2014).

Identifying pitfalls, ranging from those of a specific programming language to pitfalls of an entire computer science class, has also been a long-sought task of researchers (Du Boulay, 1986 and Spacco et al., 2006). When building and analyzing the findings of Marmoset, Spacco et al. set up repositories for student code that allowed the researchers to analyze intermediate student programs and not just final submissions

(Spacco et al., 2006). Garner, Haden, and Robins categorize programming pitfalls into twenty-seven categories built from the error reports gathered within a Java-based CS 1 course (Garner et al., 2005). As we will show in the WWWMC Tool Development section, many of the categories we have chosen for our question sets overlap the outlined categories of Garner et al.

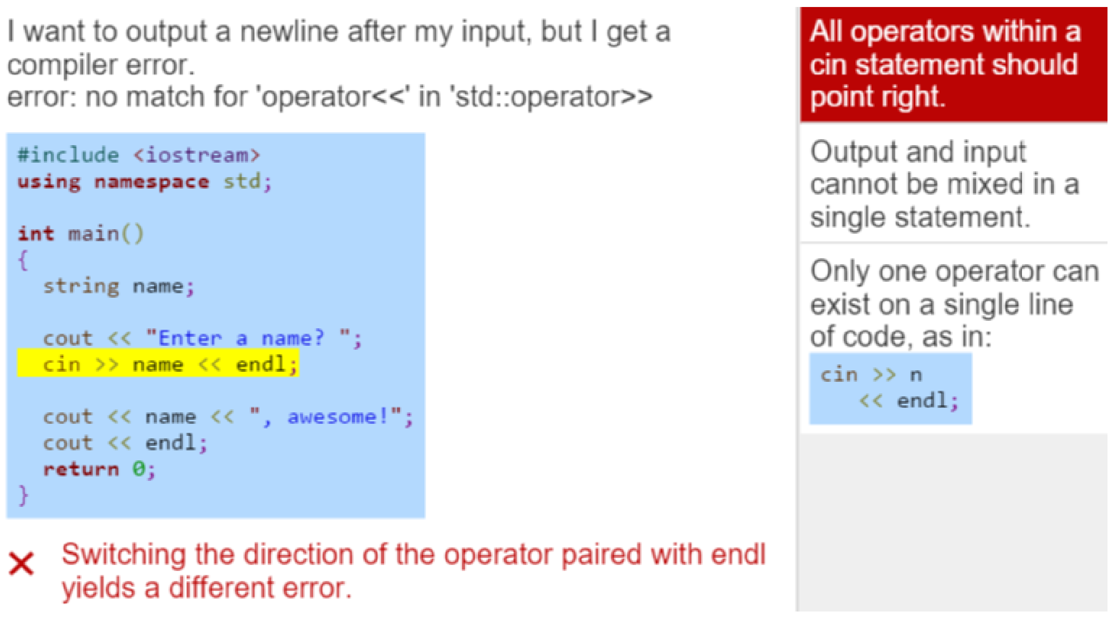
Once misconceptions are known, incorporating pedagogy into courses is an active pursuit of not only computer science education (Denny et al., 2012; Du Boulay, 1986; Ginat & Shmalo, 2013; Mathis, 1974; Murphy et al., 2010) but other educational fields as well (Barke, 2015 and Rogers et al., 2000). Pair debugging by Murphy et al. shows how to incorporate debugging into the commonly utilized pedagogical technique of pair programming (Murphy et al., 2010). Simon et al. developed several videos to help CS 1 students when debugging programs (Simon et al., 2007). Each of these pedagogical techniques has had a modicum of success. However, a few are only feasible in a smaller classroom, and others use different programming languages than C++.

WWWMC Tool Development

Traditionally with student-instructor interactions, a student presents a problem description, problematic code, and potential solutions to the instructor who provides feedback and guidance. This process can be time consuming as mastery of the specific piece of knowledge may require several such interactions. Furthermore, multiple students may ask similar questions, requiring the instructor to provide the same guidance multiple times. The goal of “What's Wrong With My Code”, is to streamline these multi-part and potentially multi-day teaching moments into a single WWWMC question. An individual

WWWMC question allows all students to encounter the same problem and receive feedback.

We created the WWWMC problems to have four distinct parts, as shown in Figure 2. The left column initially contains the problem or question, as well as the problematic code. The right column contains potential answers. Each time a student selects an answer in the right column, the left column updates with an explanation for why the choice is correct or incorrect. Also, we use textual coloring and answer highlighting to emphasize whether the selection is correct or incorrect. All our problems come from a CS 1 course using the C++ programming language.


I want to output a newline after my input, but I get a compiler error.
error: no match for 'operator<<' in 'std::operator>>

```
#include <iostream>
using namespace std;

int main()
{
  string name;

  cout << "Enter a name? ";
  cin >> name << endl;

  cout << name << ", awesome!";
  cout << endl;
  return 0;
}
```

✗ Switching the direction of the operator paired with endl yields a different error.

All operators within a cin statement should point right.

Output and input cannot be mixed in a single statement.

Only one operator can exist on a single line of code, as in:

```
cin >> n
   << endl;
```

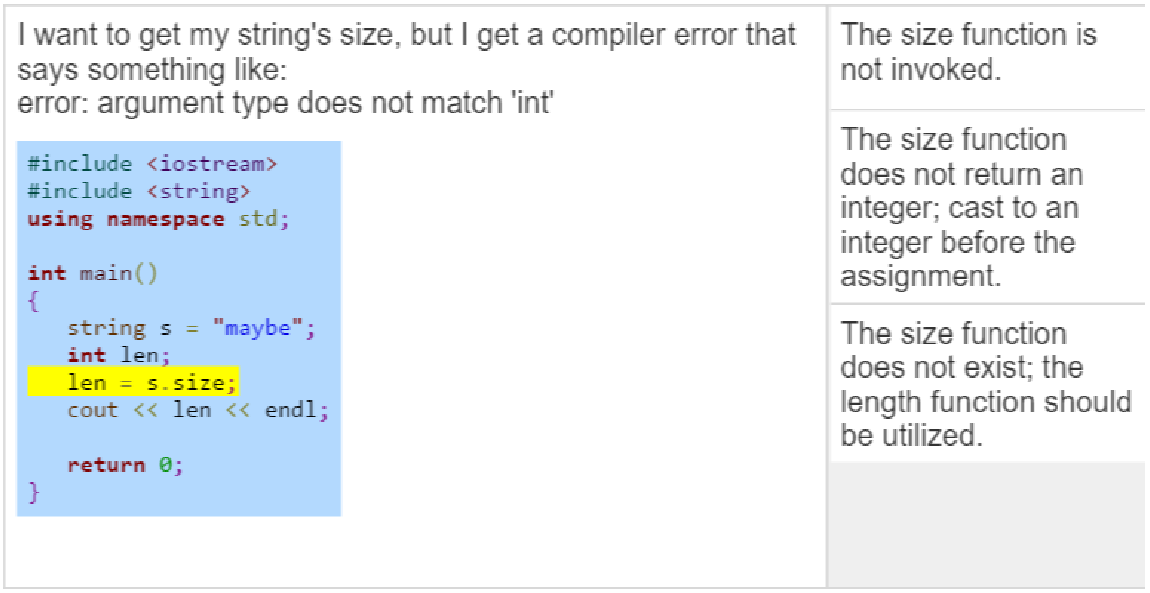
Figure 2: Example "What's Wrong With My Code" problem

The first part of the WWWMC question is a student question paired with code derived from a student submission. We gathered student programming questions from two years from course forum interactions, program submissions, and office hour inquiries. From this, we isolated several pervasive and recurring issues to build our question sets around. The question for each WWWMC problem covers a distinct issue that many students encounter when learning to program for the first time. The problems use C++ syntax, but the concepts are portable to other introductory programming languages, such as Java. We developed over fifty problems and categorized them into various CS 1 teaching topics, as shown in Table 1. Many of the categories and errors overlap with the previous categorization by Garner, Haden, and Robin (Garner et al., 2005). The first aspect of a traditional “What's Wrong” interaction is the student presenting the problem to an instructor. In a WWWMC exercise, we present the problem description and code from the student perspective. We edited the problem description for spelling, grammar, and conciseness. Despite some alterations, we emphasize posing each question as a student would. We trimmed the code portion of the questions to exclude any unnecessary pieces of code to help focus the question but maintained the full program aspect by presenting a piece of code that could be copied and pasted into an editor, compiled, and executed. Lastly, we added code highlighting to focus the student on the error.

The categories shown in Table 1 contain several distinct errors from both runtime and compile-time error groupings. For our purposes, when referring to runtime, we mean any error that occurs during program execution, such as logical bugs or thrown exceptions. Also, we have several style questions to help build a basis of proper style for a novice programmer. For example, in Figure 2, we see a compile-time error from the input and output problems. This is a common error that occurs when novice students attempt to combine input and output statements when first learning. In Figure 4, we see an example of a runtime error resulting from using variables before values have been assigned to them.

1. Style	4. Random Numbers	7. Loops
2. Basic Input & Output	5. Branching	8. Functions
3. Basic Variables and Math	6. String Member Functions	9. Vectors

Table 1: "What's Wrong With My Code" Question Categories



I want to get my string's size, but I get a compiler error that says something like:
error: argument type does not match 'int'

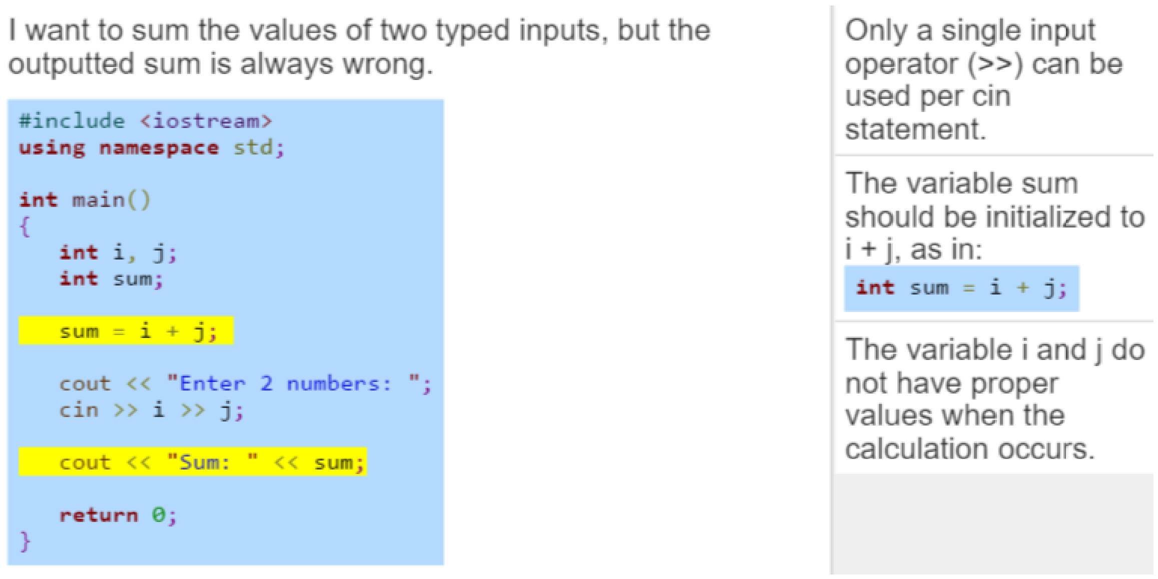
```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string s = "maybe";
    int len;
    len = s.size;
    cout << len << endl;

    return 0;
}
```

The size function is not invoked.

The size function does not return an integer; cast to an integer before the assignment.

The size function does not exist; the length function should be utilized.

Figure 3: Example compile-time error from the String Member Functions category.


I want to sum the values of two typed inputs, but the outputted sum is always wrong.

```
#include <iostream>
using namespace std;

int main()
{
   int i, j;
   int sum;

   sum = i + j;

   cout << "Enter 2 numbers: ";
   cin >> i >> j;

   cout << "Sum: " << sum;

   return 0;
}
```

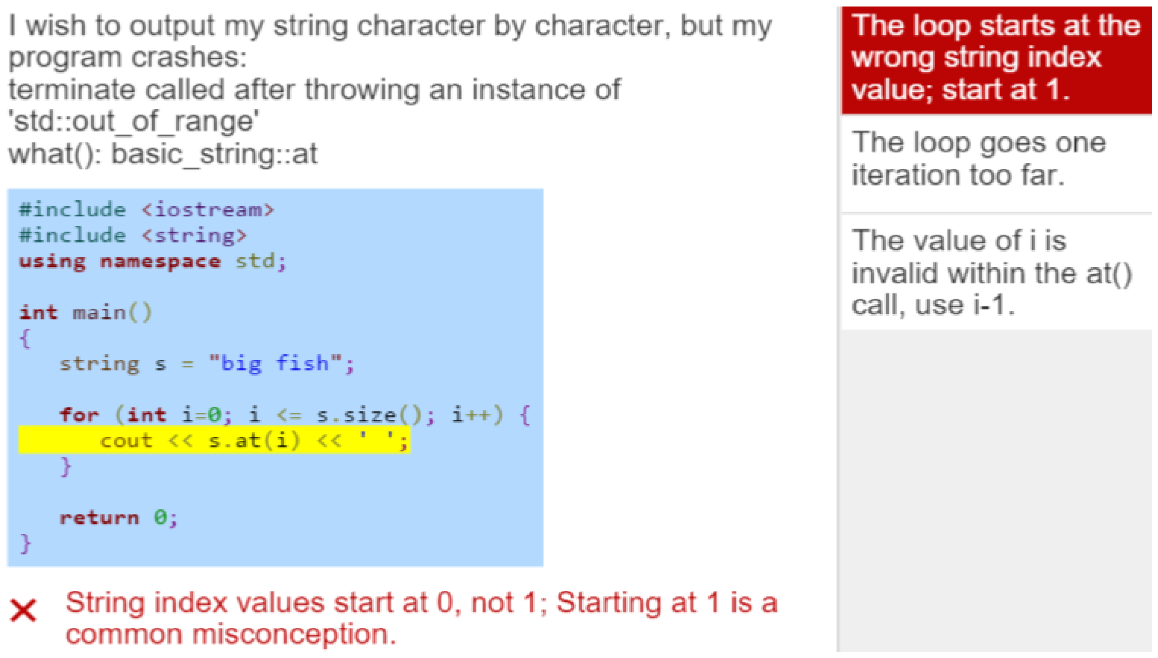
Only a single input operator (>>) can be used per cin statement.

The variable sum should be initialized to i + j, as in:

```
int sum = i + j;
```

The variable i and j do not have proper values when the calculation occurs.

Figure 4: What's Wrong With My Code problem from the Basic Variables and Math category


I wish to output my string character by character, but my program crashes:
terminate called after throwing an instance of 'std::out_of_range'
what(): basic_string::at

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string s = "big fish";

   for (int i=0; i <= s.size(); i++) {
      cout << s.at(i) << ' ';
   }

   return 0;
}
```

✗ String index values start at 0, not 1; Starting at 1 is a common misconception.

The loop starts at the wrong string index value; start at 1.

The loop goes one iteration too far.

The value of i is invalid within the at() call, use i-1.

Figure 5: Example runtime error from the String Member Functions category

If we delve into the String Member Functions category, we see more examples of the mix of runtime and compile-time errors. These include syntax errors, such as leaving off parentheses when invoking a function (shown in Figure 3) and runtime errors, such as the out-of-range exception (shown in Figure 5). A combination of compile-time and runtime errors allows students to learn about multiple types of errors while gaining knowledge of how to fix specific errors.

As shown in Figures 2, 3, 4, and 5, each question has several potential solutions. The solutions use student-centric wording just as in the problem descriptions. As with the questions, we gathered many of the solutions from correct and incorrect student answers to other student's inquiries about incorrect code. Additionally, we derived solutions from correct and incorrect implementations of student programs. Most solutions are
a simple description of what is wrong with the code. Occasionally, a description requires additional clarification with a code snippet (as seen in Figure 4). We present the potential solutions in a multiple-choice layout. Using multiple-choice allows the student to easily navigate through all the explanations by selecting the various solutions. Our approach of providing errant code alongside multiple-choice options exposes the student to questions they might not otherwise encounter during the course.

Each potential solution gives feedback. When the student selects a potential solution, we show the corresponding feedback and explanation. We provide all feedback and explanation statements from the instructor's perspective; we show examples of feedback in Figure 2 and Figure 5. The explanation describes why the solution is correct or incorrect as if an instructor is talking to a beginning programmer. Additionally, the explanation utilizes a green checkmark or red 'x' as well as colored text to denote whether a solution is correct or incorrect.

Lastly, the explanations attempt to avoid using too much computer science terminology. This is done to simplify the explanation and avoid confusing the student with unfamiliar words or programming terms. The goal of the explanation is to provide feedback on the potential solution and why the solution is correct or incorrect, not to teach computer science terminology and definitions. The feedback provides immediate justification for the correctness of a solution, rather than merely marking a question right or wrong.

Study Implementation

We performed our initial study in two quarters, Spring and Fall. In each study, students completed a pretest, a randomly assigned either an experimental or control treatment, and a post-test. The experimental treatment option was the “What's Wrong With My Code” exercises. In the Spring study, the control treatment used several exercises with the Codelab (Arnow & Barshay, 1999) instructional programming environment. In the Fall study, the control treatment used a home-grown automated assessment system to replace submission to the Codelab environment. The exercises used in the Fall study mimicked the Spring’s Codelab exercises but provided only correct or incorrect marking with no guided feedback. We developed the second control treatment for the Fall study because instructors opted not to use Codelab after the Spring quarter. In each study, the control attempts to teach items similar to the WWWMC questions.

Each study contained five parts, presented through a single web page, one part at a time. We integrated the parts with a Google Form to allow data and timestamp collection after each section. The five parts of the study included: background survey, pretest, lesson or instruction, post-test, and follow-up survey. At the beginning of the course, we provide a unique four-digit ID to each student. We used the ID within the study to anonymize the results, assign course credit to the participants, and maintain continuity in data collection across the multiple parts of the study.

The pretest and post-test were the same series of ten questions, comprising three multiple-choice questions and seven free-response essay questions. Each of the seven free-response questions required the student to write no more than a few lines of code. Figure 6 shows two free-response questions. The third item in Figure 6 is an example of a multiple-choice question. We informed participants that all questions in the pretest and post-test are optional. Additionally, we informed all participants that they receive credit based on participation, not performance on either the pretest or post-test.

1. Write an expression that calculates the floating-point value of the fraction 1213 / 57101.

2. Given the code above, write the code to store the floating-point average of x, y, and z in the variable avg.

3. Which of the following is a character literal?

Figure 6: Example written response (1&2) and multiple choice (3) test questions. In question 2, code precedes the question to declare and initialize variables. In question 3, we present several C++ literals as multiple-choice options.

To avoid grading bias toward a specific study, pretest over post-test, or toward a specific lesson, the student answers were all combined regardless of the test, lesson, or quarter. After merging the test responses, we randomized the collected student answers before grading. To establish consistency in grading, we established a standard rubric for each written question allowing partial credit between a score of 0 (no credit) and 1 (full credit).

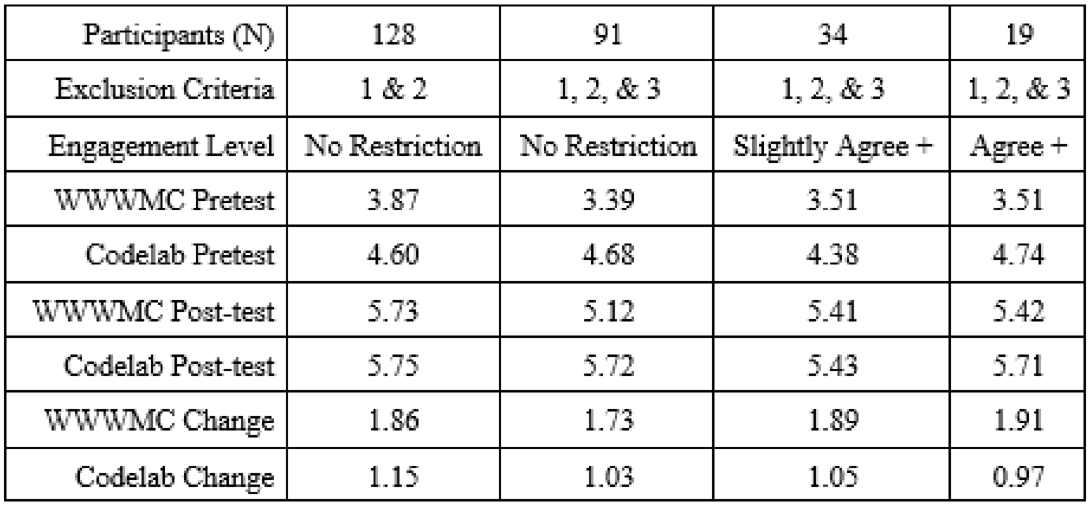
Study Participation Breakdown

Our study population included all students enrolled in the Introduction to Computer Science course at the University of California, Riverside (UCR). Introduction to Computer Science is the first course taken by all computer science majors at UCR. We performed the study in two different quarters, but the on-track Fall quarter had many more computer science majors. Even with the increase in computer science majors during the Fall study, most participants did not have prior programming experience (66% of the 333 participants had no prior experience). Not every participant completed the study correctly. For example, some participants completed the background section and skipped the other sections. Additionally, some students in the course opted to skip the entire study.

The Spring study had 201 total participants, seven of which were computer science majors. The class breakdown of participants was 23% Freshmen, 34% Sophomores, 20% Juniors, 19% Seniors, and 4% being either non-matriculated or outside the typical classifications. The Fall study had 333 total participants, 80 of whom were computer science majors. The yearly distributions were 53% Freshmen, 17% Sophomores, 14% Juniors, 9% Seniors, and 7% other.

After the study, we identified the exclusion criteria. To be included in the study: (1) the student must complete all parts of the study, (2) the student must complete the pretest, lesson, and post-test within eighty minutes, and (3) the student must have had no prior programming experience. The first criterion ensures that the student was engaged in the study enough to complete the five required steps. Our second criterion comes from our study instructions, which asked all participants to complete the study in a single sitting. An eighty-minute allotment allowed twenty minutes each for the pretest and post-test, along with forty minutes for the lesson. Over 90% of the students that completed the study were able to do so in under eighty minutes. Many of the remaining students took several hours to complete all the parts, indicating they did not complete the study in one sitting. Lastly, to control for knowledge variation between students with and without prior programming experience, we included participants with no prior programming experience.

With exclusion criteria 1 and 2, the Spring study had 128 eligible participants. After applying all three exclusion criteria, 91 participants remained. With the Fall study, 267 participants remained after applying exclusion criteria 1 and 2, and 172 participants remained after applying all three exclusion criteria.



Participants (N)	128	91	34	19
Exclusion Criteria	1 & 2	1, 2, & 3	1, 2, & 3	1, 2, & 3
Engagement Level	No Restriction	No Restriction	Slightly Agree +	Agree +
WWWMC Pretest	3.87	3.39	3.51	3.51
Codelab Pretest	4.60	4.68	4.38	4.74
WWWMC Post-test	5.73	5.12	5.41	5.42
Codelab Post-test	5.75	5.72	5.43	5.71
WWWMC Change	1.86	1.73	1.89	1.91
Codelab Change	1.15	1.03	1.05	0.97

Table 2: Scores for Spring study out of 10 points

Results and Analysis

We analyzed the results of the study across three different scores: the average pretest score, the average post-test score, and the performance improvement from pretest to post-test in the Spring and Fall studies. Additionally, during the Spring quarter, we used participant engagement scores, which we gathered during a follow-up survey using a six-point Likert scale question with no neutral option.

As shown in Table 2, both the control lesson and the WWWMC lesson showed improvements to the students' test scores in the Spring study. Students taking the WWWMC lesson posted a statistically better average improvement of 1.73 points (on a scale of 0 to 10) versus students taking the alternate lesson who posted an average improvement of 1.03 points (p-value 0.033). The 1.73 point improvement corresponds to a 48% improvement on the pretest score and an actual grade percentage increase of 17.3%. When examining the post-test for differences to determine if the students achieved a different level of knowledge, we saw no significant difference between the post-test averages for the two groups despite the average score being slightly lower for the WWWMC group than for the Codelab group (p-value 0.162).

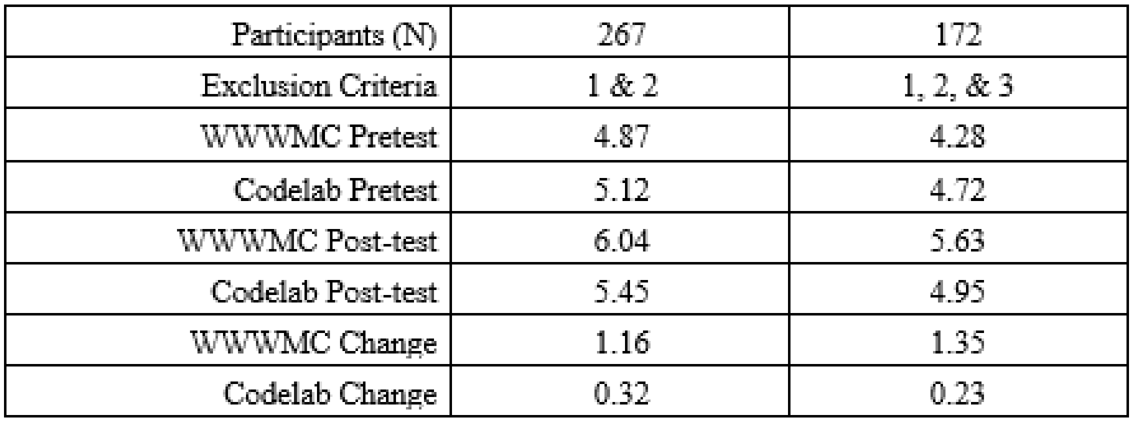
In the Spring study, we also evaluated whether students who were more engaged with each of the lessons benefited more. First, we examined if there was a difference in engagement between the two groups of students and found no significant difference (p-value 0.47). We then compared the performance of students who reported that they at least slightly agreed that they were engaged in the treatment. In this case, the WWWMC group had an average score increase of 1.89, an average pretest score of 3.51, and an average post-test score of 5.41. The Codelab group had an average improved score of 1.05, with averages scores of 4.38 and 5.43 for the pretest and post-test, respectively. These differences were non-significant, but this may be due to small population size, as only 34 students met the criteria for inclusion in this analysis.

When we go one step further and only consider individuals that replied "agree" or "strongly agree" to the engagement question, the scores improve even more. The students in the WWWMC group had an average improvement of 1.91 points with average scores of 3.51 and 5.42 for the pretest and post-test. However, the scores for the Codelab group are similar to those with lower levels of engagement, with an average of 0.97 points of score improvement and pretest and post-test scores of 4.74 and 5.71. However, the differences between the groups are non-significant, most likely due to the small population that met the criteria for inclusion in this analysis.

Table 2 shows that increased engagement corresponds to increased improvements for students in the WWWMC group, but not for students in the control group. While this evidence lacks statistical significance, it suggests that increased engagement in the WWWMC treatment may contribute to improved outcomes.

The Fall study had results similar to those of the Spring study, as shown in Table 3. The WWWMC group showed significant improvement from pretest to post-test (p-value < 0.001). However, for the control group, while there were improvements from pretest to post-test, these were not statistically significant.

The lack of improvement may have been due to the students performing a task that had minimal feedback to help them solve the problem, additionally having only a single exercise to complete may have prevented the students from encountering all the potential errors before completing the submission.



Participants (N)	267	172
Exclusion Criteria	1 & 2	1, 2, & 3
WWWMC Pretest	4.87	4.28
Codelab Pretest	5.12	4.72
WWWMC Post-test	6.04	5.63
Codelab Post-test	5.45	4.95
WWWMC Change	1.16	1.35
Codelab Change	0.32	0.23

Table 3: Scores for Fall study out of 10 points

Students taking the WWWMC lesson in the Fall posted a statistically better average improvement score of 1.16 points (a 24% improvement from the pretest) when using two of the three exclusion criteria. When we apply all three exclusion criteria, the improvement is 1.35 points (a 32% improvement from the pretest). Those are increases of 11.6% and 13.5%, respectively. Additionally, the students with the WWWMC treatment benefited with significantly stronger post-test scores when compared to students with the control treatment (p-value < 0.001).

A common sentiment among the follow-up feedback for the WWWMC treatment stated that students enjoy knowing why something is correct versus only getting points. A common myth, often reinforced by correct/incorrect auto graders, is that if a student got the answer correct, then he or she clearly understands why. The explanations for correct and incorrect answers create teachable moments, reiterating knowledge the student may not have fully grasped. Of the students who took the WWWMC treatment, 78% of the students stated they explored both correct and incorrect answers to read the provided instructor feedback.

The results from Spring and Fall allowed us to conclude that the “What's Wrong With My Code” questions were a positive influence on the success of the students. The WWWMC treatment provided significant changes in scores, often increasing the student's post-test score an entire letter grade. With this knowledge in mind, we moved to the third part of our implementation: integrating WWWMC exercises across the entire introductory computer science course.

Full Course Integration

Working with Zyante (zyBooks, 2015), the research team created a supplemental zyBook for the students of the Winter and Spring 2 quarters. We named the zyBook PreLab to designate the time for the completion of all activities in the supplemental text. The “What's Wrong With My Code” chapter had ten sections (one per week of the quarter) comprising an introduction and nine sections containing WWWMC exercises. The introduction explained the purpose of the WWWMC questions and offered a few tips for effective use of the questions, based on prior feedback from students. Instructors incorporated all of the exercises in our supplemental textbook into the required weekly student workload.

To compare the effectiveness of the fully integrated WWWMC problems to previous studies, we use the survey and post-test from the Spring and Fall studies. With no easy control for the consumption of knowledge beyond the current workload, we did not conduct a pretest in the two terms using fully integrated WWWMC exercises (Winter and Spring 2). We compare the singular survey to the previous post-test scores because students would complete all relevant WWWMC questions before the singular survey. When analyzing the data, we applied exclusion criteria 1 & 3. We modify exclusion criteria 1 to require that all WWWMC exercises be completed before the survey. Because the exercises in this study did not mandate completion in a single sitting, we did not use exclusion criteria 2.

	Winter	Spring 2
Participants (N)	118	147
Exclusion Criteria	1 & 3	1 & 3
WWWMC Test	6.75	6.46

Table 4: Scores for Winter and second Spring study out of 10 points.

Table 4 shows us the results of the Winter and Spring 2 studies. The Winter study had 118 participants, 4 of which were computer science majors. The Spring 2 study had 147 participants, 14 of which were computer science majors. We administered a single test using the post-test questions from the Spring and Fall studies. Students completed the singular test for Winter and Spring 2 in the same timeframe in the academic term as the post-test of the Spring and Fall studies. The average scores (in a range from 0 to 10) for the test administered in the Winter and Spring 2 studies were 6.75 and 6.46 points, respectively. When we compared these results to the post-test results of Fall and Spring, the Winter and Spring 2 scores were significantly better than the Spring and the Fall scores, with p-values less than 0.001 for comparisons with Winter and p-values less than 0.009 for comparisons with Spring 2. These test results show that the complete integration of WWWMC into the course further benefited the students.

In addition to the one test, we collected all the errors that the students encountered during all four quarters. For errors covered within “What's Wrong With My Code” exercises we analyzed whether the percent of students that encountered the error was different during full quarter integration that the percent of students that encountered the error given no exposure to “What's Wrong With My Code” treatment during the quarters before full concept integration. We collected errors by redirecting the compiler (g++) with a script that automatically filled in a Google form when an errant compilation occurred. The redirection script allowed us to gather errors for students, in all four terms, that used the online IDE for the course. When students initially set up the script, we triggered a Google form to show the instructors that a student set up an online IDE workspace. We calculated percentages for each error string in Table 5 from the total number of participants that set up our version of the IDE with compiler redirection. This means we excluded students enrolled in the course who use another IDE because we do not have error encounter information for those students

Error Search String	**Spring 1***	**Fall***	**Winter**	**Spring 2**
expected ';' before '{'	93.69%	59.63%	76.00%	79.39%
no return statement in function returning non-void	47.75%	43.58%	99.11%	97.33%
expected primary-expression before '<<'	63.06%	44.95%	58.22%	60.69%
operator<<	56.76%	67.89%	85.78%	87.02%
operator>>	75.68%	91.74%	96.00%	96.56%
invalid use of member	79.68%	68.35%	83.11%	85.50%
void value not ignored	97.30%	91.28%	100.00%	100.00%
cannot be used as a function	81.98%	69.72%	92.44%	91.98%
assignment of read-only variable	100.00%	95.41%	98.22%	99.24%

Table 5: Percent of students to not encounter the error per quarter, a higher percentage is better. *Spring 1 & Fall data from students not exposed to WWWMC treatment.

In Table 5, we show the percent of students that never encounter the various types of errors in each quarter. The error search strings listed in Table 5 are the strings used to search the compiler output. A few error strings map to multiple compiler errors, such as “operator<<” which will find all errors containing improper use of the C++ output operator. We group these similar errors into a single count because the groups of WWWMC exercises covered several different potential errors on a concept, such as the proper use of the output operator.

As shown in Table 5, full integration does not win out in every comparison, but often, the comparison results are in favor of the quarters with full WWWMC integration. For example, errors containing the input operator >> (row 5 in Table 5) showed significantly reduced encounter rates during the fully integrated quarters of Winter and Spring 2. However, one error that showed a significant reduction in the control quarters was the expected semicolon error (first row in Table 5). For this error, the rates for Winter and Spring 2 (the quarters with WWWMC exercises) were similar, but the percent of students in Spring 1 that did not encounter the error was significantly better than the fully integrated quarters. Similar results that favor quarters without WWWMC exercises include errors for expecting primary expressions before the << operator (row 3 in Table 5) and assignment into a read-only variable (last row in Table 5).

The fully integrated quarters were significantly better in six cases, and Spring 1 was significantly better in two cases (p-values < 0.01). We highlighted the significant differences in Table 5. The rows in Table 5 with the following search strings showed significant improvements: expected ‘;’ before ‘{‘, no return statement, operator<<,

operator>> (Spring 1), invalid use of member, void value not ignored, and cannot be used as a function. The positive results showing the percent of students that never encountered an error help further support our claim that “What's Wrong With My Code” exercises benefit the students. Additional analysis of the association between treatment and errors is in Appendix A.

Future Work

We hope to work with Zyante (zyBooks, 2015) textbooks to create a free online textbook containing all the “What’s Wrong With My Code” problems. At present, the questions are available only upon request. The free zyBook would allow open access to our WWWMC problem sets. We sorted our questions into the concept groups described in the WWWMC Tool Development section. Additionally, we would like to translate the questions into other programming languages leading to an entire zyBook of WWWMC problems with chapters for different programming languages.

We would like to break off a subset of WWWMC specifically for style. Style is often a "learn through experience" endeavor. We believe that using WWWMC type questions can help eliminate some of the programming style frustrations encountered by novice programmers when adapting to the programming style of an instructor.

Lastly, we are investigating whether our “What’s Wrong With My Code” problem set can develop into a more structured tutoring system. We are unsure whether this would create further benefit for students, but it is certainly worth investigating, given our positive results thus far.

Conclusion

The “What's Wrong With My Code” teaching instrument proved beneficial to student learning. In all the studies we conducted, the students showed improvement from pretest to post-test when using the WWWMC lessons. As we continued to develop and enhance this tool, the scores for the students using the WWWMC tool also improved. The improvement to student test scores for the WWWMC lesson participants was significantly better than scores for students taking the control lesson in the Fall study with p-values less than 0.001. Additionally, we show reduced error encounters for errors covered by the WWWMC exercises in the quarters with full integration of our “What’s Wrong With My Code” exercises.

Ultimately, we believe our more guided approach with curated problems and solution-explanation pairings helps minimize some of the known drawbacks in automatically assessing open-ended programming problems. First, our problems provide concise instructor feedback. Second, the streamlined nature of presenting all the pieces a -- the problem the errant code, the solution, and the feedback -- in one exercise helps to reinforce the connection between all the pieces. Third, to make the questions easily understandable, we frame them from the student’s perspective. If the student is still struggling, WWWMC provides the student with the code, solutions, and feedback to provide contextual support in understanding the problem. Lastly, every student that completes an exercise will have seen the problem, solution, feedback coupling. By contrast, students with traditional instructional methods may fail to learn an important piece of knowledge if they happen not to encounter a particular mistake in a traditional programming exercise. In summary, our results demonstrate that the integration of “What's Wrong With My Code” problems within an introductory programming course is beneficial to student learning.

References

Arnow, D., & Barshay, O. (1999, November). WebToTeach: An interactive focused programming exercise system. In FIE'99 Frontiers in Education. 29th Annual Frontiers in Education Conference. Designing the Future of Science and Engineering Education. Conference Proceedings (IEEE Cat. No. 99CH37011 (Vol. 1, pp. 12A9-39). IEEE.

Barke, H. D. (2015). Learners Ideas, Misconceptions, and Challenge. Chemistry education, 395-420.

Denny, P., Luxton-Reilly, A., & Tempero, E. (2012, July). All syntax errors are not equal. In Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education (pp. 75-80). ACM.

Du Boulay, B. (1986). Some difficulties of learning to program. Journal of Educational Computing Research, 2(1), 57-73.

Edgcomb, A., & Vahid, F. (2014, June). Effectiveness of online textbooks vs. interactive web-native content. In 2014 ASEE Annual Conference.

Edwards, S. H., & Perez-Quinones, M. A. (2008, June). Web-CAT: automatically grading programming assignments. In ACM SIGCSE Bulletin (Vol. 40, No. 3, pp. 328-328). ACM.

Garner, S., Haden, P., & Robins, A. (2005, January). My program is correct but it doesn't run: a preliminary investigation of novice programmers' problems. In Proceedings of the 7th Australasian conference on Computing education-Volume 42 (pp. 173-180). Australian Computer Society, Inc.

Ginat, D., & Shmalo, R. (2013, March). Constructive use of errors in teaching CS1. In Proceeding of the 44th ACM technical symposium on Computer science education (pp. 353-358). ACM.

Farrell, R. G., Anderson, J. R., & Reiser, B. J. (1984, August). An Interactive Computer-Based Tutor for LISP. In AAAI (pp. 106-109).

Mathis, R. F. (1974). Teaching debugging. ACM SIGCSE Bulletin, 6(1), 59-63.

Murphy, L., Fitzgerald, S., Hanks, B., & McCauley, R. (2010, August). Pair debugging: a transactive discourse analysis. In Proceedings of the Sixth international workshop on Computing education research (pp. 51-58). ACM.

Rogers, F., Huddle, P. A., & White, M. D. (2000). Using a teaching model to correct known misconceptions in electrochemistry. Journal of chemical education, 77(1), 104.

Simon, B., Fitzgerald, S., McCauley, R., Haller, S., Hamer, J., Hanks, B., ... & Thomas, L. (2007, December). Debugging assistance for novices: a video repository. In ACM SIGCSE Bulletin (Vol. 39, No. 4, pp. 137-151). ACM.

Spacco, J., Hovemeyer, D., Pugh, W., Emad, F., Hollingsworth, J. K., & Padua-Perez, N. (2006, June). Experiences with marmoset: designing and using an advanced submission and testing system for programming courses. In ACM SIGCSE Bulletin (Vol. 38, No. 3, pp. 13-17). ACM.

Vanlehn, K., Lynch, C., Schulze, K., Shapiro, J. A., Shelby, R., Taylor, L., ... & Wintersgill, M. (2005). The Andes physics tutoring system: Lessons learned. International Journal of Artificial Intelligence in Education, 15(3), 147-204.

Wood, S. L. (1996). A new approach to interactive tutorial software for engineering education. IEEE transactions on Education, 39(3), 399-408.

zyBooks. (2015). Animated Interactive Learning. Retrieved February, 2016, from http://www.zyante.com

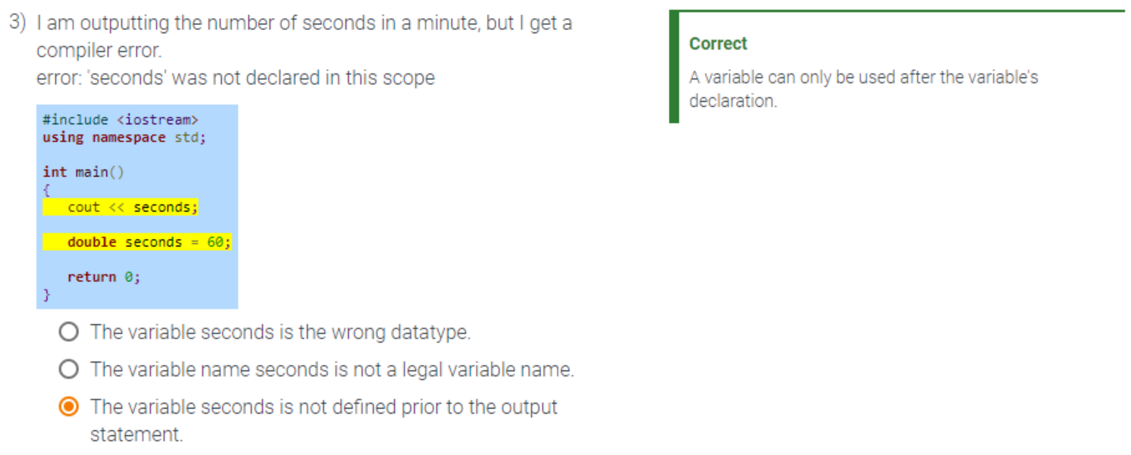
Chapter 3: Improving Student Self-efficacy Using Exercises with Erroneous Code

Abstract

Research has shown that self-efficacy, a student's confidence in his or her ability, contributes to student success. Here, we examine if teaching students to recognize common programming errors increases their self-efficacy in an introductory programming course. We provided students with exercises containing examples of typical novice programming errors and methods for fixing them. At the beginning of the course, we assessed self-efficacy and administered a test to measure programming competence. We repeated these measurements at the end of the course. To assess self-efficacy, students rated their confidence in completing various programming tasks on a scale of 0 to 100. The test had fourteen questions; twelve were short answer questions, and two were multiple-choice questions. Short answer questions required the students to write at most a few lines of C++ code, with most requiring only a single line. We analyzed student self-efficacy in relation to the pre- and post-test, assignment grade, final exam grade, and overall course grade. Our research reinforces prior research demonstrating a correlation between student assessment grades and student self-efficacy. We also found that our exercises provided significant self-efficacy improvements during an experimental term as compared to a control term across our eleven measurers for student self-efficacy (p-value < 0.05).

Introduction

Believing in oneself can go a long way. Throughout lower-level computer science courses, instructors will note that being confident will, in turn, yield benefits to student performance on assessments. However, encouraging students to have confidence in their abilities can only go so far, especially when the students are faced with finding and fixing errors in their code. In our past research, we demonstrated that presenting students with examples of erroneous code and asking them to identify the errors improved students' performance in an introductory computer science course (Koehler, 2016). In Chapter 2, we described our "What's Wrong With My Code" system. An example "What's Wrong With My Code" exercise is in Figure 7, and Appendix D contains the full set of exercises. Figure 7 shows an example of one of these exercises. Here we examine if this system improves student self-efficacy in an introductory computer science course.



3) I am outputting the number of seconds in a minute, but I get a compiler error.
error: 'seconds' was not declared in this scope

```
#include <iostream>
using namespace std;

int main()
{
   cout << seconds;

   double seconds = 60;

   return 0;
}
```

- The variable seconds is the wrong datatype.
- The variable name seconds is not a legal variable name.
- [✓ The variable seconds is not defined prior to the output statement.]

Correct

A variable can only be used after the variable's declaration.

Figure 7: Example of one of our exercises.

Background

Research both inside and outside of computer science has examined student self-efficacy and ways of improving it. The concept of self-efficacy, introduced by Albert Bandura (1986), comprises one's confidence in one's ability to complete a task. Following up on this initial concept description, Zimmerman, Bandura, and Martinez-Pons (1992) studied the interplay between goal setting and student self-efficacy. Outside of computer science, a multitude of discipline-specific studies evaluated the relationship between self-efficacy and error detection (Zamora et al., 2018; King et al., 2013; Kluge et al., 2011; Winne and Nesbit, 2009; and Lorenzet et al., 2005).

Specifically looking at computer science education, we find prior work evaluating course changes and the use of specific instruments or teaching style changes to affect self-efficacy. For example, Kinnunen and Simon (2011) analyzed prior results about student perceptions and self-efficacy evaluations in terms of Bandura's established theories to outline future course modifications and interventions. Ramalingam and Wiedenbeck (1998) developed a 32-item scale to measure programming self-efficacy in introductory computer science courses. Many studies use Ramalingam and Wiedenbeck's all-encompassing scale to evaluate student self-efficacy on a wide range of programming tasks of the sort used in a semester-long introductory computer science course. Wilson and Shrock (2001), Campbell et al. (2016), and Lambert (2015) all evaluated several factors of success in introductory computer science courses, including self-efficacy. All three studies reinforced prior research correlating student performance and student self-efficacy. Etsey and Coady (2017) evaluated whether students understand the connection between exam performance and problem-solving, and they discovered that students who perform poorly on the final exam are also confident that looking at solutions to problems prepares a student to solve future similar problems. Zingaro (2014) determined that peer instruction improves student self-efficacy significantly.

Study Design

Our study evaluates student self-efficacy improvements for students taking a ten-week introductory programming course using the C++ programming language. (Such courses are often called "CS 1" despite the actual designation in a course catalog.) At the beginning of the course, we conducted a survey to assess student self-efficacy across several tasks. Additionally, we administered a test to measure programming competence. We repeated our measurements at the end of the course. We also used a survey to gather additional demographic information such as major and prior programming history. We conducted the study in three different terms, one control term and two experimental terms. Each of the three terms had the same course structure and materials, with the experimental terms adding our erroneous code exercises to the weekly student workload. The same instructor taught all three courses with the same course layout and materials.

Instrument Development

Bandura (2006) outlined a method of developing self-efficacy scales. We utilize Bandura's guide to implement our eleven-question survey to measure student self-efficacy. The survey measures a student's self-efficacy in C++ programming skills learned in a CS 1 course in C++ as well as general skills developed in any CS 1 course. As suggested by Bandura (2006), we asked students to rate their confidence from 0 to

100 on specific tasks after completing a familiarization question. The familiarization question introduces the student to rate their confidence level to complete a common task. Our familiarization question asks students to rate their confidence (from 0 to 100) in their ability to pick up weights of different sizes varying from ten pounds to 500 pounds. Our self-efficacy scale asked students to rate their confidence in the following eleven tasks.

1. Program with proper style.
2. Tell another student what proper style is.
3. Outline a series of steps to solve the problem.
4. Write a program in code-like statements (pseudocode).
5. Write a C++ program to solve the problem on a computer with a compiler.
6. Write a solution to the problem on paper using proper C++.
7. Describe your solution to another student without showing them any code.
8. Discover errors in a program without a compiler (g++).
9. Fix compile-time errors.
10. Fix runtime or logic errors.
11. Describe how to fix an error to another student without providing them any code.

Control (n = 57)		**Experiment 1** (n = 111)		**Experiment 2** (n = 67)	
Week 1	0.948	*Week 1*	0.961	*Week 1*	0.979
Week 10	0.966	*Week 10*	0.956	*Week 10*	0.954

Table 6: Cronbach's alpha calculations per instrument use.

Instrument Reliability

We performed a reliability analysis by calculating the Cronbach's alpha for each use of the 11-item instrument. We performed this calculation six times, once for the pretest and once for the post-test, for all three terms. As shown in Table 6, all of the alphas are over 0.90, with five of the six being over 0.95. Thus, the reliability of our scale is in line with other self-efficacy measurement instruments utilized in computer science and computing education. Ramalingam and Wiedenbeck (1998) developed a 32-item scale to measure programming self-efficacy with an overall alpha of 0.98, and instruments prior to Ramalingam and Wiedenbeck's reported alphas ranging from 0.92 to 0.97 (Torkzadeh and Koufteros, 1994; Murphy et al., 1989; Loyd and Gressard, 1984).

Results and Analyses

We compare results across three terms, all taught by the same instructor. However, the enrolled student population contained zero computer science majors in the control term as well as one of the two experimental terms, and under ten majors in the other experimental term. Since zero computer science majors existed in the control term, we exclude all computer science majors during our analyses across terms. Additionally, to control for variable levels of programming knowledge in students with prior programming knowledge, we exclude all students with prior programming knowledge from our analyses. After the two population restrictions, the control term had 57 participants remaining, the first experimental term had 111 participants remaining, and the second experimental term had 67 participants remaining.

As expected from prior research on student self-efficacy improvements in education, student self-efficacy increased significantly from week 1 to week 10 for all three terms. Table 7 shows the mean and standard deviation of the 11 self-efficacy measures from week 10 for all three terms. The table also includes the gains from week one to week ten. To control for differing week-one scores between the control and experimental groups, we performed an ANCOVA analysis to determine the statistical significance for the differences in the week-10 scores for each of the measures. Additionally, we computed each student's overall self-efficacy score by taking the average of all eleven measures for that student.

We performed an ANCOVA analysis to determine the significance of the difference in the overall week-10 student self-efficacy scores across the three terms (shown in the "All Measures" row of Table 7). For the first experimental term, all measures except measure one show statistically significant improvements in student self-efficacy scores (p-values < 0.05) compared to the control. For the second experimental term, all eleven measures show statistically significant improvements (p-values < 0.05) compared to the control term. As the only difference between the control and experimental course offerings was the inclusion of our exercises, we can conclude that our exercises are beneficial to improving student self-efficacy.

	Control		Experiment 1		Experiment 2	
Measure	Mean	Std. Dev	Mean	Std. Dev	Mean	Std. Dev
1	73.16 (15.79)	20.04	75.85 (34.86)	18.19	74.79* (54.04)	17.17
2	64.21 (15.18)	21.79	69.99** (35.03)	20.51	69.63** (52.97)	20.06
3	66.75 (7.28)	23.33	74.02** (23.60)	21.37	75.67** (44.78)	17.27
4	64.47 (14.12)	20.57	73.19** (35.77)	22.09	72.91** (48.88)	19.25
5	67.95 (6.81)	19.82	76.67** (39.15)	19.15	74.55** (56.87)	18.44
6	63.47 (6.02)	20.10	69.23** (33.30)	20.94	70.00** (51.49)	18.43
7	60.18 (8.26)	21.64	65.46* (30.05)	23.88	65.55** (47.72)	22.91
8	55.70 (8.95)	24.68	63.42** (30.05)	20.79	62.40** (43.85)	22.11
9	65.26 (8.60)	25.723	70.66* (37.64)	21.58	69.75* (52.57)	21.45
10	64.91 (10.61)	22.77	68.55* (35.11)	20.70	68.58** (52.42)	20.87
11	56.14 (8.12)	25.44	62.75** (32.20)	22.07	59.78** (44.34)	24.16
All Measures	63.84 (9.98)	19.38	70.00** (33.34)	17.57	69.42** (49.99)	16.80

Table 7*:* End of term self-efficacy averages per measure with the score change between week 1 and week 10 shown in parentheses.

** indicates p-value < 0.01, * indicates p-value < 0.05

We also evaluated the correlation between student self-efficacy scores and a student’s programming assignment’s grade, midterm grade, and final exam grade. We found that the correlations were significant for all three terms (p-values < 0.05). These findings reinforce previously mentioned related work examining the correlations between learning outcomes and self-efficacy.

Limitations

One issue with our style of study design is the self-selection bias introduced by students completing surveys. To counteract the bias, the instructor provided extra credit to students participating in our study. Our study spanned only three terms, and when we control for the instructor, three possible terms remain for comparison. Selecting courses taught by this instructor resulted in student population controls that eliminated computer science majors and students with prior programming knowledge before our analyses. Having additional terms would allow the selection of a different instructor and potentially allow the inclusion of the two excluded populations.

Conclusion

In this study, we analyze the effect of using exercises containing erroneous code to improve student self-efficacy in an introductory computer science course. Students in the two experimental terms used our “What's Wrong With My Code” exercises as part of their weekly workload. Students in the control term did not use our exercises. Instruction in the three terms was otherwise identical. Students completed a survey at the start and end of the quarter to self-assess their confidence levels from zero to one hundred on eleven different tasks. Self-efficacy improved for all eleven tasks across all three terms.

For ten of the eleven measures and the overall student self-efficacy score, both experimental terms showed statistically significant improvements in student self-efficacy compared to the control term (p values < 0.05). The other measure showed statistically significant improvement in one of the two experimental terms (p value < 0.05). With this evidence, we conclude that our exercises do contribute to the improvement of student self-efficacy in an introductory computer science course. These results demonstrate one more benefit for including our “What's Wrong With My Code” exercises in introductory computer science courses.

References

Bandura, A. (1986). The explanatory and predictive scope of self-efficacy theory. Journal of social and clinical psychology, 4(3), 359-373.

Bandura, A. (1986). Social foundations of thought and action. Englewood Cliffs, NJ, 1986.

Bandura, Albert. "Guide for constructing self-efficacy scales." Self-efficacy beliefs of adolescents 5.307-337 (2006).

Estey, A., & Coady, Y. (2017, June). Study Habits, Exam Performance, and Confidence: How Do Workflow Practices and Self-Efficacy Ratings Align?. In Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education (pp. 158-163).

Campbell, J., Horton, D., & Craig, M. (2016, July). Factors for success in online CS1. In Proceedings of the 2016 acm conference on innovation and technology in computer science education (pp. 320-325).

King, A., Holder, M. G., & Ahmed, R. A. (2013). Errors as allies: error management training in health professions education. BMJ Qual Saf, 22(6), 516-519.

Kinnunen, P., & Simon, B. (2011, August). CS majors' self-efficacy perceptions in CS1: results in light of social cognitive theory. In Proceedings of the seventh international workshop on Computing education research (pp. 19-26).

Kluge, A., Ritzmann, S., Burkolter, D., & Sauer, J. (2011). The interaction of drill and practice and error training with individual differences. Cognition, Technology & Work, 13(2), 103-120.

Koehler, A. T. (2016). What's wrong with my code (wwwmc). In 2016 ASEE Annual Conference & Exposition, number (Vol. 10, p. 27196).

Lambert, L. (2015). Factors that predict success in CS1. Journal of Computing Sciences in Colleges, 31(2), 165-171.

Lorenzet, S. J., Salas, E., & Tannenbaum, S. I. (2005). Benefiting from mistakes: The impact of guided errors on learning, performance, and self-efficacy. Human Resource Development Quarterly, 16(3), 301-322.

Loyd, B. H., & Gressard, C. (1984). Reliability and factorial validity of computer attitude scales. Educational and psychological measurement, 44(2), 501-505.

Murphy, C. A., Coover, D., & Owen, S. V. (1989). Development and validation of the computer self-efficacy scale. Educational and Psychological measurement, 49(4), 893-899.

Torkzadeh, G., & Koufteros, X. (1994). Factorial validity of a computer self-efficacy scale and the impact of computer training. Educational and psychological measurement, 54(3), 813-821.

Winne, P. H., & Nesbit, J. C. (2009). 14 Supporting Self-Regulated Learning with Cognitive Tools. Handbook of metacognition in education, 259.

Zamora, Á., Súarez, J. M., & Ardura, D. (2018). A model of the role of error detection and self-regulation in academic performance. The Journal of Educational Research, 111(5), 595-602.

Zimmerman, B. J., Bandura, A., & Martinez-Pons, M. (1992). Self-motivation for academic attainment: The role of self-efficacy beliefs and personal goal setting. American educational research journal, 29(3), 663-676.

Zingaro, D. (2014, March). Peer instruction contributes to self-efficacy in CS1. In Proceedings of the 45th ACM technical symposium on Computer science education (pp. 373-378).

Chapter 4: Teaching Programming Style in CS 1

Abstract

Students in introductory computer science courses (CS 1) typically receive little formal instruction in proper programming style. Students may gain a limited understanding of proper style by reading code samples, observing an instructor write code with proper style, or by receiving feedback on homework submissions. In our research, we evaluate the effectiveness of an alternative pedagogical approach in which we provide students with brief instruction on proper style, and then students critique and fix examples containing improper style. Our research answers three questions: First, will students use proper style if they know each program receives a style grade? Second, does the student's ability to use proper style correlate with academic performance? Third, do our exercises increase the student's ability to employ proper style? We investigated the first two of our research questions using data from three CS 1 courses with distinct forms of style assessment: 1) no style grading and no feedback, 2) automated style grading with feedback, and 3) hand-graded style with feedback. We investigated the third research question by augmenting the first two course forms with our pedagogical approach. In all courses, students use the same textbook, complete similar assignments, receive a programming style guide at the beginning of the term, and experience similar examples in lectures. We found that simply having a style grade is insufficient. Our results show that automatic assessment with feedback is the best route to proper style adoption. Our approach shows promise in increasing student use of proper programming style on programming assignments with and without automatic assessment.

Introduction

Teaching programming style is an afterthought for many introductory computer science courses (CS 1). Programming style is never the primary subject of a text or lecture, and students learn proper programming style only through years of watching, reading, and doing rather than through instruction and assessment. Students may eventually obtain an understanding of and reasoning behind proper programming style, but the process can leave students frustrated, cause unnecessary delays in debugging, and increase the time needed for instructors to provide help. Teaching and enforcing proper programming style at the CS 1 level can set a student on the correct path of using proper style and reaping the benefits of proper style from the beginning of a student’s computer science education.

In this paper, we present our research that evaluates the usefulness of assessing proper programming style and present our exercises as a teaching implement that integrates into current introductory computer science courses without the need for content replacement. Our research answers three questions: First, will students use proper style if they know each program receives a style grade? Second, does the student's ability to use proper style correlate with academic performance? Third, do our exercises increase the student's ability to employ proper style? We investigated the first two of our research questions using data from three CS 1 courses with three distinct forms of style assessment: 1) no style grading and no feedback, 2) automated style grading with feedback, and 3) hand-graded style with feedback. We investigated the third research question by using our style exercises with the first two of these forms of style assessment.

Background

Dennis Ritchie invented the C programming language in 1974, and the language became popular throughout the 1980s after the first release of *The C Programming Language* (K&R C) in 1978 (Kernighan and Ritchie, 2006). Between the invention of C and the release of K&R C, Kernighan and Plauger (1974) investigated what proper programming style and structure should be and made their arguments with a series of examples. However, a dearth of research into what proper style is and how to teach it has continued to exist after Kernighan and Plauger's work. Jumping a decade, Oman and Crook (1988) separated style into two categories, typographical and structural, in their pursuit of creating better style guidelines and automatic style assessment. Following their work, Oman and Cook (1990) created a taxonomy for programming style and demonstrated, for the first time in decades, the impact of teaching programming style.

Since Oman and Cook's work, the C programming language is now the parent language of many current programming languages, including C++ and Java, which are common programming language choices for introductory computer science courses. To support growing class sizes, a handful of researchers have looked for alternative methods of teaching style, such as automatic assessment tools to help grade style and provide feedback. For example, Ala-Mutka et al. (2005) developed an automatic C++ style analyzer for deployment in introductory computer science courses. In "Effectively teaching coding standards in programming," Xiaosong Li and Christine Prasad (2005) question several students about coding standards to determine the importance of style from the student's perspective. They discovered that students believe in assessing and teaching coding standards, but students prefer to learn through examples and practice. As a follow-up, Li (2006) used peer review as a teaching and assessment method for programming style. Li's case study provided anecdotal results demonstrating the usefulness of peer review. However, a student that does not understand style may not be able to peer review another student's style. We believe teaching style directly through our exercises is a more straightforward approach to build from the results of Li and Prasad (2005).

Previous research has outlined several potential benefits of teaching programming standards. To this prior research, we provide our pedagogical contribution of teaching style through exercises containing erroneous style examples for use at the CS 1 level.

Exercise Development & Implementation

The APA Dictionary of Psychology defines cognitive load as the amount of mental resources required to complete a task (VandenBos, 2007). Cognitive load has three distinct types, intrinsic, extraneous, and germane. Extraneous cognitive load relates to the presentation style of the material and the effort placed into parsing or interpreting the presentation. Chandler and Sweller (1991) use cognitive load theory to outline guidance for exercise creation, concentrating on reducing extraneous load by reducing the exercise presentation complexity. Within our university's version of CS 1, instructors provide a style guide to students as the primary instructional tool for teaching proper programming style. We used Chandler and Sweller's guidance to reduce the style guide from 7 pages of text and examples to create the following 15 concrete rules for proper programming style in our version of CS 1.

1. Indent each block of code 3 spaces beyond the previous indentation level. A block of code is code between a pair of left and right curly braces. One indentation level is three spaces. The global block, function headers, and everything outside function bodies, start with zero indentation.
2. Only one curly brace can exist on a single line.
3. Indent any line that continues from a prior line one level beyond the original line.
4. Comments and vertical whitespace should help identify and visually separate logical code blocks.
5. Initialize variables to literals, not expressions.
6. Variables must have well thought out names, and names are usually nouns.
7. Variables must use camel casing style casing. The first letter is lowercase, and then the first letter in each new word of the variable name is uppercase, such as personName.
8. A conditional expression should not contain any comparisons to the literal values of true or false.
9. All branches and loops must use curly braces to enclose code.
10. The opening curly brace for all branches and loops must be on the same line and one space after the terminating parenthesis of the conditional expression.
11. Indent all terminating curly braces to the initial indentation level, and the curly braces must be alone on the syntax line.
12. Function headers have zero indentation.

13. Functions have an opening curly brace immediately following the header, or the curly brace may go on the line after the function header by itself.

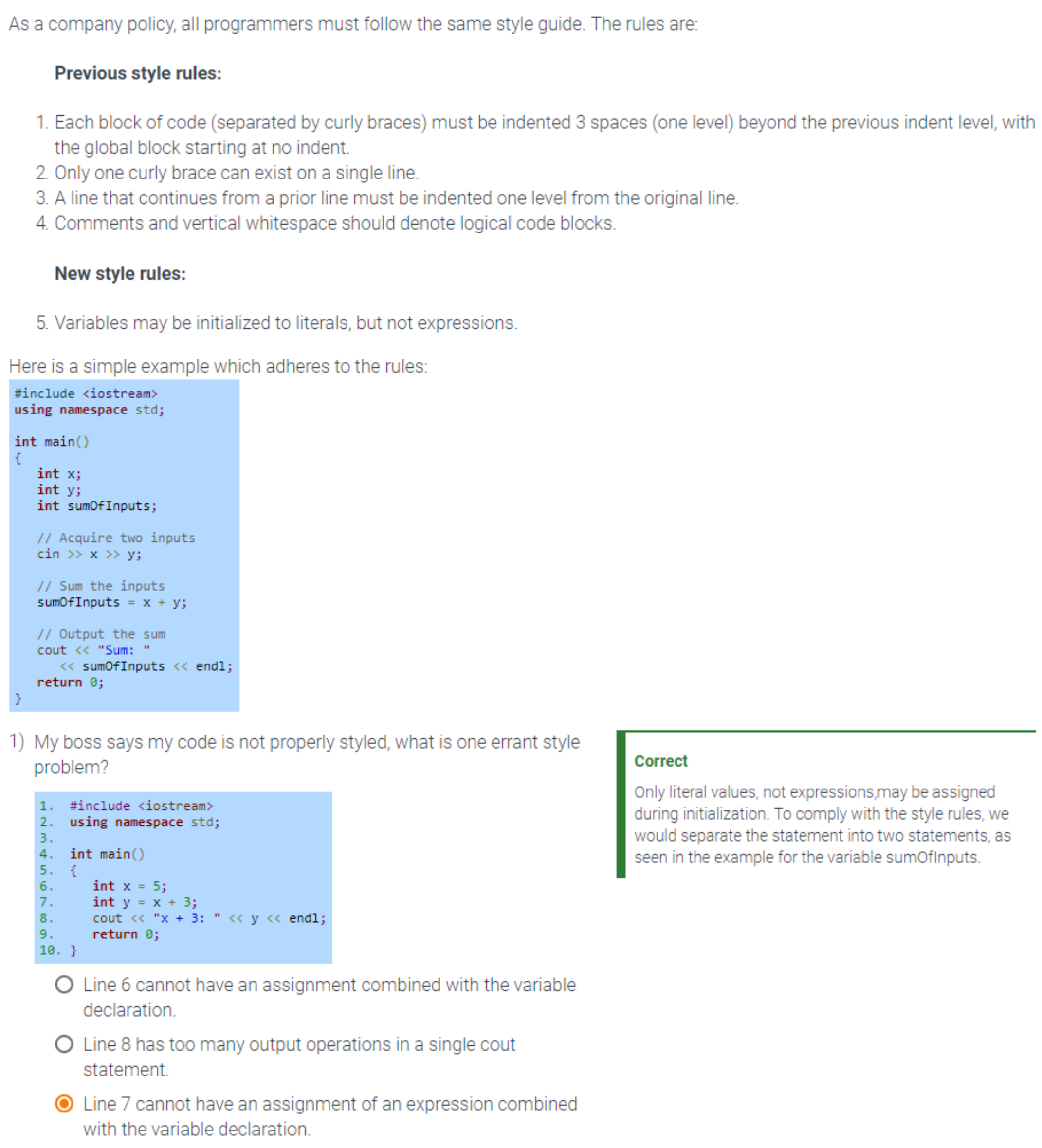
14. If a function header is too long, continue the function header on the next line, but remember to indent the continuation line one indentation level.

15. The function body or implementation code should follow all previously outlined style guides.

Our past research establishes the benefits of teaching with examples of errors and establishes a core exercise structure to use when building exercises (Koehler 2016). We used our prior research to develop eight exercises to teach programming style. We integrated the eight exercises into the weekly student workload across the first five weeks of the term. In our CS 1, all syntax learned after week 5 follows one or more style rules established in the first five weeks of exercises, such as when students learn about vectors. Students complete our exercises shortly after learning new C++ syntax. Figure 8 shows one example of the exercises (all eight are in Appendix E). As illustrated in Figure 8, our exercises present style rules sequentially, and each rule is presented along with a list of rules that have been previously taught. We designed this style of presentation to reduce unnecessary cognitive load.

Research Questions

Our research answers three questions: First, will students use proper style if they know they will receive a style grade on each assignment? Second, does the student's ability to use proper style correlate with academic performance? Third, do our exercises increase the student's ability to employ proper style? We investigated the first two of our research questions using data from three CS 1 courses with distinct forms of style assessment: 1) no style grading and no feedback, 2) automated style grading with feedback, and 3) hand-graded style with feedback. We investigated the third research question by using our style exercises with the first two of these forms of style assessment.



As a company policy, all programmers must follow the same style guide. The rules are:

Previous style rules:

1. Each block of code (separated by curly braces) must be indented 3 spaces (one level) beyond the previous indent level, with the global block starting at no indent.
2. Only one curly brace can exist on a single line.
3. A line that continues from a prior line must be indented one level from the original line.
4. Comments and vertical whitespace should denote logical code blocks.

New style rules:

5. Variables may be initialized to literals, but not expressions.

Here is a simple example which adheres to the rules:

```
#include <iostream>
using namespace std;

int main()
{
   int x;
   int y;
   int sumOfInputs;

   // Acquire two inputs
   cin >> x >> y;

   // Sum the inputs
   sumOfInputs = x + y;

   // Output the sum
   cout << "Sum: "
      << sumOfInputs << endl;
   return 0;
}
```

1) My boss says my code is not properly styled, what is one errant style problem?

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main()
5.  {
6.     int x = 5;
7.     int y = x + 3;
8.     cout << "x + 3: " << y << endl;
9.     return 0;
10. }
```

- Line 6 cannot have an assignment combined with the variable declaration.
- Line 8 has too many output operations in a single cout statement.
- [✓ Line 7 cannot have an assignment of an expression combined with the variable declaration.]

Correct

Only literal values, not expressions,may be assigned during initialization. To comply with the style rules, we would separate the statement into two statements, as seen in the example for the variable sumOfInputs.

Figure 8: Example of an exercise teaching programming style for variables.

Methodology

To compare style across terms, we created a style checker to mark seven types of style errors covered by the fifteen rules. Our checker output the number of style errors per group and the total style errors for a student’s submission. Each of the following style checks is specific to our CS 1 course implementation; other instructors may choose different rules for grading. Our rules, test harnesses, and approach to grading can serve as a model for other approaches to grading style. The Python code we use to test style errors is included in Appendix B. The seven style elements we check are:

1. Global variables. Our CS 1 instructors prohibit the use of global variables. Students may use global variables to circumvent the need to learn scope rules and to learn how to send and return values from functions.
2. Existence of comments. Checking the usefulness of comments is a difficult task. Our CS 1 instructors simply want to make sure students use comments.
3. Line length must be 80 characters or less. Our CS 1 instructors use a common programming standard of avoiding long lines of syntax. Our CS 1 instructors define long lines as syntax lines with more than 80 characters.
4. Indentation must use spaces. Our CS 1 instructors allow only the use of spaces for indentation, and tabs are prohibited.
5. Proper conditional expressions. Our CS 1 instructors do not allow students to create conditional expressions with direct comparisons to literal values of true or false. A conditional expression using the literal value of true or false is improper.

6. One curly brace per syntax line. Our CS 1 instructors require that no more than one curly brace is used on a line to reduce code complexity.
7. Proper indentation. Our CS 1 instructors allow consistent indentation between two and six spaces. Our style checker attempts to determine the number of spaces used for indentation in each file. If a value cannot be determined, the checker uses three spaces as a default.

	Term:	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**
	On-track	▪		▪		▪		▪		
	Style Guide	▪	▪	▪	▪	▪	▪	▪	▪	▪
	Style Exercises							▪	▪	▪
	Style Grade	▪	▪			▪	▪			▪
Feedback on programming style in homework assignments	*Human Generated*	▪	▪							
	Machine Generated					▪	▪			▪
	none			▪	▪			▪	▪	

Table 8: Important characteristics of the course offerings included in this study.

Our study included student program submissions from seven offerings of CS 1. Table 8 lists the important differences between these course offerings. Some of the courses were offered during the Fall quarter. We refer to these as on-track offerings (A, C, E, and G), as that is when first-year students in computer science are supposed to take the course. In all terms, students were provided with a six-page style guide. During our style exercise development, we used this guide as the basis of our fifteen style rules. We deployed our style exercises during three of the terms: G, H, and I. Five terms (A, B, E, F, and I) had programming assignments with a style grade. All of the terms with a style grade incorporated style feedback to justify the assigned style grade. Two types of style feedback were provided to students: human-generated and machine-generated. In terms A and B, instructors provided human-generated feedback on student programming style when hand grading student submissions. In terms E, F, and I, the autograder provided machine-generated style feedback immediately following each student submission. In terms, C, D, G, and H, students received no programming style feedback on their submissions, nor did they receive a grade for style. In terms that used the autograder, students were allowed unlimited attempts to submit programming assignments. When possible, we gathered data from both the first and final submission to the autograder.

Term Type	**Term Description and Submissions Description**	**Style Feedback**	**Style Grade**	**Average Errors per Submission** (Std Dev)	**Average non-Indent Errors per Submission** (Std Dev)
A	Hand-graded: One Submission	After Only Submission	Yes	41.69 (72.66)	12.18 (35.24)
C	Ungraded: One Submission	None	No	41.58 (69.03)	6.06 (23.58)
E	Autograded: Final Submission	After Each Submission	Yes	1.28 (8.33)	0.18 (2.15)
G	Style Exercises: Autograded Final Submission	None	No	26.97 (32.41)	7.42 (12.97)

Table 9: Average number of style errors per student assignment submission.

Will students use proper style if they know each program receives a style grade?

Our first research question asks whether students will use proper programming style if style is graded. To ensure consistent demographics, for this analysis, we consider data from only the on-track terms, A, C, E, and G. For two of these terms, A and E, the students received a style grade on assignments. In term A, students received handwritten style feedback when instructors returned graded assignments one week after submission. In term E, students received machine-generated feedback immediately after each submission to the autograder. In terms C and G, students received no style feedback or style grade. In term G, our style exercises were presented in the textbook alongside other required exercises.

Table 9 provides data about the frequency of style errors in students' programs. To determine the impact of receiving a grade, we compare the frequency of style errors for terms where assignments received no style grade (C and G) to terms where they did (A and E). In all these comparisons, we used a student t-test to determine significance.

Students in term C averaged 41.58 style errors per submission, and students in term A averaged 41.69 errors. This difference is non-significant using a p-value cutoff of 0.05. Students in term G averaged 26.97 style errors per submission. This average is significantly less than that of term A (p-value < 0.01). Students in term E averaged 1.28 style errors per submission. This average is significantly less than the average of term C and the average of term G (p-values < 0.01).

These results do not support the hypothesis that receiving a style grade leads to improved use of style. For example, there was no significant difference in the number of errors between terms A and C, even though only term A include a grade for style. Likewise, there were significantly fewer errors in term G than for term A, even though term A included a grade, but G did not. This latter fact suggests that other factors besides style grade, may have a greater influence on students' use of style. However, the grade does have some influence. We found that for term E, the average number of errors on the first submission was 6.65 and on the last submission was 1.46. The latter is significantly smaller than the former. This suggests that students were explicitly attempting to improve their style. The most likely reason for doing that would be to improve their grade.

The vast majority of style errors were related to indentation. Because of this, including indentation errors in our analysis may obscure our ability to examine student's knowledge of the other style rules. Thus, here we repeat our analysis, excluding indentation errors. Students in term A averaged 12.18 non-indentation style errors per submission while students in term C averaged 6.06. This latter is significantly less than the former (p-value < 0.01). Students in term G averaged 7.52 non-indentation style errors per submission. This average is significantly less than that of term A (p-value < 0.01). Students in term E averaged 0.18 non-indentation style errors per submission. This average is significantly less than the average of term C and the average of term G (p-values < 0.01).

Once again, these results do not support the hypothesis that receiving a style grade leads to improved use of style. For example, there were significantly fewer errors in terms C and G than for term A, even though term A included a grade, but C and G did not.

	Course Grade		**Assignment Grade**	
Term (n)	***All Style Errors***	***non-Indent Errors***	***All Style Errors***	***non-Indent Errors***
E (314)	-0.193*	-0.178*	-0.151*	-0.190*
C (520)	-0.043	0.080	0.114*	0.144*
G (455)	-0.173*	0.015	-0.026	0.014

Table 10: Pearson correlation calculations between style and course/assignments grade. * p-value < 0.01.

Does the student's ability to use proper style correlate with academic performance?

Our next research question is to determine if a student’s ability to use proper style correlates with academic performance. For this analysis, we examine the correlation between the total number of style errors a student made during the term and both the student’s course grade and the average grade across all programming assignments. Table 10 shows the Pearson correlation coefficients and corresponding p-values. As before, we consider only the on-track terms, except for term A, which is excluded because we could not obtain a gradebook for that term. Also, we consider correlations for both all style errors and only non-indentation style errors.

For term E, all correlations are negative and significant (p-value < 0.01). For term C, the correlations with assignment grades are positive and significant. However, the correlations with course grade are non-significant. For term G, only the correlation for the number of style errors (including indentation errors) and course grade is negative and significant. The other correlations are non-significant.

For term E, the number of style errors does correlate significantly and negatively with grade, indicating that a student's ability to employ proper style does correlate positively with performance. For the other terms, there was no strong evidence to support a correlation between proper use of style and performance. Thus, the evidence supporting the hypothesis that proper use of style relates to performance is, at best, mixed.

Term	**Style Autograded**	**Style Exercises**	**Average Errors per Submission** (Std Dev)	**Average non-Indent Errors per Submission** (Std Dev)
C	No	No	41.58 (69.03)	6.06 (23.58)
F	Yes	No	10.61 (27.32)	12.18 (35.24)
G	No	Yes	26.97 (32.41)	7.42 (12.97)
I	Yes	Yes	8.35 (18.76)	12.18 (35.24)

Table 11: Average style grades per submission for style exercise effectiveness evaluation.

Do our exercises increase the student's ability to employ proper programming style?

Our next research question examines if our exercises significantly improve the use of proper programming style. For this analysis, we compare term C to G and term F to I. Terms C and G differ only in that G included our style exercises, but C did not. In both terms C and G, students received no feedback on style and no grade for style. Likewise, terms G and I differ only in that I included our style exercises, but F did not. In both terms F and I, students received machine-generated feedback on style and a grade for style.

Table 11 shows the frequencies of errors for these four terms. For terms F and I, we consider the number of errors on the first submission to the autograder. This represents their ability to apply proper style before receiving feedback from the autograder. On average, students in term C had 41.58 errors per submission, while students in term G had 26.97 errors per submission. This difference is significant at p-value < 0.01. In this comparison, our style exercises have a medium Cohen's d effect size of 0.439, which exceeds Hattie's educational effectiveness threshold of 0.40 (Hattie 2009). Likewise, students in term F averaged 10.61 errors per submission, while students in term I averaged 8.35 errors per submission. This difference is significant at p-value < 0.05. In this comparison, our style exercises have a minimal Cohen's d effect size of 0.097. In both cases, the style scores for the students in terms where our exercises were significantly better than for students in similar terms without our exercises. Therefore, we conclude that our exercises do increase a student's ability to employ proper programming style.

Limitations

Our autograder may be considered too harsh. The most problematic style marking is the score given to indentation errors. Many of the style score averages are drastically reduced when indentation errors are ignored. This is why, in multiple analyses, we provide statistics for style scores with and without indentation errors. Our anecdotal interactions with instructors suggest that improper indentation is the top complaint about student programming style, so we believe the exclusion of indentation from any style assessment would be a misstep. In future research, hand-grading a randomized selection of programming assignments and comparing hand-graded style scores to autograded scores could help alleviate the sentiment that autograding style can be too harsh.

Future

Many anecdotes suggest that proper programming style improves code readability and reduces debugging times. Future investigations should take an empirical look at many of these anecdotes to provide concrete evidence to either back up this gut feeling or disprove it.

Conclusions

We used our "What's Wrong With My Code?" methodology to develop eight style exercises to teach fifteen style rules during the first five weeks of a ten-week course. Our analysis compares data across seven years of programming submissions to answer three research questions.

First, we investigated whether assigning a grade for programming style as part of a programming assignment would result in students using proper programming style. Our analysis compared several terms with and without a style grade. From our analysis, we do not believe that giving a grade for style has a strong effect on students' use of proper programming style. However, we found that there was some effect as we found that students’ final submissions to the autograder were significantly better than their first submissions, with a likely motivation being improving the style grade.

Our second research question investigated whether a correlation exists between students’ use of proper style and students’ performance in a course. For three different on-track term types, we calculated the Pearson correlation coefficients relating student style scores to both the course grade and the average assignment grade. Our results show that a student’s overall performance for a term with autograding did significantly correlate with student performance, but for other terms, there was no strong evidence to support a strong relationship.

Our final research question investigated whether students in terms that included our style exercises employed proper programming style more often than students in terms without these exercises. We examined both terms that used an autograder and those that did not. Our results show that students in terms with our exercises had significantly improved style scores compared to students in other similar terms. These results suggest that our exercises did improve students’ use of proper programming style.

Based on the results in this paper, we recommend that all CS 1 instructors use our exercises or similarly developed exercises to teach programming style. Additionally, if possible, instructors should automatically provide style feedback for each submission and allow students to fix style before the final assignment submission. Finally, we encourage instructors to use a style grade for each assignment as a means of motivating students to fix style problems caught by the autograder.

References

Ala-Mutka, K., Uimonen, T., & Jarvinen, H. M. (2004). Supporting students in C++ programming courses with automatic program style assessment. Journal of Information Technology Education: Research, 3(1), 245-262.

Garner, S., Haden, P., & Robins, A. (2005, January). My program is correct but it doesn't run: a preliminary investigation of novice programmers' problems. In Proceedings of the 7th Australasian conference on Computing education-Volume 42 (pp. 173-180).

Ginat, D., & Shmalo, R. (2013, March). Constructive use of errors in teaching CS1. In Proceeding of the 44th ACM technical symposium on Computer science education (pp. 353-358).

Kernighan, B. W., & Plauger, P. J. (1974). Programming style: Examples and counterexamples. ACM Computing Surveys (CSUR), 6(4), 303-319.

Kernighan, B. W., & Ritchie, D. M. (2006). *The C programming language*.

Koehler, A. T. (2016). What's wrong with my code (wwwmc). In 2016 ASEE Annual Conference & Exposition, number (Vol. 10, p. 27196).

Oman, P. W., & Cook, C. R. (1988). A paradigm for programming style research. ACM Sigplan Notices, 23(12), 69-78.

Oman, P. W., & Cook, C. R. (1990, January). A taxonomy for programming style. In Proceedings of the 1990 ACM annual conference on Cooperation (pp. 244-250).

Renkl, A., & Atkinson, R. K. (2010). Learning from worked-out examples and problem solving.

Li, X. (2006, October). Using peer review to assess coding standards-a case study. In Proceedings. Frontiers in Education. 36th Annual Conference (pp. 9-14). IEEE.

Li, X., & Prasad, C. (2005, October). Effectively teaching coding standards in programming. In Proceedings of the 6th conference on Information technology education (pp. 239-244).

Chapter 5: Conclusions

In this dissertation, we used past research in computer science, education theory, and cognitive load theory to develop and establish a methodology for incorporating examples of student errors into computer science education. Our methodology incorporates five steps:

1. Determine a set of common student errors. These are errors made by differing students term after term, but the errors may be specific to a course or field.
2. Collect a group of prior student submissions containing errors from the set of errors in step 1. Curate each submission such that the submission contains only a single error and minimizes the amount of information presented with the error. For example, a student submission may reduce to just the errant lines of code and a handful of other lines to create relevant a program structure.
3. Gather 3 or 4 student resolutions (only one correct) for each errant submission created in step 2. The student-suggested resolutions are a crucial component to invoking self-explanation.
4. Create an instructor response per resolution from step 3. The instructor response provides feedback for each potential solution and establishes why a solution is correct or incorrect.
5. Use the content created in steps 1 through 4 and our exercise design to create exercises. Spread the exercises across the term and integrate the exercises into the weekly student workload alongside current teaching practices.

We built a standardized exercise structure based on prior research in education, cognitive load theory, and computer science. The structure incorporates student submissions containing erroneous code, past student solutions presented during student-instructor interactions, and instructor feedback. Using our methodology, we created over fifty "What's Wrong With My Code" exercises to teach programming syntax and several additional exercises to teach programming style for use in an introductory computer science course based on the C++ programming language.

Student Performance Improvement Study

In our first study, we evaluated the effect of using our "What's Wrong With My Code" on student performance and reducing students' programming errors. Our study design was based on a pretest, a lesson, and a post-test. We randomly assigned one of two lessons to each student. The first lesson contained typical course programming exercises and used the native course programming environment to complete the lesson. The other lesson was a series of our "What's Wrong With My Code" exercises. In the first term of the study, the students using our exercises showed improvement over the control group in post-test scores. Using the average of the student improvement scores (change from pretest to post-test), we calculated a Cohen's d effect size of 0.56. However, an ANCOVA analysis could not prove the post-test scores to be significantly different between the control group and the experimental group. In the study's second term, the experimental group's post-test scores were significantly better than the post-test scores of the control group (p-value of 0.001). For the second term, the Cohen's d calculation using the average of student improvement scores yielded an effect size of 0.42. Each of the calculated effect sizes exists in the moderate treatment range of 0.4 to 0.6 for education, and both clear Hattie's proposed 0.40 educational importance threshold (Hattie, 2009). The student improvements, combined with the effect sizes, show that our exercises provide beneficial contributions to student learning.

After our positive initial result, we integrated our exercises into the students' weekly course workload in subsequent course offerings. In those offerings, instructors chose to drop the exercises offered to the control group in our first study. After students completed the “What's Wrong With My Code” exercises from the previous study (spread across multiple weeks of instruction), they were given a quiz. The quiz used the questions from the previous study's post-test. However, a controlled pretest could not be completed due to the exercises being spread across the early weeks of the course. Quiz scores of students in courses with the full implementation were similar or better than the average of the experimental group scores for prior studies, suggesting our full implementation was working as expected.

Reduction in Errors Encountered

In this analysis, we cataloged nine types of errors displayed during student compilation using g++. We gathered the errors from four academic quarters, including two control quarters and two experimental quarters, which used our exercises as part of the weekly student coursework. We found that most of the error types occurred less frequently in the quarters using our exercises than in the control quarters. For five of the nine error types, both experimental quarters showed a statistically significant reduction in error encounters ($p\text{-values} < 0.01$), and for another error type, one of the two experimental quarters showed a statistically significant reduction in error encounters (p-value < 0.01) while the other did not. Two error types were more common during the experimental quarters than during the control quarters.

Additionally, we evaluated the association between the treatment (our exercises and the control) and sixteen errors. Our analysis, presented in Appendix A, uses a chi-squared test to show a statistically significant association between our exercises and nine of the sixteen errors. This suggests that our treatment directly affected the error rates on those nine errors.

Improved Student Self-efficacy

In our study on improving student self-efficacy, we compared experimental quarters that used our “What's Wrong With My Code” exercises as part of the weekly student workload to a control quarter that did not. Students completed a survey at the start and end of the quarter to self-assess their confidence levels from zero to one hundred on eleven different tasks. All eleven tasks showed improvement in student self-efficacy across all the quarters. The improvement was statistically significant for ten of the eleven measures and the overall student self-efficacy score in both experimental terms (p-values < 0.05). The remaining measure improved in both terms, but significantly improved in only one of the two experimental terms (p-value < 0.05). Based on this evidence, we conclude that our exercises contribute to the improvement of student self-efficacy in an introductory computer science course.

Using Proper Programming Style

We used our “What's Wrong With My Code” methodology to develop eight programming style exercises to teach fifteen style rules during the first five weeks of a ten-week quarter. Our analysis compares data across seven years of programming submissions. Employing the exercises adds only a minimal amount of additional work to the student workload but yields a Cohen's d effect size of 0.36 for improving the use of proper style, which is lower than Hattie's educational importance threshold of 0.40 (Hattie, 2009). Our analysis shows that students in our experimental term had better style scores on their initial submission to the autograder, even when style was not a part of the overall assignment or course grade. Likewise, when our exercises were used in course with feedback from an autograder and with a grade assigned for style, we observed significantly improved style scores on both the initial and final submissions to the autograder for several programming assignments (p-values < 0.05).

The benefits demonstrated in Chapter 4 and summarized above make our style exercises an ideal choice over the currently established standards of a long style guide or not teaching programming style at all. Furthermore, to encourage even better use of proper programming style, our results indicate that instructors should automatically assess style and provide immediate feedback with the ability to fix the style errors and resubmit. A style grade or style for an assignment should be used to motivate students to fix errors caught by the autograder.

Limitations & Future Directions

Many of our studies rely on students to provide self-evaluation or self-assessment. One known limitation to this style of data collection is self-selection bias. When possible, course instructors provided extra credit to students that participated in our research studies to help reduce self-selection bias, and an alternate option for extra credit was offered to students that did not wish to participate in the study. Self-assessment can also be unreliable. For example, in our self-efficacy study, our control group's confidence scores for the pretest were significantly higher than for the experimental terms, even though we performed the pretests at the same time for all three terms. Self-assessment is necessary for certain types of data, but ensuring students understand the evaluations and are honest is important. Instructors provided participation credit for our studies, and we provided explanations at the beginning of each study to help the student understand that the correctness of their answers was not going to affect their participation credit.

We use CS 1 for all our studies, and to help control for changes in student population across terms, we used several exclusion criteria. Using exclusion criteria can limit the generalization of the results. Future studies should be longitudinal to allow comparison of several similar terms across multiple academic years without exclusion criteria. Additionally, future studies should evaluate courses beyond CS 1.

Instructor changes can lead to uncontrollable threats to validity. We could not control the change of teaching assistants for the course offerings used in our studies, but our analyses compare courses with the same programming assignments, textbook, and primary instructor.

We analyzed error reduction on errors related to or loosely related to the common errors encountered by our students. This narrowing of the field limits our analysis to these specific errors, and thus limits the generality of our findings. Our analysis focused on the frequency of occurrence of various types of errors. It would also be beneficial to examine other measures of performance, such as the effects of our exercises on the time required to fix an error or debug a program.

Contributions

This dissertation establishes a methodology for the incorporation of examples of student errors within instructional materials in introductory computer science courses. We used the methodology to create over fifty lightweight exercises that use erroneous code from students and created a series of similar exercises to teach programming style. Our exercises proved to be effective at reducing students' programming and style errors. These exercises could be easily incorporated into existing introductory programming courses that use the C++ programming language. Our methodology could also be adapted to other topics for use in other computer science courses.

Key Takeaways

With our contributions in mind, we recommend the following for all computer science instructors:

1. Instructors should teach using examples of students' errors. This is a must for future course development at all levels of computer science education. However, teaching using errors should supplement and not replace current instruction.

2. Instructors should consider using our methodology to develop their own exercises to integrate into their courses.

Additionally, the following recommendations apply specifically to instructors teaching introductory courses using the C++ programming language:

3. Our exercises should be distributed across the entire CS 1 course. For instructors using zyBooks, you can file a request to add our “What’s Wrong With My Code” exercises to your book. However, with the sale of zyBooks in 2019, we are not sure how long this option will be available.
4. At a minimum, instructors should consider teaching programming style with our exercises or similarly designed exercises. To further improve students’ ability to use proper programming style, a grade should be assigned for programming style, and student submissions should be evaluated with an automatic style checker, and feedback should be provided immediately.

References

Adams, D. M., McLaren, B. M., Durkin, K., Mayer, R. E., Rittle-Johnson, B., Isotani, S., & Van Velsen, M. (2014). Using erroneous examples to improve mathematics learning with a web-based tutoring system. Computers in Human Behavior, 36, 401-411.

Barke, H. D. (2015). Learners Ideas, Misconceptions, and Challenge. Chemistry education, 395-420.

Chandler, P., & Sweller, J. (1991). Cognitive load theory and the format of instruction. Cognition and instruction, 8(4), 293-332.

Chi, M. T., De Leeuw, N., Chiu, M. H., & LaVancher, C. (1994). Eliciting self-explanations improves understanding. Cognitive science, 18(3), 439-477.

Denny, P., Luxton-Reilly, A., & Tempero, E. (2012, July). All syntax errors are not equal. In Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education (pp. 75-80). ACM.

Du Boulay, B. (1986). Some difficulties of learning to program. Journal of Educational Computing Research, 2(1), 57-73.

Durkin, K., & Rittle-Johnson, B. (2012). The effectiveness of using incorrect examples to support learning about decimal magnitude. Learning and Instruction, 22(3), 206-214.

Ginat, D., & Shmalo, R. (2013, March). Constructive use of errors in teaching CS1. In Proceeding of the 44th ACM technical symposium on Computer science education (pp. 353-358). ACM.

Große, C. S., & Renkl, A. L. E. X. A. N. D. E. R. (2004). Learning from worked examples: What happens if errors are included. Instructional design for effective and enjoyable computer-supported learning, 356-364.

Hattie, J. A. C. (2009). *Visible learning: a synthesis of over 800 meta-analyses relating to achievement*. New York: Routledge.

Johnson, C. I., & Mayer, R. E. (2010). Applying the self-explanation principle to multimedia learning in a computer-based game-like environment. Computers in Human Behavior, 26(6), 1246-1252.

Koehler, A. T. (2016). What's wrong with my code (wwwmc). In 2016 ASEE Annual Conference & Exposition, number (Vol. 10, p. 27196).

Kopp, V., Stark, R., & Fischer, M. R. (2008). Fostering diagnostic knowledge through computer-supported, case-based worked examples: effects of erroneous examples and feedback. Medical education, 42(8), 823-829.

Mathis, R. F. (1974). Teaching debugging. ACM SIGCSE Bulletin, 6(1), 59-63.

Murphy, L., Fitzgerald, S., Hanks, B., & McCauley, R. (2010, August). Pair debugging: a transactive discourse analysis. In Proceedings of the Sixth international workshop on Computing education research (pp. 51-58). ACM.

Renkl, A., & Atkinson, R. K. (2010). Learning from worked-out examples and problem solving.

Rogers, F., Huddle, P. A., & White, M. D. (2000). Using a teaching model to correct known misconceptions in electrochemistry. Journal of chemical education, 77(1), 104.

Sweller, J. (1988). Cognitive load during problem solving: Effects on learning. Cognitive science, 12(2), 257-285.

VandenBos, G. R. (2007). APA dictionary of psychology. American Psychological Association.

Appendix A: Results & Analyses Addendum to Chapter 2

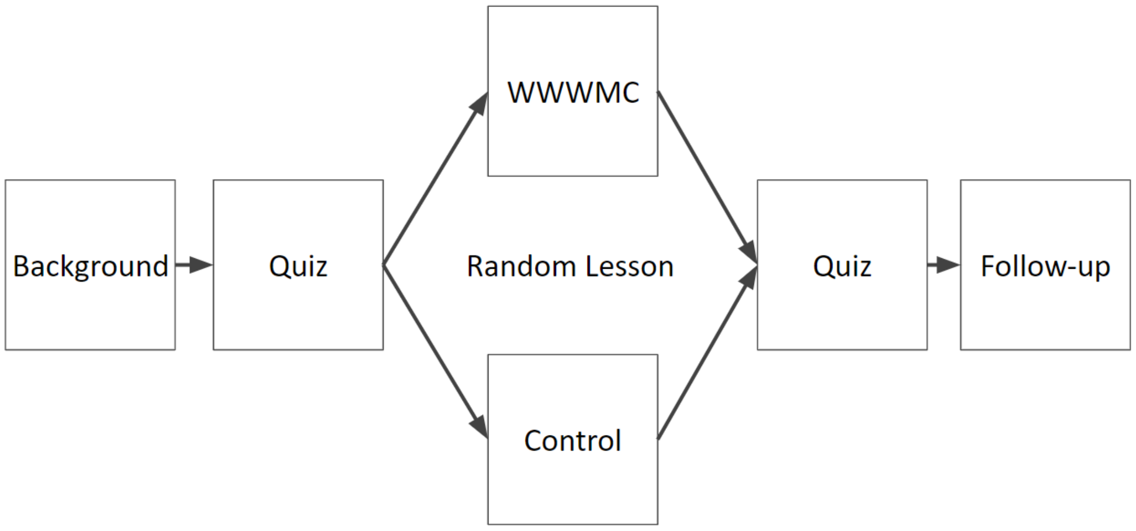


Figure 9: Flow chart outlining study evaluating student performance increase.

Term A – Participants: 108	Pre-	SD	Post-	SD	Δ	SD
Control (54)	4.67	1.79	5.76	1.77	1.09	1.47
What's Wrong With My Code (54)	3.90	1.89	5.92	2.04	2.02	1.81

Table 12: Term A mean scores for pre- and post-test out of 10.

Term B – Participants: 269	Pre-	SD	Post-	SD	Δ	SD
Control (144)	5.13	2.25	5.45	2.18	0.32	1.72
What's Wrong With My Code (123)	4.89	2.10	6.04*	2.45	1.16*	2.25

Table 13: Term B mean scores for pre- and post-test out of 10.

*Significantly different (p-value < 0.001)

ANCOVA Analysis of Student Performance Increases

As discussed in Chapter 2, we performed a study during a single week of the quarter to analyze the performance increases of students on a post-test after taking either a control or experimental lesson containing “What’s Wrong With My Code” exercises. Figure 9 demonstrates the student flow through the study. The results of the study are in

Table 12 and Table 13. Term A utilized a control of Codelab exercises, and Term B used a control of a lab implementation that mimicked the previously used Codelab exercises on a homegrown automated marking and feedback system. We gave a ten-question pre- and post-test before and after the student completed the randomly assigned lesson. The ten-question test contained seven written and three multiple-choice questions. Each question was worth one point, and we gave partial credit to all written questions.

Due to the differing pretest scores, we performed an additional analysis to determine if statistically significant differences exist between the two groups of student post-test scores. We performed an ANCOVA analysis on the post-test scores comparing students in the control lesson group to students in the experimental lesson group using the pre-test as the covariate. Student scores for the "What's Wrong With My Code" experimental lesson were higher than the student scores for the control lesson in both Terms A and B. However, the ANCOVA analysis shows that only Term B qualified as a statistically significant improvement (p-value < 0.001).

Effect Size Analysis

In addition to the ANCOVA analysis, we evaluated the effect size of the "What's Wrong With My Code" lesson using the average of all the student score improvements from pretest to post-test. The delta column in Table 12 and Table 13 displays the average improvement out of 10 points. For Term A, we achieve an effect size of 0.56, and for Term B, we achieve an effect size of 0.42. Both of our calculated effect sizes meet Hattie's educational effectiveness threshold of 0.40 (Hattie, 2009).

Treatment # (%) of students that did not encounter						
	Error Search String	Control n=329	WWWMC n=487	Chi-Square	p-value	Phi
1	no input files	10 (3.04%)	29 (5.95%)	3.667	0.056	0.067
2*	expected ';' before '{'	234 (71.12%)	379 (77.82%)	4.715	0.030	0.076
3*	invalid operands of types 'double' and 'int' to binary 'operator%'	266 (80.85%)	318 (65.30%)	23.343	0.000	-0.169
4	was not declared in this scope	39 (11.85%)	76 (15.61%)	2.283	0.131	0.053
5*	no return statement in function returning non-void	148 (44.98%)	478 (98.15%)	310.723	0.000	0.617
6*	expected primary-expression before '<<'	168 (51.06%)	290 (59.55%)	5.740	0.017	0.084
7*	operator<<	211 (64.13%)	421 (86.45%)	55.980	0.000	0.262
8*	no matching function for call to	220 (66.87%)	383 (78.64%)	14.115	0.000	0.132
9*	invalid use of member	237 (72.04%)	411 (84.39)	18.341	0.000	0.150
10*	invalid conversion from 'const char*' to 'char'	288 (87.54%)	382 (78.44%)	11.065	0.001	-0.116
11	invalid operands of types 'double' and 'double' to binary 'operator%'	278 (84.50%)	421 (86.45%)	0.607	.436	0.027
12*	void value not ignored	307 (93.31%)	487 (100%)	33.468	0.000	.203
13	invalid operands of types 'int' and 'double' to binary 'operator%'	297 (90.27%)	439 (90.14%)	0.004	0.951	-0.002
14*	cannot be used as a function	242 (73.56%)	449 (92.20%)	52.597	0.000	0.254
15*	operator>>	284 (86.32%)	469 (96.30%)	27.459	0.000	0.183
16	assignment of read-only variable	319 (96.96%)	481 (98.77%)	3.337	0.068	0.064

Table 14: Chi-Squared scores and Phi calculations.
*Significant difference, p-value < 0.05

Error Reduction - Chi-Squared Analysis

We gathered all g++ error output across four terms, two control terms, and two experimental terms. In the experimental terms, students completed our “What’s Wrong With My Code” exercises as part of their weekly course workload. We counted specific errors encountered by each student by searching for a specific string in a log of all the g++ compilation results collected over an entire term. Each of the sixteen different search strings (seen in Table 14), identifies a different error output by the g++ command. For our analysis, we performed a chi-squared test to determine if one of the specific errors is associated with either the control group or the experiment group. Table 14 contains the chi-squared and Phi scores for all sixteen error strings. Additionally, Table 14 contains the number of students that did not encounter the error and the percent of the total students for the control or experimental grouping.

Five error strings (numbers 1, 4, 11, 13, and 16) did not have a significant association with either the control or experimental offerings. The five error strings range from highly encountered errors to barely encountered errors. At the high end, error string number 1 describes the error output by g++ when the input file name has a typo. Students in both types of offerings encountered this at a high rate, with only 3.04% of control group students and 5.95% of experimental group students not encountering the error. On the other end of the spectrum, error string number 16 describes the error output by g++ when a programmer attempts to change the value of a constant variable after the constant is initially set. For error number 16, 96.96% of control group students and 98.77% of experimental group students did not encounter the error.

Two error strings (numbers 3 and 10) are significantly associated with the control offerings of the course (p-values < 0.001). For error string number 3, 80.85% of control group students and 65.30% of experimental group students did not encounter the error for mismatched modulo operands. For error string number 10, 87.54% of control group students and 78.44% of experimental group students did not encounter the g++ error output for a string literal assignment into a character variable

Nine error strings (numbers 2, 5, 6, 7, 8, 9, 12, 14, and 15) are significantly associated with the experimental offerings of the course (p-values < 0.001 to 0.05). The differences in percentages of students that did not encounter one of the individual errors ranged from 6.69% on error string number 2 to 53.17% on error string number 5.

Conclusion

Overall, our additional analyses reinforce the findings of Chapter 2. Our ANCOVA analysis showed significant performance improvements. Our effect size calculations are higher than the educational effectiveness threshold. Lastly, our chi-squared analysis indicates a relationship between several error strings and our experimental course offering containing "What's Wrong With My Code" exercises. Together these findings combined with the finding in Chapter 2 emphasize the usefulness of our exercises within an introductory computer science course.

Appendix B: Style Checker Code

Automatic Style Checker

The following sections contain subtests called by our style checker. We use Python for all our test harnesses. We extracted the code from a full test harness, and therefore use by others may require source code modification.

Comments Existence Check

```
##
# @brief sub test to check whether comments exist in source file
#           which are not part of the assignment header
#
# @param test the test part containing properties to fill out
# @param source source object containing name, location & content splits
# @param deduction the deduction to take off per discovered problem
#
# @return true if the source contains comments, otherwise false
#
def comments_exist(test, source, deduction):
    OK = 0
    ERROR = 1

    # comments existence depends on header existing

    # if the header is not used in the quarter
    # the comments check can be disabled by returning True immediately
    #return True

    # determine the length of the comments not in header
    if len(source.comments) == 0:
        test.score = -1 * deduction
        return False

    return True
```

Figure 10: Code to check whether the source file has programming comments or not.

Line Length Check

```
##
# @brief sub test to check whether long lines exist in the program
#
# @param test the test part object to update with score
# @param source source object containing name, location & content splits
# @param max_len maximum length of a line
# @param deduction the deduction to take off per discovered problem
#
def long_check(test, source, max_len, deduction):
    from system.utils import expand_all_tabs

    line_nums = []

    # open and read in the source code file
    file = open(source.file_loc)
    contents = file.read()
    file.close()

    # split the lines into a list
    lines = contents.split("\n")
    expand_all_tabs(lines, source.indent_size)

    # create a list of line numbers which are long
    for (i, line) in enumerate(lines):
        if len(line.rstrip()) > max_len:
            line_nums.append(i+1)

    # set the test score and return the line number list
    test.score = -1 * deduction * len(line_nums)
    return line_nums
```

Figure 11: Code to return lines numbers of lines exceeding 80 characters in length.

Check for Improper Conditional Expressions

```
##
# @brief sub test to check whether improper conditionals exist
#
#       Improper conditions are "x == true", "true == x",
#       "x == false", "false == x".
#
# @param test the test part object to update with score
# @param source source object containing name, location & content splits
# @param deduction the deduction to take off per discovered problem
#
# @return a list containing line numbers in file that
#           have improper conditional
#
def improper_bool(test, source, deduction):
    line_nums = []
    improper = ["== true", "true ==", "== false", "false =="]

    # open and read source code
    file = open(source.file_loc)
    contents = file.read()
    file.close()

    # separate source code file into lines
    lines = contents.split("\n")
    for (i, line) in enumerate(lines):
        found_improper = False

        # determine if line number contains an improper condition
        for cond in improper:
            if line.replace(" ", "").find(cond.replace(" ", "")) != -1:
                found_improper = True
                break

        # add line number of improper conditional expression
        if found_improper:
            line_nums.append(i+1)

    # calculate score and return list of errant line numbers
    test.score = -1 * deduction * len(line_nums)
    return line_nums
```

Figure 12: Code to return line numbers of conditional expressions that compare to literal values of true or false.

Check for Global Variables

```
##
# @brief sub test to check whether global variables exist in the program
#
# @param test the test part object to update with score
# @param harness_dir directory containing the test harness
# @param source source object containing name, location & content splits
# @param env environment variables dict from JSON config of test suite
# @param deduction the deduction to take off per discovered problem
#
# @return a list containing all the names of global variables,
#          None if sys err
#
def globals_exist(test, harness_dir, source, env, deduction):
    import os
    import shutil
    import re
    from system.utils import which
    from system.procs import check_call

    # intialize to empty list of global variables
    vars = []

    files_to_remove = []

    # make sure the commands exist
    nm = which("nm")
    gpp = which(env["compiler"])
    if nm == None or gpp == None:
        return None

    # check if working directory exists, if not make one
    working_dir = os.path.join(harness_dir, env["working_dir"])
    if not os.path.exists(working_dir):
        os.mkdir(working_dir)
        made_working = True
    else:
        made_working = False

    FNULL = open(os.devnull, 'w')
    try:
        # compile the object file
        # set up path to includes directory
        include_path = os.path.join(harness_dir, env["includes_dir"])
        object_file = source.name[0:source.name.find(".")] + ".o"
        obj_loc = os.path.join(working_dir, object_file)
        files_to_remove.append(obj_loc)
        check_call([gpp, "-c", "-o", obj_loc,
                     "-I", include_path, source.file_loc],
                    stdout=FNULL, stderr=FNULL)
```

Figure 13: Part 1 of the code to return a string of all global variable names.

```
        # get object symbols
        symf = os.path.join(working_dir, "symbols.txt")
        files_to_remove.append(symf)
        cmd = " ".join([nm, obj_loc])
        with open(symf, 'w') as sym_file:
            check_call([nm, obj_loc], stdout=sym_file, stderr=FNULL)

        # search for global variables line in symbols file
        var_lines = []
        with open(symf, 'r') as sym_file:
            for line in sym_file:
                if re.search('[0-9A-Fa-f]* [BCDGRS]', line):
                    var_lines.append(line)

        # append last item (the variable name) to variables list
        for line in var_lines:
            if len(line) > 0:
                vars.append(line.strip().split(" ")[-1])

    except SystemError as e:
        FNULL.close()
        if made_working:
            shutil.rmtree(working_dir)
        elif len(files_to_remove) > 0:
            for f in files_to_remove:
                if os.path.isfile(f):
                    os.remove(f)
        return None

    # remove working directory if created it
    #otherwise remove files created
    if made_working:
        shutil.rmtree(working_dir)
    elif len(files_to_remove) > 0:
        for f in files_to_remove:
            if os.path.isfile(f):
                os.remove(f)

    FNULL.close()

    # calculate score and return string containg global variable names
    test.score = -1 * deduction * len(vars)
    return vars
```

Figure 14: Part 2 of the code to return a string of all global variable names.

Check for Tab Characters

```
##
# @brief sub test to check whether tabs exist in code (we use spaces)
#
# @param test the test part object to update with score
# @param source source object containing name, location & content splits
# @param deduction the deduction to take off per discovered problem
#
# @return a list containing line numbers containing 1+ tab characters
#
def find_tabs(test, source, deduction):
    line_nums = []

    # open and read source code file
    file = open(source.file_loc)
    contents = file.read()
    file.close()

    lines = contents.split("\n")
    for (i, line) in enumerate(lines):
        # determine if a tab exists on the line
        if line.find('\t') != -1:
            line_nums.append(i+1)

    # calculate the score and return list of line numbers with tab(s)
    test.score = -1 * deduction * len(line_nums)
    return line_nums
```

Figure 15: Code to return all source code line numbers containing a tab character.

Check for Proper Line Indentation

```
def one_curly_per_line(source, test=None, deduction=0):
    """
    Determines if one curly brace exists per line

    """

    bad_lines = []

    # open and read source file
    contents = open(source.file_loc).read()

    # create list of line numbers where multiple braces exist
    lines = contents.split('\n')
    for (i, line) in enumerate(lines):
        num_braces = line.count("{") + line.count("}")
        if num_braces > 1:
            bad_lines.append(i)

    # not having a single curly per line halts certain style checks
    if len(bad_lines) > 0:
        source.style_halt = True

    if test is not None:
        test.score = -1 * deduction * len(bad_lines)

    # return list of line numbers that have more than one curly brace
    return bad_lines
```

Figure 16: Part 1 of indentation check - verify single curly brace per line.

```python
##
# @brief create a list of indentation Block objects from source
# @param source source object containing name, location & other members
# @return a list of levels as defined in the expanded brief
def create_blocks(source):
    """
    Create a list of levels, each level contains all lines in source
    file for that level indentation block.
    """
    from system.utils import expand_all_tabs

    # Indentation is dependent on the absence of other style problems
    if source.style_halt:
        return (False, [])

    indeces = []
    blocks = []

    # TODO implement try block
    f = open(source.file_loc)
    contents = f.read()
    f.close()

    # global level
    level = 0
    begin = 1
    blocks.append(Block(level, begin))
    indeces.append(0)


    # loop over all the lines in the file splitting up levels
    file_lines = contents.split("\n")
    expand_all_tabs(file_lines, source.indent_size)
    for (i, line) in enumerate(file_lines):
        # determine if end of level exists on line
        if line.find("}") != -1:
            # make sure not to pop global off in case of mismatch braces
            if len(indeces) > 1:
                # only pop if belong within indent level one up
                if line.lstrip().find("}") == 0:
                    indeces.pop()
                    popped = True

        # add line to current level
        cur = indeces[-1]
        blocks[cur].lines.append(SourceLine(i+1, line))

        # determine if new level needs to be created
        if line.find("{") != -1:
            # create a new object for discovered level
            level = blocks[cur].level + 1
            begin = i+1

            # update lists of tuples and indeces
            indeces.append(len(blocks))
            blocks.append(Block(level, begin))

        # determine if end of level exists on line and haven't popped
        if line.find("}") != -1 and not popped:
            # make sure not to pop global in case of mismatch braces
            if len(indeces) > 1:
                indeces.pop()

        popped = False
    # return created blocks
    return (len(blocks) != 0, blocks)
```

Figure 17: Part 2 of indentation check - create blocks to separate indentation levels.

```
##
# @brief function determines if the spacing of all lines within the
#           various indentation blocks are correct.
#
#        Currently uses the create_levels function & one_curly_per_line
#        functions as the single curly restraint is necessary at the
#        time for proper block initialization.
#
# @param test the test part object to update with score
# @param source source object containing name, location & other members
# @param deduction the deduction to take off per discovered problem
#
# @return tuple containing True/False & list of line numbers where each
#         number is the start of an incorrectly spaced indentation block
#
def correct_indent(test, source, deduction):
    """
    Go through the lines in each level, determine if spacing is correct.

    """

    bad_lines = []

    # Verify one curly brace per line
    l = one_curly_per_line(source)

    # Correct spacing is dependent on other style tests passing
    if source.style_halt:
        return (False, [])

    # Create a list of indentation blocks
    (ret, blocks) = create_blocks(source)

    # False value indicates error in levels creation, test fails
    if not ret:
        return (False, [])

    for cur_block in blocks:
        sp = cur_block.level * source.indent_size
        for (i, line) in enumerate(cur_block.lines):
            strip_len = len(line.content.strip())
            if strip_len > 0:
                indent_amount = line.content.find(line.content.lstrip()[0])
                if sp != indent_amount:
                    if i-1 >= 0:
                        prev = cur_block.lines[i-1].content
                        if not (prev.strip() == "{" or prev.strip() == "}"):
                            if prev.find(";") == -1:
                                if sp+source.indent_size != indent_amount:
                                    bad_lines.append(line.num)
                            else:
                                bad_lines.append(line.num)
                        else:
                            bad_lines.append(line.num)
                    else:
                        bad_lines.append(line.num)

    # test completed and return list of incorrectly indented lines
    bad_l = sorted(list(set(bad_lines)))
    test.score = -1 * deduction * len(bad_l)
    return (True, bad_l)
```

Figure 18: Part 3 of indentation check - primary function to create a list of improperly indented lines.

Appendix C: Additional Charts and Graphs

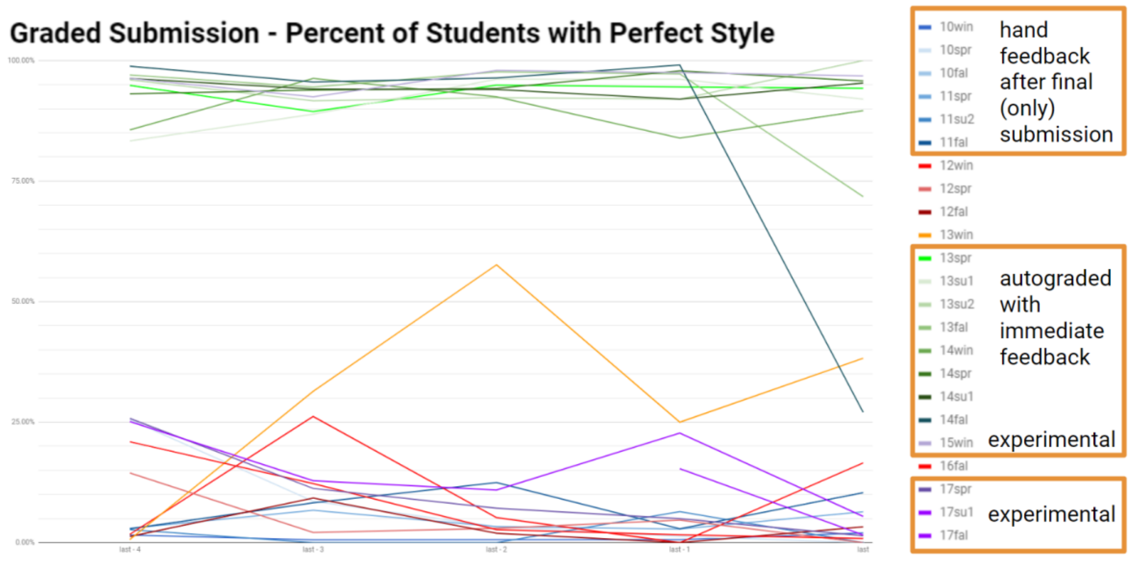


Figure 19: Style scores for the final submission for the last five assignments in the quarter across seven years.

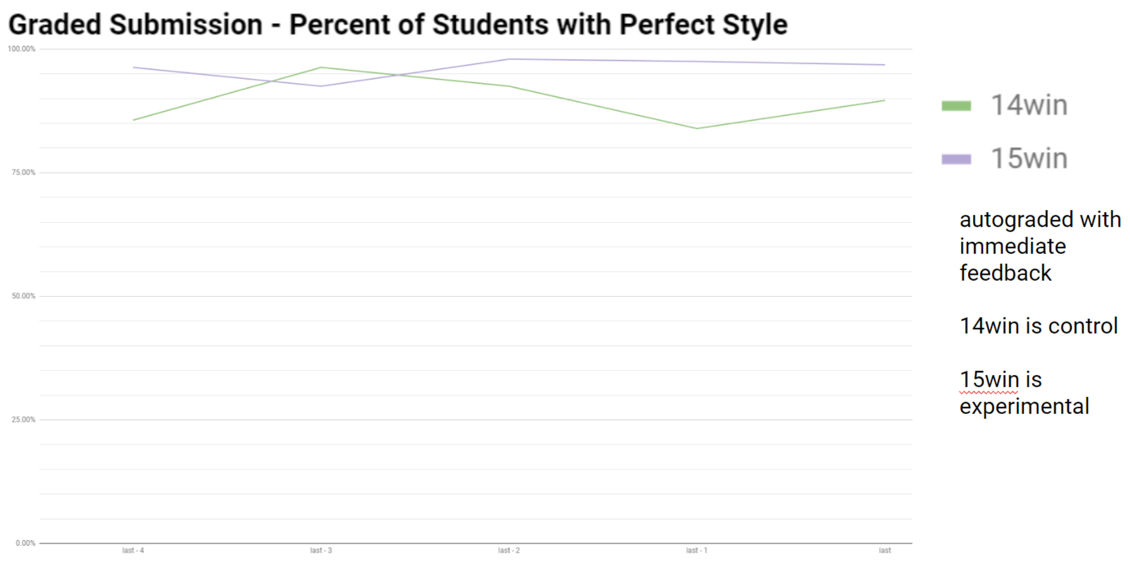


Figure 20: Style scores for the last five assignments in the quarter. Comparing two terms (types F and I) with automatic feedback on unlimited submissions.

Autograded Assn Number:	**8**	**7**	**6**	**5**	**4**
Term F (Control)					
N	154	138	172	182	158
Average # of Errors	30.00	12.67	10.65	13.83	7.13
Std Dev	57.13	28.76	22.28	26.28	15.43
Term I (Experimental)	✓	✓		✓	✓
N	187	200	169	199	214
Average # of Errors	4.01	5.37	11.52	15.87	5.95
Std Dev	10.33	13.76	22.27	27.69	9.54

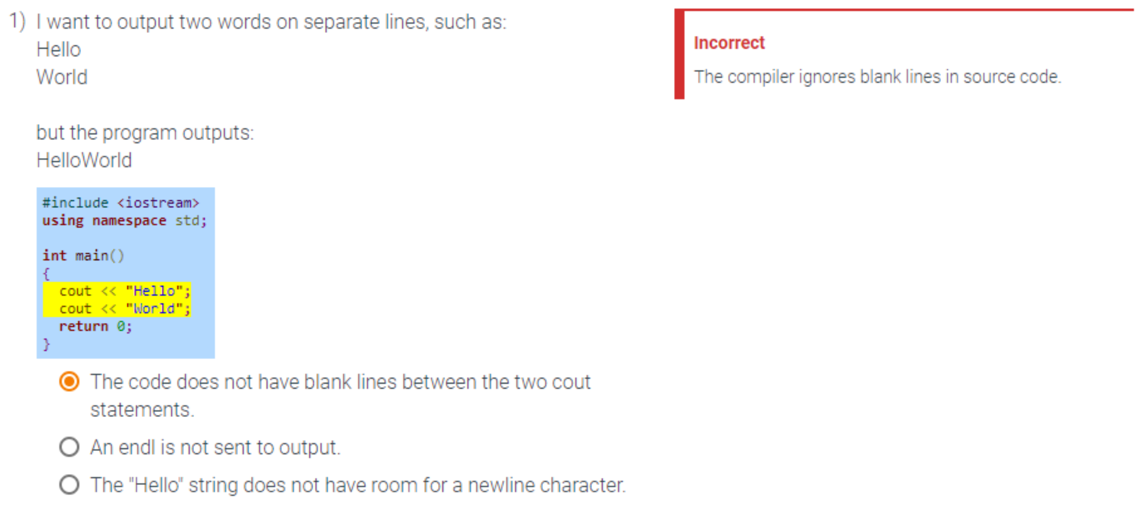
Table 15: Autograded terms (F and I), style scores on the first submission of unlimited submissions. ✓ indicates significant difference in errors (p-value < 0.01).

Autograded Assn Number:	**8**	**7**	**6**	**5**	**4**
Term F (Control)					
N	154	137	173	162	188
Average # of Errors	5.22	1.00	1.05	3.51	3.86
Std Dev	18.94	6.74	8.46	14.33	24.49
Term I (Experimental)					✓
N	216	200	196	201	189
Average # of Errors	0.56	2.01	0.58	0.16	0.35
Std Dev	3.47	10.22	6.81	1.61	2.75

Table 16: Autograded terms (F and I), style scores on the final submission of unlimited submissions. ✓ indicates significant difference in errors (p-value < 0.01).

Appendix D: What's Wrong With My Code Exercises

Basic Output Exercises



1) I want to output two words on separate lines, such as:
Hello
World

but the program outputs:
HelloWorld

```
#include <iostream>
using namespace std;

int main()
{
  cout << "Hello";
  cout << "World";
  return 0;
}
```

[✓ The code does not have blank lines between the two cout statements.]

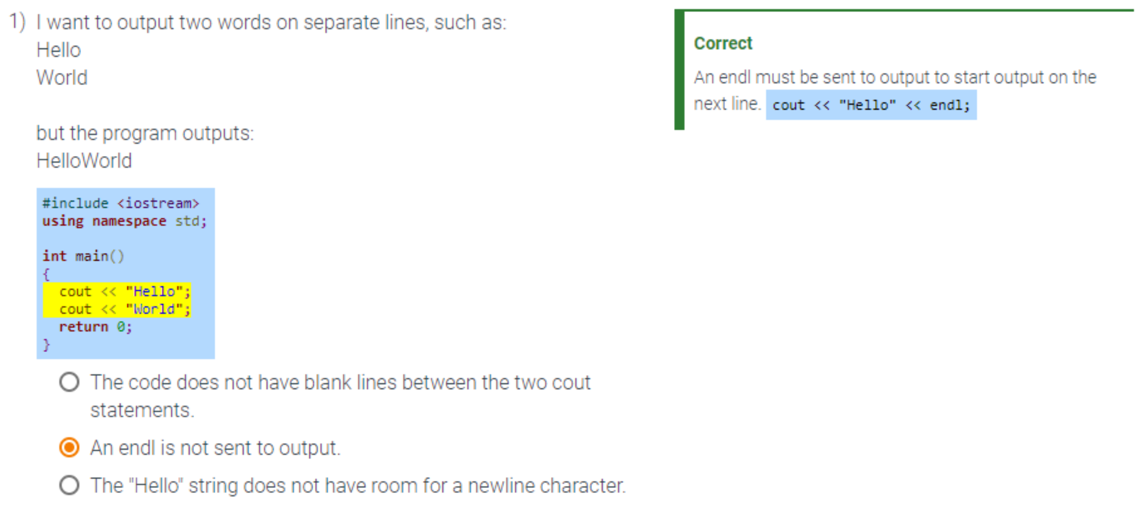
An endl is not sent to output.

The "Hello" string does not have room for a newline character.

Incorrect

The compiler ignores blank lines in source code.

Figure 21: Basic Output, Question #1, Answer #1



1) I want to output two words on separate lines, such as:
Hello
World

but the program outputs:
HelloWorld

```
#include <iostream>
using namespace std;

int main()
{
  cout << "Hello";
  cout << "World";
  return 0;
}
```

The code does not have blank lines between the two cout statements.

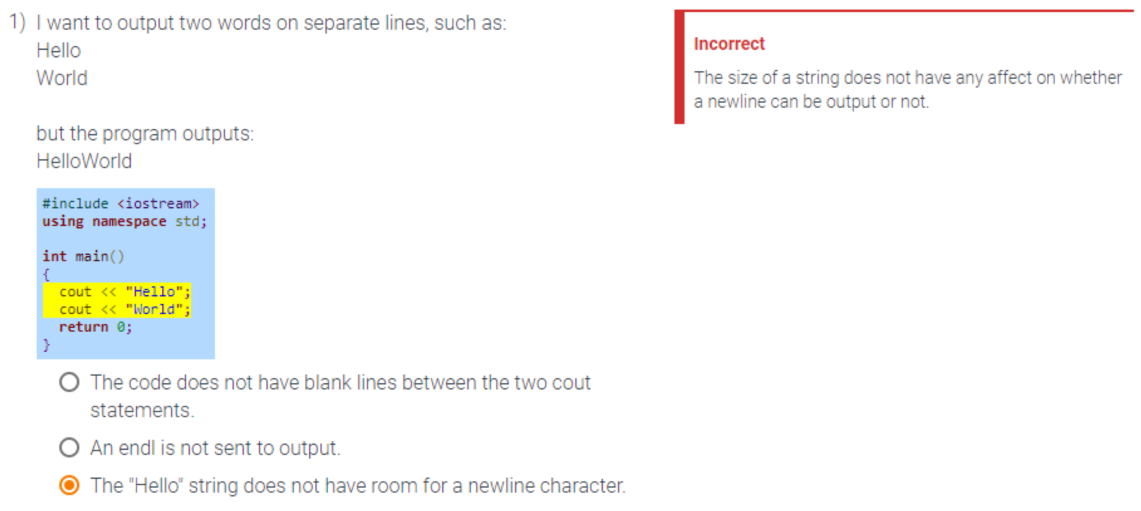
[✓ An endl is not sent to output.]

The "Hello" string does not have room for a newline character.

Correct

An endl must be sent to output to start output on the next line. `cout << "Hello" << endl;`

Figure 22: Basic Output, Question #1, Answer #2



1) I want to output two words on separate lines, such as:
Hello
World

but the program outputs:
HelloWorld

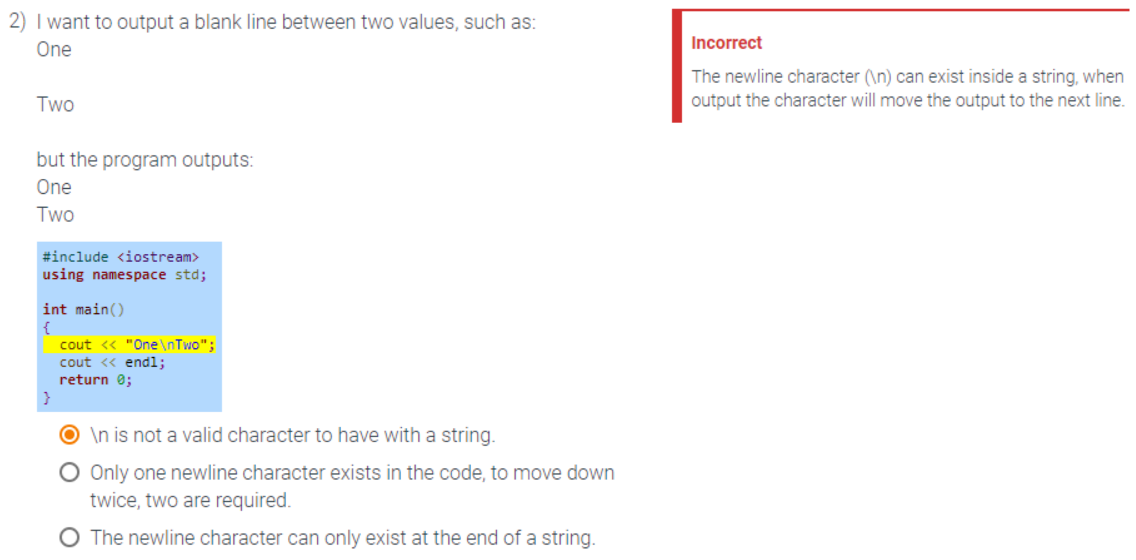
```
#include <iostream>
using namespace std;

int main()
{
   cout << "Hello";
   cout << "World";
   return 0;
}
```

- The code does not have blank lines between the two cout statements.
- An endl is not sent to output.
- [✓ The "Hello" string does not have room for a newline character.]

Incorrect

The size of a string does not have any affect on whether a newline can be output or not.

Figure 23: Basic Output, Question #1, Answer #3



2) I want to output a blank line between two values, such as:
One

Two

but the program outputs:
One
Two

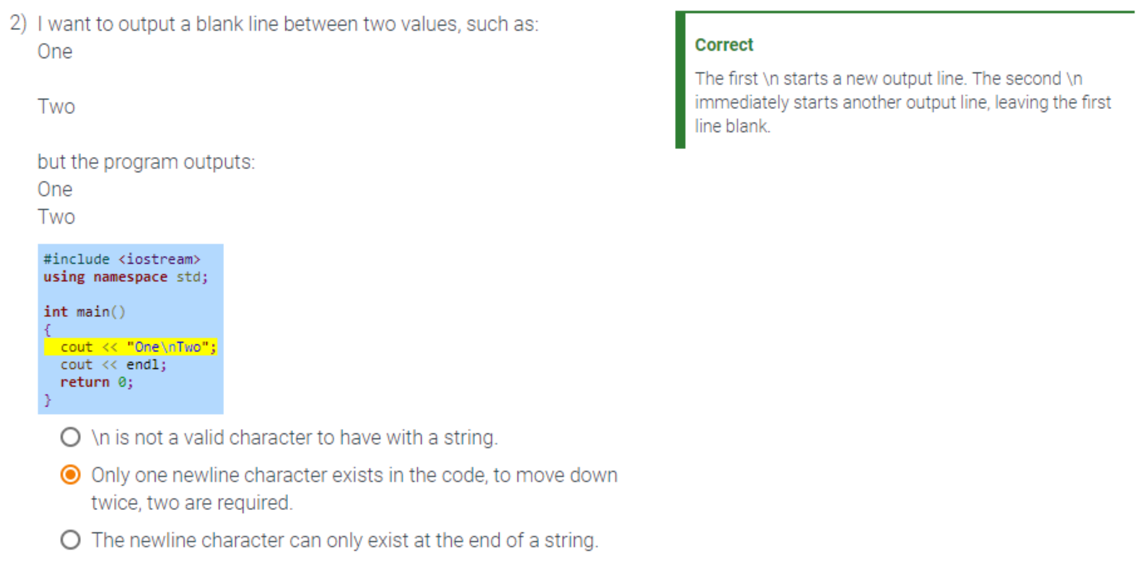
```
#include <iostream>
using namespace std;

int main()
{
   cout << "One\nTwo";
   cout << endl;
   return 0;
}
```

- [✓ \n is not a valid character to have with a string.]
- Only one newline character exists in the code, to move down twice, two are required.
- The newline character can only exist at the end of a string.

Incorrect

The newline character (\n) can exist inside a string, when output the character will move the output to the next line.

Figure 24: Basic Output, Question #2, Answer #1


2) I want to output a blank line between two values, such as:
One

Two

but the program outputs:
One
Two

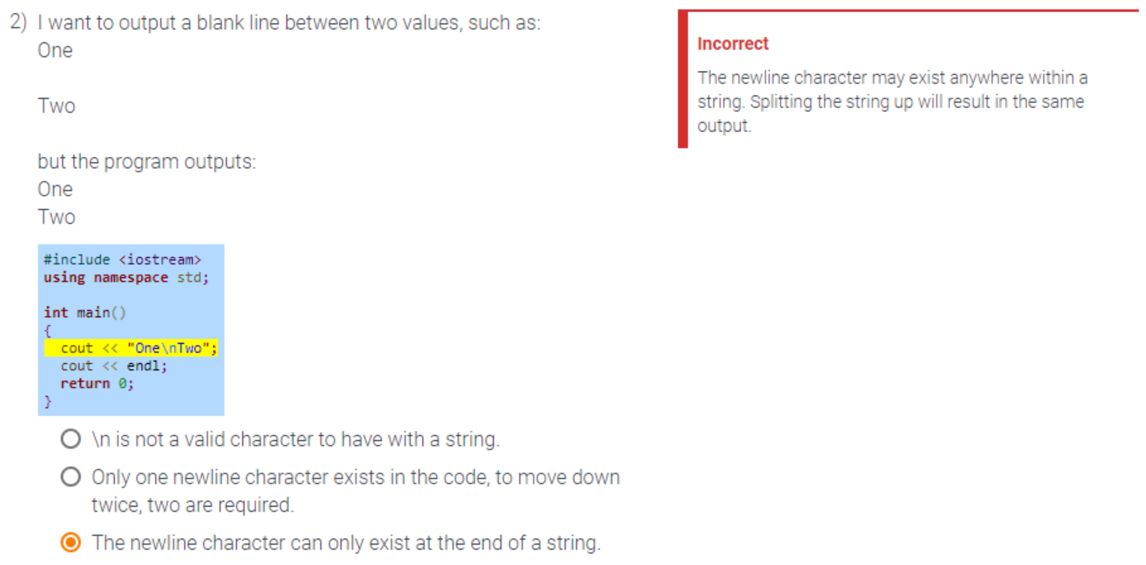
```
#include <iostream>
using namespace std;

int main()
{
   cout << "One\nTwo";
   cout << endl;
   return 0;
}
```

- [ ] \n is not a valid character to have with a string.
- [✓ Only one newline character exists in the code, to move down twice, two are required.]
- [ ] The newline character can only exist at the end of a string.

Correct

The first \n starts a new output line. The second \n immediately starts another output line, leaving the first line blank.

Figure 25: Basic Output, Question #1, Answer #2


2) I want to output a blank line between two values, such as:
One

Two

but the program outputs:
One
Two

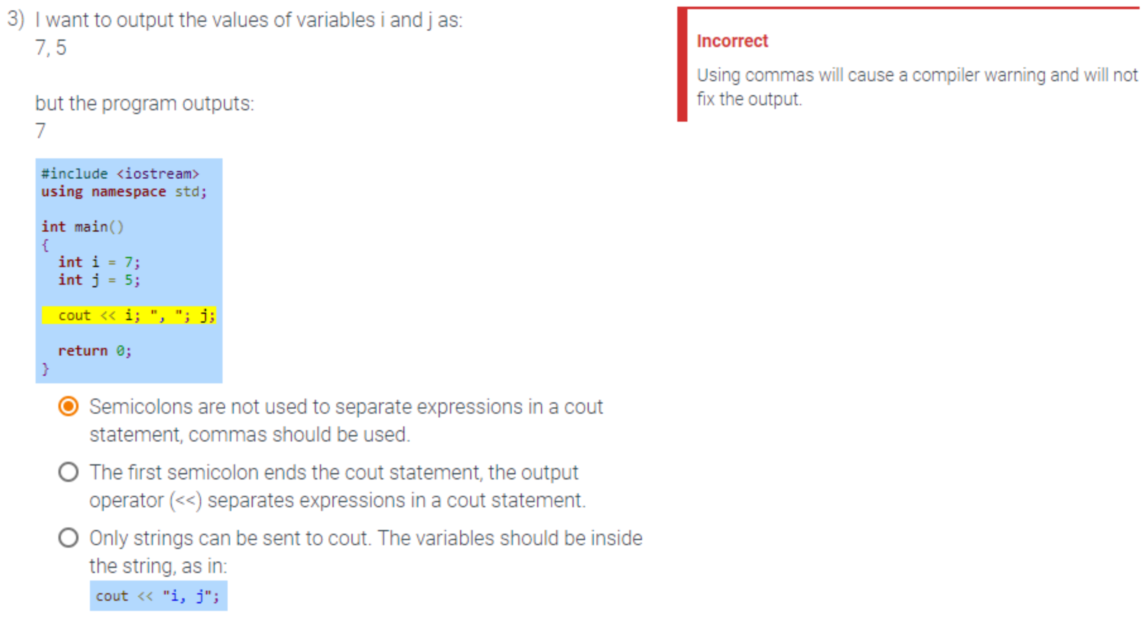
```
#include <iostream>
using namespace std;

int main()
{
   cout << "One\nTwo";
   cout << endl;
   return 0;
}
```

- [ ] \n is not a valid character to have with a string.
- [ ] Only one newline character exists in the code, to move down twice, two are required.
- [✓ The newline character can only exist at the end of a string.]

Incorrect

The newline character may exist anywhere within a string. Splitting the string up will result in the same output.

Figure 26: Basic Output, Question #2, Answer #3



3) I want to output the values of variables i and j as:
7, 5

but the program outputs:
7

```
#include <iostream>
using namespace std;

int main()
{
  int i = 7;
  int j = 5;

  cout << i; ", "; j;

  return 0;
}
```

[✓ Semicolons are not used to separate expressions in a cout statement, commas should be used.]

The first semicolon ends the cout statement, the output operator (<<) separates expressions in a cout statement.

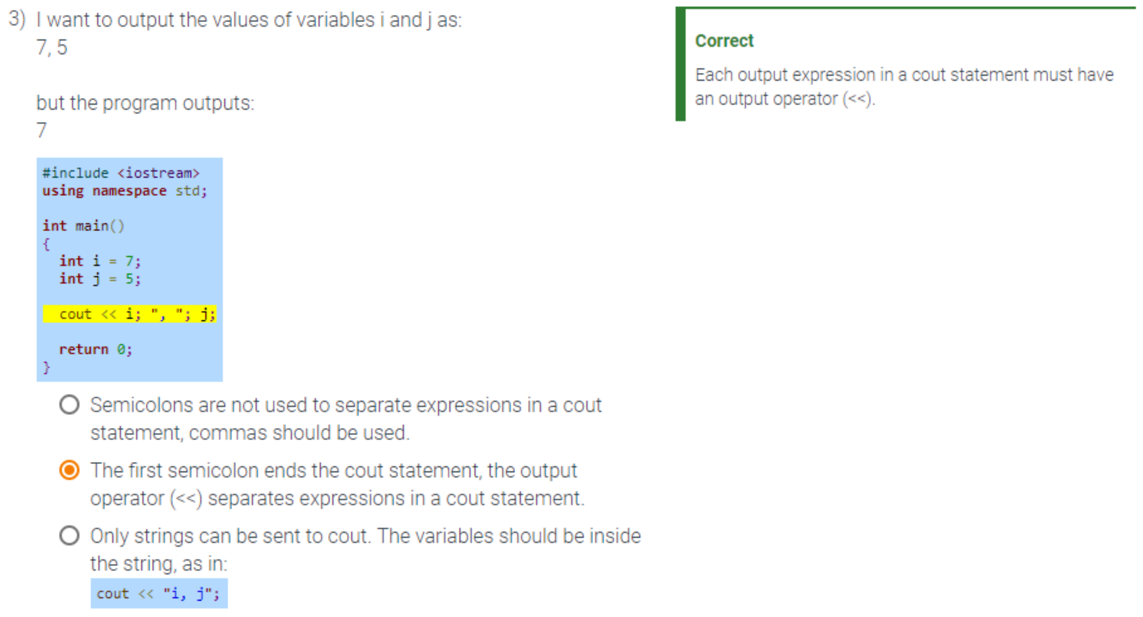
Only strings can be sent to cout. The variables should be inside the string, as in:

```
cout << "i, j";
```

Incorrect

Using commas will cause a compiler warning and will not fix the output.

Figure 27: Basic Output, Question #3, Answer #1



3) I want to output the values of variables i and j as:
7, 5

but the program outputs:
7

```
#include <iostream>
using namespace std;

int main()
{
  int i = 7;
  int j = 5;

  cout << i; ", "; j;

  return 0;
}
```

Semicolons are not used to separate expressions in a cout statement, commas should be used.

[✓ The first semicolon ends the cout statement, the output operator (<<) separates expressions in a cout statement.]

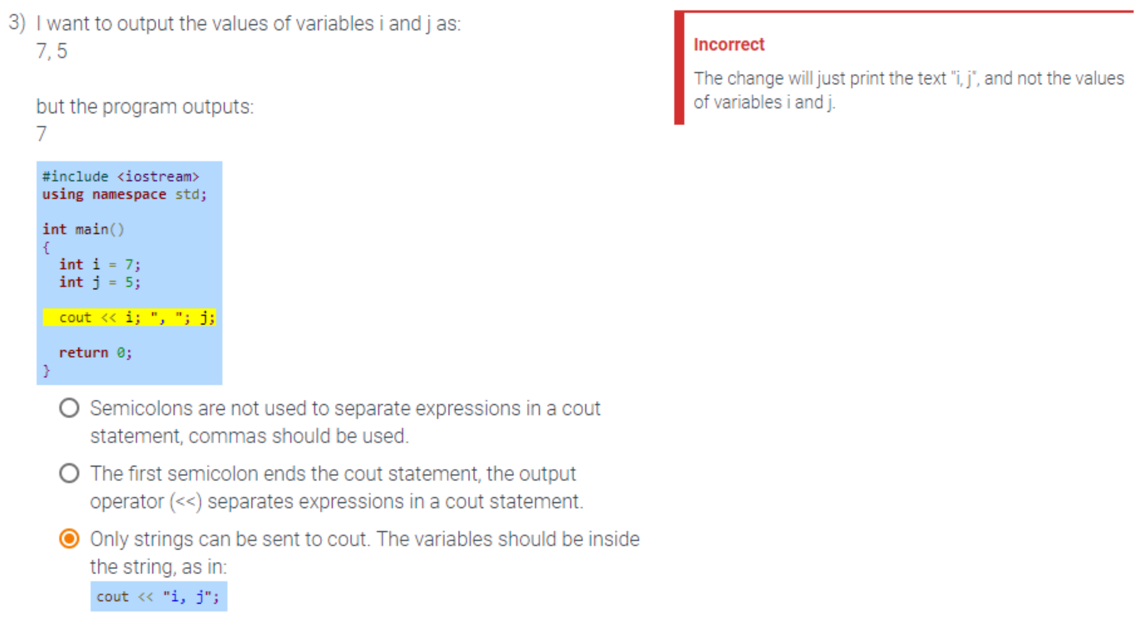
Only strings can be sent to cout. The variables should be inside the string, as in:

```
cout << "i, j";
```

Correct

Each output expression in a cout statement must have an output operator (<<).

Figure 28: Basic Output, Question #3, Answer #2



3) I want to output the values of variables i and j as:
7, 5

but the program outputs:
7

```
#include <iostream>
using namespace std;

int main()
{
  int i = 7;
  int j = 5;

  cout << i; ", "; j;

  return 0;
}
```

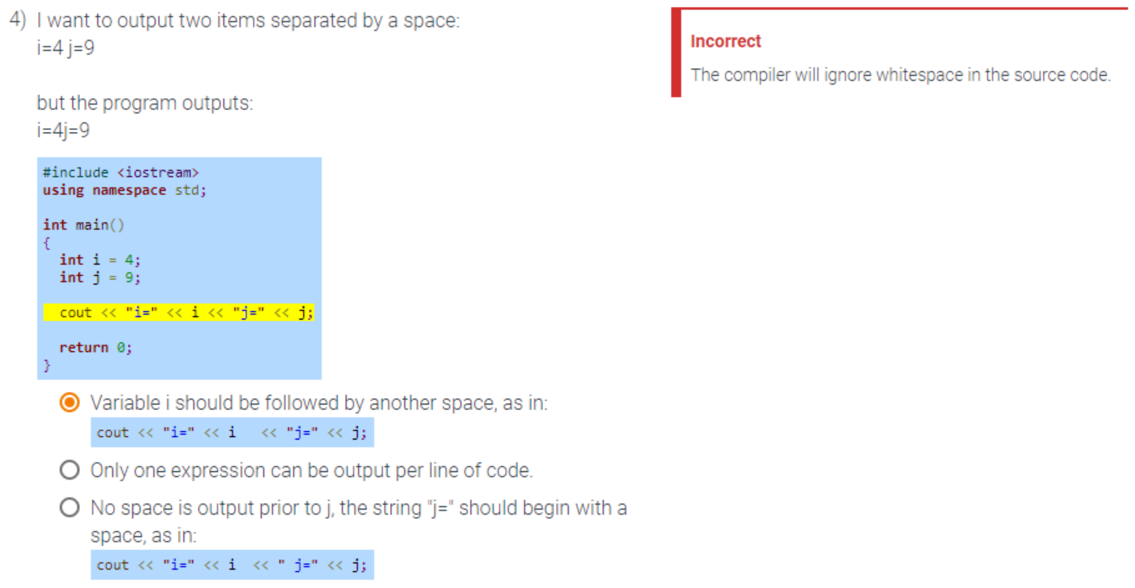
- ○ Semicolons are not used to separate expressions in a cout statement, commas should be used.
- ○ The first semicolon ends the cout statement, the output operator (<<) separates expressions in a cout statement.
- [✓ Only strings can be sent to cout. The variables should be inside the string, as in:]

```
cout << "i, j";
```

Incorrect

The change will just print the text "i, j", and not the values of variables i and j.

Figure 29: Basic Output, Question #3, Answer #3



4) I want to output two items separated by a space:
i=4 j=9

but the program outputs:
i=4j=9

```
#include <iostream>
using namespace std;

int main()
{
  int i = 4;
  int j = 9;

  cout << "i=" << i << "j=" << j;

  return 0;
}
```

- [✓ Variable i should be followed by another space, as in:]

```
cout << "i=" << i   << "j=" << j;
```

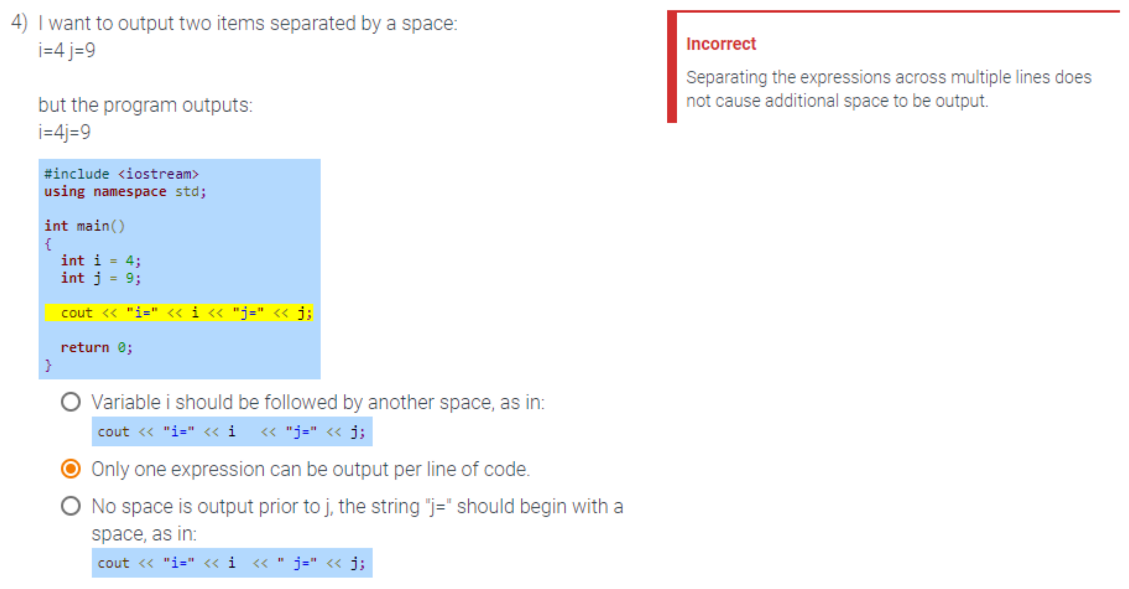
- ○ Only one expression can be output per line of code.
- ○ No space is output prior to j, the string "j=" should begin with a space, as in:

```
cout << "i=" << i  << " j=" << j;
```

Incorrect

The compiler will ignore whitespace in the source code.

Figure 30: Basic Output, Question #4, Answer #1



4) I want to output two items separated by a space:
i=4 j=9

but the program outputs:
i=4j=9

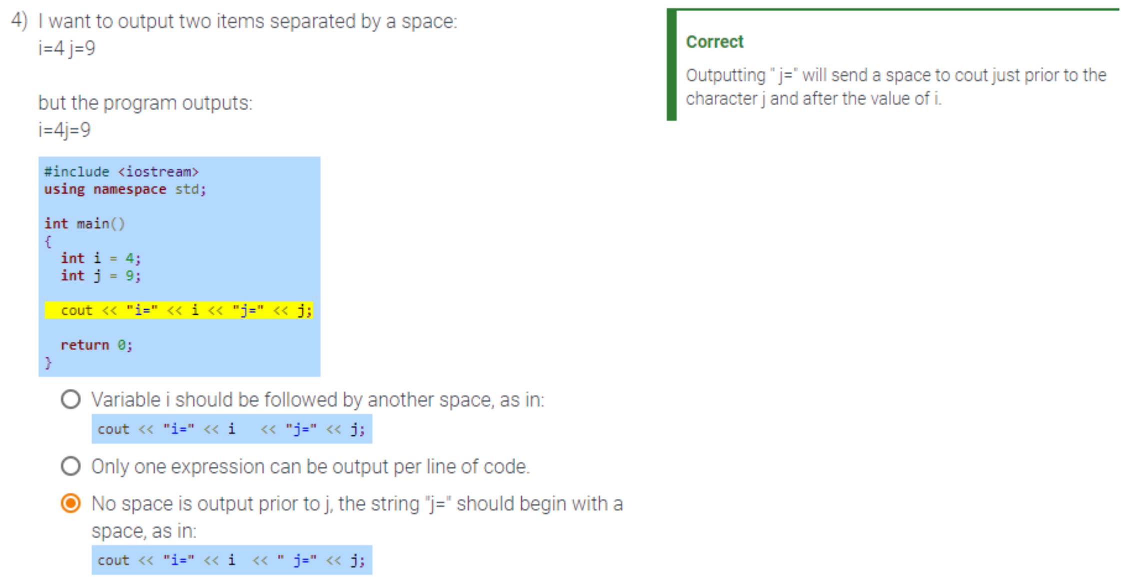
```
#include <iostream>
using namespace std;

int main()
{
  int i = 4;
  int j = 9;

  cout << "i=" << i << "j=" << j;

  return 0;
}
```

- ○ Variable i should be followed by another space, as in:
  `cout << "i=" << i  << "j=" << j;`
- ◉ Only one expression can be output per line of code.
- ○ No space is output prior to j, the string "j=" should begin with a space, as in:
  `cout << "i=" << i  << " j=" << j;`

Incorrect

Separating the expressions across multiple lines does not cause additional space to be output.

Figure 31: Basic Output, Question #4, Answer #2



4) I want to output two items separated by a space:
i=4 j=9

but the program outputs:
i=4j=9

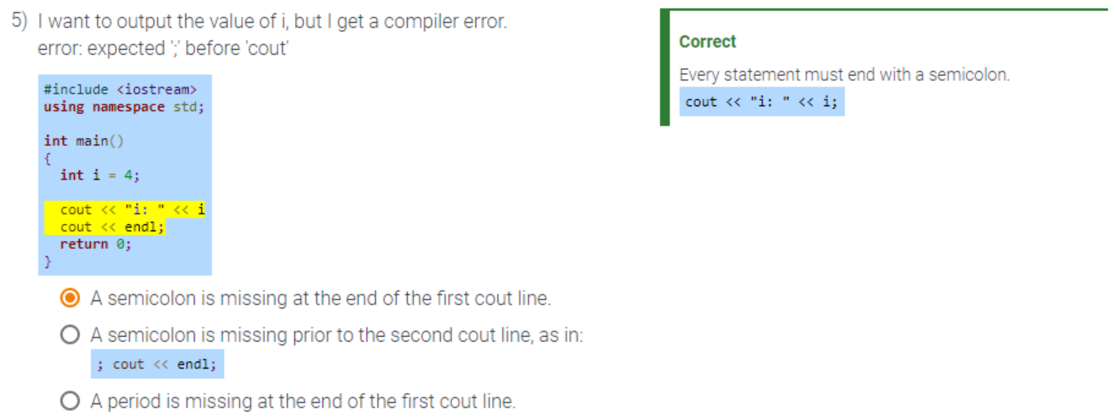
```
#include <iostream>
using namespace std;

int main()
{
  int i = 4;
  int j = 9;

  cout << "i=" << i << "j=" << j;

  return 0;
}
```

- ○ Variable i should be followed by another space, as in:
  `cout << "i=" << i  << "j=" << j;`
- ○ Only one expression can be output per line of code.
- ◉ No space is output prior to j, the string "j=" should begin with a space, as in:
  `cout << "i=" << i  << " j=" << j;`

Correct

Outputting " j=" will send a space to cout just prior to the character j and after the value of i.

Figure 32: Basic Output, Question #4, Answer #3



5) I want to output the value of i, but I get a compiler error.
error: expected ';' before 'cout'

```
#include <iostream>
using namespace std;

int main()
{
  int i = 4;

  cout << "i: " << i
  cout << endl;
  return 0;
}
```

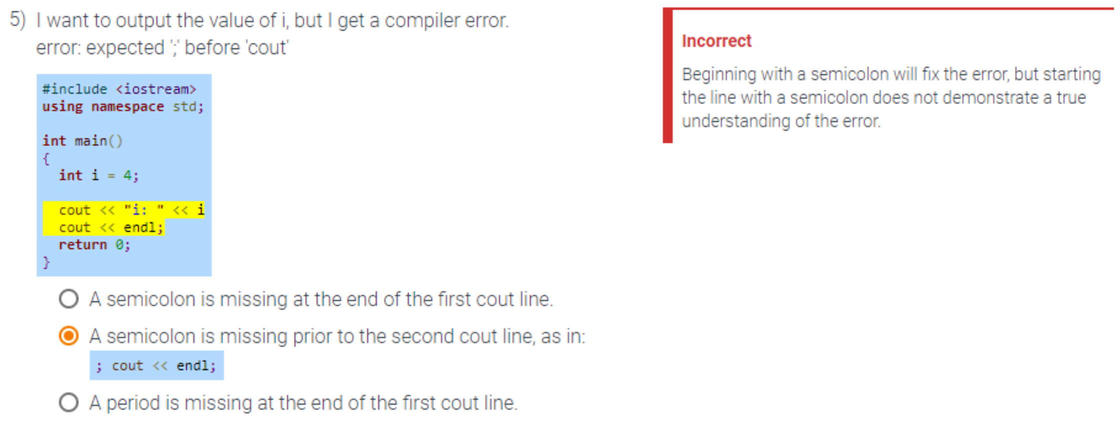
- [✓ A semicolon is missing at the end of the first cout line.]
- A semicolon is missing prior to the second cout line, as in:
  `; cout << endl;`
- A period is missing at the end of the first cout line.

Correct

Every statement must end with a semicolon.

`cout << "i: " << i;`

Figure 33: Basic Output, Question #5, Answer #1



5) I want to output the value of i, but I get a compiler error.
error: expected ';' before 'cout'

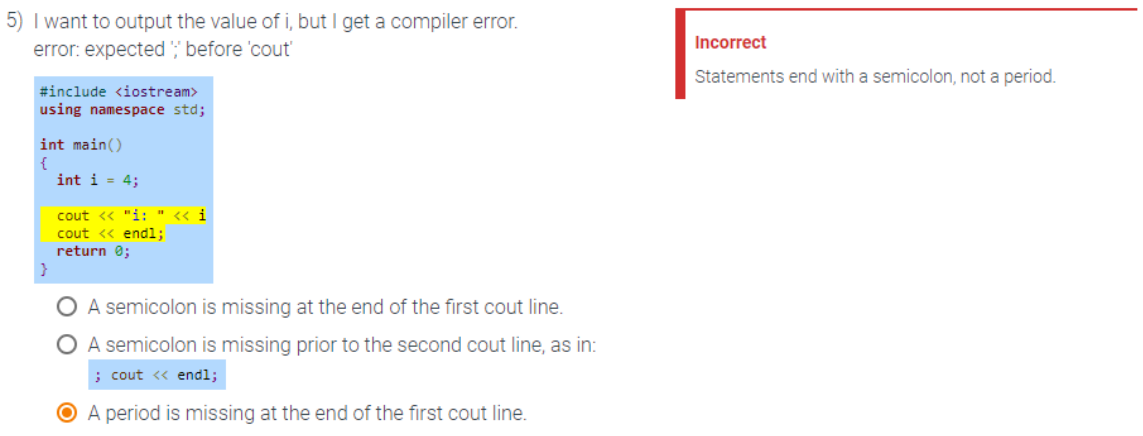
```
#include <iostream>
using namespace std;

int main()
{
  int i = 4;

  cout << "i: " << i
  cout << endl;
  return 0;
}
```

- A semicolon is missing at the end of the first cout line.
- [✓ A semicolon is missing prior to the second cout line, as in:]
  `; cout << endl;`
- A period is missing at the end of the first cout line.

Incorrect

Beginning with a semicolon will fix the error, but starting the line with a semicolon does not demonstrate a true understanding of the error.

Figure 34: Basic Output, Question #5, Answer #2



5) I want to output the value of i, but I get a compiler error.
error: expected ';' before 'cout'

```
#include <iostream>
using namespace std;

int main()
{
  int i = 4;

  cout << "i: " << i
  cout << endl;
  return 0;
}
```

- A semicolon is missing at the end of the first cout line.
- A semicolon is missing prior to the second cout line, as in:

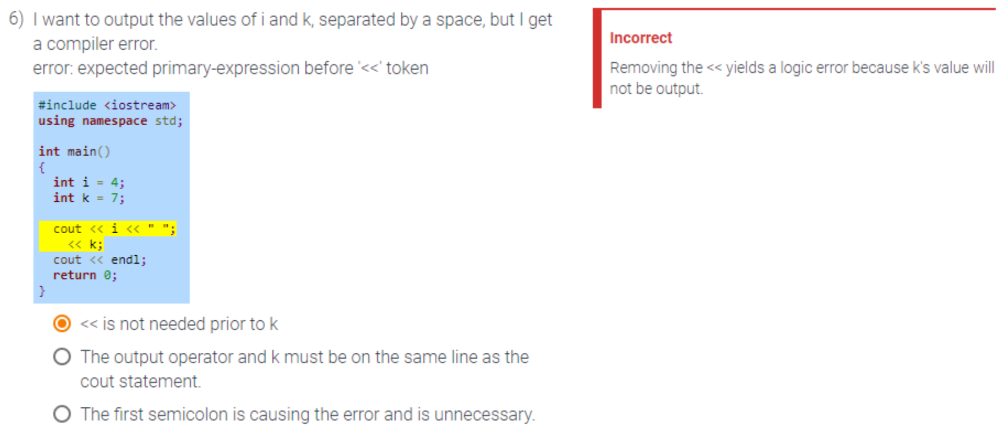
```
; cout << endl;
```

- [✓ A period is missing at the end of the first cout line.]

Incorrect

Statements end with a semicolon, not a period.

Figure 35: Basic Output, Question #5, Answer #3



6) I want to output the values of i and k, separated by a space, but I get a compiler error.
error: expected primary-expression before '<<' token

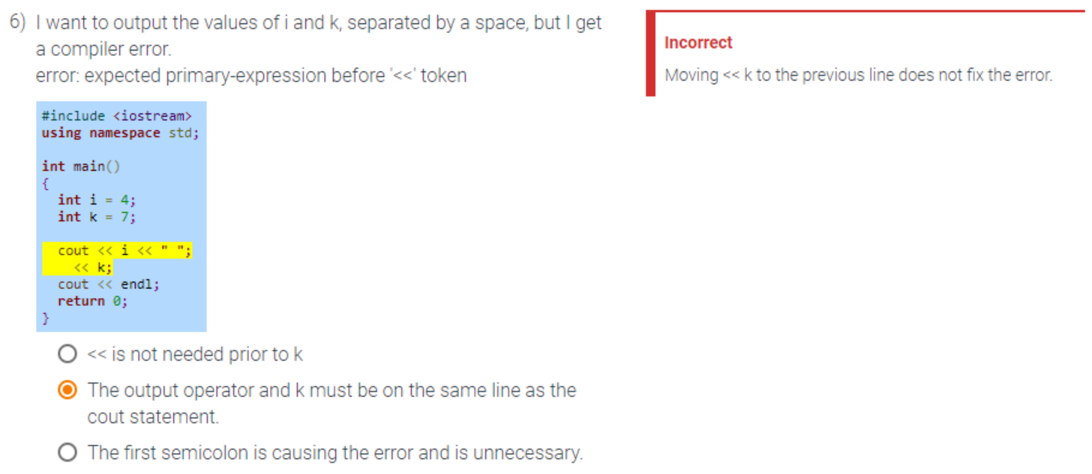
```
#include <iostream>
using namespace std;

int main()
{
  int i = 4;
  int k = 7;

  cout << i << " ";
    << k;
  cout << endl;
  return 0;
}
```

- [✓ << is not needed prior to k]
- The output operator and k must be on the same line as the cout statement.
- The first semicolon is causing the error and is unnecessary.

Incorrect

Removing the << yields a logic error because k's value will not be output.

Figure 36: Basic Output, Question #6, Answer #1



6) I want to output the values of i and k, separated by a space, but I get a compiler error.
error: expected primary-expression before '<<' token

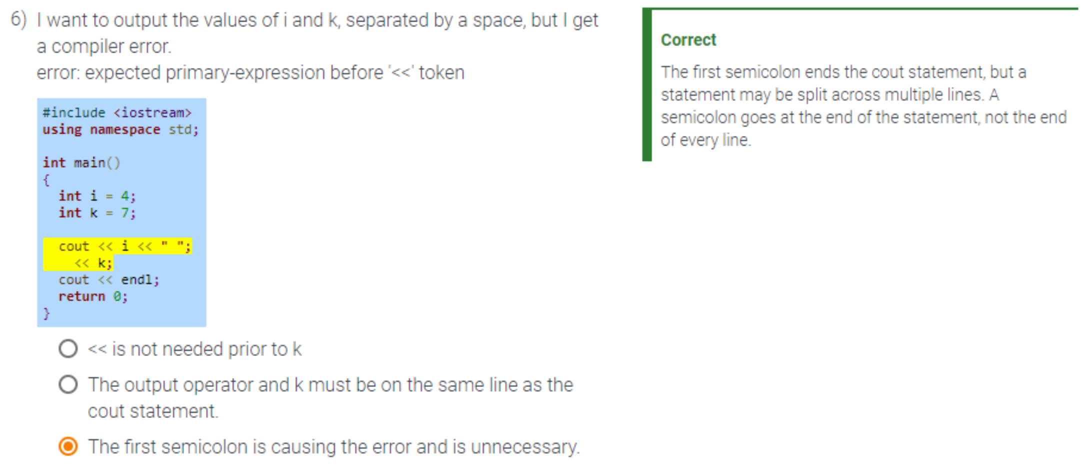
```
#include <iostream>
using namespace std;

int main()
{
  int i = 4;
  int k = 7;

  cout << i << " ";
    << k;
  cout << endl;
  return 0;
}
```

- ○ << is not needed prior to k
- [✓ The output operator and k must be on the same line as the cout statement.]
- ○ The first semicolon is causing the error and is unnecessary.

Incorrect

Moving << k to the previous line does not fix the error.

Figure 37: Basic Output, Question #6, Answer #2



6) I want to output the values of i and k, separated by a space, but I get a compiler error.
error: expected primary-expression before '<<' token

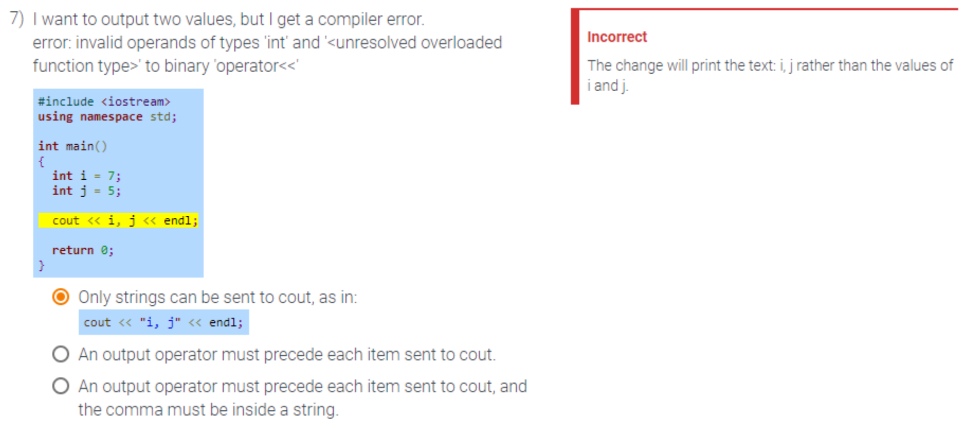
```
#include <iostream>
using namespace std;

int main()
{
  int i = 4;
  int k = 7;

  cout << i << " ";
    << k;
  cout << endl;
  return 0;
}
```

- ○ << is not needed prior to k
- ○ The output operator and k must be on the same line as the cout statement.
- [✓ The first semicolon is causing the error and is unnecessary.]

Correct

The first semicolon ends the cout statement, but a statement may be split across multiple lines. A semicolon goes at the end of the statement, not the end of every line.

Figure 38: Basic Output, Question #6, Answer #3



7) I want to output two values, but I get a compiler error.
error: invalid operands of types 'int' and '<unresolved overloaded function type>' to binary 'operator<<'

```
#include <iostream>
using namespace std;

int main()
{
  int i = 7;
  int j = 5;

  cout << i, j << endl;

  return 0;
}
```

[✓ Only strings can be sent to cout, as in:]

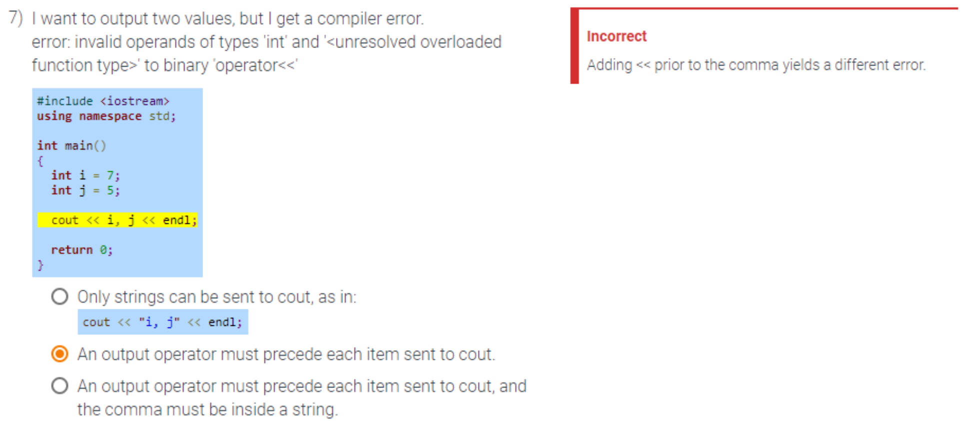
```
cout << "i, j" << endl;
```

- An output operator must precede each item sent to cout.
- An output operator must precede each item sent to cout, and the comma must be inside a string.

Incorrect

The change will print the text: i, j rather than the values of i and j.

Figure 39: Basic Output, Question #7, Answer #1



7) I want to output two values, but I get a compiler error.
error: invalid operands of types 'int' and '<unresolved overloaded function type>' to binary 'operator<<'

```
#include <iostream>
using namespace std;

int main()
{
  int i = 7;
  int j = 5;

  cout << i, j << endl;

  return 0;
}
```

- Only strings can be sent to cout, as in:

```
cout << "i, j" << endl;
```

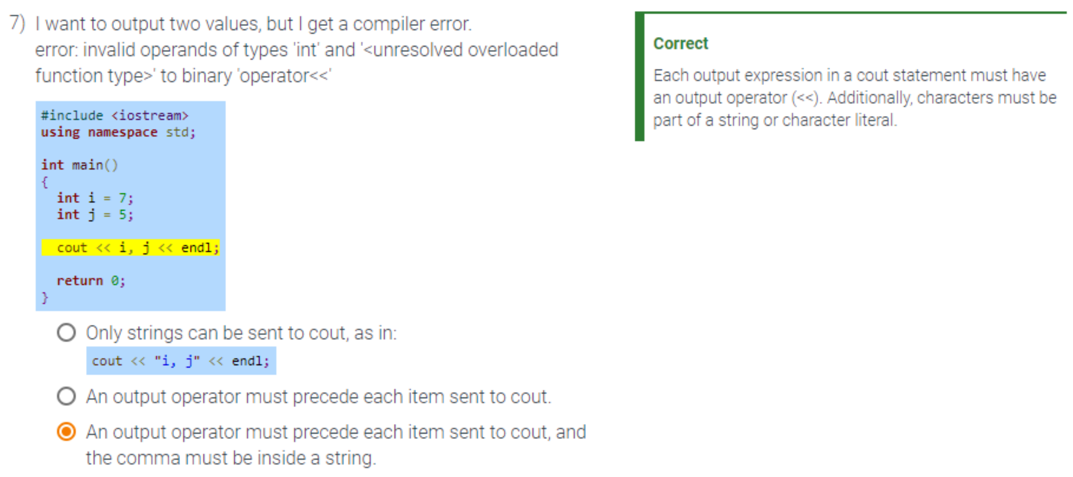
[✓ An output operator must precede each item sent to cout.]

- An output operator must precede each item sent to cout, and the comma must be inside a string.

Incorrect

Adding << prior to the comma yields a different error.

Figure 40: Basic Output, Question #7, Answer #2



7) I want to output two values, but I get a compiler error.
error: invalid operands of types 'int' and '<unresolved overloaded function type>' to binary 'operator<<'

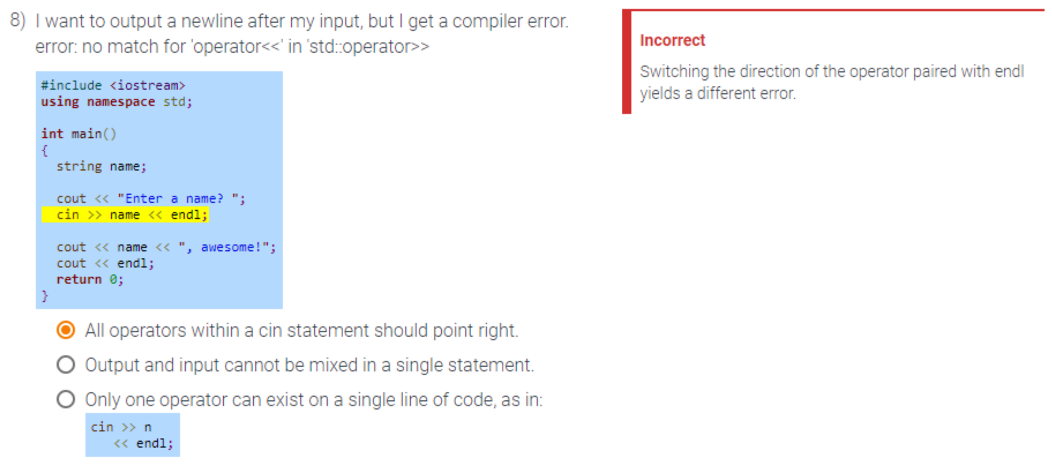
```
#include <iostream>
using namespace std;

int main()
{
  int i = 7;
  int j = 5;

  cout << i, j << endl;

  return 0;
}
```

- ○ Only strings can be sent to cout, as in:
  ```
  cout << "i, j" << endl;
  ```
- ○ An output operator must precede each item sent to cout.
- ● An output operator must precede each item sent to cout, and the comma must be inside a string.

Correct

Each output expression in a cout statement must have an output operator (<<). Additionally, characters must be part of a string or character literal.

Figure 41: Basic Output, Question #7, Answer #3



8) I want to output a newline after my input, but I get a compiler error.
error: no match for 'operator<<' in 'std::operator>>

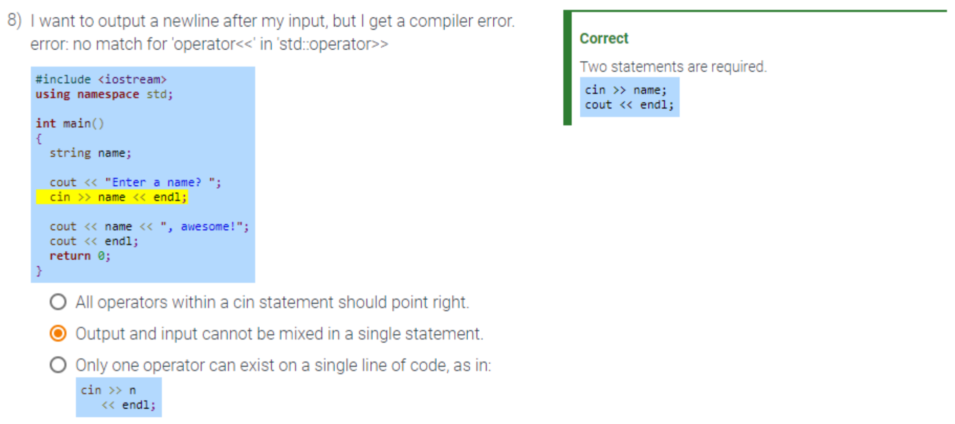
```
#include <iostream>
using namespace std;

int main()
{
  string name;

  cout << "Enter a name? ";
  cin >> name << endl;

  cout << name << ", awesome!";
  cout << endl;
  return 0;
}
```

- ● All operators within a cin statement should point right.
- ○ Output and input cannot be mixed in a single statement.
- ○ Only one operator can exist on a single line of code, as in:
  ```
  cin >> n
     << endl;
  ```

Incorrect

Switching the direction of the operator paired with endl yields a different error.

Figure 42: Basic Output, Question #8, Answer #1


8) I want to output a newline after my input, but I get a compiler error.
error: no match for 'operator<<' in 'std::operator>>

```
#include <iostream>
using namespace std;

int main()
{
  string name;

  cout << "Enter a name? ";
  cin >> name << endl;

  cout << name << ", awesome!";
  cout << endl;
  return 0;
}
```

- All operators within a cin statement should point right.
- [✓ Output and input cannot be mixed in a single statement.]
- Only one operator can exist on a single line of code, as in:

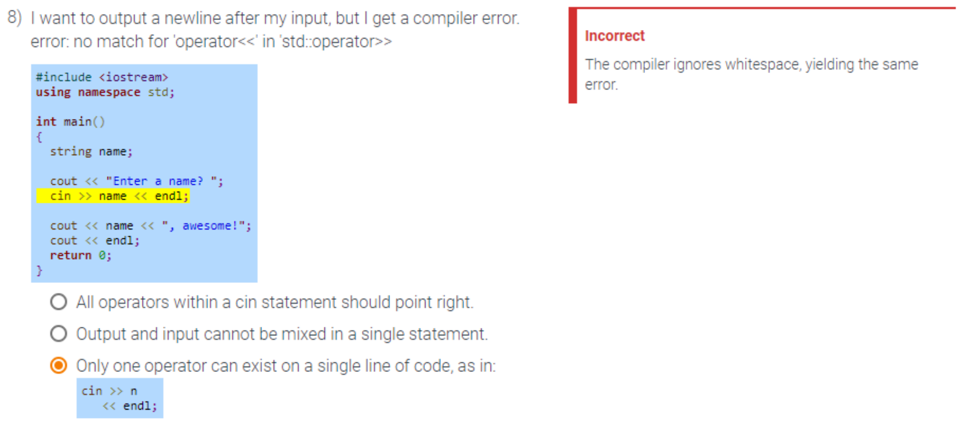
```
cin >> n
   << endl;
```

Correct

Two statements are required.

```
cin >> name;
cout << endl;
```

Figure 43: Basic Output, Question #8, Answer #2


8) I want to output a newline after my input, but I get a compiler error.
error: no match for 'operator<<' in 'std::operator>>

```
#include <iostream>
using namespace std;

int main()
{
  string name;

  cout << "Enter a name? ";
  cin >> name << endl;

  cout << name << ", awesome!";
  cout << endl;
  return 0;
}
```

- All operators within a cin statement should point right.
- Output and input cannot be mixed in a single statement.
- [✓ Only one operator can exist on a single line of code, as in:]

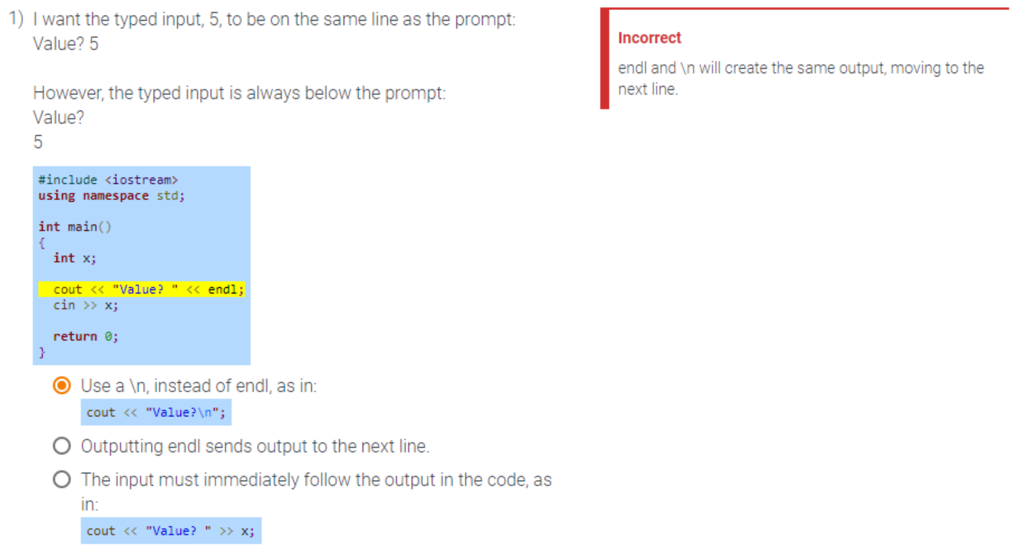
```
cin >> n
   << endl;
```

Incorrect

The compiler ignores whitespace, yielding the same error.

Figure 44: Basic Output, Question #8, Answer #3

Basic Input Exercises



1) I want the typed input, 5, to be on the same line as the prompt:
Value? 5

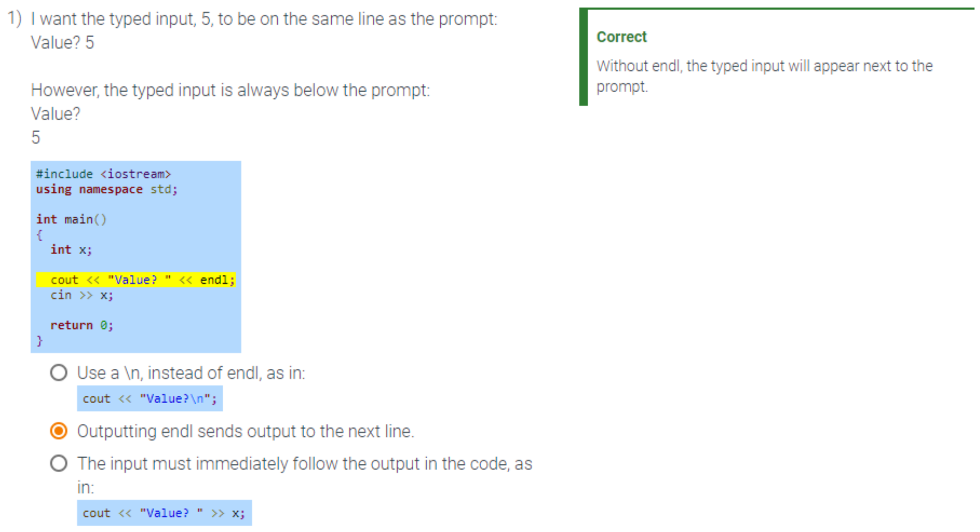
However, the typed input is always below the prompt:
Value?
5

```
#include <iostream>
using namespace std;

int main()
{
  int x;

  cout << "Value? " << endl;
  cin >> x;

  return 0;
}
```

- [✓ Use a \n, instead of endl, as in:]
  `cout << "Value?\n";`
- Outputting endl sends output to the next line.
- The input must immediately follow the output in the code, as in:
  `cout << "Value? " >> x;`

Incorrect
endl and \n will create the same output, moving to the next line.

Figure 45: Basic Input, Question #1, Answer #1



1) I want the typed input, 5, to be on the same line as the prompt:
Value? 5

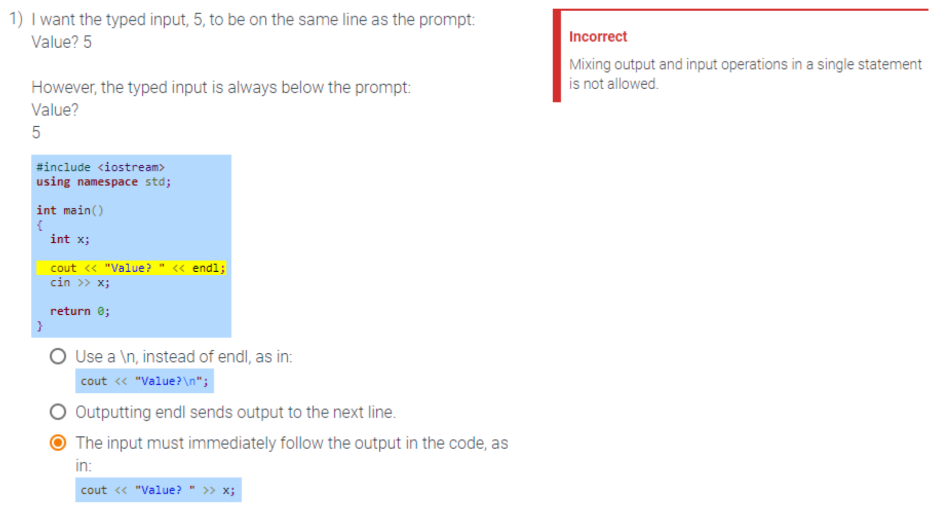
However, the typed input is always below the prompt:
Value?
5

```
#include <iostream>
using namespace std;

int main()
{
  int x;

  cout << "Value? " << endl;
  cin >> x;

  return 0;
}
```

- Use a \n, instead of endl, as in:
  `cout << "Value?\n";`
- [✓ Outputting endl sends output to the next line.]
- The input must immediately follow the output in the code, as in:
  `cout << "Value? " >> x;`

Correct
Without endl, the typed input will appear next to the prompt.

Figure 46: Basic Input, Question #1, Answer #2



1) I want the typed input, 5, to be on the same line as the prompt:
Value? 5

However, the typed input is always below the prompt:
Value?
5

```
#include <iostream>
using namespace std;

int main()
{
  int x;

  cout << "Value? " << endl;
  cin >> x;

  return 0;
}
```

- Use a \n, instead of endl, as in:

```
cout << "Value?\n";
```

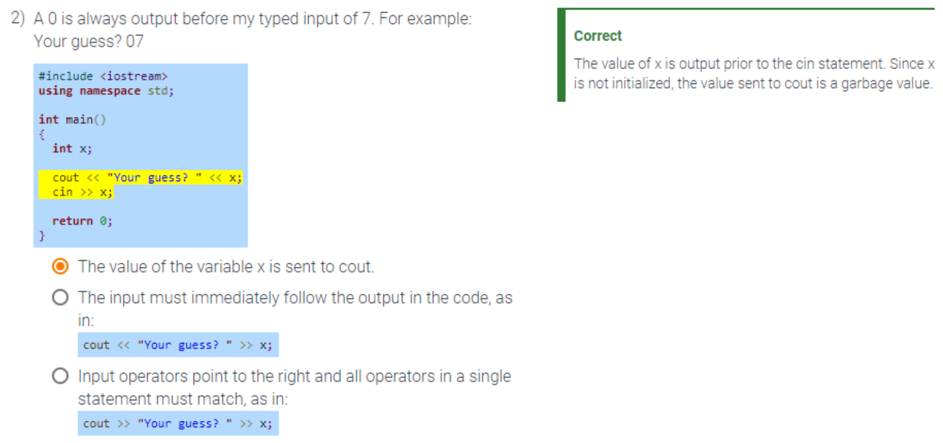
- Outputting endl sends output to the next line.
- [✓ The input must immediately follow the output in the code, as in:]

```
cout << "Value? " >> x;
```

Incorrect

Mixing output and input operations in a single statement is not allowed.

Figure 47: Basic Input, Question #1, Answer #3



2) A 0 is always output before my typed input of 7. For example:
Your guess? 07

```
#include <iostream>
using namespace std;

int main()
{
  int x;

  cout << "Your guess? " << x;
  cin >> x;

  return 0;
}
```

- [✓ The value of the variable x is sent to cout.]
- The input must immediately follow the output in the code, as in:

```
cout << "Your guess? " >> x;
```

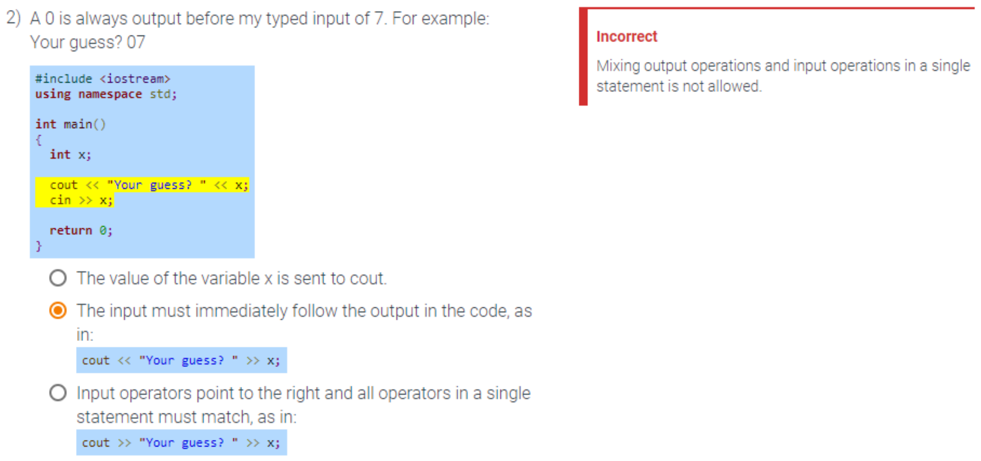
- Input operators point to the right and all operators in a single statement must match, as in:

```
cout >> "Your guess? " >> x;
```

Correct

The value of x is output prior to the cin statement. Since x is not initialized, the value sent to cout is a garbage value.

Figure 48: Basic Input, Question #2, Answer #1



2) A 0 is always output before my typed input of 7. For example:
Your guess? 07

```
#include <iostream>
using namespace std;

int main()
{
  int x;

  cout << "Your guess? " << x;
  cin >> x;

  return 0;
}
```

( ) The value of the variable x is sent to cout.

[✓ The input must immediately follow the output in the code, as in:]

```
cout << "Your guess? " >> x;
```

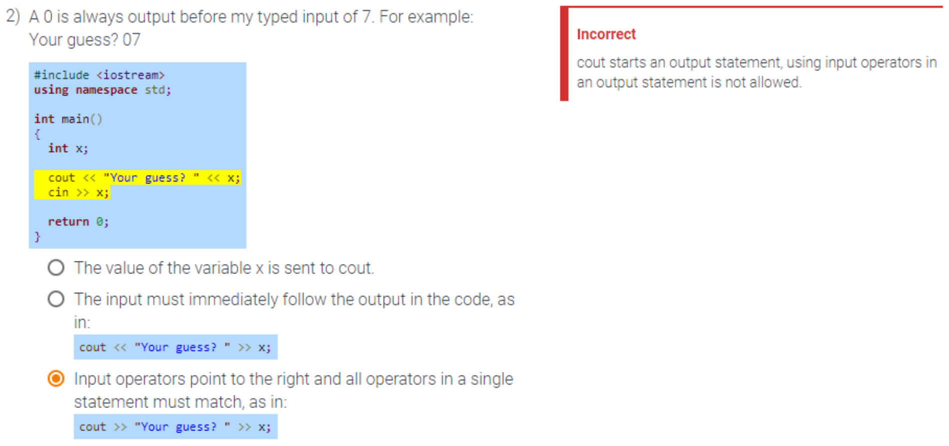
( ) Input operators point to the right and all operators in a single statement must match, as in:

```
cout >> "Your guess? " >> x;
```

Incorrect

Mixing output operations and input operations in a single statement is not allowed.

Figure 49: Basic Input, Question #2, Answer #2



2) A 0 is always output before my typed input of 7. For example:
Your guess? 07

```
#include <iostream>
using namespace std;

int main()
{
  int x;

  cout << "Your guess? " << x;
  cin >> x;

  return 0;
}
```

( ) The value of the variable x is sent to cout.

( ) The input must immediately follow the output in the code, as in:

```
cout << "Your guess? " >> x;
```

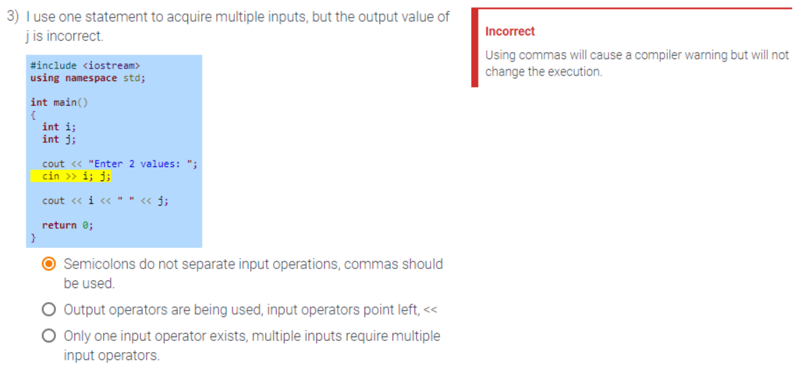
[✓ Input operators point to the right and all operators in a single statement must match, as in:]

```
cout >> "Your guess? " >> x;
```

Incorrect

cout starts an output statement, using input operators in an output statement is not allowed.

Figure 50: Basic Input, Question #2, Answer #3


3) I use one statement to acquire multiple inputs, but the output value of j is incorrect.

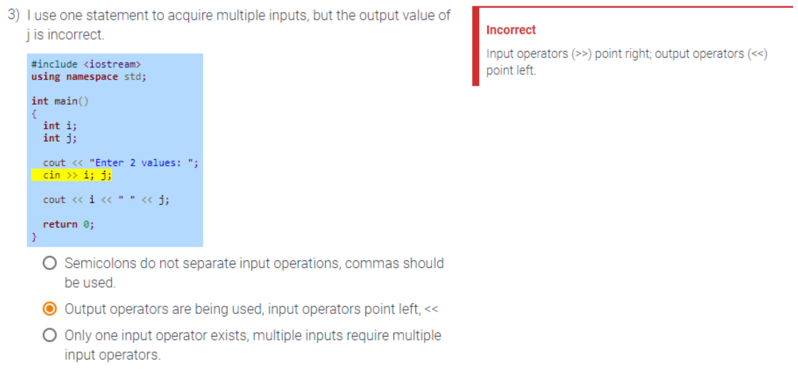
```
#include <iostream>
using namespace std;

int main()
{
  int i;
  int j;

  cout << "Enter 2 values: ";
  cin >> i; j;

  cout << i << " " << j;

  return 0;
}
```

- [✓ Semicolons do not separate input operations, commas should be used.]
- Output operators are being used, input operators point left, <<
- Only one input operator exists, multiple inputs require multiple input operators.

Incorrect

Using commas will cause a compiler warning but will not change the execution.

Figure 51: Basic Input, Question #3, Answer #1


3) I use one statement to acquire multiple inputs, but the output value of j is incorrect.

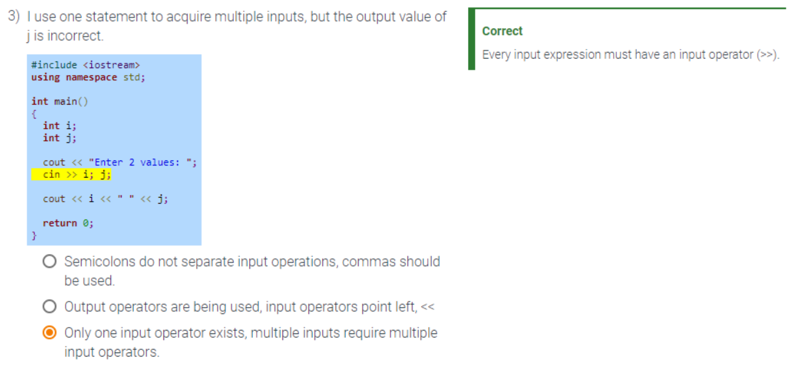
```
#include <iostream>
using namespace std;

int main()
{
  int i;
  int j;

  cout << "Enter 2 values: ";
  cin >> i; j;

  cout << i << " " << j;

  return 0;
}
```

- Semicolons do not separate input operations, commas should be used.
- [✓ Output operators are being used, input operators point left, <<]
- Only one input operator exists, multiple inputs require multiple input operators.

Incorrect

Input operators (>>) point right; output operators (<<) point left.

Figure 52: Basic Input, Question #3, Answer #2


3) I use one statement to acquire multiple inputs, but the output value of j is incorrect.

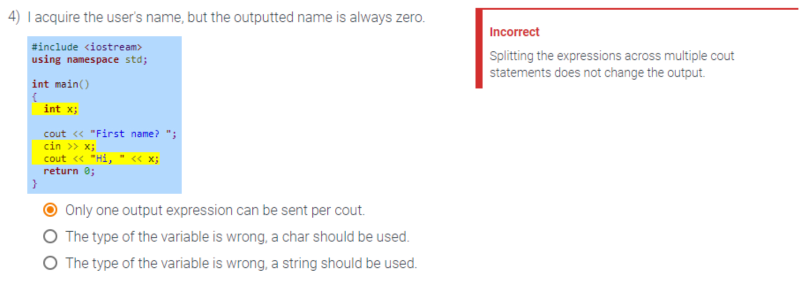
```
#include <iostream>
using namespace std;

int main()
{
  int i;
  int j;

  cout << "Enter 2 values: ";
  cin >> i; j;

  cout << i << " " << j;

  return 0;
}
```

- Semicolons do not separate input operations, commas should be used.
- Output operators are being used, input operators point left, <<
- [✓ Only one input operator exists, multiple inputs require multiple input operators.]

Correct

Every input expression must have an input operator (>>).

Figure 53: Basic Input, Question #3, Answer #3



4) I acquire the user's name, but the outputted name is always zero.

```
#include <iostream>
using namespace std;

int main()
{
  int x;

  cout << "First name? ";
  cin >> x;
  cout << "Hi, " << x;
  return 0;
}
```

[✓ Only one output expression can be sent per cout.]

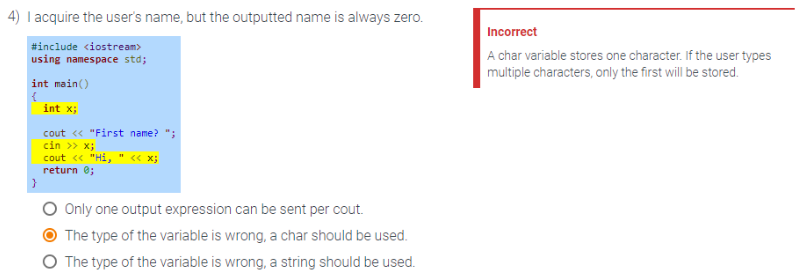
The type of the variable is wrong, a char should be used.

The type of the variable is wrong, a string should be used.

Incorrect

Splitting the expressions across multiple cout statements does not change the output.

Figure 54: Basic Input, Question #4, Answer #1



4) I acquire the user's name, but the outputted name is always zero.

```
#include <iostream>
using namespace std;

int main()
{
  int x;

  cout << "First name? ";
  cin >> x;
  cout << "Hi, " << x;
  return 0;
}
```

Only one output expression can be sent per cout.

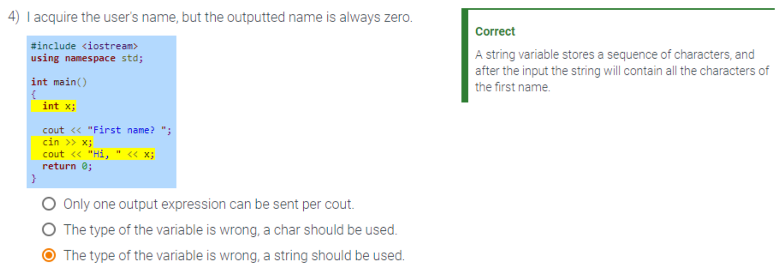
[✓ The type of the variable is wrong, a char should be used.]

The type of the variable is wrong, a string should be used.

Incorrect

A char variable stores one character. If the user types multiple characters, only the first will be stored.

Figure 55: Basic Input, Question #4, Answer #2



4) I acquire the user's name, but the outputted name is always zero.

```
#include <iostream>
using namespace std;

int main()
{
  int x;

  cout << "First name? ";
  cin >> x;
  cout << "Hi, " << x;
  return 0;
}
```

Only one output expression can be sent per cout.

The type of the variable is wrong, a char should be used.

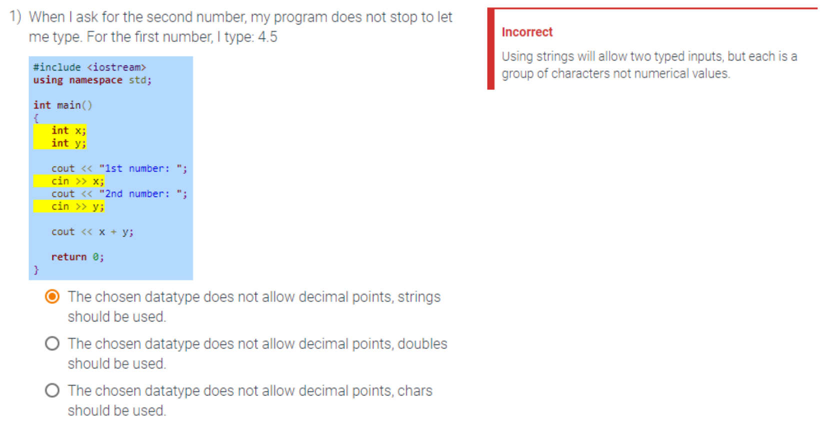
[✓ The type of the variable is wrong, a string should be used.]

Correct

A string variable stores a sequence of characters, and after the input the string will contain all the characters of the first name.

Figure 56: Basic Input, Question #4, Answer #3

Basic Math Exercises



1) When I ask for the second number, my program does not stop to let me type. For the first number, I type: 4.5

```
#include <iostream>
using namespace std;

int main()
{
    int x;
    int y;

    cout << "1st number: ";
    cin >> x;
    cout << "2nd number: ";
    cin >> y;

    cout << x + y;

    return 0;
}
```

[✓ The chosen datatype does not allow decimal points, strings should be used.]

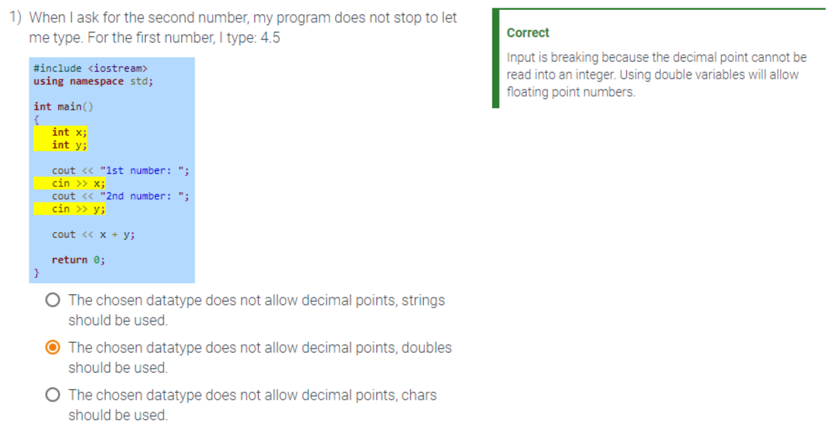
The chosen datatype does not allow decimal points, doubles should be used.

The chosen datatype does not allow decimal points, chars should be used.

Incorrect

Using strings will allow two typed inputs, but each is a group of characters not numerical values.

Figure 57: Basic Math, Question #1, Answer #1



1) When I ask for the second number, my program does not stop to let me type. For the first number, I type: 4.5

```
#include <iostream>
using namespace std;

int main()
{
    int x;
    int y;

    cout << "1st number: ";
    cin >> x;
    cout << "2nd number: ";
    cin >> y;

    cout << x + y;

    return 0;
}
```

The chosen datatype does not allow decimal points, strings should be used.

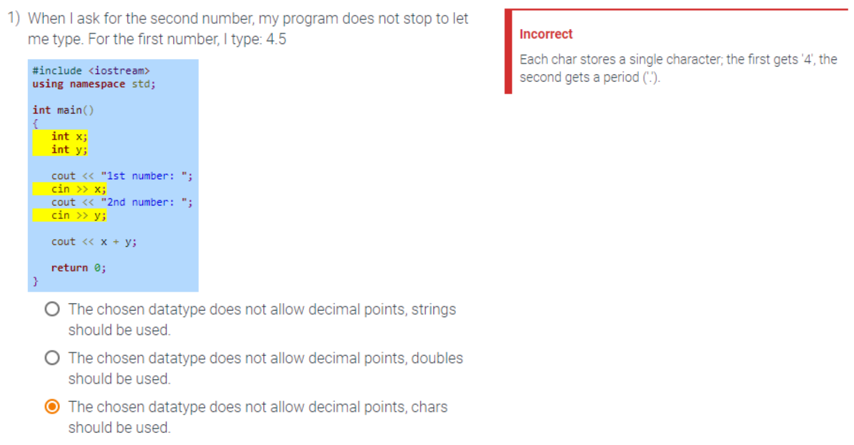
[✓ The chosen datatype does not allow decimal points, doubles should be used.]

The chosen datatype does not allow decimal points, chars should be used.

Correct

Input is breaking because the decimal point cannot be read into an integer. Using double variables will allow floating point numbers.

Figure 58: Basic Math, Question #1, Answer #2


1) When I ask for the second number, my program does not stop to let me type. For the first number, I type: 4.5

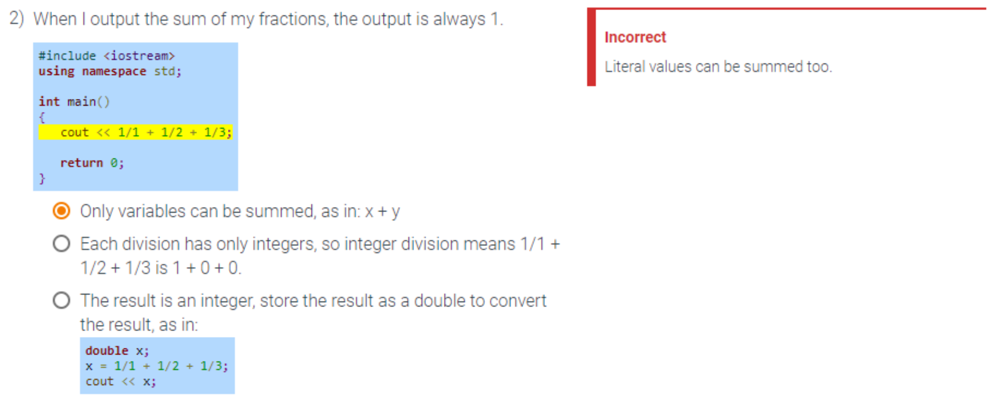
```
#include <iostream>
using namespace std;

int main()
{
   int x;
   int y;

   cout << "1st number: ";
   cin >> x;
   cout << "2nd number: ";
   cin >> y;

   cout << x + y;

   return 0;
}
```

- The chosen datatype does not allow decimal points, strings should be used.
- The chosen datatype does not allow decimal points, doubles should be used.
- [✓ The chosen datatype does not allow decimal points, chars should be used.]

Incorrect

Each char stores a single character; the first gets '4', the second gets a period ('.').

Figure 59: Basic Math, Question #1, Answer #3


2) When I output the sum of my fractions, the output is always 1.

```
#include <iostream>
using namespace std;

int main()
{
   cout << 1/1 + 1/2 + 1/3;

   return 0;
}
```

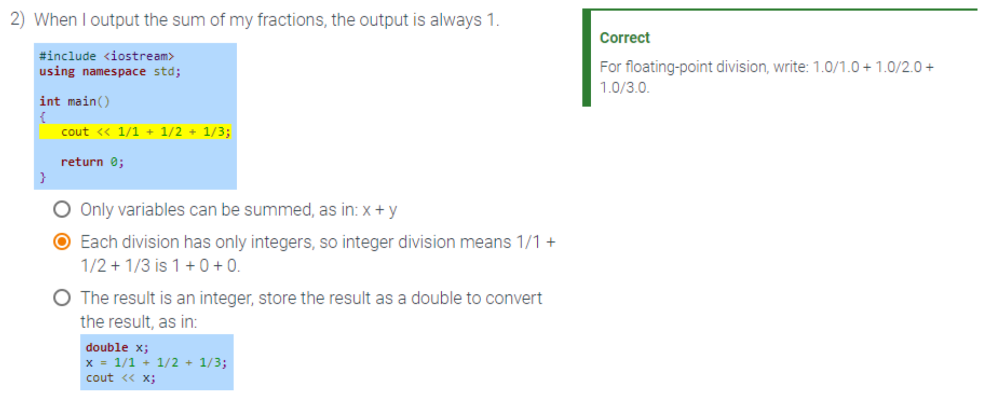
- [✓ Only variables can be summed, as in: x + y]
- Each division has only integers, so integer division means 1/1 + 1/2 + 1/3 is 1 + 0 + 0.
- The result is an integer, store the result as a double to convert the result, as in:

```
double x;
x = 1/1 + 1/2 + 1/3;
cout << x;
```

Incorrect

Literal values can be summed too.

Figure 60: Basic Math, Question #2, Answer #1



2) When I output the sum of my fractions, the output is always 1.

```
#include <iostream>
using namespace std;

int main()
{
   cout << 1/1 + 1/2 + 1/3;

   return 0;
}
```

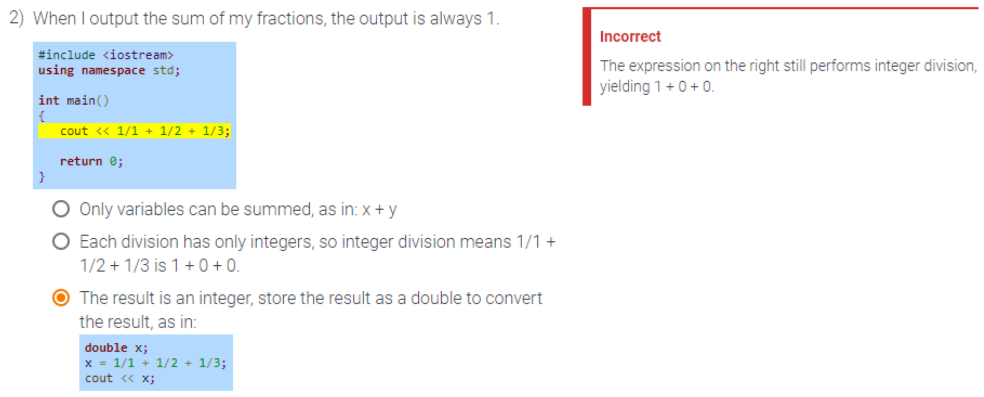
- ○ Only variables can be summed, as in: x + y
- ◉ Each division has only integers, so integer division means 1/1 + 1/2 + 1/3 is 1 + 0 + 0.
- ○ The result is an integer, store the result as a double to convert the result, as in:

```
double x;
x = 1/1 + 1/2 + 1/3;
cout << x;
```

Correct

For floating-point division, write: 1.0/1.0 + 1.0/2.0 + 1.0/3.0.

Figure 61: Basic Math, Question #2, Answer #2



2) When I output the sum of my fractions, the output is always 1.

```
#include <iostream>
using namespace std;

int main()
{
   cout << 1/1 + 1/2 + 1/3;

   return 0;
}
```

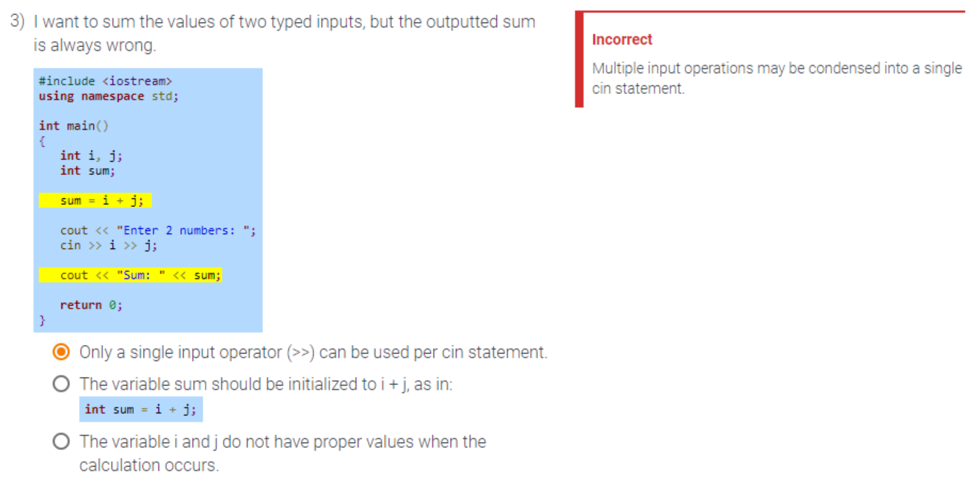
- ○ Only variables can be summed, as in: x + y
- ○ Each division has only integers, so integer division means 1/1 + 1/2 + 1/3 is 1 + 0 + 0.
- ◉ The result is an integer, store the result as a double to convert the result, as in:

```
double x;
x = 1/1 + 1/2 + 1/3;
cout << x;
```

Incorrect

The expression on the right still performs integer division, yielding 1 + 0 + 0.

Figure 62: Basic Math, Question #2, Answer #3



3) I want to sum the values of two typed inputs, but the outputted sum is always wrong.

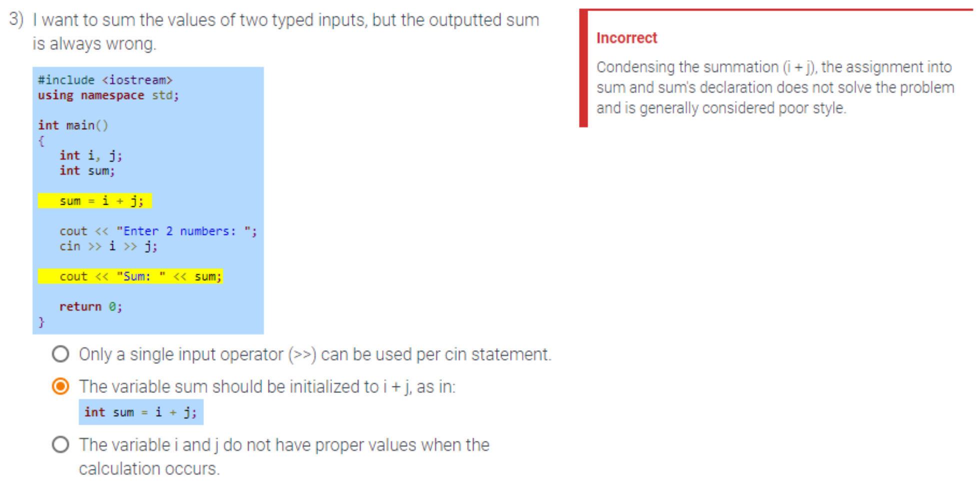
```
#include <iostream>
using namespace std;

int main()
{
   int i, j;
   int sum;

   sum = i + j;

   cout << "Enter 2 numbers: ";
   cin >> i >> j;

   cout << "Sum: " << sum;

   return 0;
}
```

- [✓ Only a single input operator (>>) can be used per cin statement.]
- The variable sum should be initialized to i + j, as in: `int sum = i + j;`
- The variable i and j do not have proper values when the calculation occurs.

Incorrect

Multiple input operations may be condensed into a single cin statement.

Figure 63: Basic Math, Question #3, Answer #1



3) I want to sum the values of two typed inputs, but the outputted sum is always wrong.

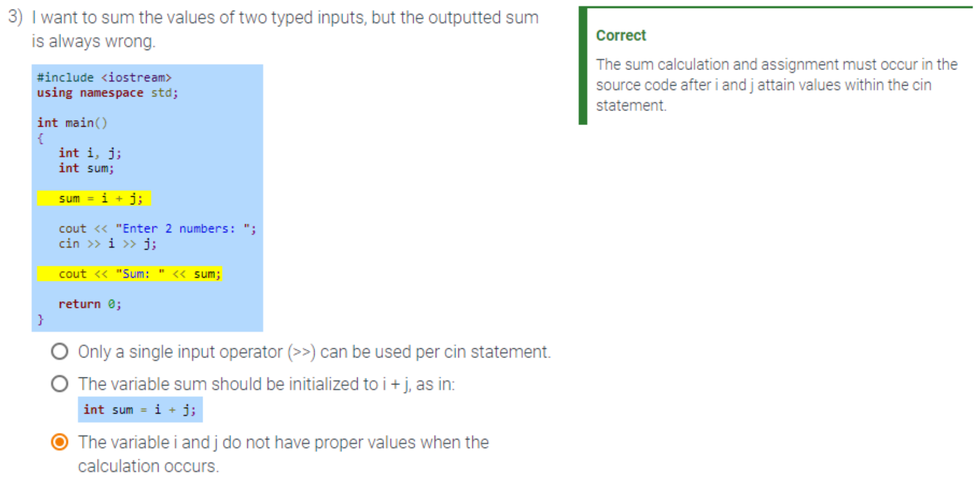
```
#include <iostream>
using namespace std;

int main()
{
   int i, j;
   int sum;

   sum = i + j;

   cout << "Enter 2 numbers: ";
   cin >> i >> j;

   cout << "Sum: " << sum;

   return 0;
}
```

- Only a single input operator (>>) can be used per cin statement.
- [✓ The variable sum should be initialized to i + j, as in: `int sum = i + j;`]
- The variable i and j do not have proper values when the calculation occurs.

Incorrect

Condensing the summation (i + j), the assignment into sum and sum's declaration does not solve the problem and is generally considered poor style.

Figure 64: Basic Math, Question #3, Answer #2



3) I want to sum the values of two typed inputs, but the outputted sum is always wrong.

```
#include <iostream>
using namespace std;

int main()
{
   int i, j;
   int sum;

   sum = i + j;

   cout << "Enter 2 numbers: ";
   cin >> i >> j;

   cout << "Sum: " << sum;

   return 0;
}
```

○ Only a single input operator (>>) can be used per cin statement.

○ The variable sum should be initialized to i + j, as in:

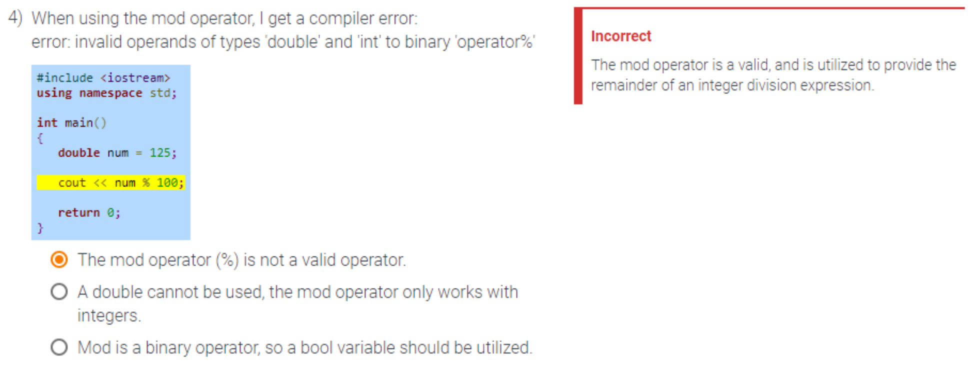
```
int sum = i + j;
```

● The variable i and j do not have proper values when the calculation occurs.

Correct

The sum calculation and assignment must occur in the source code after i and j attain values within the cin statement.

Figure 65: Basic Math, Question #3, Answer #3



4) When using the mod operator, I get a compiler error:
error: invalid operands of types 'double' and 'int' to binary 'operator%'

```
#include <iostream>
using namespace std;

int main()
{
   double num = 125;

   cout << num % 100;

   return 0;
}
```

● The mod operator (%) is not a valid operator.

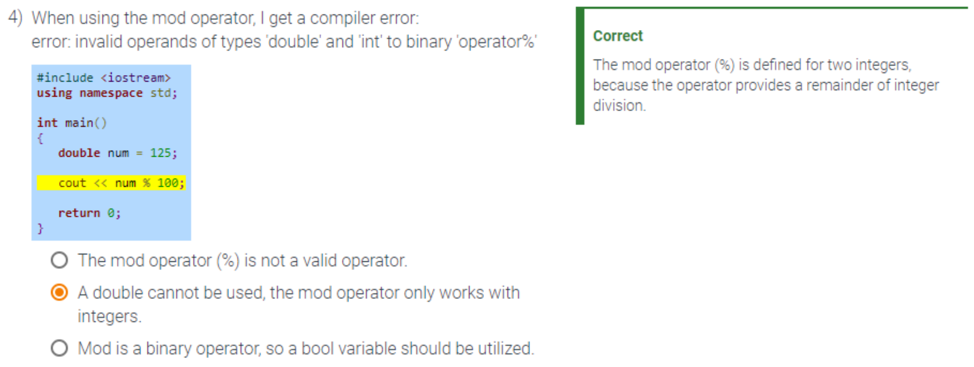
○ A double cannot be used, the mod operator only works with integers.

○ Mod is a binary operator, so a bool variable should be utilized.

Incorrect

The mod operator is a valid, and is utilized to provide the remainder of an integer division expression.

Figure 66: Basic Math, Question #4, Answer #1



4) When using the mod operator, I get a compiler error:
error: invalid operands of types 'double' and 'int' to binary 'operator%'

```
#include <iostream>
using namespace std;

int main()
{
   double num = 125;

   cout << num % 100;

   return 0;
}
```

○ The mod operator (%) is not a valid operator.

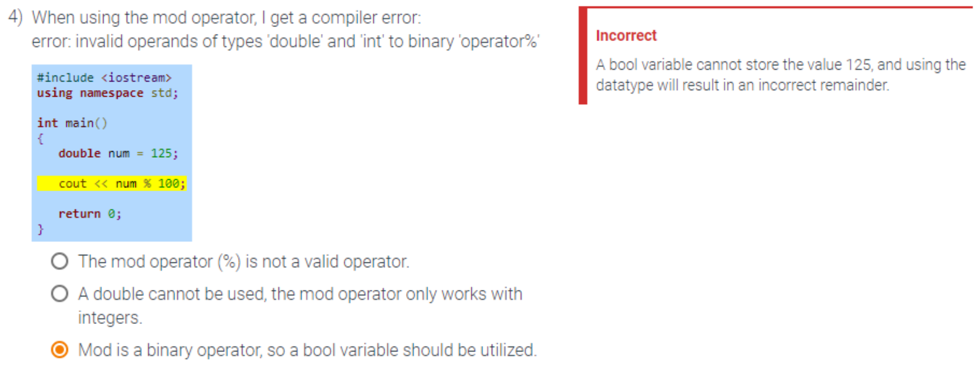
● A double cannot be used, the mod operator only works with integers.

○ Mod is a binary operator, so a bool variable should be utilized.

Correct

The mod operator (%) is defined for two integers, because the operator provides a remainder of integer division.

Figure 67: Basic Math, Question #4, Answer #2



4) When using the mod operator, I get a compiler error:
error: invalid operands of types 'double' and 'int' to binary 'operator%'

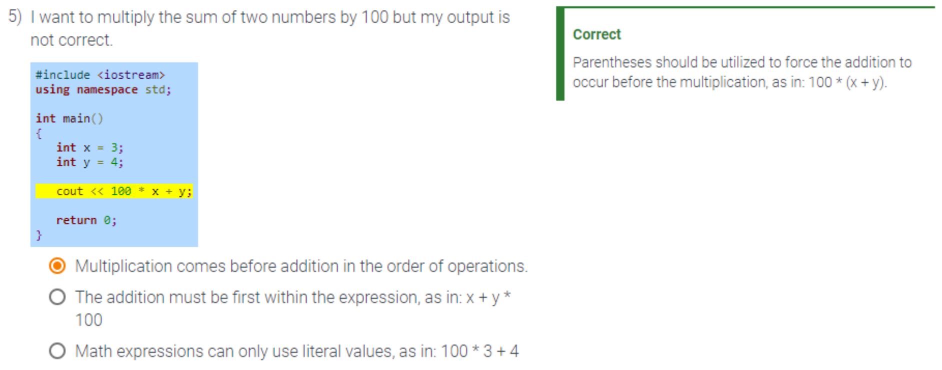
```
#include <iostream>
using namespace std;

int main()
{
    double num = 125;

    cout << num % 100;

    return 0;
}
```

- The mod operator (%) is not a valid operator.
- A double cannot be used, the mod operator only works with integers.
- [✓ Mod is a binary operator, so a bool variable should be utilized.]

Incorrect

A bool variable cannot store the value 125, and using the datatype will result in an incorrect remainder.

Figure 68: Basic Math, Question #4, Answer #3



5) I want to multiply the sum of two numbers by 100 but my output is not correct.

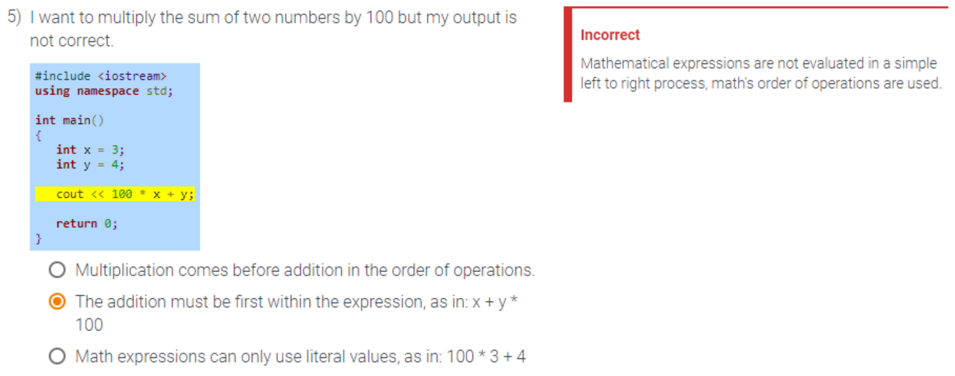
```
#include <iostream>
using namespace std;

int main()
{
    int x = 3;
    int y = 4;

    cout << 100 * x + y;

    return 0;
}
```

- [✓ Multiplication comes before addition in the order of operations.]
- The addition must be first within the expression, as in: x + y * 100
- Math expressions can only use literal values, as in: 100 * 3 + 4

Correct

Parentheses should be utilized to force the addition to occur before the multiplication, as in: 100 * (x + y).

Figure 69: Basic Math, Question #5, Answer #1



5) I want to multiply the sum of two numbers by 100 but my output is not correct.

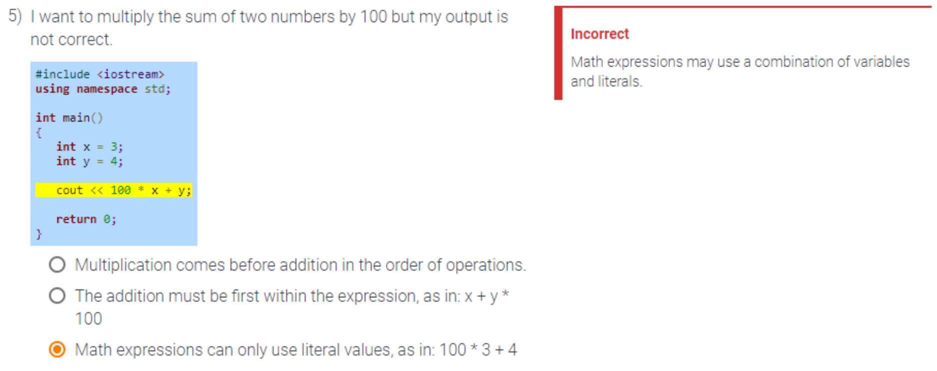
```
#include <iostream>
using namespace std;

int main()
{
    int x = 3;
    int y = 4;

    cout << 100 * x + y;

    return 0;
}
```

- Multiplication comes before addition in the order of operations.
- [✓ The addition must be first within the expression, as in: x + y * 100]
- Math expressions can only use literal values, as in: 100 * 3 + 4

Incorrect

Mathematical expressions are not evaluated in a simple left to right process, math's order of operations are used.

Figure 70: Basic Math, Question #5, Answer #2


5) I want to multiply the sum of two numbers by 100 but my output is not correct.

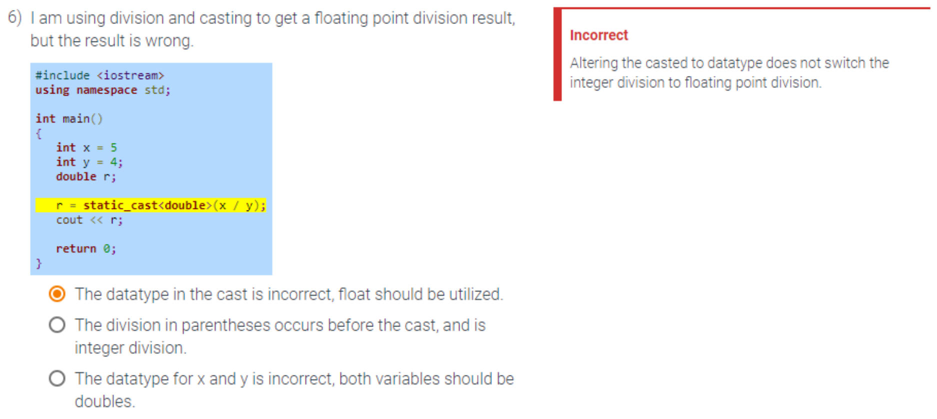
```
#include <iostream>
using namespace std;

int main()
{
   int x = 3;
   int y = 4;

   cout << 100 * x + y;

   return 0;
}
```

- ○ Multiplication comes before addition in the order of operations.
- ○ The addition must be first within the expression, as in: x + y * 100
- [✓ Math expressions can only use literal values, as in: 100 * 3 + 4]

Incorrect

Math expressions may use a combination of variables and literals.

Figure 71: Basic Math, Question #5, Answer #3


6) I am using division and casting to get a floating point division result, but the result is wrong.

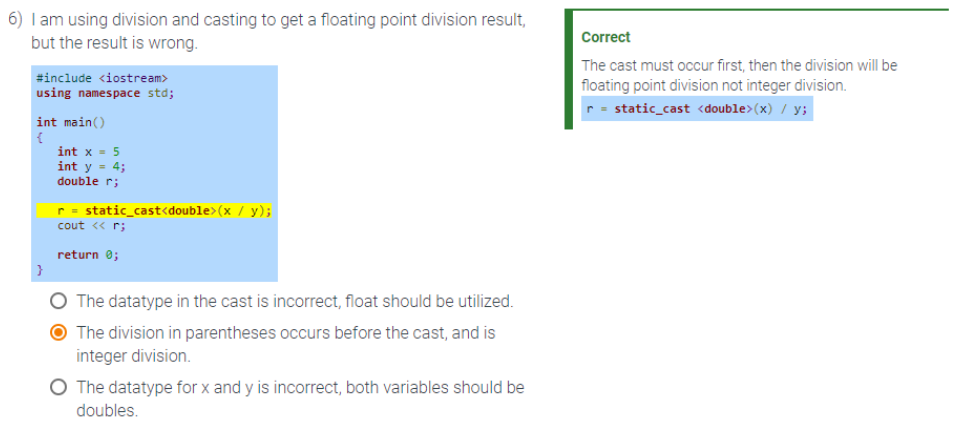
```
#include <iostream>
using namespace std;

int main()
{
   int x = 5
   int y = 4;
   double r;

   r = static_cast<double>(x / y);
   cout << r;

   return 0;
}
```

- [✓ The datatype in the cast is incorrect, float should be utilized.]
- ○ The division in parentheses occurs before the cast, and is integer division.
- ○ The datatype for x and y is incorrect, both variables should be doubles.

Incorrect

Altering the casted to datatype does not switch the integer division to floating point division.

Figure 72: Basic Math, Question #6, Answer #1


6) I am using division and casting to get a floating point division result, but the result is wrong.

```
#include <iostream>
using namespace std;

int main()
{
    int x = 5
    int y = 4;
    double r;

    r = static_cast<double>(x / y);
    cout << r;

    return 0;
}
```

○ The datatype in the cast is incorrect, float should be utilized.

[✓ The division in parentheses occurs before the cast, and is integer division.]

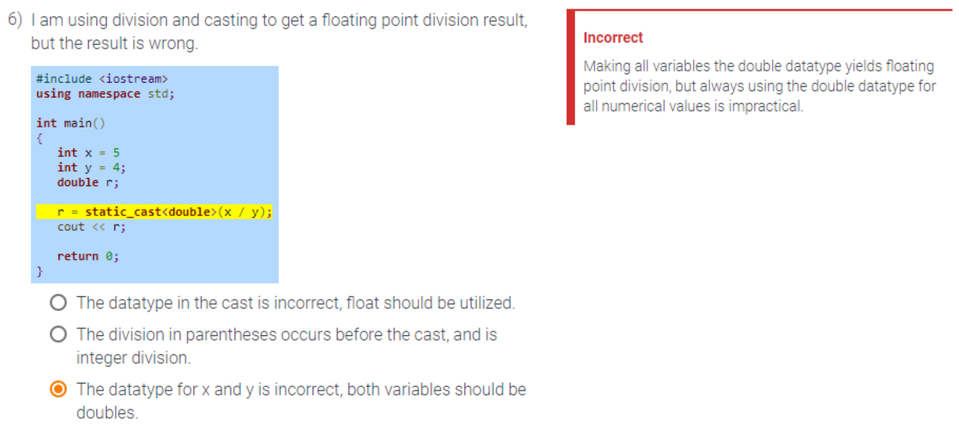
○ The datatype for x and y is incorrect, both variables should be doubles.

Correct

The cast must occur first, then the division will be floating point division not integer division.

```
r = static_cast <double>(x) / y;
```

Figure 73: Basic Math, Question #6, Answer #2


6) I am using division and casting to get a floating point division result, but the result is wrong.

```
#include <iostream>
using namespace std;

int main()
{
    int x = 5
    int y = 4;
    double r;

    r = static_cast<double>(x / y);
    cout << r;

    return 0;
}
```

○ The datatype in the cast is incorrect, float should be utilized.

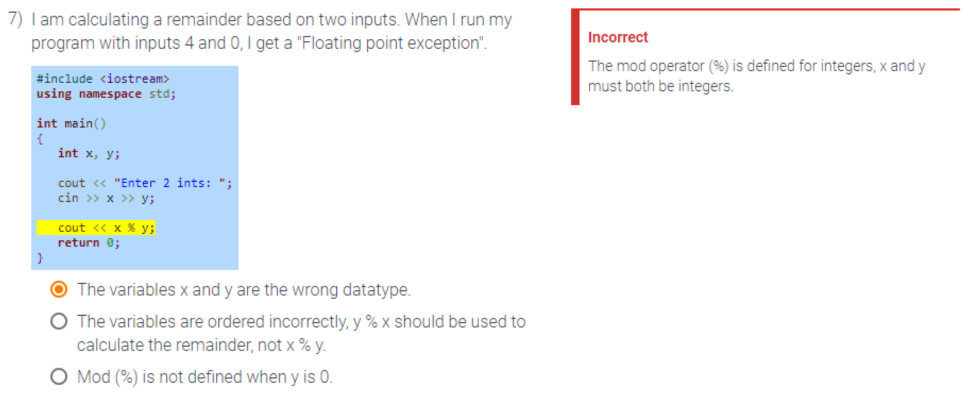
○ The division in parentheses occurs before the cast, and is integer division.

[✓ The datatype for x and y is incorrect, both variables should be doubles.]

Incorrect

Making all variables the double datatype yields floating point division, but always using the double datatype for all numerical values is impractical.

Figure 74: Basic Math, Question #6, Answer #3


7) I am calculating a remainder based on two inputs. When I run my program with inputs 4 and 0, I get a "Floating point exception".

```
#include <iostream>
using namespace std;

int main()
{
    int x, y;

    cout << "Enter 2 ints: ";
    cin >> x >> y;

    cout << x % y;
    return 0;
}
```

[✓ The variables x and y are the wrong datatype.]

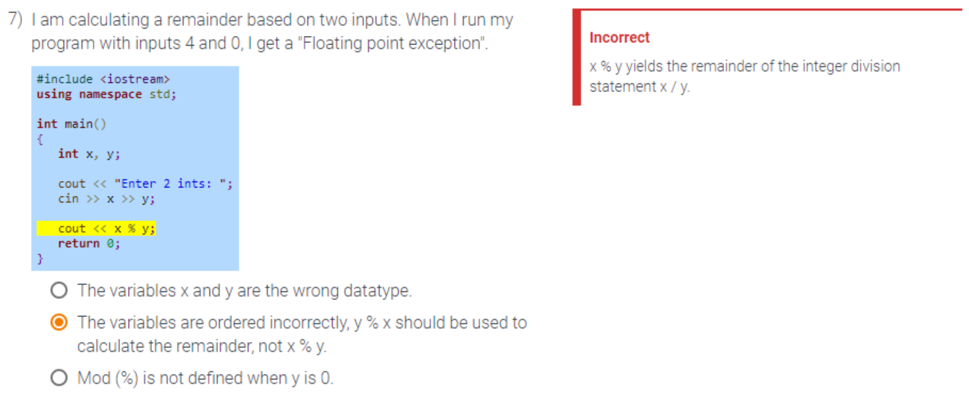
The variables are ordered incorrectly, y % x should be used to calculate the remainder, not x % y.

Mod (%) is not defined when y is 0.

Incorrect

The mod operator (%) is defined for integers, x and y must both be integers.

Figure 75: Basic Math, Question #7, Answer #1


7) I am calculating a remainder based on two inputs. When I run my program with inputs 4 and 0, I get a "Floating point exception".

```
#include <iostream>
using namespace std;

int main()
{
    int x, y;

    cout << "Enter 2 ints: ";
    cin >> x >> y;

    cout << x % y;
    return 0;
}
```

The variables x and y are the wrong datatype.

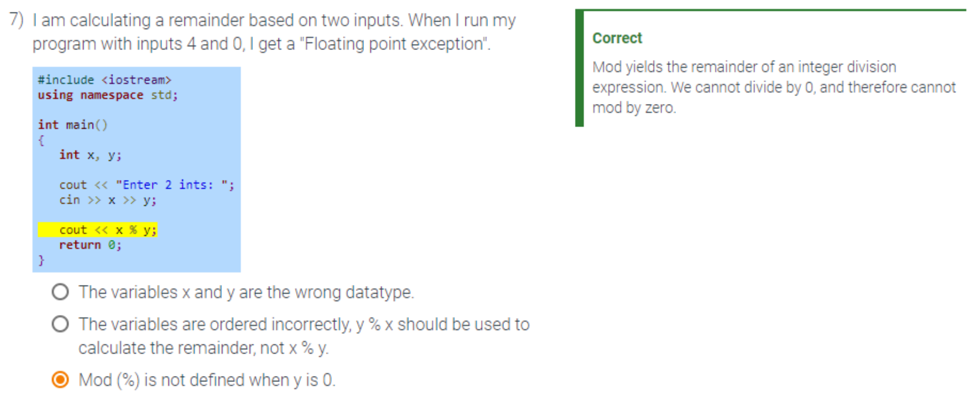
[✓ The variables are ordered incorrectly, y % x should be used to calculate the remainder, not x % y.]

Mod (%) is not defined when y is 0.

Incorrect

x % y yields the remainder of the integer division statement x / y.

Figure 76: Basic Math, Question #7, Answer #2


7) I am calculating a remainder based on two inputs. When I run my program with inputs 4 and 0, I get a "Floating point exception".

```
#include <iostream>
using namespace std;

int main()
{
    int x, y;

    cout << "Enter 2 ints: ";
    cin >> x >> y;

    cout << x % y;
    return 0;
}
```

The variables x and y are the wrong datatype.

The variables are ordered incorrectly, y % x should be used to calculate the remainder, not x % y.

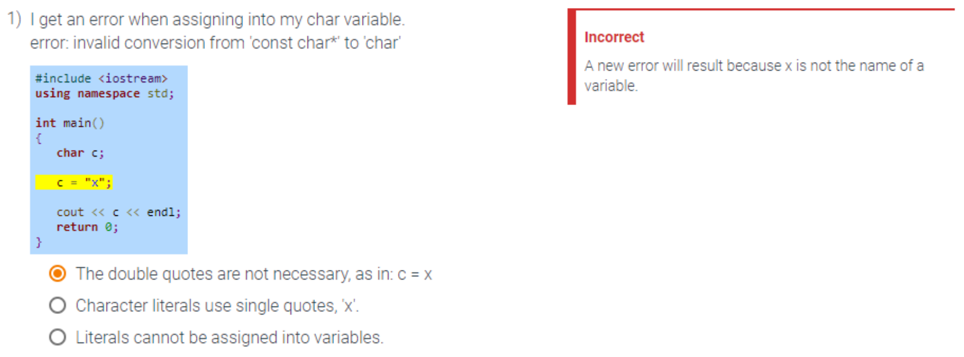
[✓ Mod (%) is not defined when y is 0.]

Correct

Mod yields the remainder of an integer division expression. We cannot divide by 0, and therefore cannot mod by zero.

Figure 77: Basic Math, Question #7, Answer #3

Basic Variable Exercises



1) I get an error when assigning into my char variable.
error: invalid conversion from 'const char*' to 'char'

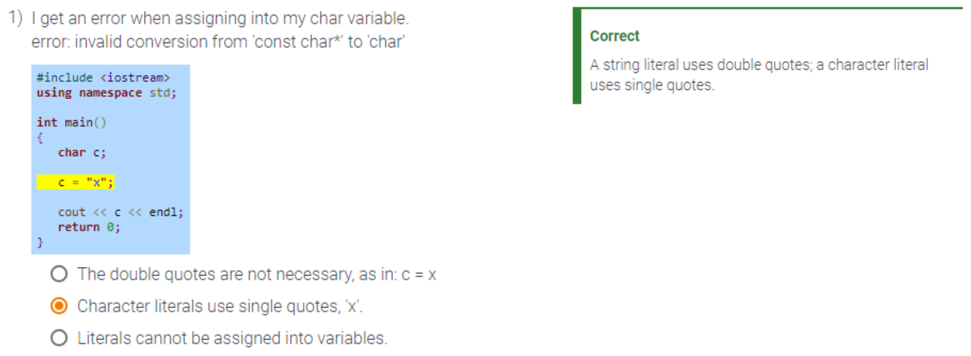
```
#include <iostream>
using namespace std;

int main()
{
   char c;

   c = "x";

   cout << c << endl;
   return 0;
}
```

- [✓ The double quotes are not necessary, as in: c = x]
- Character literals use single quotes, 'x'.
- Literals cannot be assigned into variables.

Incorrect

A new error will result because x is not the name of a variable.

Figure 78: Basic Variable, Question #1, Answer #1



1) I get an error when assigning into my char variable.
error: invalid conversion from 'const char*' to 'char'

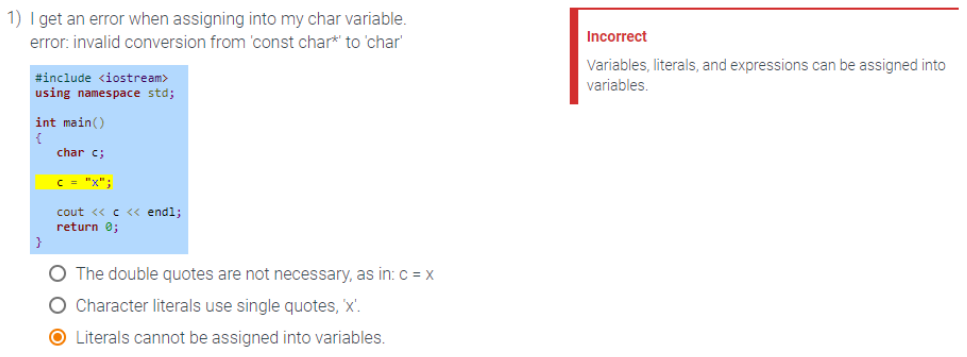
```
#include <iostream>
using namespace std;

int main()
{
   char c;

   c = "x";

   cout << c << endl;
   return 0;
}
```

- The double quotes are not necessary, as in: c = x
- [✓ Character literals use single quotes, 'x'.]
- Literals cannot be assigned into variables.

Correct

A string literal uses double quotes; a character literal uses single quotes.

Figure 79: Basic Variable, Question #1, Answer #2



1) I get an error when assigning into my char variable.
error: invalid conversion from 'const char*' to 'char'

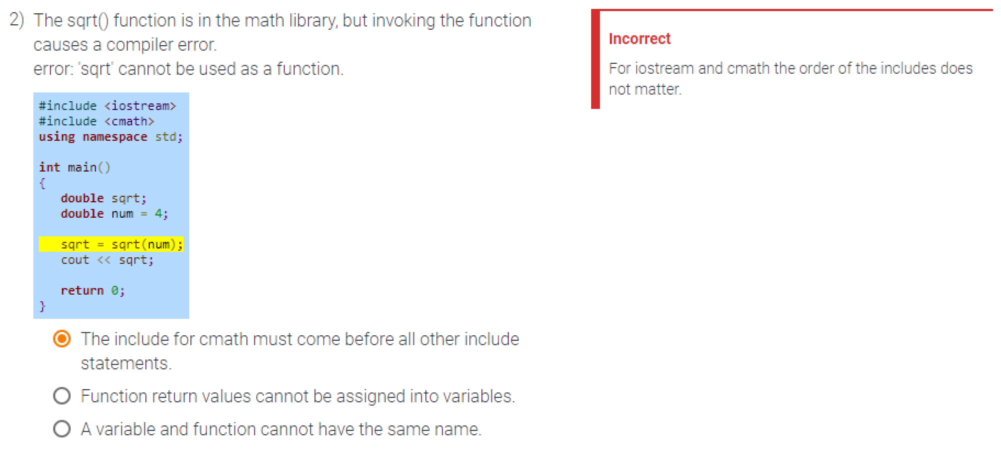
```
#include <iostream>
using namespace std;

int main()
{
   char c;

   c = "x";

   cout << c << endl;
   return 0;
}
```

- The double quotes are not necessary, as in: c = x
- Character literals use single quotes, 'x'.
- [✓ Literals cannot be assigned into variables.]

Incorrect

Variables, literals, and expressions can be assigned into variables.

Figure 80: Basic Variable, Question #1, Answer #3



2) The sqrt() function is in the math library, but invoking the function causes a compiler error.
error: 'sqrt' cannot be used as a function.

```
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
   double sqrt;
   double num = 4;

   sqrt = sqrt(num);
   cout << sqrt;

   return 0;
}
```

[✓ The include for cmath must come before all other include statements.]

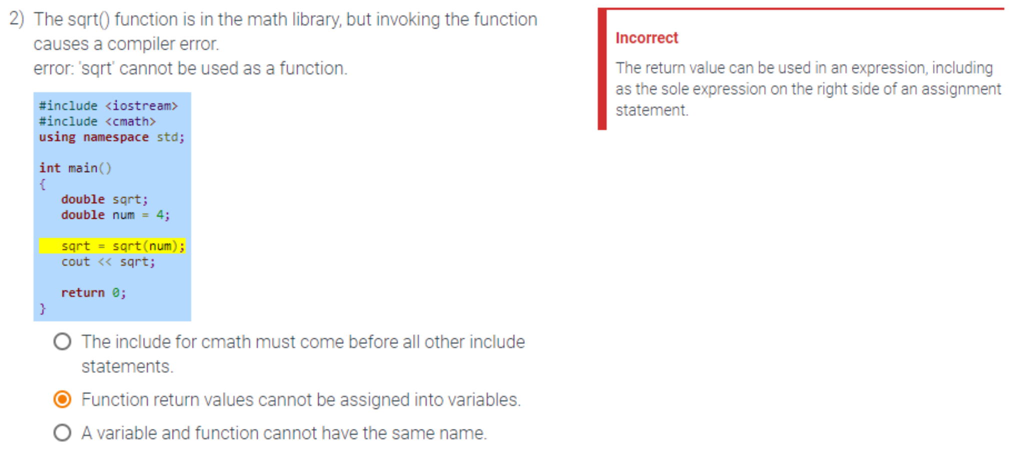
Function return values cannot be assigned into variables.

A variable and function cannot have the same name.

Incorrect

For iostream and cmath the order of the includes does not matter.

Figure 81: Basic Variable, Question #2, Answer #1



2) The sqrt() function is in the math library, but invoking the function causes a compiler error.
error: 'sqrt' cannot be used as a function.

```
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
   double sqrt;
   double num = 4;

   sqrt = sqrt(num);
   cout << sqrt;

   return 0;
}
```

The include for cmath must come before all other include statements.

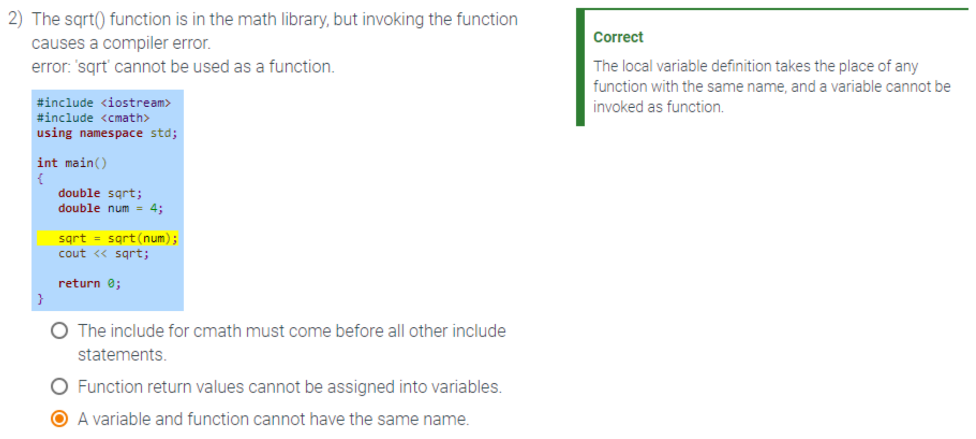
[✓ Function return values cannot be assigned into variables.]

A variable and function cannot have the same name.

Incorrect

The return value can be used in an expression, including as the sole expression on the right side of an assignment statement.

Figure 82: Basic Variable, Question #2, Answer #2



2) The sqrt() function is in the math library, but invoking the function causes a compiler error.
error: 'sqrt' cannot be used as a function.

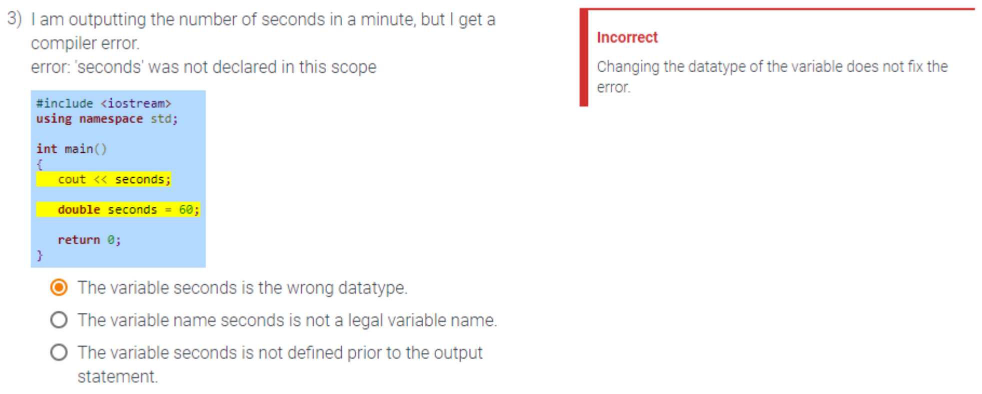
```
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
   double sqrt;
   double num = 4;

   sqrt = sqrt(num);
   cout << sqrt;

   return 0;
}
```

- The include for cmath must come before all other include statements.
- Function return values cannot be assigned into variables.
- [✓ A variable and function cannot have the same name.]

Correct

The local variable definition takes the place of any function with the same name, and a variable cannot be invoked as function.

Figure 83: Basic Variable, Question #1, Answer #3



3) I am outputting the number of seconds in a minute, but I get a compiler error.
error: 'seconds' was not declared in this scope

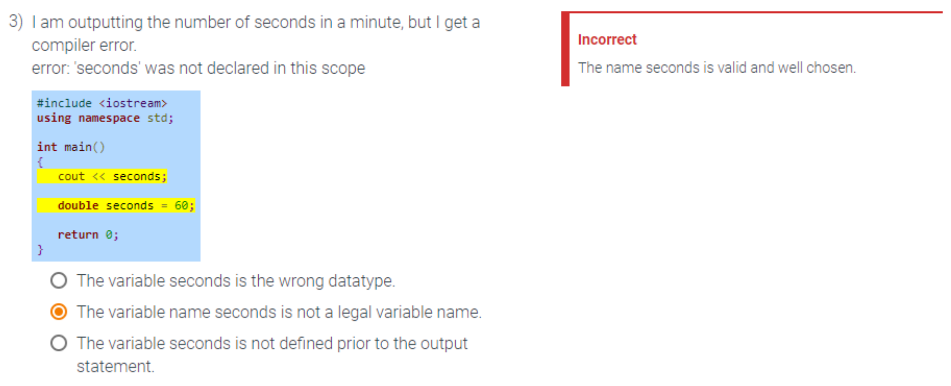
```
#include <iostream>
using namespace std;

int main()
{
   cout << seconds;

   double seconds = 60;

   return 0;
}
```

- [✓ The variable seconds is the wrong datatype.]
- The variable name seconds is not a legal variable name.
- The variable seconds is not defined prior to the output statement.

Incorrect

Changing the datatype of the variable does not fix the error.

Figure 84: Basic Variable, Question #3, Answer #1



3) I am outputting the number of seconds in a minute, but I get a compiler error.
error: 'seconds' was not declared in this scope

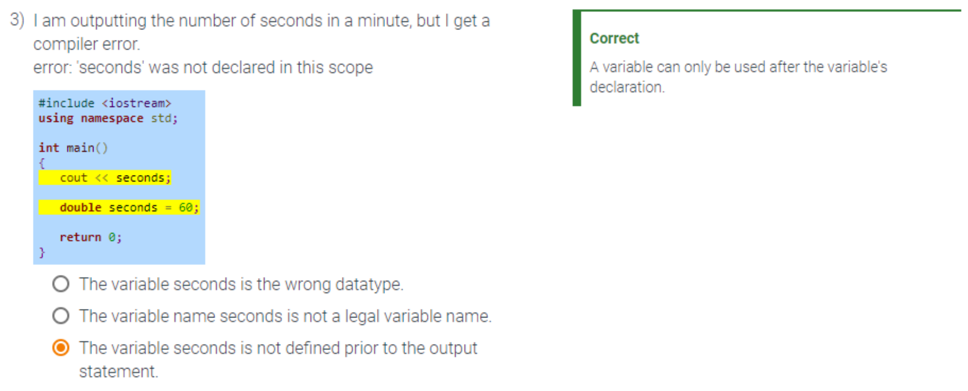
```
#include <iostream>
using namespace std;

int main()
{
   cout << seconds;

   double seconds = 60;

   return 0;
}
```

- The variable seconds is the wrong datatype.
- [✓ The variable name seconds is not a legal variable name.]
- The variable seconds is not defined prior to the output statement.

Incorrect

The name seconds is valid and well chosen.

Figure 85: Basic Variable, Question #3, Answer #2



3) I am outputting the number of seconds in a minute, but I get a compiler error.
error: 'seconds' was not declared in this scope

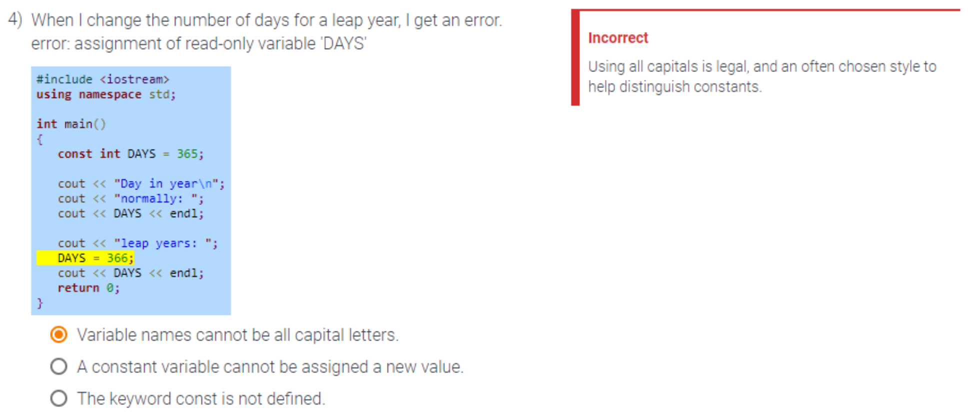
```
#include <iostream>
using namespace std;

int main()
{
   cout << seconds;

   double seconds = 60;

   return 0;
}
```

- The variable seconds is the wrong datatype.
- The variable name seconds is not a legal variable name.
- [✓ The variable seconds is not defined prior to the output statement.]

Correct

A variable can only be used after the variable's declaration.

Figure 86: Basic Variable, Question #3, Answer #3



4) When I change the number of days for a leap year, I get an error.
error: assignment of read-only variable 'DAYS'

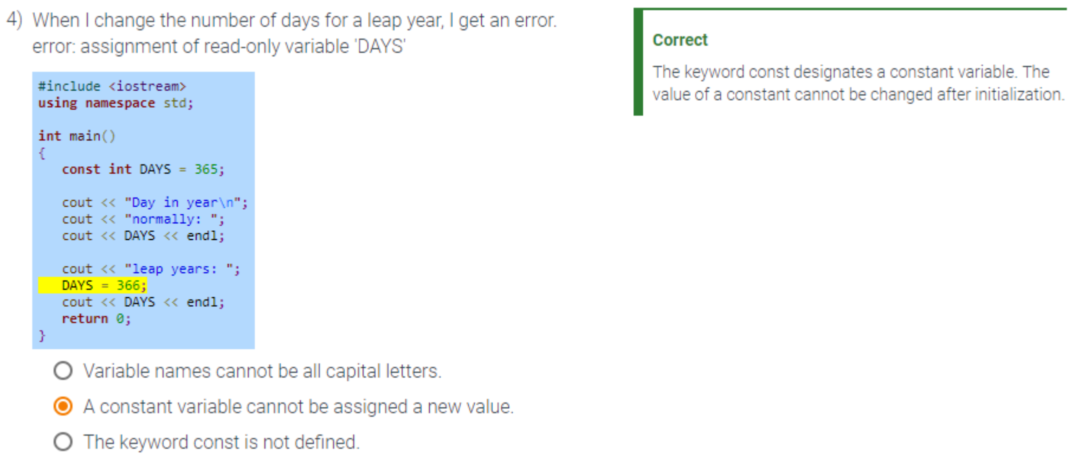
```
#include <iostream>
using namespace std;

int main()
{
   const int DAYS = 365;

   cout << "Day in year\n";
   cout << "normally: ";
   cout << DAYS << endl;

   cout << "leap years: ";
   DAYS = 366;
   cout << DAYS << endl;
   return 0;
}
```

- [✓ Variable names cannot be all capital letters.]
- A constant variable cannot be assigned a new value.
- The keyword const is not defined.

Incorrect

Using all capitals is legal, and an often chosen style to help distinguish constants.

Figure 87: Basic Variable, Question #4, Answer #1



4) When I change the number of days for a leap year, I get an error.
error: assignment of read-only variable 'DAYS'

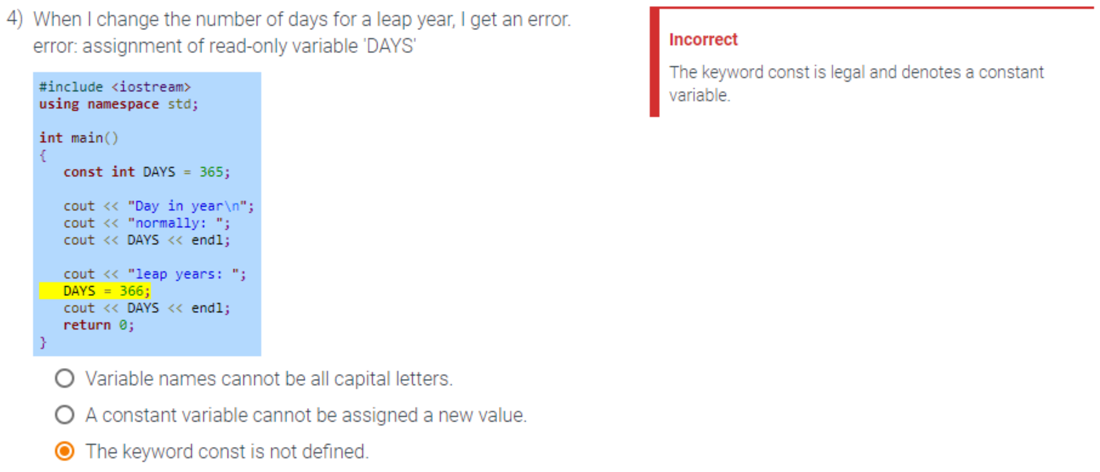
```
#include <iostream>
using namespace std;

int main()
{
   const int DAYS = 365;

   cout << "Day in year\n";
   cout << "normally: ";
   cout << DAYS << endl;

   cout << "leap years: ";
   DAYS = 366;
   cout << DAYS << endl;
   return 0;
}
```

○ Variable names cannot be all capital letters.
● A constant variable cannot be assigned a new value.
○ The keyword const is not defined.

Correct

The keyword const designates a constant variable. The value of a constant cannot be changed after initialization.

Figure 88: Basic Variable, Question #4, Answer #2



4) When I change the number of days for a leap year, I get an error.
error: assignment of read-only variable 'DAYS'

```
#include <iostream>
using namespace std;

int main()
{
   const int DAYS = 365;

   cout << "Day in year\n";
   cout << "normally: ";
   cout << DAYS << endl;

   cout << "leap years: ";
   DAYS = 366;
   cout << DAYS << endl;
   return 0;
}
```

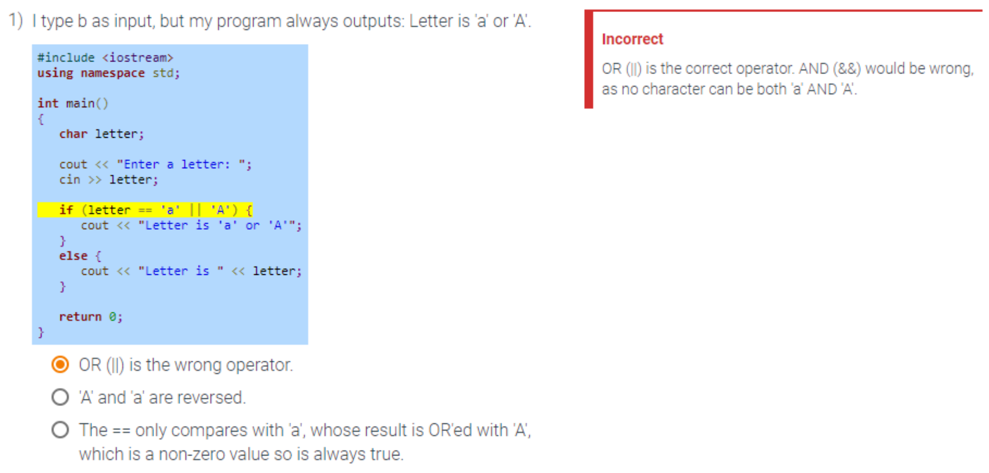
○ Variable names cannot be all capital letters.
○ A constant variable cannot be assigned a new value.
● The keyword const is not defined.

Incorrect

The keyword const is legal and denotes a constant variable.

Figure 89: Basic Variable, Question #4, Answer #3

Branching Exercises



1) I type b as input, but my program always outputs: Letter is 'a' or 'A'.

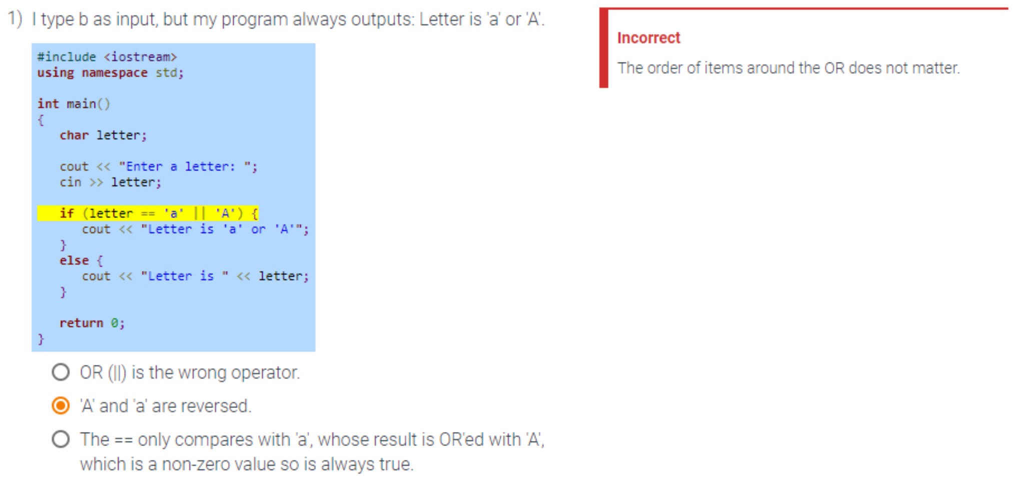
```
#include <iostream>
using namespace std;

int main()
{
   char letter;

   cout << "Enter a letter: ";
   cin >> letter;

   if (letter == 'a' || 'A') {
      cout << "Letter is 'a' or 'A'";
   }
   else {
      cout << "Letter is " << letter;
   }

   return 0;
}
```

- [✓ OR (||) is the wrong operator.]
- 'A' and 'a' are reversed.
- The == only compares with 'a', whose result is OR'ed with 'A', which is a non-zero value so is always true.

Incorrect

OR (||) is the correct operator. AND (&&) would be wrong, as no character can be both 'a' AND 'A'.

Figure 90: Branching, Question #1, Answer #1



1) I type b as input, but my program always outputs: Letter is 'a' or 'A'.

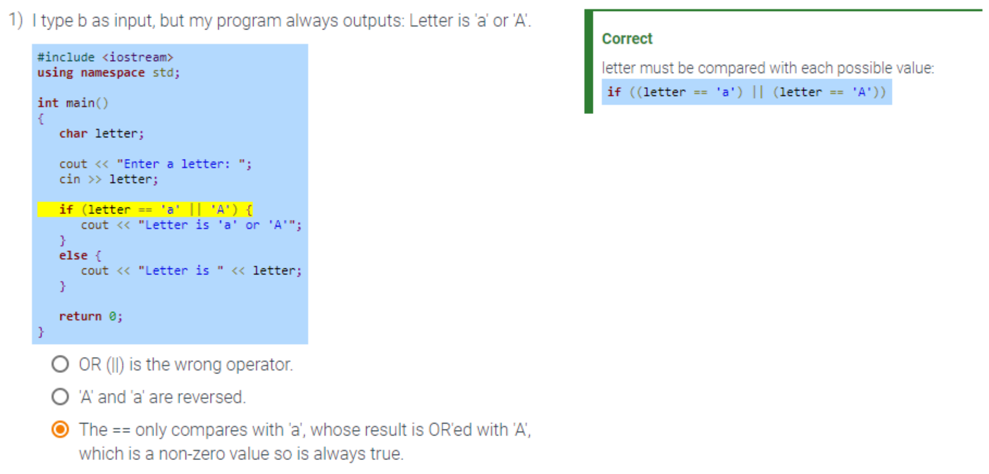
```
#include <iostream>
using namespace std;

int main()
{
   char letter;

   cout << "Enter a letter: ";
   cin >> letter;

   if (letter == 'a' || 'A') {
      cout << "Letter is 'a' or 'A'";
   }
   else {
      cout << "Letter is " << letter;
   }

   return 0;
}
```

- OR (||) is the wrong operator.
- [✓ 'A' and 'a' are reversed.]
- The == only compares with 'a', whose result is OR'ed with 'A', which is a non-zero value so is always true.

Incorrect

The order of items around the OR does not matter.

Figure 91: Branching, Question #1, Answer #2



1) I type b as input, but my program always outputs: Letter is 'a' or 'A'.

```
#include <iostream>
using namespace std;

int main()
{
   char letter;

   cout << "Enter a letter: ";
   cin >> letter;

   if (letter == 'a' || 'A') {
      cout << "Letter is 'a' or 'A'";
   }
   else {
      cout << "Letter is " << letter;
   }

   return 0;
}
```

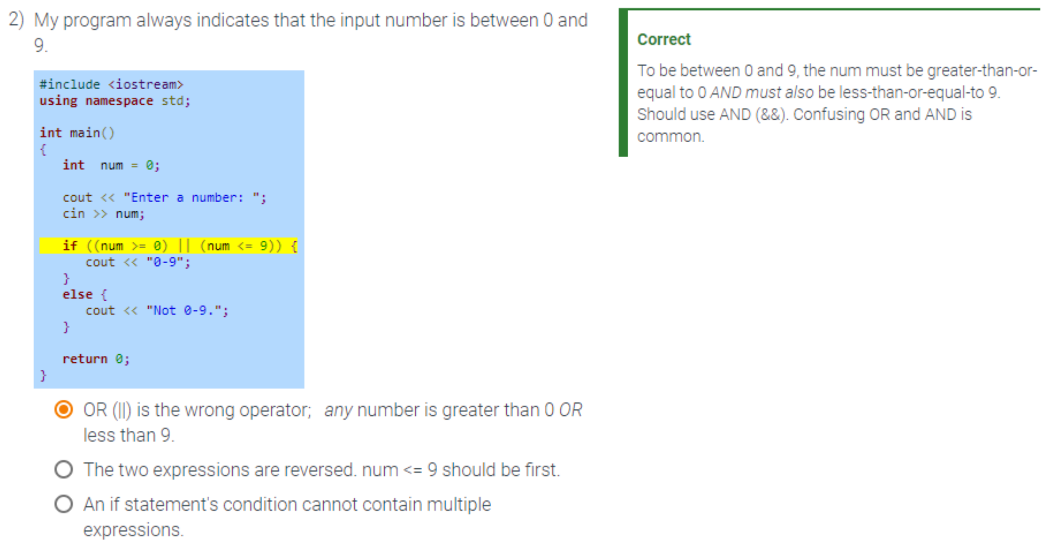
- OR (||) is the wrong operator.
- 'A' and 'a' are reversed.
- [✓ The == only compares with 'a', whose result is OR'ed with 'A', which is a non-zero value so is always true.]

Correct

letter must be compared with each possible value:

`if ((letter == 'a') || (letter == 'A'))`

Figure 92: Branching, Question #1, Answer #3



2) My program always indicates that the input number is between 0 and 9.

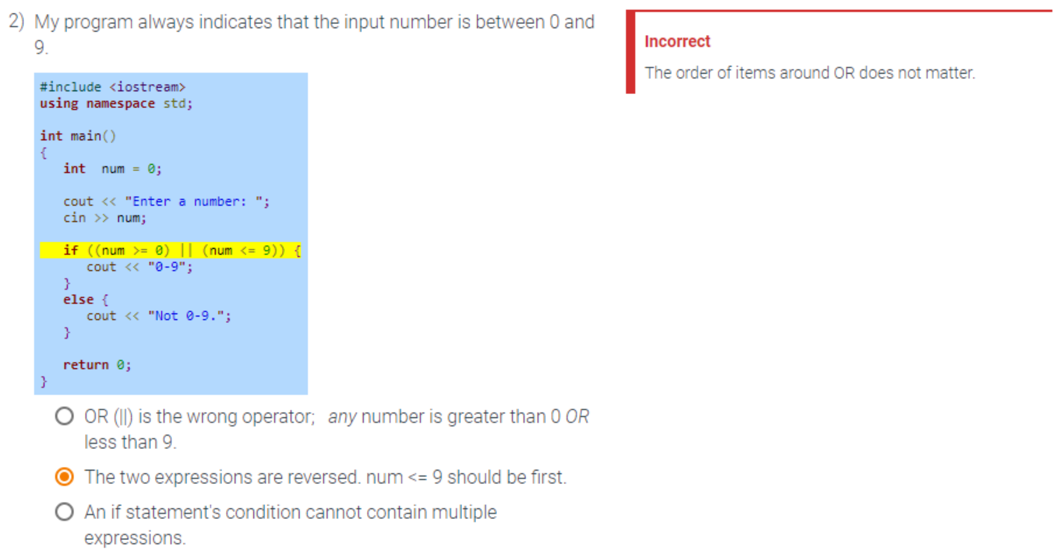
```
#include <iostream>
using namespace std;

int main()
{
   int  num = 0;

   cout << "Enter a number: ";
   cin >> num;

   if ((num >= 0) || (num <= 9)) {
      cout << "0-9";
   }
   else {
      cout << "Not 0-9.";
   }

   return 0;
}
```

- [✓ OR (||) is the wrong operator; *any* number is greater than 0 *OR* less than 9.]
- The two expressions are reversed. num <= 9 should be first.
- An if statement's condition cannot contain multiple expressions.

Correct

To be between 0 and 9, the num must be greater-than-or-equal to 0 *AND must also* be less-than-or-equal-to 9. Should use AND (&&). Confusing OR and AND is common.

Figure 94: Branching, Question #2, Answer #1



2) My program always indicates that the input number is between 0 and 9.

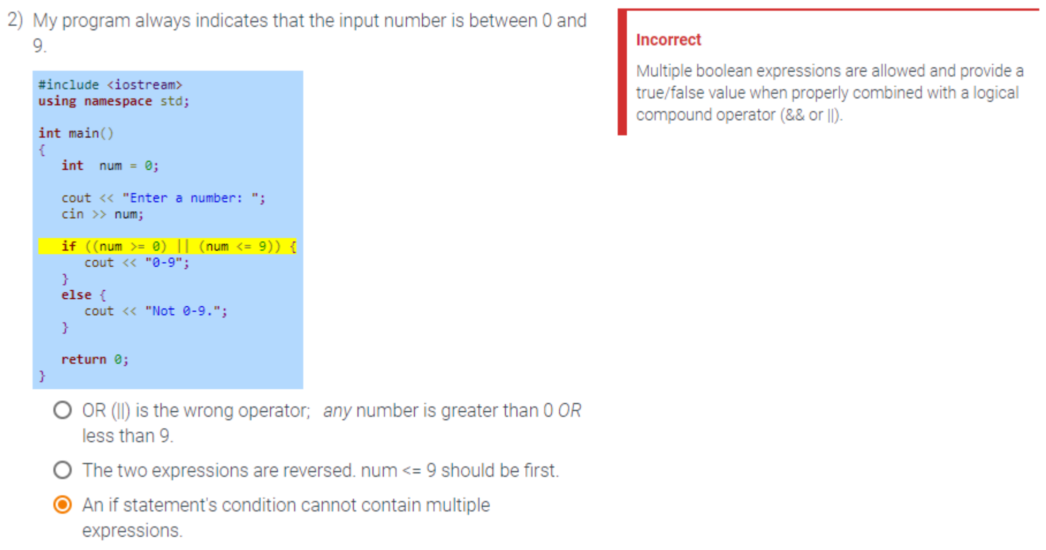
```
#include <iostream>
using namespace std;

int main()
{
   int  num = 0;

   cout << "Enter a number: ";
   cin >> num;

   if ((num >= 0) || (num <= 9)) {
      cout << "0-9";
   }
   else {
      cout << "Not 0-9.";
   }

   return 0;
}
```

- ○ OR (||) is the wrong operator; *any* number is greater than 0 *OR* less than 9.
- [✓ The two expressions are reversed. num <= 9 should be first.]
- ○ An if statement's condition cannot contain multiple expressions.

Incorrect

The order of items around OR does not matter.

Figure 93: Branching, Question #2, Answer #3



2) My program always indicates that the input number is between 0 and 9.

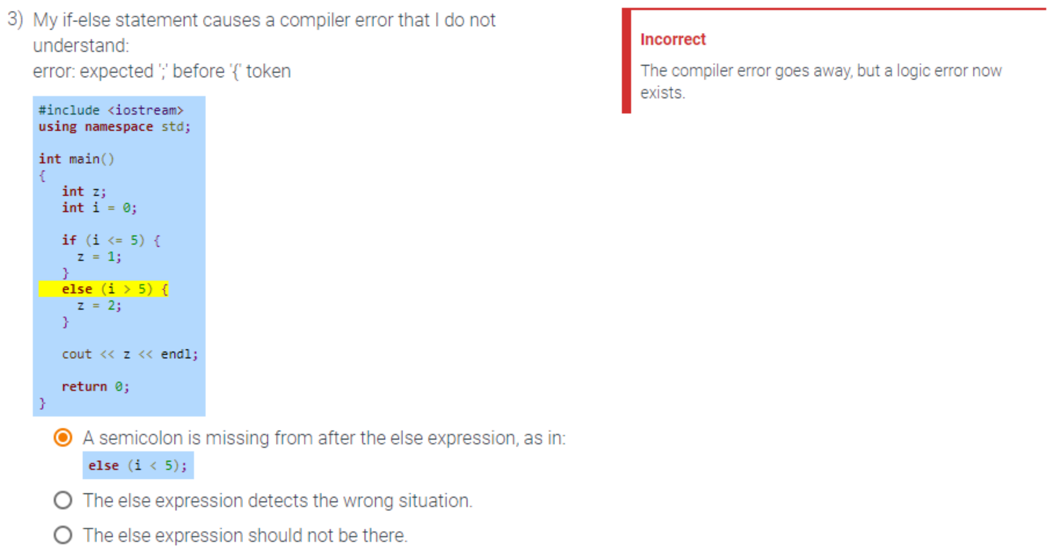
```
#include <iostream>
using namespace std;

int main()
{
   int  num = 0;

   cout << "Enter a number: ";
   cin >> num;

   if ((num >= 0) || (num <= 9)) {
      cout << "0-9";
   }
   else {
      cout << "Not 0-9.";
   }

   return 0;
}
```

- ○ OR (||) is the wrong operator; *any* number is greater than 0 *OR* less than 9.
- ○ The two expressions are reversed. num <= 9 should be first.
- [✓ An if statement's condition cannot contain multiple expressions.]

Incorrect

Multiple boolean expressions are allowed and provide a true/false value when properly combined with a logical compound operator (&& or ||).

Figure 95: Branching, Question #2, Answer #3



3) My if-else statement causes a compiler error that I do not understand:
error: expected ';' before '{' token

```
#include <iostream>
using namespace std;

int main()
{
   int z;
   int i = 0;

   if (i <= 5) {
     z = 1;
   }
   else (i > 5) {
     z = 2;
   }

   cout << z << endl;

   return 0;
}
```

[✓ A semicolon is missing from after the else expression, as in:]

```
else (i < 5);
```

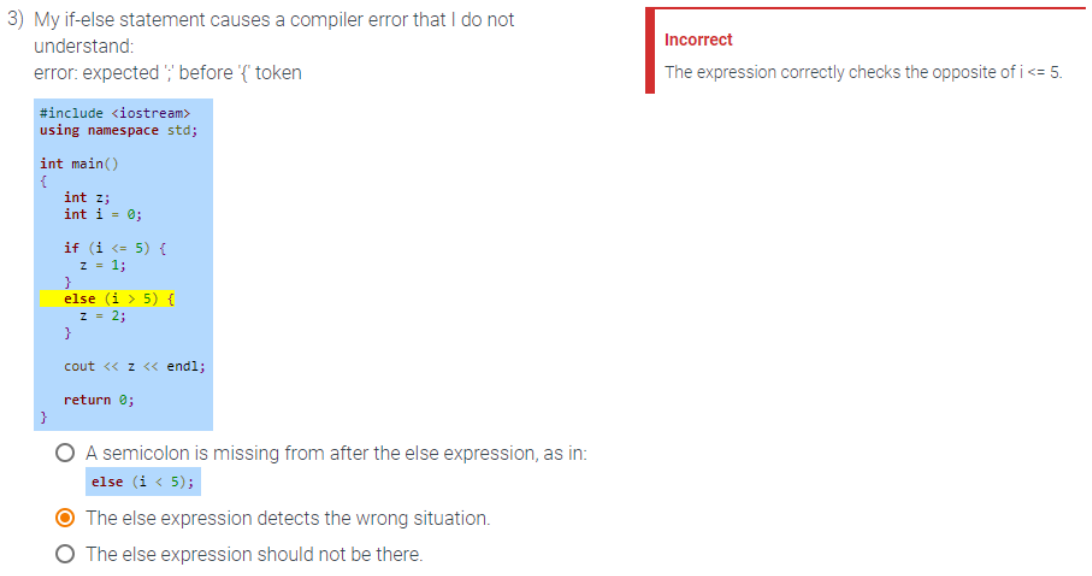
The else expression detects the wrong situation.

The else expression should not be there.

Incorrect

The compiler error goes away, but a logic error now exists.

Figure 96: Branching, Question #3, Answer #1



3) My if-else statement causes a compiler error that I do not understand:
error: expected ';' before '{' token

```
#include <iostream>
using namespace std;

int main()
{
   int z;
   int i = 0;

   if (i <= 5) {
     z = 1;
   }
   else (i > 5) {
     z = 2;
   }

   cout << z << endl;

   return 0;
}
```

A semicolon is missing from after the else expression, as in:

```
else (i < 5);
```

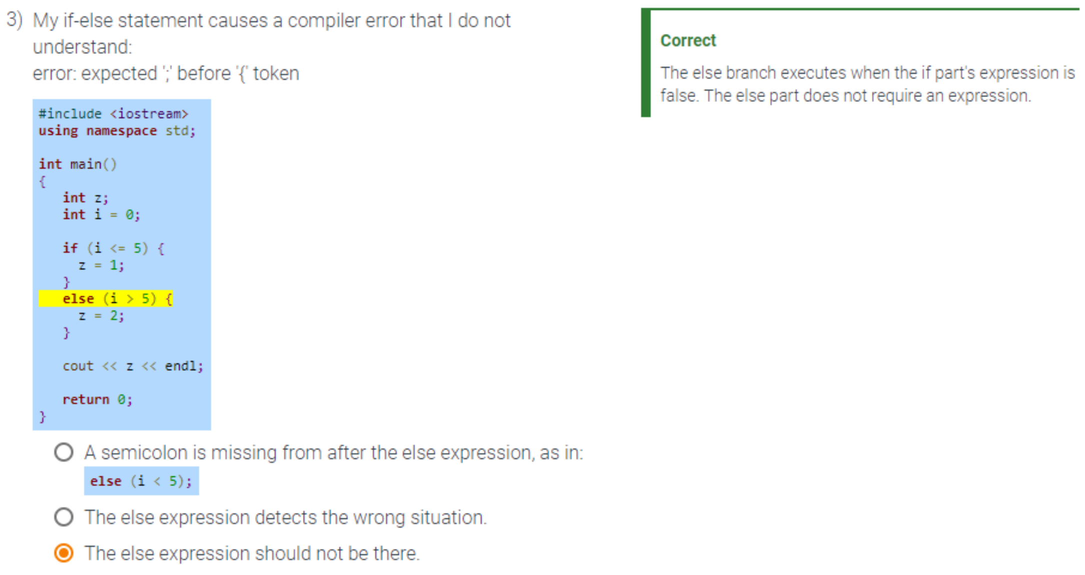
[✓ The else expression detects the wrong situation.]

The else expression should not be there.

Incorrect

The expression correctly checks the opposite of i <= 5.

Figure 97: Branching, Question #3, Answer #2



3) My if-else statement causes a compiler error that I do not understand:
error: expected ';' before '{' token

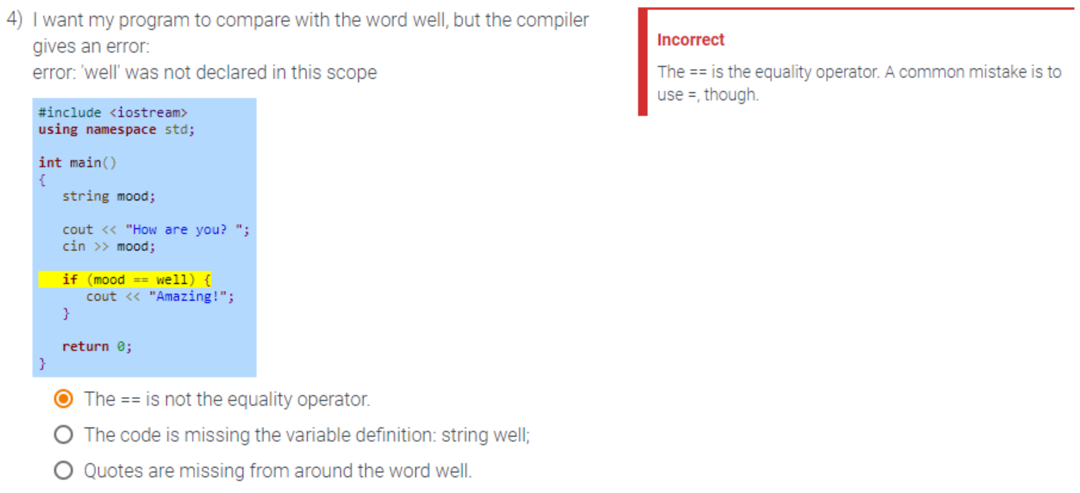
```
#include <iostream>
using namespace std;

int main()
{
   int z;
   int i = 0;

   if (i <= 5) {
     z = 1;
   }
   else (i > 5) {
     z = 2;
   }

   cout << z << endl;

   return 0;
}
```

- A semicolon is missing from after the else expression, as in:
  ```
  else (i < 5);
  ```
- The else expression detects the wrong situation.
- [✓ The else expression should not be there.]

Correct

The else branch executes when the if part's expression is false. The else part does not require an expression.

Figure 98: Branching, Question #3, Answer #3



4) I want my program to compare with the word well, but the compiler gives an error:
error: 'well' was not declared in this scope

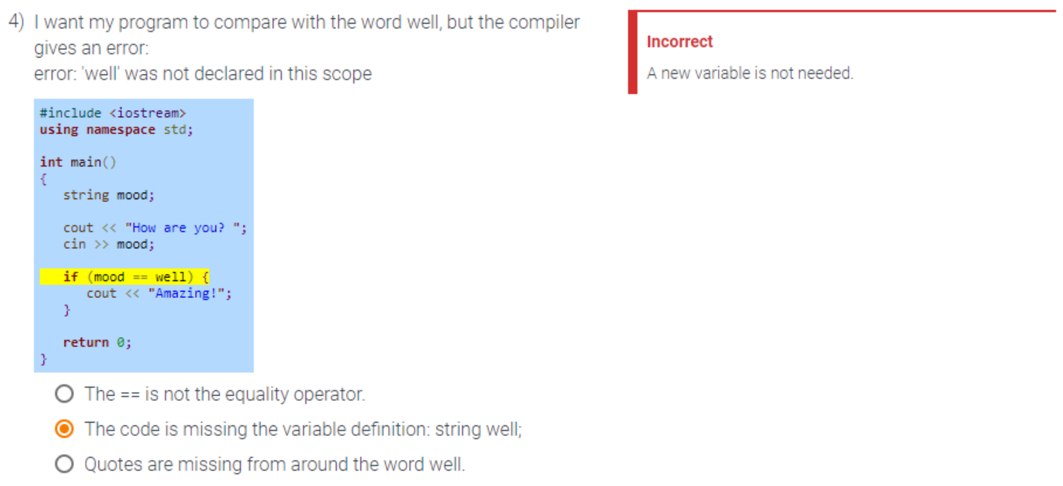
```
#include <iostream>
using namespace std;

int main()
{
   string mood;

   cout << "How are you? ";
   cin >> mood;

   if (mood == well) {
      cout << "Amazing!";
   }

   return 0;
}
```

- [✓ The == is not the equality operator.]
- The code is missing the variable definition: string well;
- Quotes are missing from around the word well.

Incorrect

The == is the equality operator. A common mistake is to use =, though.

Figure 99: Branching, Question #4, Answer #1


4) I want my program to compare with the word well, but the compiler gives an error:
error: 'well' was not declared in this scope

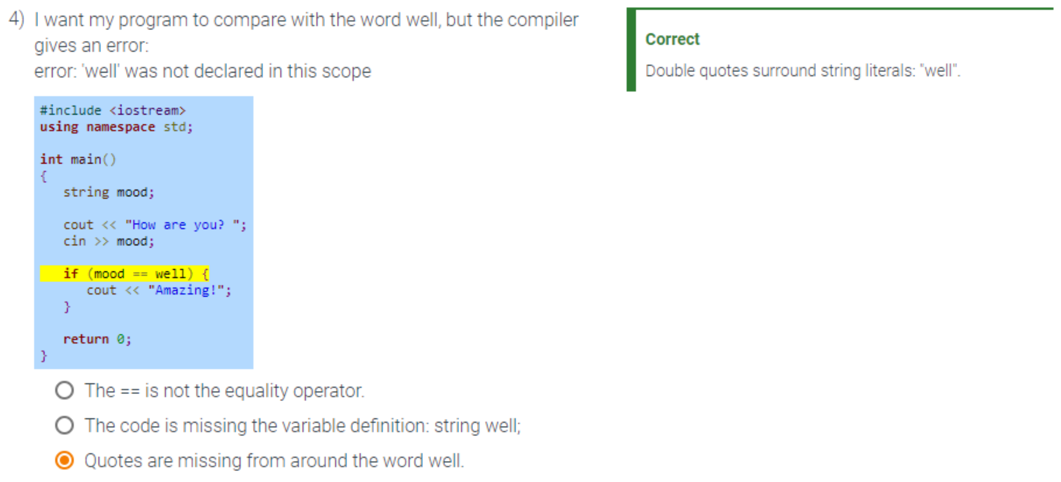
```
#include <iostream>
using namespace std;

int main()
{
   string mood;

   cout << "How are you? ";
   cin >> mood;

   if (mood == well) {
      cout << "Amazing!";
   }

   return 0;
}
```

- ○ The == is not the equality operator.
- ◉ The code is missing the variable definition: string well;
- ○ Quotes are missing from around the word well.

Incorrect

A new variable is not needed.

Figure 100: Branching, Question #4, Answer #2


4) I want my program to compare with the word well, but the compiler gives an error:
error: 'well' was not declared in this scope

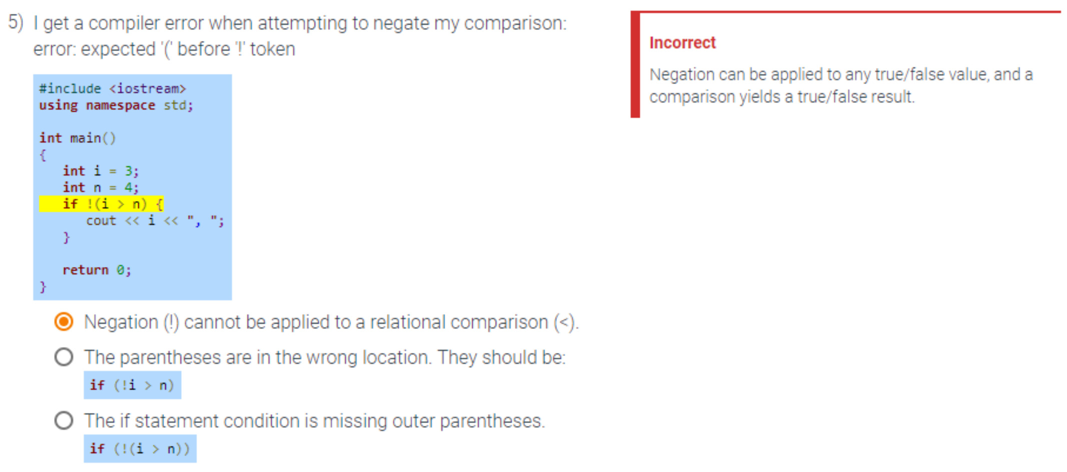
```
#include <iostream>
using namespace std;

int main()
{
   string mood;

   cout << "How are you? ";
   cin >> mood;

   if (mood == well) {
      cout << "Amazing!";
   }

   return 0;
}
```

- ○ The == is not the equality operator.
- ○ The code is missing the variable definition: string well;
- ◉ Quotes are missing from around the word well.

Correct

Double quotes surround string literals: "well".

Figure 101: Branching, Question #4, Answer #3


5) I get a compiler error when attempting to negate my comparison:
error: expected '(' before '!' token

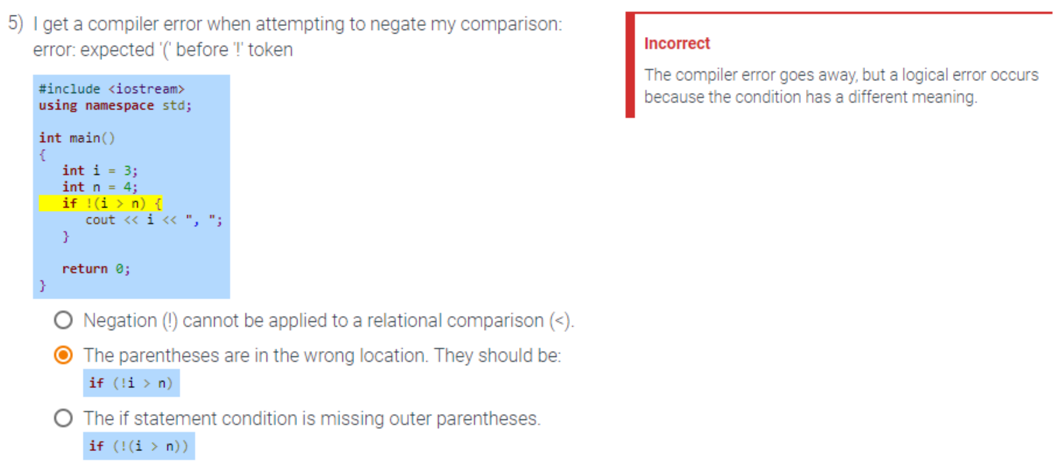
```
#include <iostream>
using namespace std;

int main()
{
   int i = 3;
   int n = 4;
   if !(i > n) {
      cout << i << ", ";
   }

   return 0;
}
```

- [✓ Negation (!) cannot be applied to a relational comparison (<).]
- The parentheses are in the wrong location. They should be:
  `if (!i > n)`
- The if statement condition is missing outer parentheses.
  `if (!(i > n))`

Incorrect

Negation can be applied to any true/false value, and a comparison yields a true/false result.

Figure 102: Branching, Question #5, Answer #1


5) I get a compiler error when attempting to negate my comparison:
error: expected '(' before '!' token

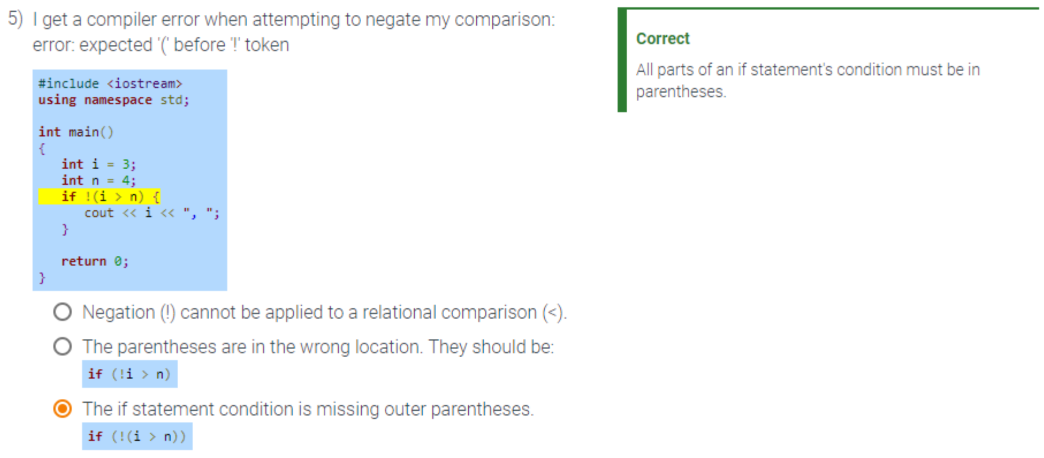
```
#include <iostream>
using namespace std;

int main()
{
   int i = 3;
   int n = 4;
   if !(i > n) {
      cout << i << ", ";
   }

   return 0;
}
```

- Negation (!) cannot be applied to a relational comparison (<).
- [✓ The parentheses are in the wrong location. They should be:]
  `if (!i > n)`
- The if statement condition is missing outer parentheses.
  `if (!(i > n))`

Incorrect

The compiler error goes away, but a logical error occurs because the condition has a different meaning.

Figure 103: Branching, Question #5, Answer #2



5) I get a compiler error when attempting to negate my comparison: error: expected '(' before '!' token

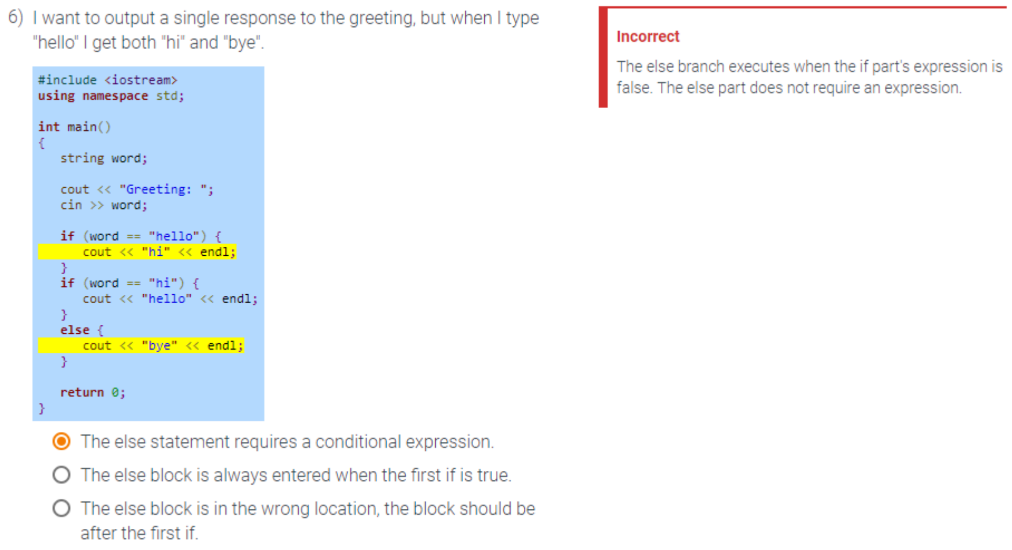
```
#include <iostream>
using namespace std;

int main()
{
   int i = 3;
   int n = 4;
   if !(i > n) {
      cout << i << ", ";
   }

   return 0;
}
```

- Negation (!) cannot be applied to a relational comparison (<).
- The parentheses are in the wrong location. They should be: `if (!i > n)`
- [✓ The if statement condition is missing outer parentheses. `if (!(i > n))`]

Correct

All parts of an if statement's condition must be in parentheses.

Figure 104: Branching, Question #5, Answer #3



6) I want to output a single response to the greeting, but when I type "hello" I get both "hi" and "bye".

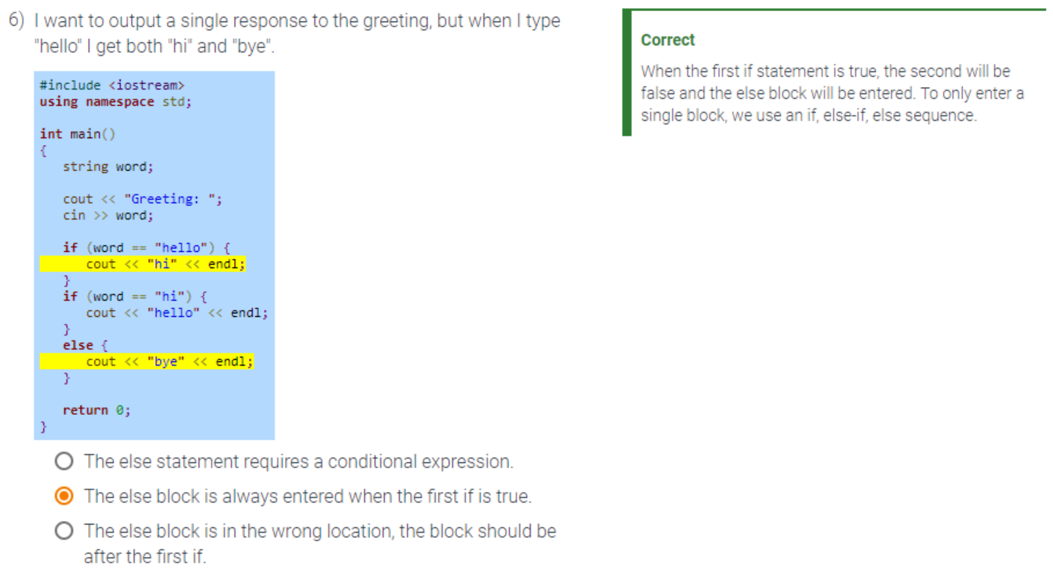
```
#include <iostream>
using namespace std;

int main()
{
   string word;

   cout << "Greeting: ";
   cin >> word;

   if (word == "hello") {
      cout << "hi" << endl;
   }
   if (word == "hi") {
      cout << "hello" << endl;
   }
   else {
      cout << "bye" << endl;
   }

   return 0;
}
```

- [✓ The else statement requires a conditional expression.]
- The else block is always entered when the first if is true.
- The else block is in the wrong location, the block should be after the first if.

Incorrect

The else branch executes when the if part's expression is false. The else part does not require an expression.

Figure 105: Branching, Question #6, Answer #1


6) I want to output a single response to the greeting, but when I type "hello" I get both "hi" and "bye".

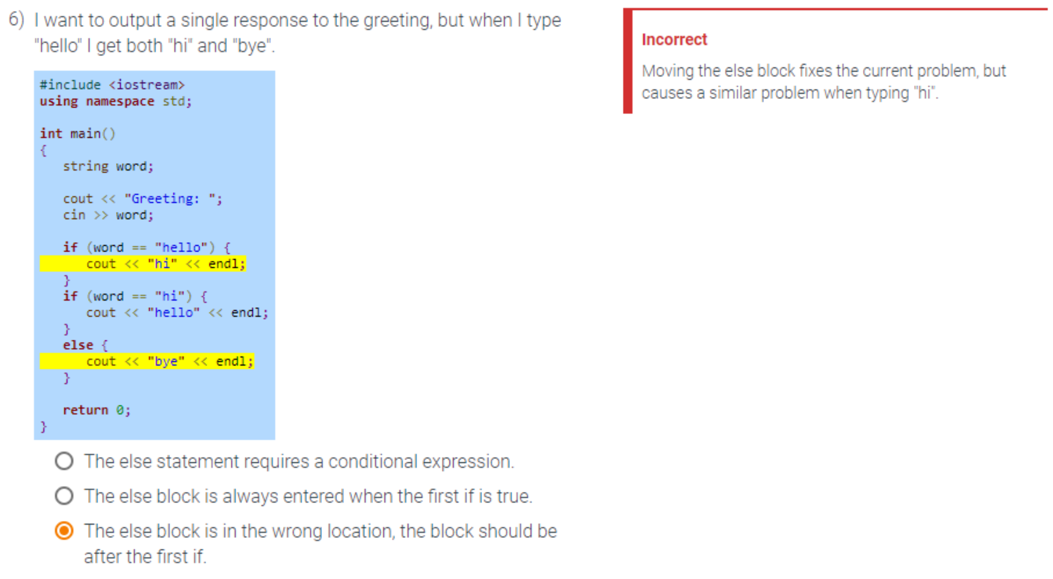
```
#include <iostream>
using namespace std;

int main()
{
   string word;

   cout << "Greeting: ";
   cin >> word;

   if (word == "hello") {
      cout << "hi" << endl;
   }
   if (word == "hi") {
      cout << "hello" << endl;
   }
   else {
      cout << "bye" << endl;
   }

   return 0;
}
```

- ○ The else statement requires a conditional expression.
- ● The else block is always entered when the first if is true.
- ○ The else block is in the wrong location, the block should be after the first if.

Correct

When the first if statement is true, the second will be false and the else block will be entered. To only enter a single block, we use an if, else-if, else sequence.

Figure 106: Branching, Question #6, Answer #2


6) I want to output a single response to the greeting, but when I type "hello" I get both "hi" and "bye".

```
#include <iostream>
using namespace std;

int main()
{
   string word;

   cout << "Greeting: ";
   cin >> word;

   if (word == "hello") {
      cout << "hi" << endl;
   }
   if (word == "hi") {
      cout << "hello" << endl;
   }
   else {
      cout << "bye" << endl;
   }

   return 0;
}
```

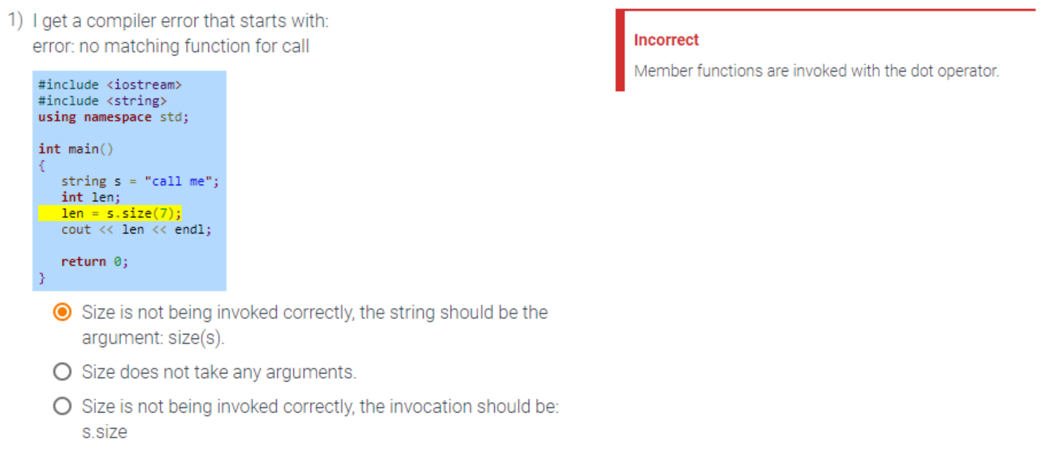
- ○ The else statement requires a conditional expression.
- ○ The else block is always entered when the first if is true.
- ● The else block is in the wrong location, the block should be after the first if.

Incorrect

Moving the else block fixes the current problem, but causes a similar problem when typing "hi".

Figure 107: Branching, Question #6, Answer #3

String Function Exercises



1) I get a compiler error that starts with:
error: no matching function for call

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string s = "call me";
   int len;
   len = s.size(7);
   cout << len << endl;

   return 0;
}
```

● Size is not being invoked correctly, the string should be the argument: size(s).

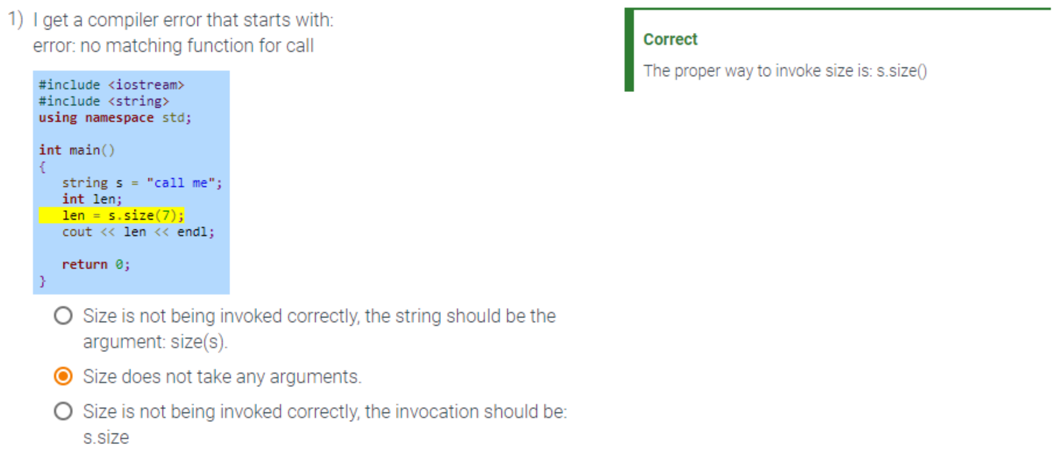
○ Size does not take any arguments.

○ Size is not being invoked correctly, the invocation should be: s.size

Incorrect

Member functions are invoked with the dot operator.

Figure 108: String Functions, Question #1, Answer #1



1) I get a compiler error that starts with:
error: no matching function for call

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string s = "call me";
   int len;
   len = s.size(7);
   cout << len << endl;

   return 0;
}
```

○ Size is not being invoked correctly, the string should be the argument: size(s).

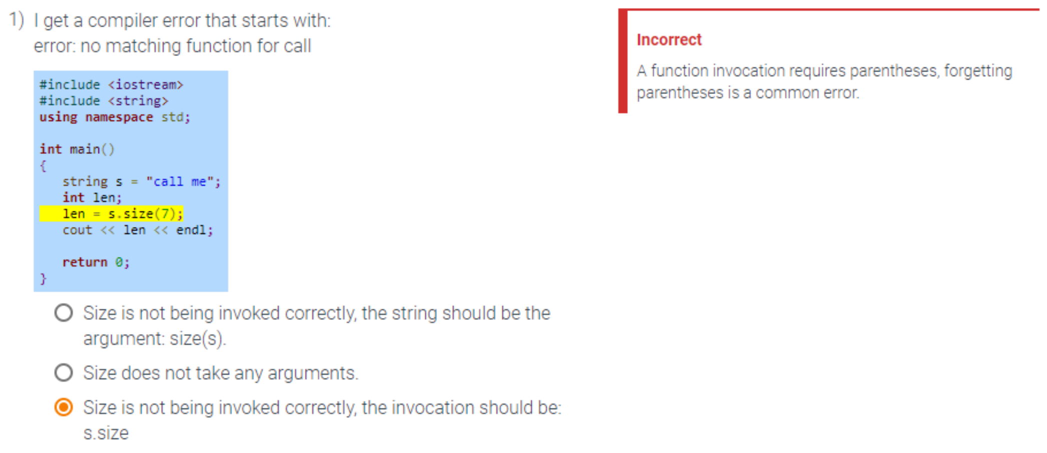
● Size does not take any arguments.

○ Size is not being invoked correctly, the invocation should be: s.size

Correct

The proper way to invoke size is: s.size()

Figure 109: String Functions, Question #1, Answer #2


1) I get a compiler error that starts with:
error: no matching function for call

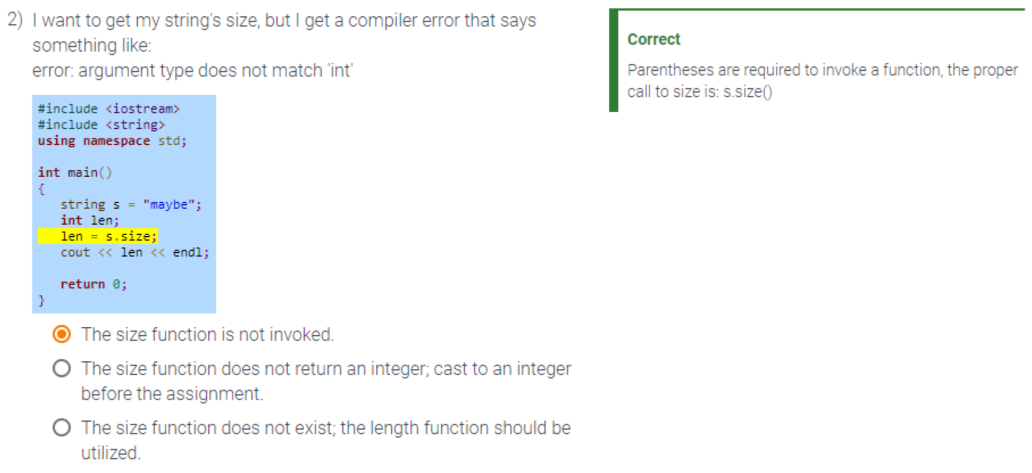
```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string s = "call me";
    int len;
    len = s.size(7);
    cout << len << endl;

    return 0;
}
```

- ○ Size is not being invoked correctly, the string should be the argument: size(s).
- ○ Size does not take any arguments.
- ● Size is not being invoked correctly, the invocation should be: s.size

Incorrect

A function invocation requires parentheses, forgetting parentheses is a common error.

Figure 110: String Functions, Question #1, Answer #3


2) I want to get my string's size, but I get a compiler error that says something like:
error: argument type does not match 'int'

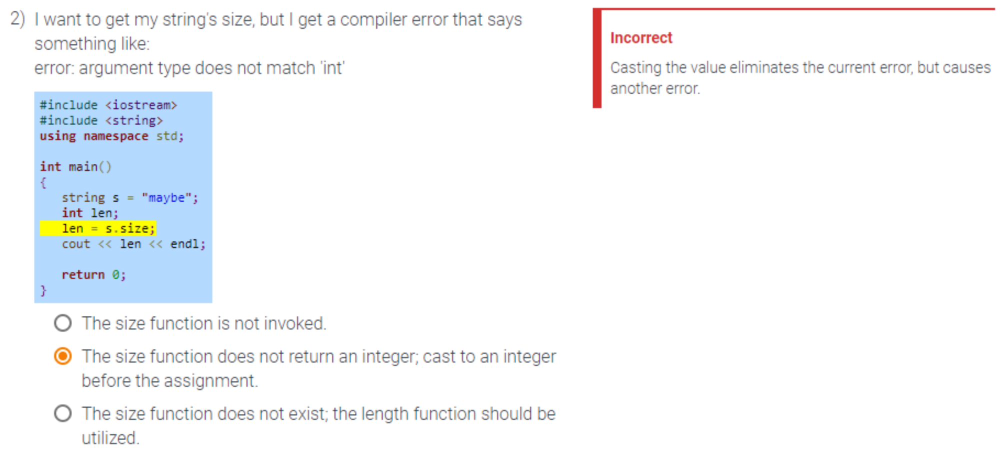
```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string s = "maybe";
    int len;
    len = s.size;
    cout << len << endl;

    return 0;
}
```

- ● The size function is not invoked.
- ○ The size function does not return an integer; cast to an integer before the assignment.
- ○ The size function does not exist; the length function should be utilized.

Correct

Parentheses are required to invoke a function, the proper call to size is: s.size()

Figure 111: String Functions, Question #2, Answer #1



2) I want to get my string's size, but I get a compiler error that says something like:
error: argument type does not match 'int'

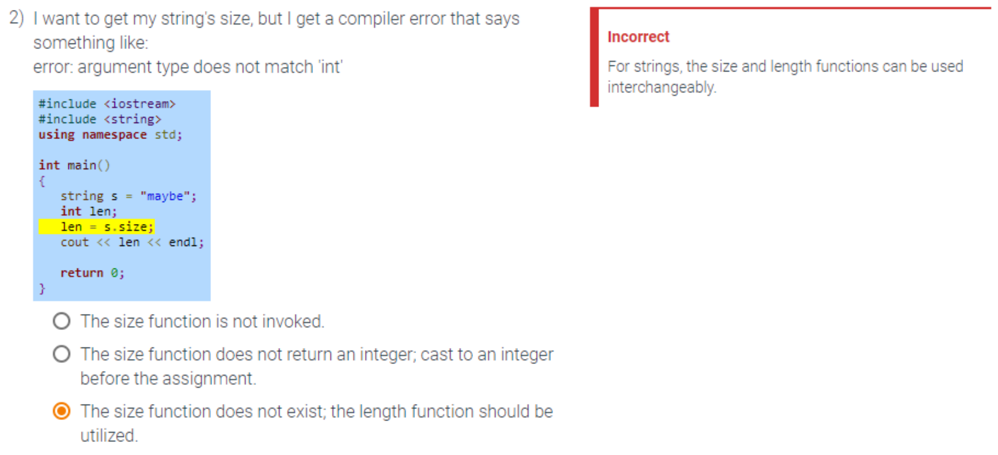
```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string s = "maybe";
   int len;
   len = s.size;
   cout << len << endl;

   return 0;
}
```

- ○ The size function is not invoked.
- [✓ The size function does not return an integer; cast to an integer before the assignment.]
- ○ The size function does not exist; the length function should be utilized.

Incorrect

Casting the value eliminates the current error, but causes another error.

Figure 112: String Functions, Question #2, Answer #2



2) I want to get my string's size, but I get a compiler error that says something like:
error: argument type does not match 'int'

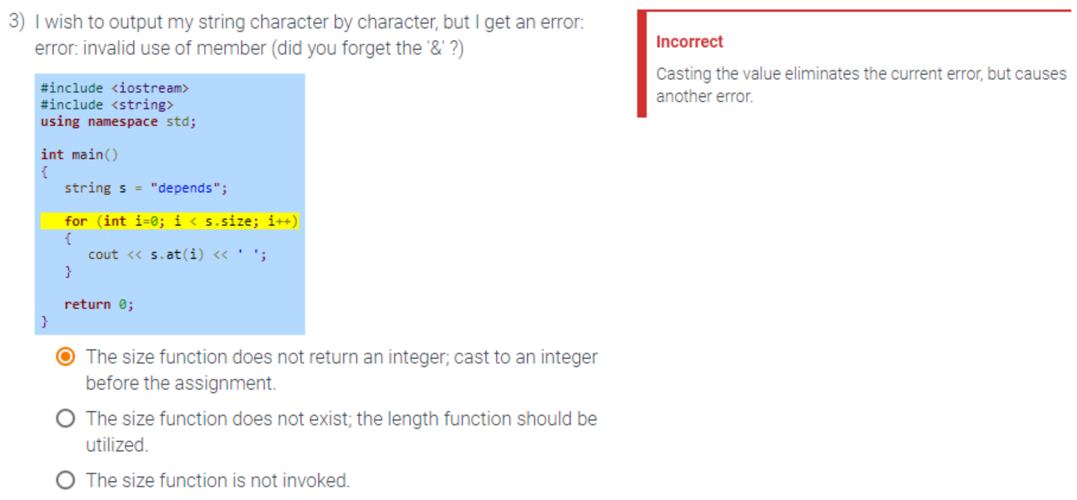
```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string s = "maybe";
   int len;
   len = s.size;
   cout << len << endl;

   return 0;
}
```

- ○ The size function is not invoked.
- ○ The size function does not return an integer; cast to an integer before the assignment.
- [✓ The size function does not exist; the length function should be utilized.]

Incorrect

For strings, the size and length functions can be used interchangeably.

Figure 113: String Functions, Question #2, Answer #3



3) I wish to output my string character by character, but I get an error:
error: invalid use of member (did you forget the '&' ?)

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string s = "depends";

   for (int i=0; i < s.size; i++)
   {
      cout << s.at(i) << ' ';
   }

   return 0;
}
```

[✓ The size function does not return an integer; cast to an integer before the assignment.]

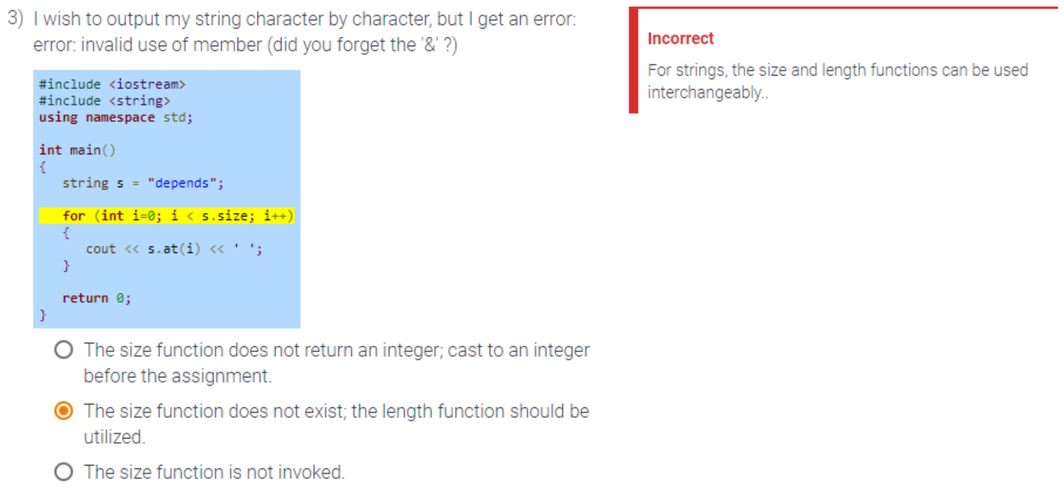
The size function does not exist; the length function should be utilized.

The size function is not invoked.

Incorrect

Casting the value eliminates the current error, but causes another error.

Figure 114: String Functions, Question #3, Answer #1



3) I wish to output my string character by character, but I get an error:
error: invalid use of member (did you forget the '&' ?)

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string s = "depends";

   for (int i=0; i < s.size; i++)
   {
      cout << s.at(i) << ' ';
   }

   return 0;
}
```

The size function does not return an integer; cast to an integer before the assignment.

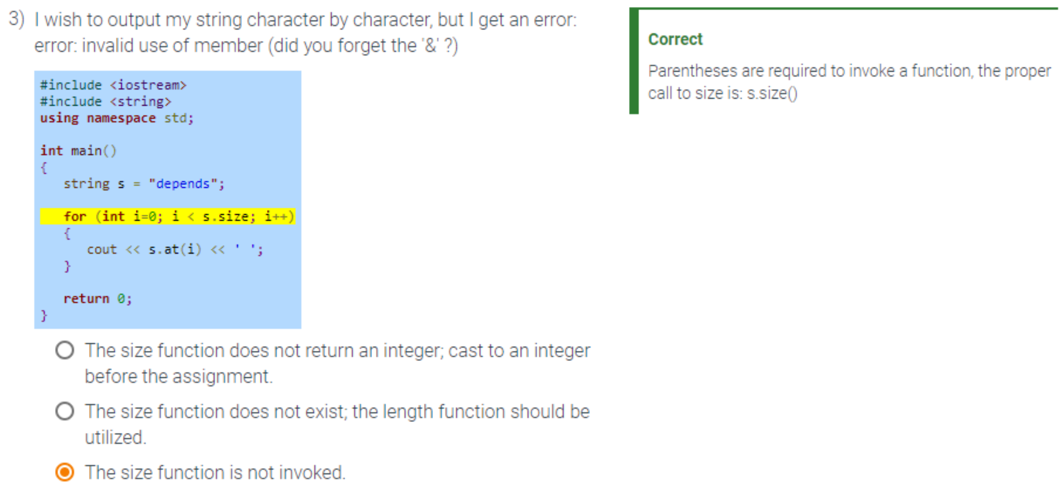
[✓ The size function does not exist; the length function should be utilized.]

The size function is not invoked.

Incorrect

For strings, the size and length functions can be used interchangeably..

Figure 115: String Functions, Question #3, Answer #2



3) I wish to output my string character by character, but I get an error:
error: invalid use of member (did you forget the '&' ?)

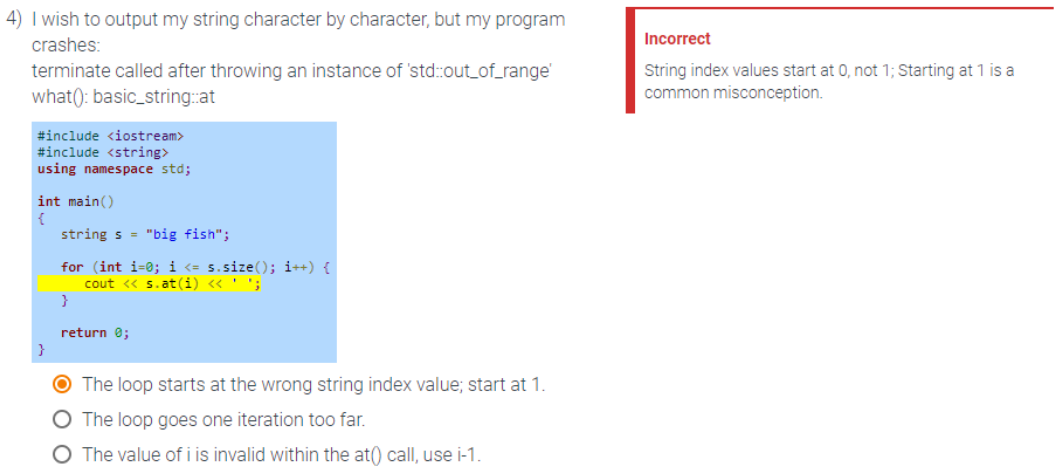
```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string s = "depends";

   for (int i=0; i < s.size; i++)
   {
      cout << s.at(i) << ' ';
   }

   return 0;
}
```

- The size function does not return an integer; cast to an integer before the assignment.
- The size function does not exist; the length function should be utilized.
- [✓ The size function is not invoked.]

Correct

Parentheses are required to invoke a function, the proper call to size is: s.size()

Figure 116: String Functions, Question #3, Answer #3



4) I wish to output my string character by character, but my program crashes:
terminate called after throwing an instance of 'std::out_of_range'
what(): basic_string::at

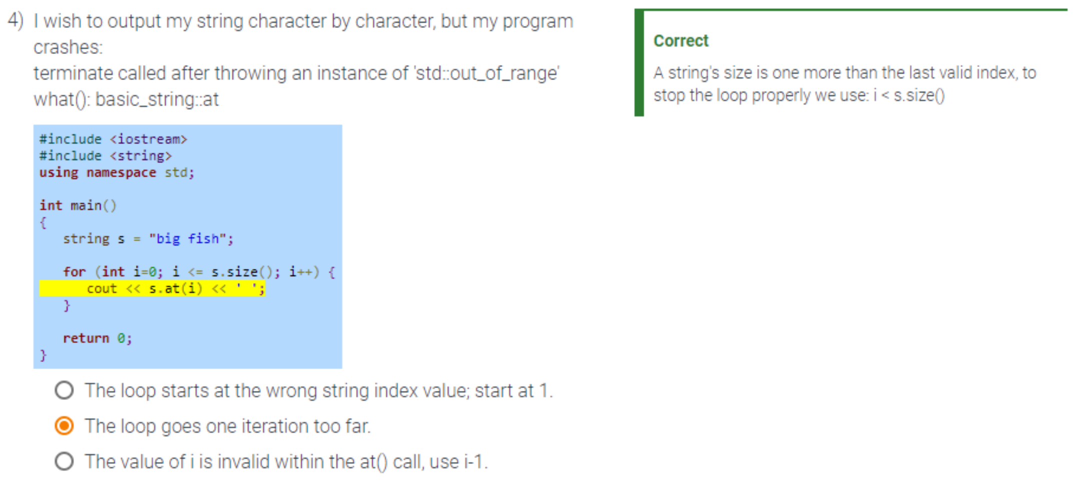
```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string s = "big fish";

   for (int i=0; i <= s.size(); i++) {
      cout << s.at(i) << ' ';
   }

   return 0;
}
```

- [✓ The loop starts at the wrong string index value; start at 1.]
- The loop goes one iteration too far.
- The value of i is invalid within the at() call, use i-1.

Incorrect

String index values start at 0, not 1; Starting at 1 is a common misconception.

Figure 117: String Functions, Question #4, Answer #1



4) I wish to output my string character by character, but my program crashes:
terminate called after throwing an instance of 'std::out_of_range'
what(): basic_string::at

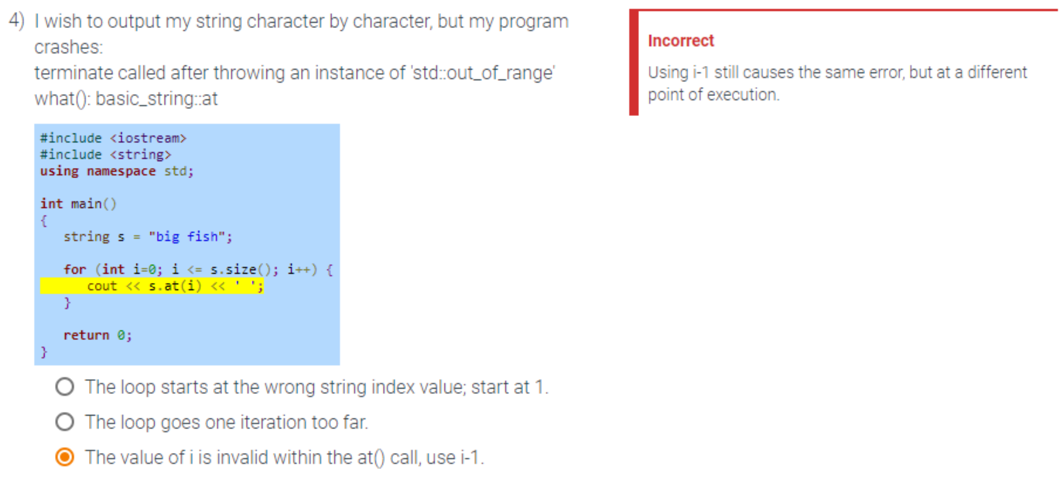
```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string s = "big fish";

   for (int i=0; i <= s.size(); i++) {
      cout << s.at(i) << ' ';
   }

   return 0;
}
```

- ○ The loop starts at the wrong string index value; start at 1.
- [✓ The loop goes one iteration too far.]
- ○ The value of i is invalid within the at() call, use i-1.

Correct

A string's size is one more than the last valid index, to stop the loop properly we use: i < s.size()

Figure 118: String Functions, Question #4, Answer #2



4) I wish to output my string character by character, but my program crashes:
terminate called after throwing an instance of 'std::out_of_range'
what(): basic_string::at

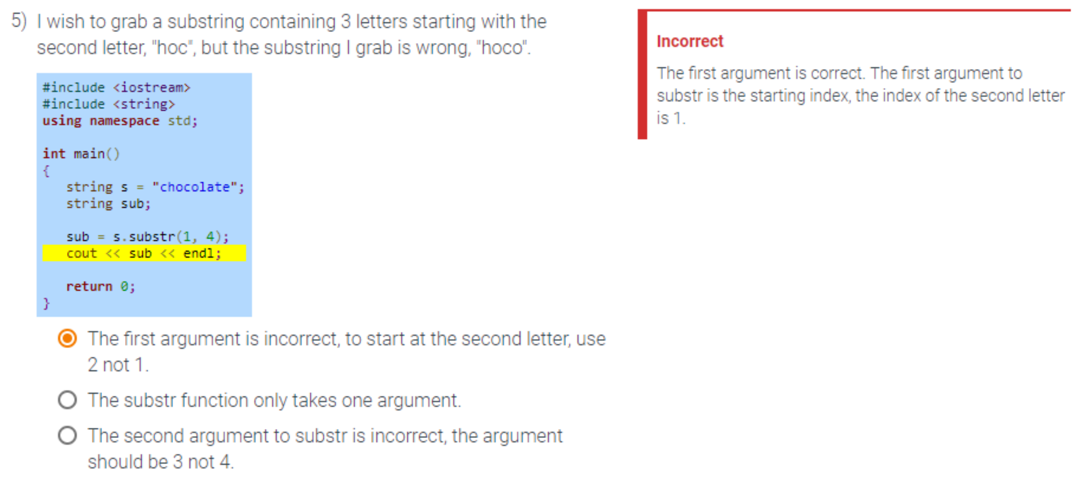
```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string s = "big fish";

   for (int i=0; i <= s.size(); i++) {
      cout << s.at(i) << ' ';
   }

   return 0;
}
```

- ○ The loop starts at the wrong string index value; start at 1.
- ○ The loop goes one iteration too far.
- [✓ The value of i is invalid within the at() call, use i-1.]

Incorrect

Using i-1 still causes the same error, but at a different point of execution.

Figure 119: String Functions, Question #4, Answer #3



5) I wish to grab a substring containing 3 letters starting with the second letter, "hoc", but the substring I grab is wrong, "hoco".

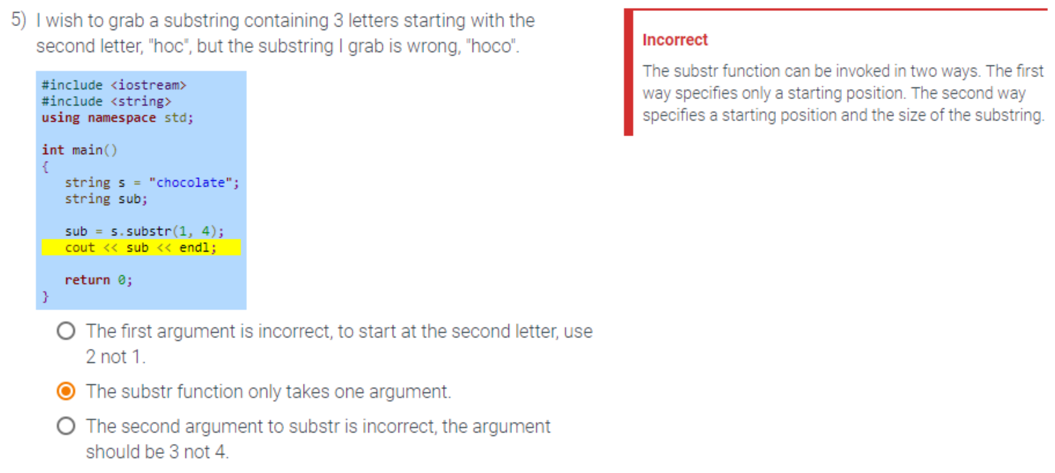
```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string s = "chocolate";
   string sub;

   sub = s.substr(1, 4);
   cout << sub << endl;

   return 0;
}
```

- [✓ The first argument is incorrect, to start at the second letter, use 2 not 1.]
- The substr function only takes one argument.
- The second argument to substr is incorrect, the argument should be 3 not 4.

Incorrect

The first argument is correct. The first argument to substr is the starting index, the index of the second letter is 1.

Figure 120: String Functions, Question #5, Answer #1



5) I wish to grab a substring containing 3 letters starting with the second letter, "hoc", but the substring I grab is wrong, "hoco".

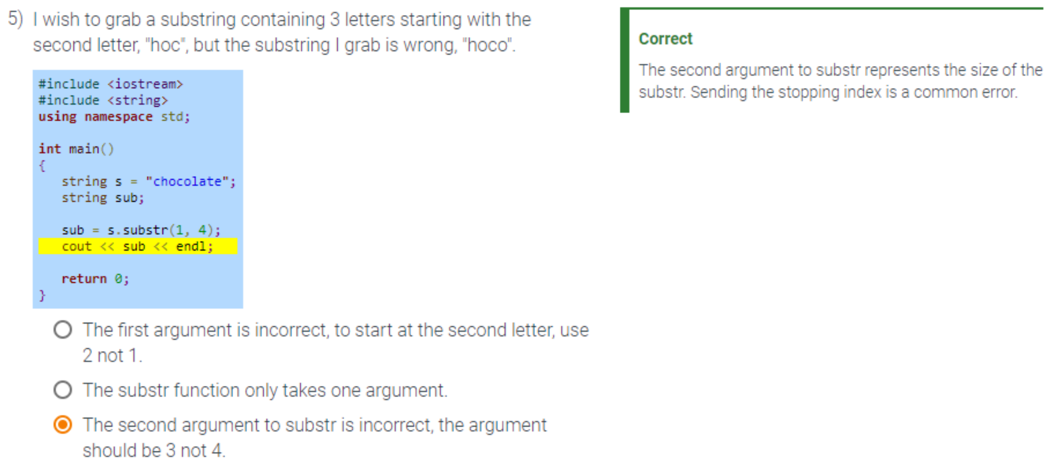
```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string s = "chocolate";
   string sub;

   sub = s.substr(1, 4);
   cout << sub << endl;

   return 0;
}
```

- The first argument is incorrect, to start at the second letter, use 2 not 1.
- [✓ The substr function only takes one argument.]
- The second argument to substr is incorrect, the argument should be 3 not 4.

Incorrect

The substr function can be invoked in two ways. The first way specifies only a starting position. The second way specifies a starting position and the size of the substring.

Figure 122: String Functions, Question #5, Answer #2



5) I wish to grab a substring containing 3 letters starting with the second letter, "hoc", but the substring I grab is wrong, "hoco".

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string s = "chocolate";
   string sub;

   sub = s.substr(1, 4);
   cout << sub << endl;

   return 0;
}
```

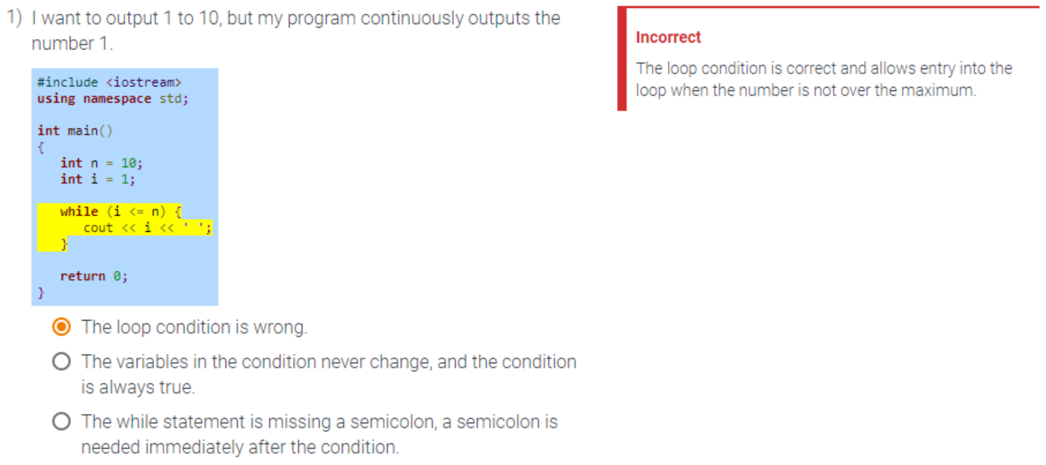
- ○ The first argument is incorrect, to start at the second letter, use 2 not 1.
- ○ The substr function only takes one argument.
- ● The second argument to substr is incorrect, the argument should be 3 not 4.

Correct

The second argument to substr represents the size of the substr. Sending the stopping index is a common error.

Figure 121: String Functions, Question #5, Answer #3

Loops Exercises



1) I want to output 1 to 10, but my program continuously outputs the number 1.

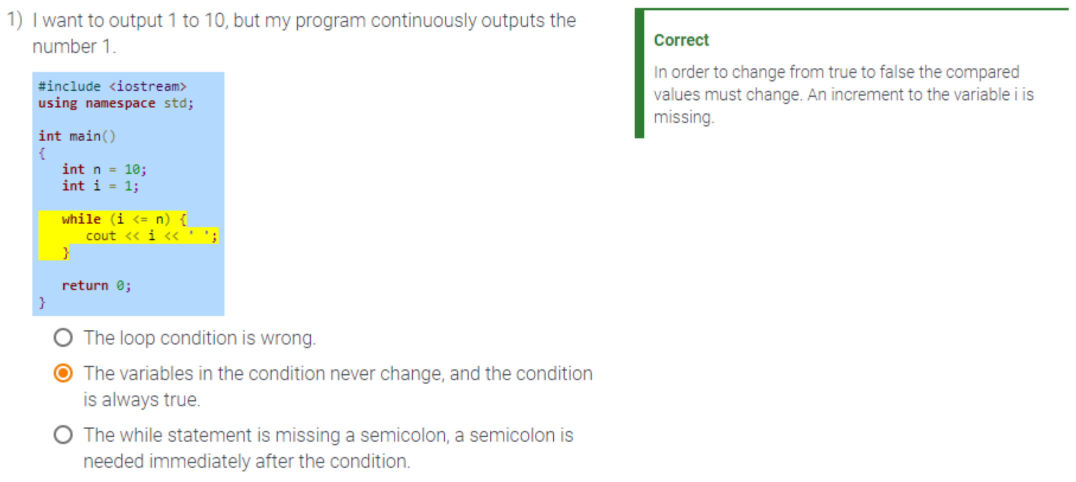
```
#include <iostream>
using namespace std;

int main()
{
   int n = 10;
   int i = 1;

   while (i <= n) {
      cout << i << ' ';
   }

   return 0;
}
```

- ● The loop condition is wrong.
- ○ The variables in the condition never change, and the condition is always true.
- ○ The while statement is missing a semicolon, a semicolon is needed immediately after the condition.

Incorrect

The loop condition is correct and allows entry into the loop when the number is not over the maximum.

Figure 123: Loops, Question #1, Answer #1



1) I want to output 1 to 10, but my program continuously outputs the number 1.

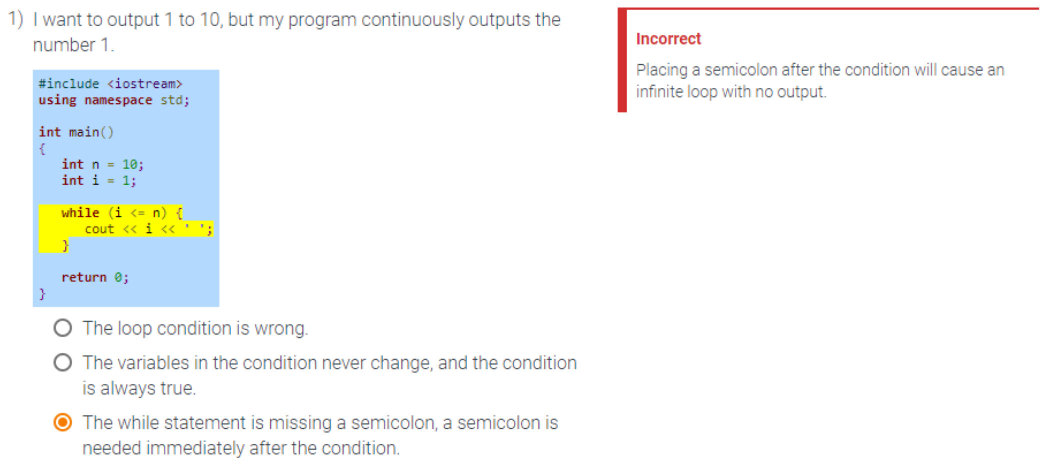
```
#include <iostream>
using namespace std;

int main()
{
    int n = 10;
    int i = 1;

    while (i <= n) {
        cout << i << ' ';
    }

    return 0;
}
```

- ○ The loop condition is wrong.
- ◉ The variables in the condition never change, and the condition is always true.
- ○ The while statement is missing a semicolon, a semicolon is needed immediately after the condition.

Correct

In order to change from true to false the compared values must change. An increment to the variable i is missing.

Figure 124: Loops, Question #1, Answer #2



1) I want to output 1 to 10, but my program continuously outputs the number 1.

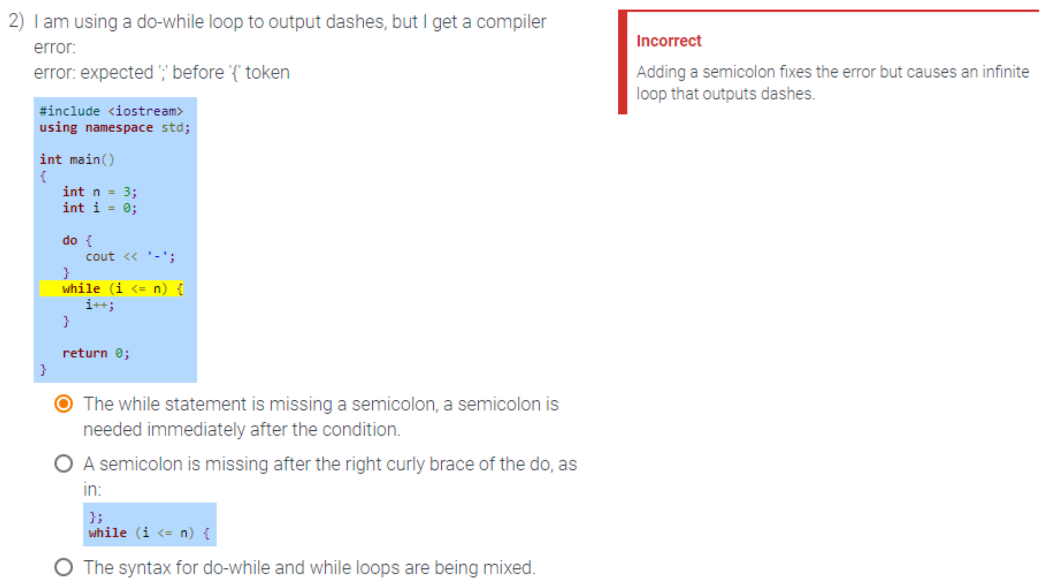
```
#include <iostream>
using namespace std;

int main()
{
    int n = 10;
    int i = 1;

    while (i <= n) {
        cout << i << ' ';
    }

    return 0;
}
```

- ○ The loop condition is wrong.
- ○ The variables in the condition never change, and the condition is always true.
- ◉ The while statement is missing a semicolon, a semicolon is needed immediately after the condition.

Incorrect

Placing a semicolon after the condition will cause an infinite loop with no output.

Figure 125: Loops, Question #1, Answer #3


2) I am using a do-while loop to output dashes, but I get a compiler error:
error: expected ';' before '{' token

```
#include <iostream>
using namespace std;

int main()
{
   int n = 3;
   int i = 0;

   do {
      cout << '-';
   }
   while (i <= n) {
      i++;
   }

   return 0;
}
```

[✓ The while statement is missing a semicolon, a semicolon is needed immediately after the condition.]

○ A semicolon is missing after the right curly brace of the do, as in:

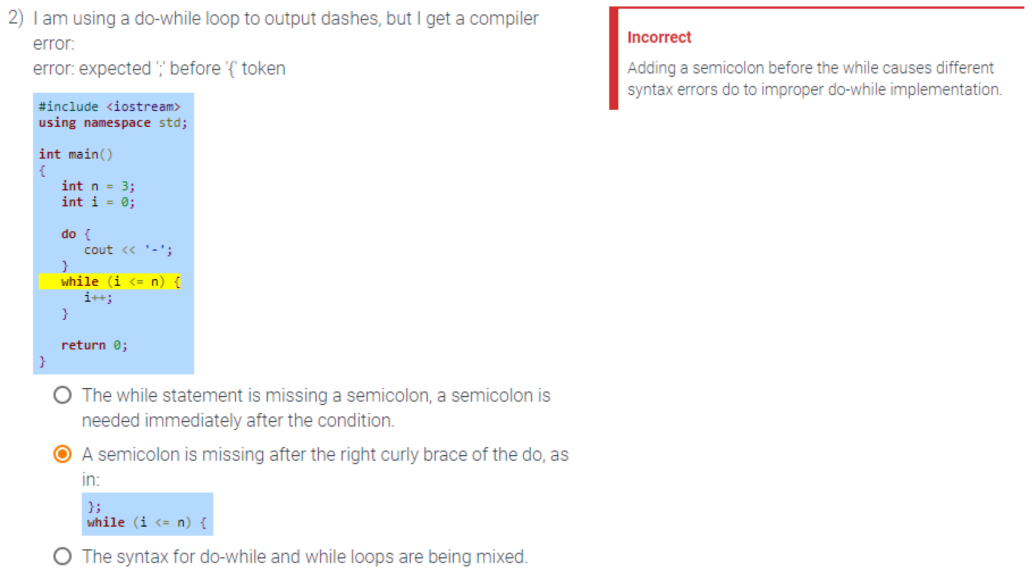
```
};
while (i <= n) {
```

○ The syntax for do-while and while loops are being mixed.

Incorrect

Adding a semicolon fixes the error but causes an infinite loop that outputs dashes.

Figure 126: Loops, Question #2, Answer #1


2) I am using a do-while loop to output dashes, but I get a compiler error:
error: expected ';' before '{' token

```
#include <iostream>
using namespace std;

int main()
{
   int n = 3;
   int i = 0;

   do {
      cout << '-';
   }
   while (i <= n) {
      i++;
   }

   return 0;
}
```

○ The while statement is missing a semicolon, a semicolon is needed immediately after the condition.

[✓ A semicolon is missing after the right curly brace of the do, as in:]

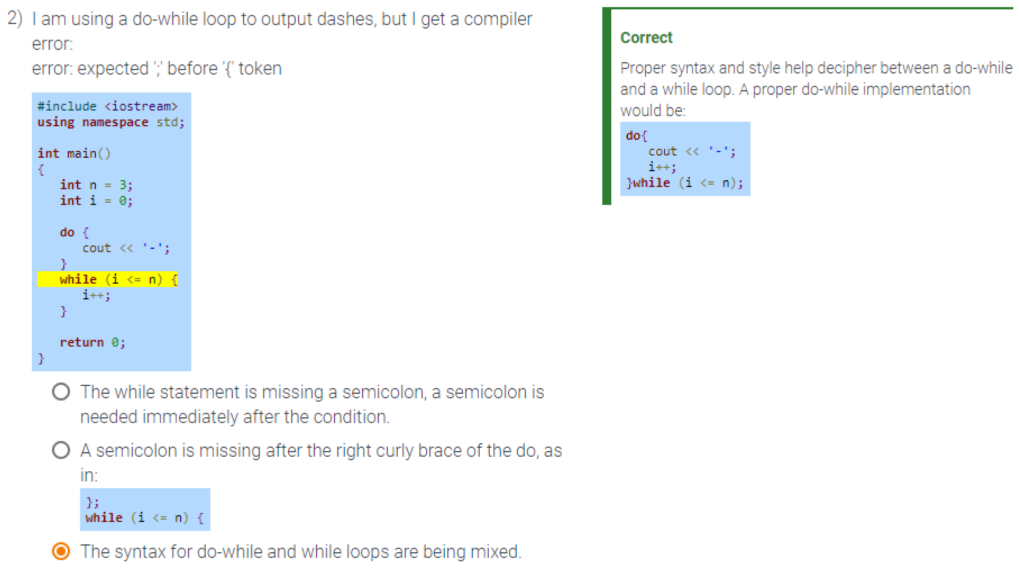
```
};
while (i <= n) {
```

○ The syntax for do-while and while loops are being mixed.

Incorrect

Adding a semicolon before the while causes different syntax errors do to improper do-while implementation.

Figure 127: Loops, Question #2, Answer #2



2) I am using a do-while loop to output dashes, but I get a compiler error:
error: expected ';' before '{' token

```
#include <iostream>
using namespace std;

int main()
{
   int n = 3;
   int i = 0;

   do {
      cout << '-';
   }
   while (i <= n) {
      i++;
   }

   return 0;
}
```

- [ ] The while statement is missing a semicolon, a semicolon is needed immediately after the condition.
- [ ] A semicolon is missing after the right curly brace of the do, as in:

```
};
while (i <= n) {
```

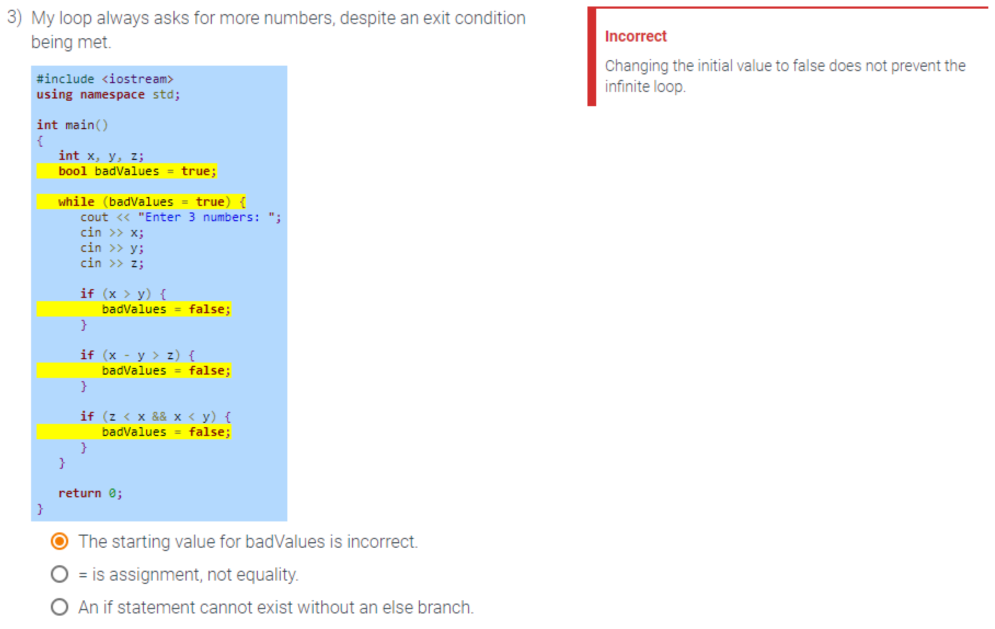
- [✓] The syntax for do-while and while loops are being mixed.

Correct

Proper syntax and style help decipher between a do-while and a while loop. A proper do-while implementation would be:

```
do{
   cout << '-';
   i++;
}while (i <= n);
```

Figure 128: Loops, Question #2, Answer #3



3) My loop always asks for more numbers, despite an exit condition being met.

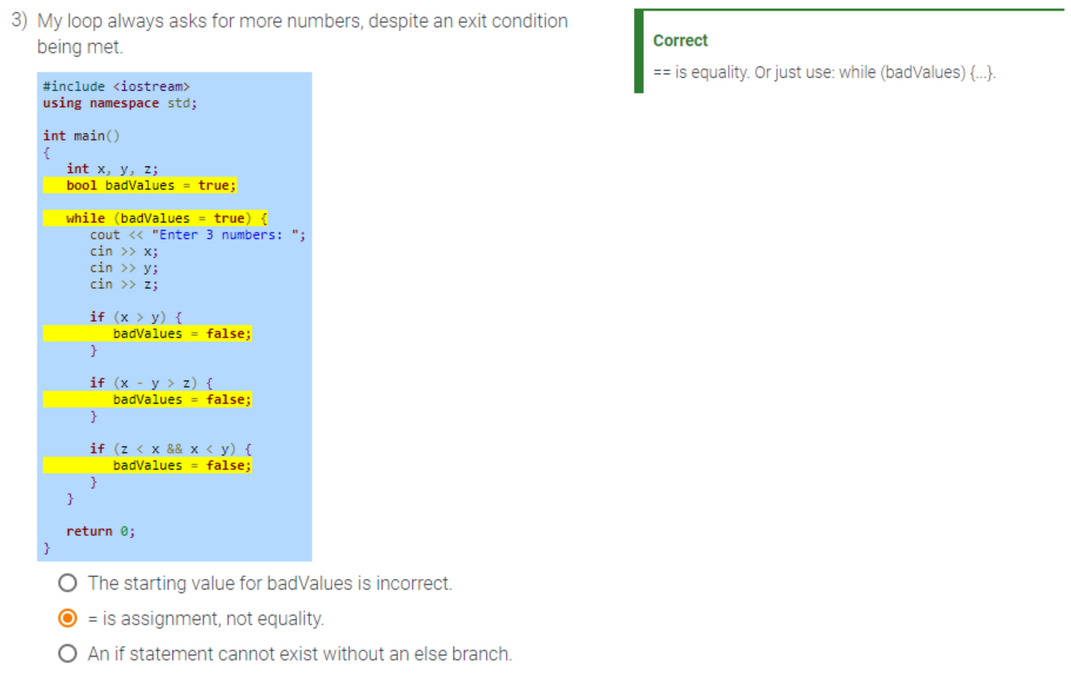
```
#include <iostream>
using namespace std;

int main()
{
   int x, y, z;
   bool badValues = true;

   while (badValues = true) {
      cout << "Enter 3 numbers: ";
      cin >> x;
      cin >> y;
      cin >> z;

      if (x > y) {
         badValues = false;
      }

      if (x - y > z) {
         badValues = false;
      }

      if (z < x && x < y) {
         badValues = false;
      }
   }

   return 0;
}
```

- [✓] The starting value for badValues is incorrect.
- [ ] = is assignment, not equality.
- [ ] An if statement cannot exist without an else branch.

Incorrect

Changing the initial value to false does not prevent the infinite loop.

Figure 129: Loops, Question #3, Answer #1



3) My loop always asks for more numbers, despite an exit condition being met.

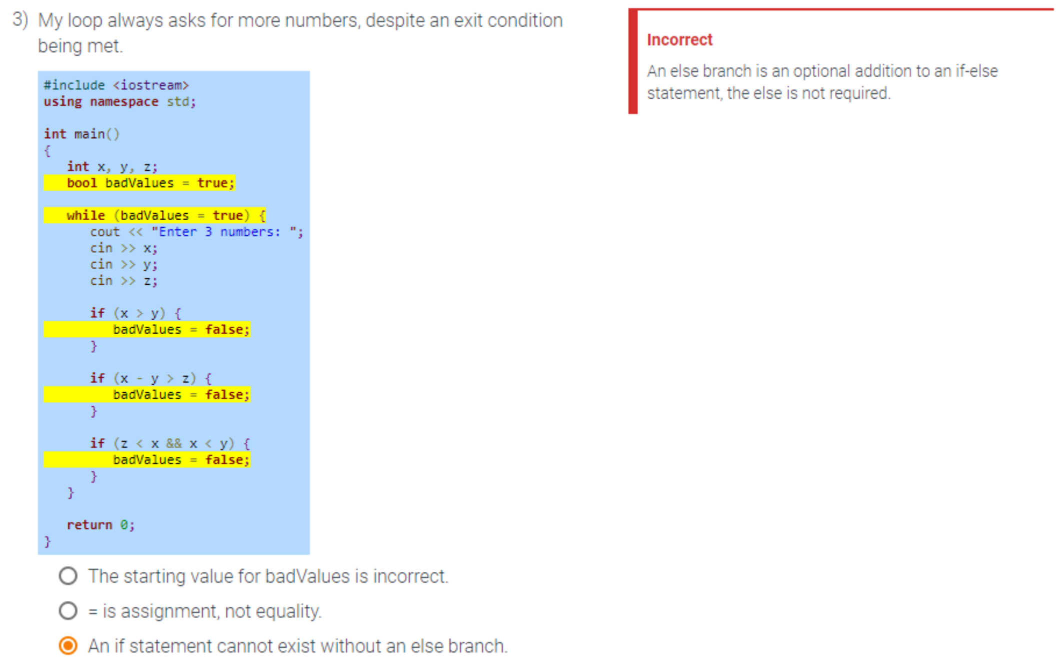
```cpp
#include <iostream>
using namespace std;

int main()
{
    int x, y, z;
    bool badValues = true;

    while (badValues = true) {
        cout << "Enter 3 numbers: ";
        cin >> x;
        cin >> y;
        cin >> z;

        if (x > y) {
            badValues = false;
        }

        if (x - y > z) {
            badValues = false;
        }

        if (z < x && x < y) {
            badValues = false;
        }
    }

    return 0;
}
```

- The starting value for badValues is incorrect.
- [✓ = is assignment, not equality.]
- An if statement cannot exist without an else branch.

Correct

== is equality. Or just use: while (badValues) {...}.

Figure 130: Loops, Question #3, Answer #2



3) My loop always asks for more numbers, despite an exit condition being met.

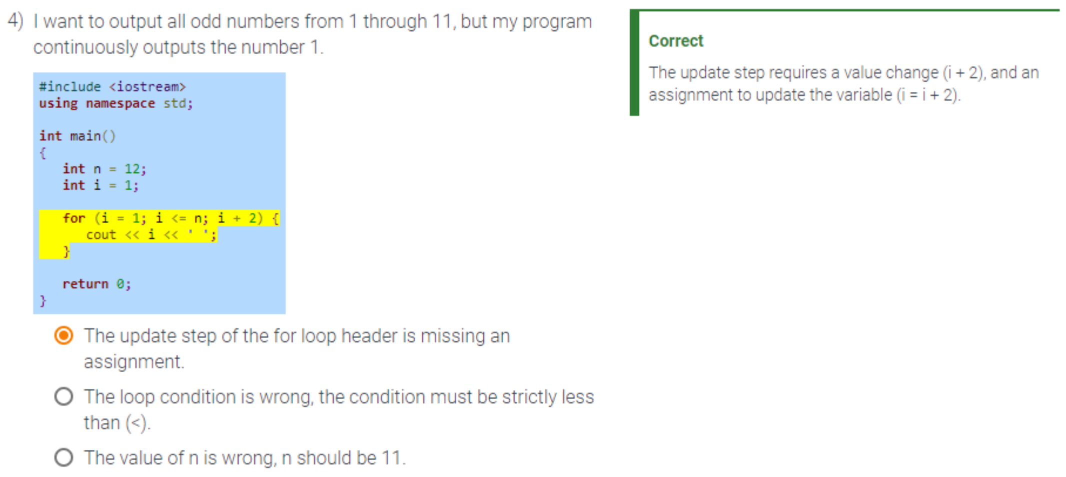
```cpp
#include <iostream>
using namespace std;

int main()
{
    int x, y, z;
    bool badValues = true;

    while (badValues = true) {
        cout << "Enter 3 numbers: ";
        cin >> x;
        cin >> y;
        cin >> z;

        if (x > y) {
            badValues = false;
        }

        if (x - y > z) {
            badValues = false;
        }

        if (z < x && x < y) {
            badValues = false;
        }
    }

    return 0;
}
```

- The starting value for badValues is incorrect.
- = is assignment, not equality.
- [✓ An if statement cannot exist without an else branch.]

Incorrect

An else branch is an optional addition to an if-else statement, the else is not required.

Figure 131: Loops, Question #3, Answer #3


4) I want to output all odd numbers from 1 through 11, but my program continuously outputs the number 1.

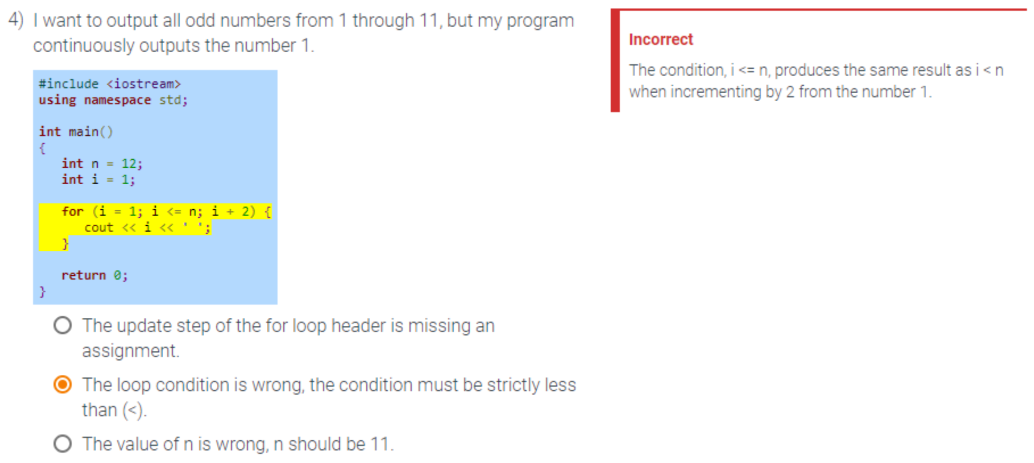
```
#include <iostream>
using namespace std;

int main()
{
    int n = 12;
    int i = 1;

    for (i = 1; i <= n; i + 2) {
        cout << i << ' ';
    }

    return 0;
}
```

- [✓ The update step of the for loop header is missing an assignment.]
- The loop condition is wrong, the condition must be strictly less than (<).
- The value of n is wrong, n should be 11.

Correct

The update step requires a value change (i + 2), and an assignment to update the variable (i = i + 2).

Figure 132: Loops, Question #4, Answer #1


4) I want to output all odd numbers from 1 through 11, but my program continuously outputs the number 1.

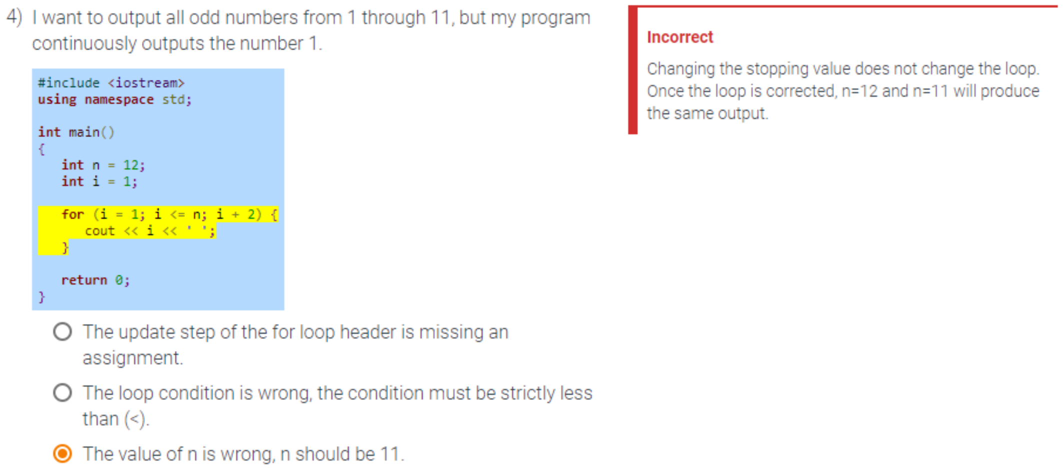
```
#include <iostream>
using namespace std;

int main()
{
    int n = 12;
    int i = 1;

    for (i = 1; i <= n; i + 2) {
        cout << i << ' ';
    }

    return 0;
}
```

- The update step of the for loop header is missing an assignment.
- [✓ The loop condition is wrong, the condition must be strictly less than (<).]
- The value of n is wrong, n should be 11.

Incorrect

The condition, i <= n, produces the same result as i < n when incrementing by 2 from the number 1.

Figure 133: Loops, Question #4, Answer #2



4) I want to output all odd numbers from 1 through 11, but my program continuously outputs the number 1.

```
#include <iostream>
using namespace std;

int main()
{
   int n = 12;
   int i = 1;

   for (i = 1; i <= n; i + 2) {
      cout << i << ' ';
   }

   return 0;
}
```

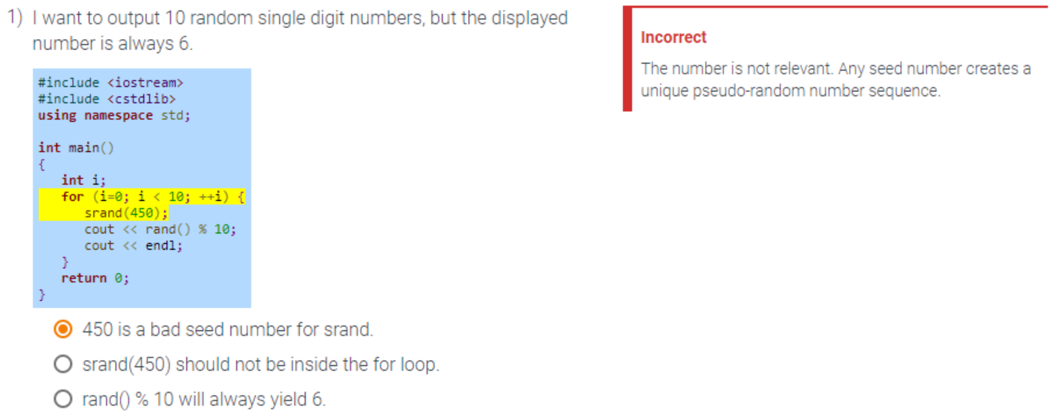
- The update step of the for loop header is missing an assignment.
- The loop condition is wrong, the condition must be strictly less than (<).
- [✓ The value of n is wrong, n should be 11.]

Incorrect

Changing the stopping value does not change the loop. Once the loop is corrected, n=12 and n=11 will produce the same output.

Figure 134: Loops, Question #4, Answer #3

Random Number Exercises



1) I want to output 10 random single digit numbers, but the displayed number is always 6.

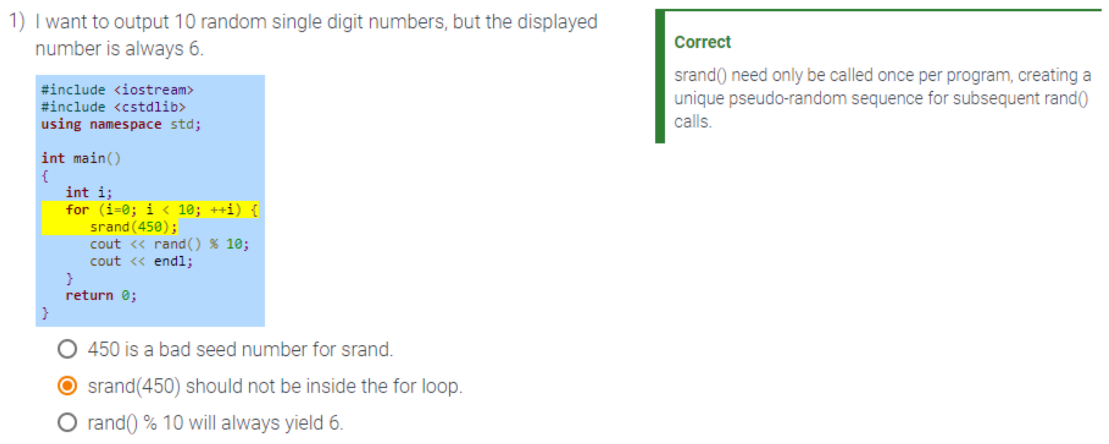
```
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
   int i;
   for (i=0; i < 10; ++i) {
      srand(450);
      cout << rand() % 10;
      cout << endl;
   }
   return 0;
}
```

- [✓ 450 is a bad seed number for srand.]
- srand(450) should not be inside the for loop.
- rand() % 10 will always yield 6.

Incorrect

The number is not relevant. Any seed number creates a unique pseudo-random number sequence.

Figure 135: Random Numbers, Question #1, Answer #1


1) I want to output 10 random single digit numbers, but the displayed number is always 6.

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
   int i;
   for (i=0; i < 10; ++i) {
      srand(450);
      cout << rand() % 10;
      cout << endl;
   }
   return 0;
}
```

○ 450 is a bad seed number for srand.

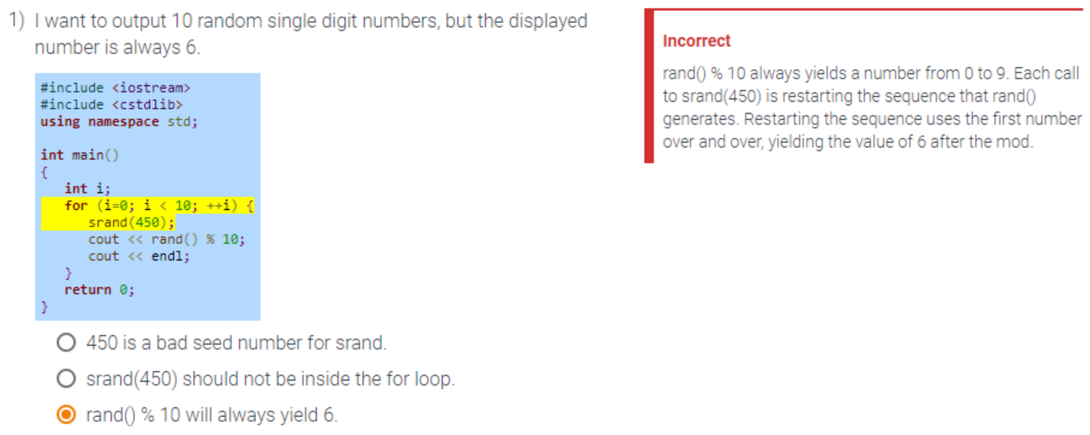
● srand(450) should not be inside the for loop.

○ rand() % 10 will always yield 6.

Correct

srand() need only be called once per program, creating a unique pseudo-random sequence for subsequent rand() calls.

Figure 136: Random Numbers, Question #1, Answer #2


1) I want to output 10 random single digit numbers, but the displayed number is always 6.

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
   int i;
   for (i=0; i < 10; ++i) {
      srand(450);
      cout << rand() % 10;
      cout << endl;
   }
   return 0;
}
```

○ 450 is a bad seed number for srand.

○ srand(450) should not be inside the for loop.

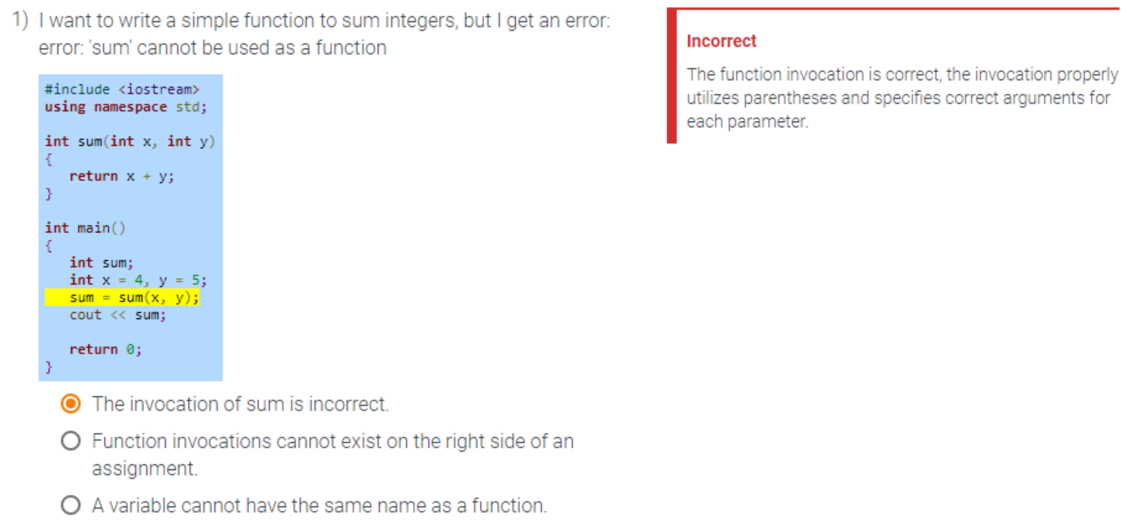
● rand() % 10 will always yield 6.

Incorrect

rand() % 10 always yields a number from 0 to 9. Each call to srand(450) is restarting the sequence that rand() generates. Restarting the sequence uses the first number over and over, yielding the value of 6 after the mod.

Figure 137: Random Numbers, Question #1, Answer #3

Function Invocation Exercises



1) I want to write a simple function to sum integers, but I get an error:
error: 'sum' cannot be used as a function

```
#include <iostream>
using namespace std;

int sum(int x, int y)
{
    return x + y;
}

int main()
{
    int sum;
    int x = 4, y = 5;
    sum = sum(x, y);
    cout << sum;

    return 0;
}
```

[✓ The invocation of sum is incorrect.]

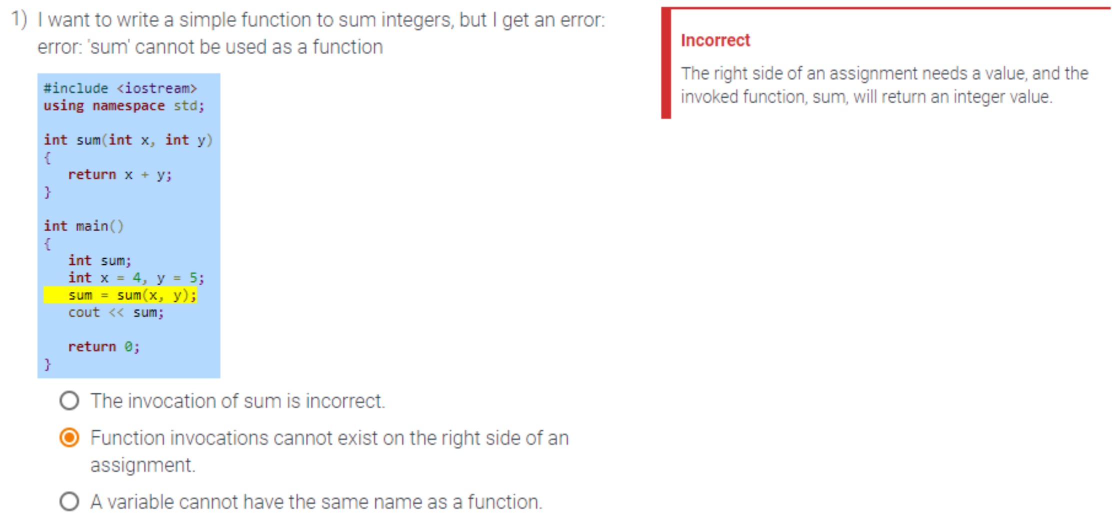
Function invocations cannot exist on the right side of an assignment.

A variable cannot have the same name as a function.

Incorrect

The function invocation is correct, the invocation properly utilizes parentheses and specifies correct arguments for each parameter.

Figure 138: Function Invocation, Question #1, Answer #1



1) I want to write a simple function to sum integers, but I get an error:
error: 'sum' cannot be used as a function

```
#include <iostream>
using namespace std;

int sum(int x, int y)
{
    return x + y;
}

int main()
{
    int sum;
    int x = 4, y = 5;
    sum = sum(x, y);
    cout << sum;

    return 0;
}
```

The invocation of sum is incorrect.

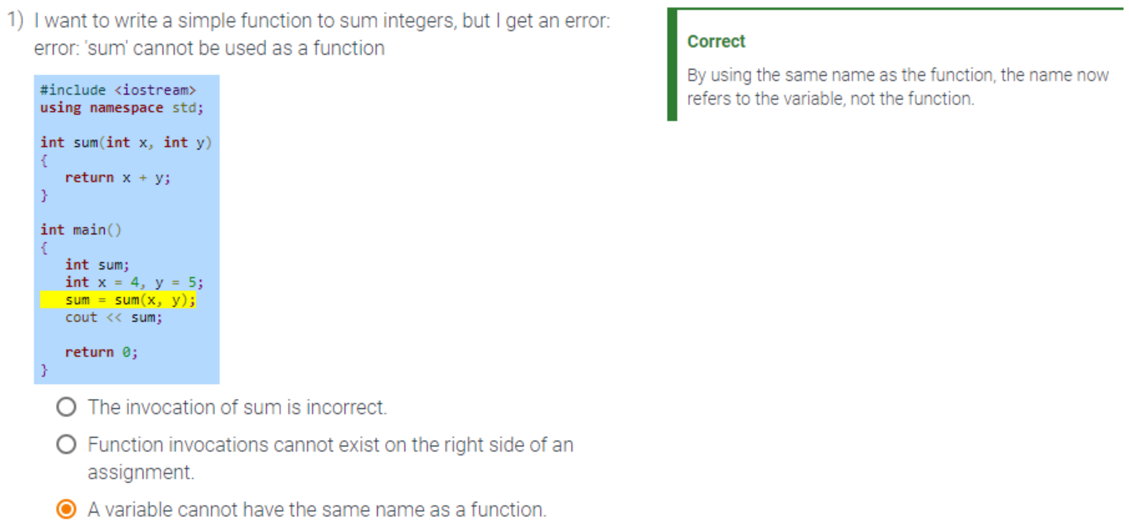
[✓ Function invocations cannot exist on the right side of an assignment.]

A variable cannot have the same name as a function.

Incorrect

The right side of an assignment needs a value, and the invoked function, sum, will return an integer value.

Figure 139: Function Invocation, Question #1, Answer #2



1) I want to write a simple function to sum integers, but I get an error:
error: 'sum' cannot be used as a function

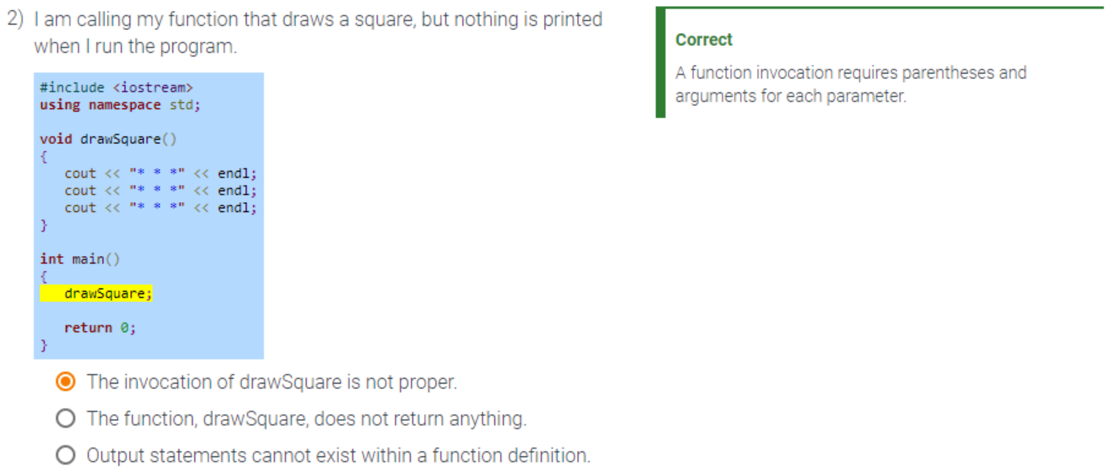
```
#include <iostream>
using namespace std;

int sum(int x, int y)
{
   return x + y;
}

int main()
{
   int sum;
   int x = 4, y = 5;
   sum = sum(x, y);
   cout << sum;

   return 0;
}
```

- The invocation of sum is incorrect.
- Function invocations cannot exist on the right side of an assignment.
- [✓ A variable cannot have the same name as a function.]

Correct

By using the same name as the function, the name now refers to the variable, not the function.

Figure 140: Function Invocation, Question #1, Answer #3



2) I am calling my function that draws a square, but nothing is printed when I run the program.

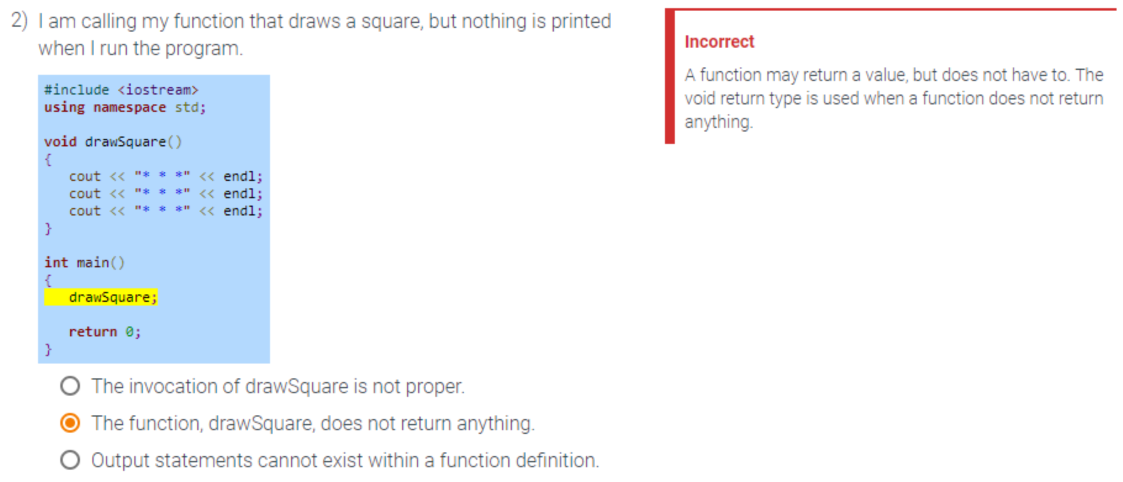
```
#include <iostream>
using namespace std;

void drawSquare()
{
   cout << "* * *" << endl;
   cout << "* * *" << endl;
   cout << "* * *" << endl;
}

int main()
{
   drawSquare;

   return 0;
}
```

- [✓ The invocation of drawSquare is not proper.]
- The function, drawSquare, does not return anything.
- Output statements cannot exist within a function definition.

Correct

A function invocation requires parentheses and arguments for each parameter.

Figure 141: Function Invocation, Question #2, Answer #1



2) I am calling my function that draws a square, but nothing is printed when I run the program.

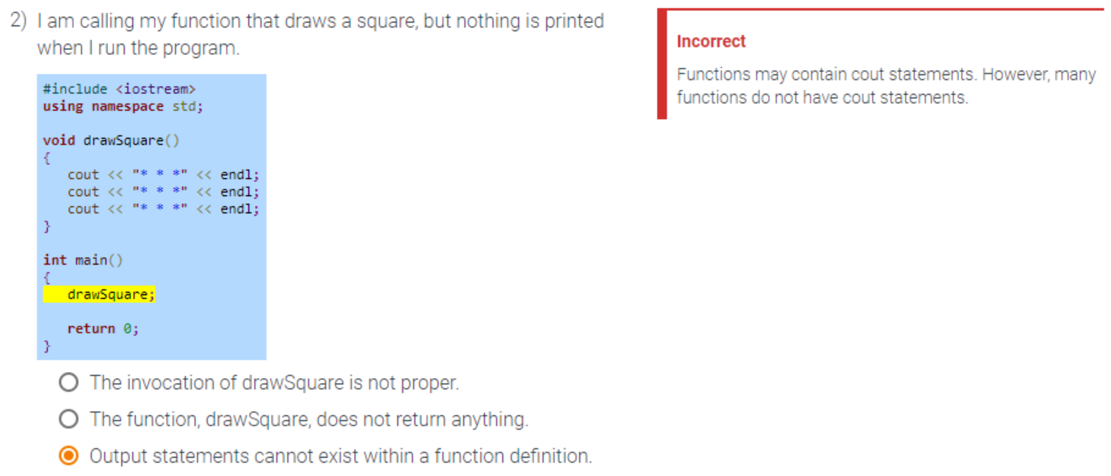
```
#include <iostream>
using namespace std;

void drawSquare()
{
    cout << "* * *" << endl;
    cout << "* * *" << endl;
    cout << "* * *" << endl;
}

int main()
{
    drawSquare;

    return 0;
}
```

- ○ The invocation of drawSquare is not proper.
- [✓ The function, drawSquare, does not return anything.]
- ○ Output statements cannot exist within a function definition.

Incorrect

A function may return a value, but does not have to. The void return type is used when a function does not return anything.

Figure 142: Function Invocation, Question #2, Answer #2



2) I am calling my function that draws a square, but nothing is printed when I run the program.

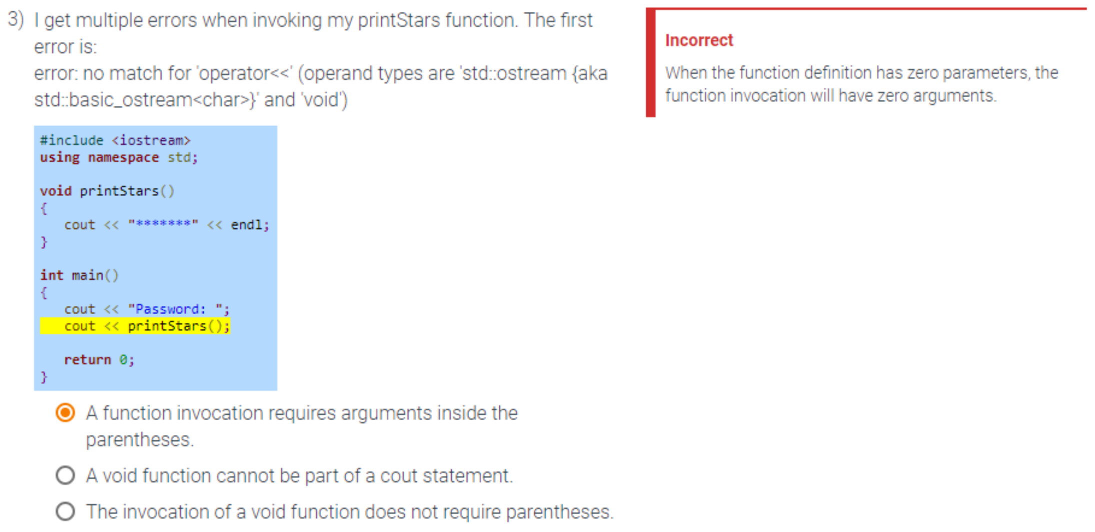
```
#include <iostream>
using namespace std;

void drawSquare()
{
    cout << "* * *" << endl;
    cout << "* * *" << endl;
    cout << "* * *" << endl;
}

int main()
{
    drawSquare;

    return 0;
}
```

- ○ The invocation of drawSquare is not proper.
- ○ The function, drawSquare, does not return anything.
- [✓ Output statements cannot exist within a function definition.]

Incorrect

Functions may contain cout statements. However, many functions do not have cout statements.

Figure 143: Function Invocation, Question #2, Answer #3



3) I get multiple errors when invoking my printStars function. The first error is:
error: no match for 'operator<<' (operand types are 'std::ostream {aka std::basic_ostream<char>}' and 'void')

```
#include <iostream>
using namespace std;

void printStars()
{
   cout << "*******" << endl;
}

int main()
{
   cout << "Password: ";
   cout << printStars();

   return 0;
}
```

[✓ A function invocation requires arguments inside the parentheses.]

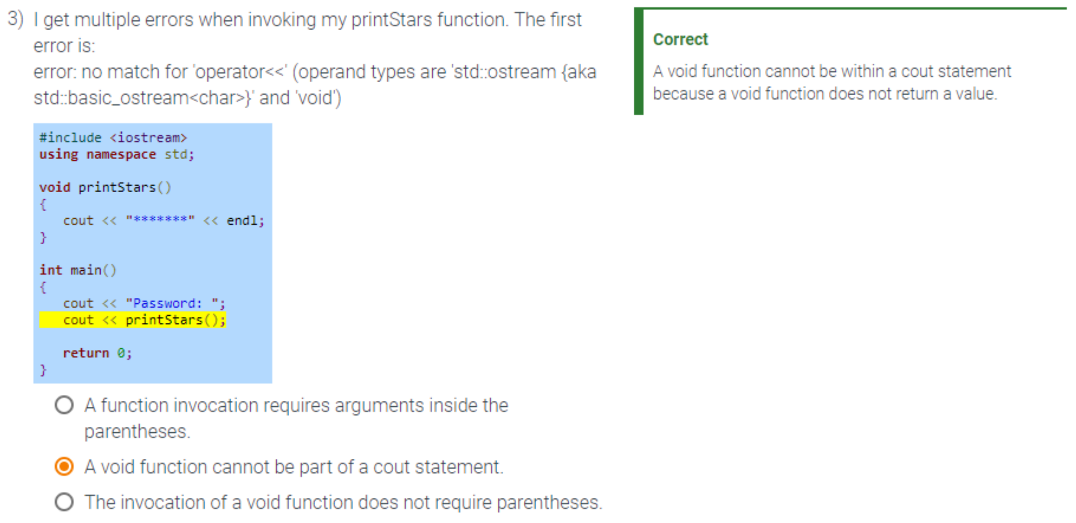
A void function cannot be part of a cout statement.

The invocation of a void function does not require parentheses.

Incorrect

When the function definition has zero parameters, the function invocation will have zero arguments.

Figure 144: Function Invocation, Question #3, Answer #1



3) I get multiple errors when invoking my printStars function. The first error is:
error: no match for 'operator<<' (operand types are 'std::ostream {aka std::basic_ostream<char>}' and 'void')

```
#include <iostream>
using namespace std;

void printStars()
{
   cout << "*******" << endl;
}

int main()
{
   cout << "Password: ";
   cout << printStars();

   return 0;
}
```

A function invocation requires arguments inside the parentheses.

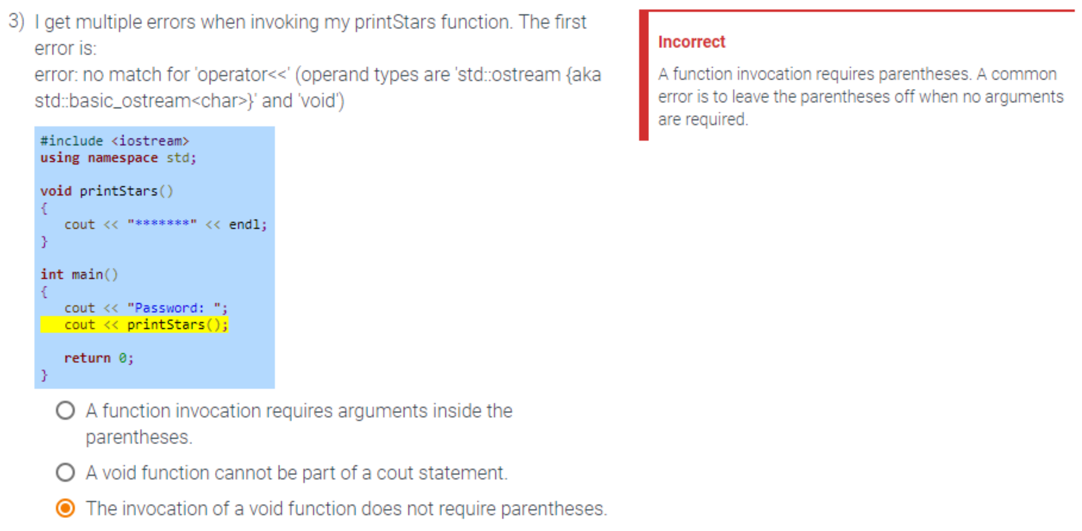
[✓ A void function cannot be part of a cout statement.]

The invocation of a void function does not require parentheses.

Correct

A void function cannot be within a cout statement because a void function does not return a value.

Figure 145: Function Invocation, Question #3, Answer #2



3) I get multiple errors when invoking my printStars function. The first error is:
error: no match for 'operator<<' (operand types are 'std::ostream {aka std::basic_ostream<char>}' and 'void')

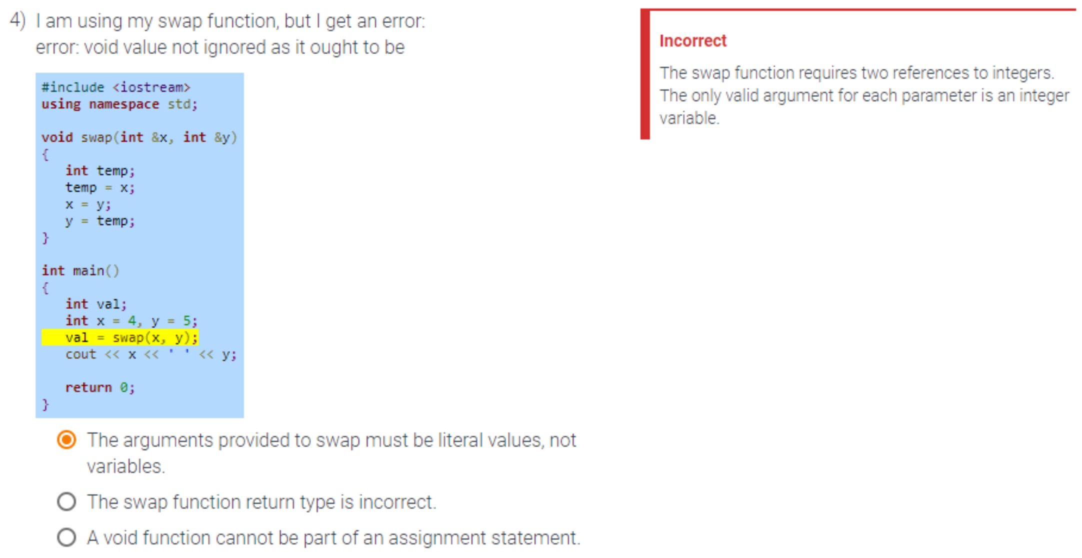
```
#include <iostream>
using namespace std;

void printStars()
{
   cout << "*******" << endl;
}

int main()
{
   cout << "Password: ";
   cout << printStars();

   return 0;
}
```

- ○ A function invocation requires arguments inside the parentheses.
- ○ A void function cannot be part of a cout statement.
- [✓ The invocation of a void function does not require parentheses.]

Incorrect

A function invocation requires parentheses. A common error is to leave the parentheses off when no arguments are required.

Figure 146: Function Invocation, Question #3, Answer #3



4) I am using my swap function, but I get an error:
error: void value not ignored as it ought to be

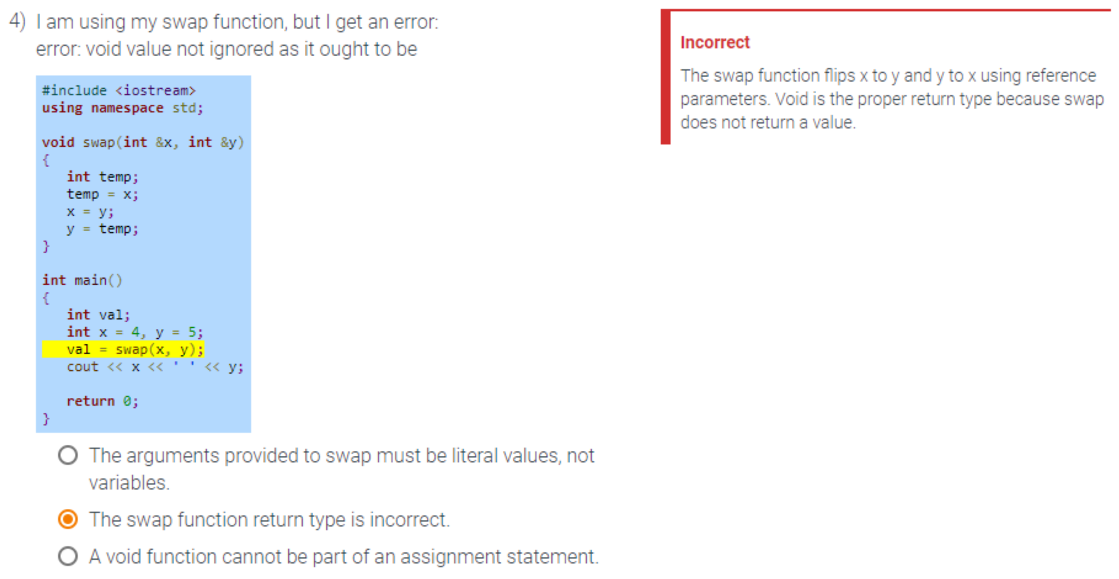
```
#include <iostream>
using namespace std;

void swap(int &x, int &y)
{
   int temp;
   temp = x;
   x = y;
   y = temp;
}

int main()
{
   int val;
   int x = 4, y = 5;
   val = swap(x, y);
   cout << x << ' ' << y;

   return 0;
}
```

- [✓ The arguments provided to swap must be literal values, not variables.]
- ○ The swap function return type is incorrect.
- ○ A void function cannot be part of an assignment statement.

Incorrect

The swap function requires two references to integers. The only valid argument for each parameter is an integer variable.

Figure 147: Function Invocation, Question #4, Answer #1



4) I am using my swap function, but I get an error:
error: void value not ignored as it ought to be

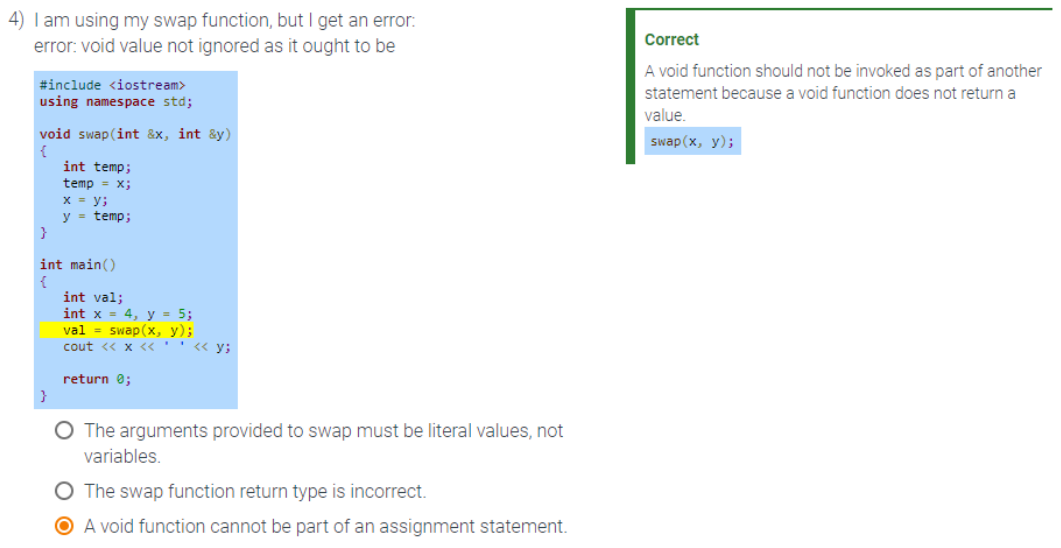
```
#include <iostream>
using namespace std;

void swap(int &x, int &y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}

int main()
{
    int val;
    int x = 4, y = 5;
    val = swap(x, y);
    cout << x << ' ' << y;

    return 0;
}
```

- ○ The arguments provided to swap must be literal values, not variables.
- ● The swap function return type is incorrect.
- ○ A void function cannot be part of an assignment statement.

Incorrect

The swap function flips x to y and y to x using reference parameters. Void is the proper return type because swap does not return a value.

Figure 148: Function Invocation, Question #4, Answer #2



4) I am using my swap function, but I get an error:
error: void value not ignored as it ought to be

```
#include <iostream>
using namespace std;

void swap(int &x, int &y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}

int main()
{
    int val;
    int x = 4, y = 5;
    val = swap(x, y);
    cout << x << ' ' << y;

    return 0;
}
```

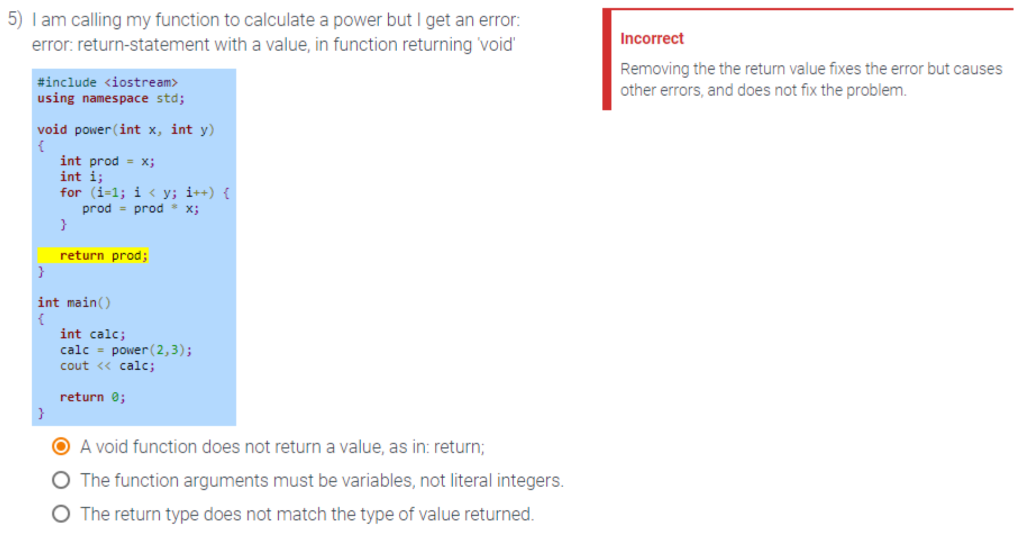
- ○ The arguments provided to swap must be literal values, not variables.
- ○ The swap function return type is incorrect.
- ● A void function cannot be part of an assignment statement.

Correct

A void function should not be invoked as part of another statement because a void function does not return a value.

```
swap(x, y);
```

Figure 149: Function Invocation, Question #4, Answer #3



5) I am calling my function to calculate a power but I get an error:
error: return-statement with a value, in function returning 'void'

```
#include <iostream>
using namespace std;

void power(int x, int y)
{
   int prod = x;
   int i;
   for (i=1; i < y; i++) {
      prod = prod * x;
   }

   return prod;
}

int main()
{
   int calc;
   calc = power(2,3);
   cout << calc;

   return 0;
}
```

[✓ A void function does not return a value, as in: return;]

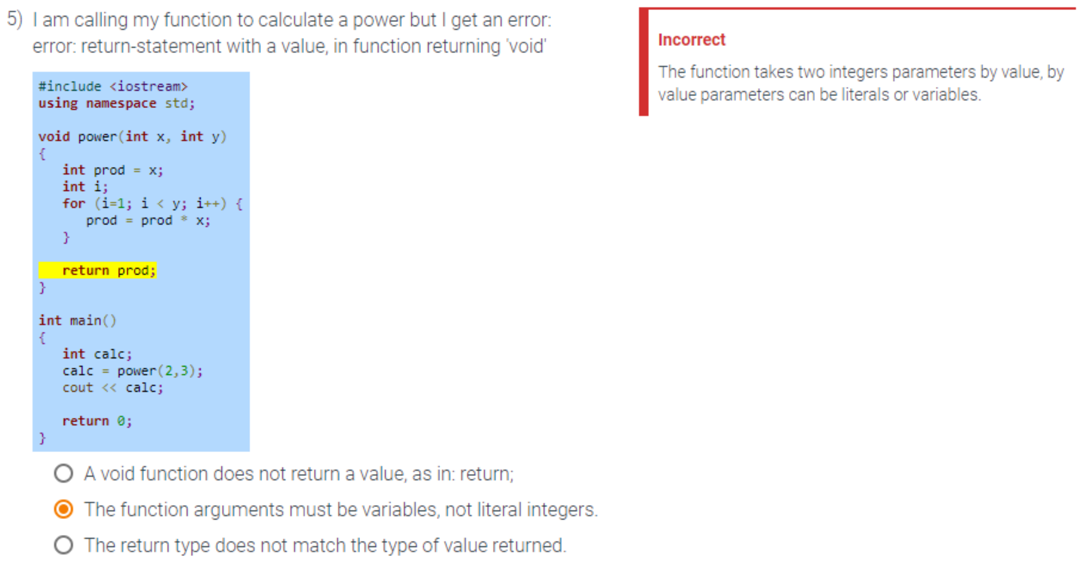
The function arguments must be variables, not literal integers.

The return type does not match the type of value returned.

Incorrect

Removing the the return value fixes the error but causes other errors, and does not fix the problem.

Figure 150: Function Invocation, Question #5, Answer #1



5) I am calling my function to calculate a power but I get an error:
error: return-statement with a value, in function returning 'void'

```
#include <iostream>
using namespace std;

void power(int x, int y)
{
   int prod = x;
   int i;
   for (i=1; i < y; i++) {
      prod = prod * x;
   }

   return prod;
}

int main()
{
   int calc;
   calc = power(2,3);
   cout << calc;

   return 0;
}
```

A void function does not return a value, as in: return;

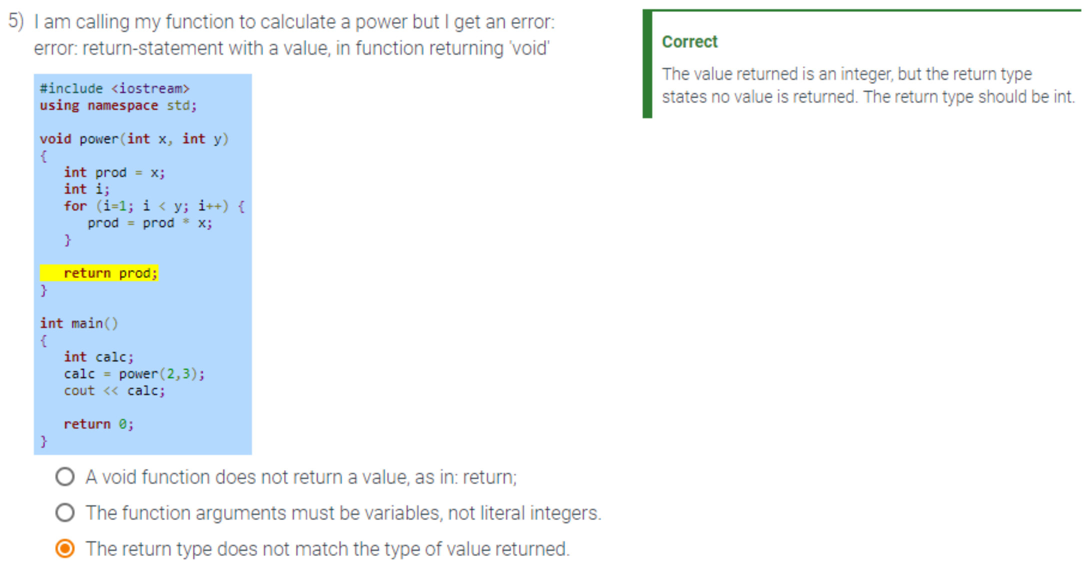
[✓ The function arguments must be variables, not literal integers.]

The return type does not match the type of value returned.

Incorrect

The function takes two integers parameters by value, by value parameters can be literals or variables.

Figure 151: Function Invocation, Question #5, Answer #2



5) I am calling my function to calculate a power but I get an error: error: return-statement with a value, in function returning 'void'

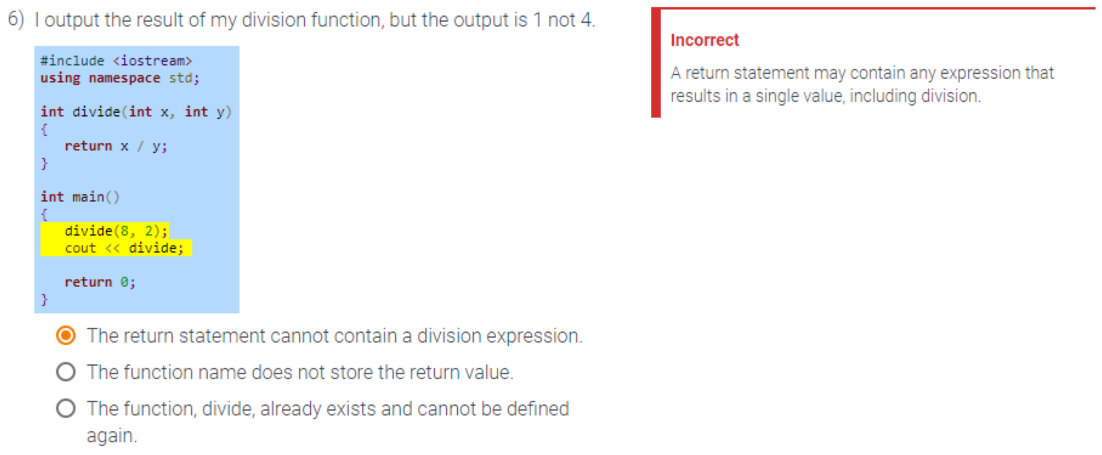
```
#include <iostream>
using namespace std;

void power(int x, int y)
{
   int prod = x;
   int i;
   for (i=1; i < y; i++) {
      prod = prod * x;
   }

   return prod;
}

int main()
{
   int calc;
   calc = power(2,3);
   cout << calc;

   return 0;
}
```

- ○ A void function does not return a value, as in: return;
- ○ The function arguments must be variables, not literal integers.
- ● The return type does not match the type of value returned.

Correct

The value returned is an integer, but the return type states no value is returned. The return type should be int.

Figure 152: Function Invocation, Question #5, Answer #3



6) I output the result of my division function, but the output is 1 not 4.

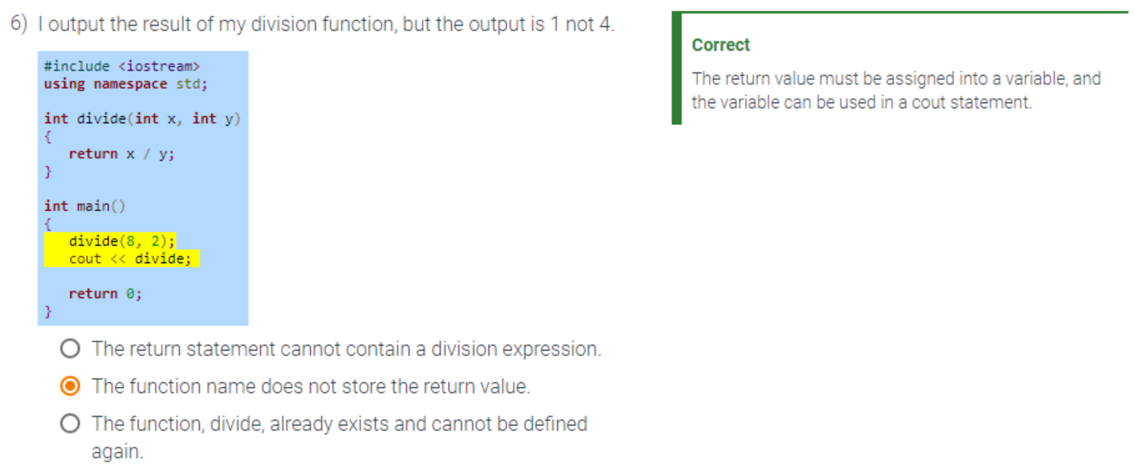
```
#include <iostream>
using namespace std;

int divide(int x, int y)
{
   return x / y;
}

int main()
{
   divide(8, 2);
   cout << divide;

   return 0;
}
```

- ● The return statement cannot contain a division expression.
- ○ The function name does not store the return value.
- ○ The function, divide, already exists and cannot be defined again.

Incorrect

A return statement may contain any expression that results in a single value, including division.

Figure 153: Function Invocation, Question #6, Answer #1



6) I output the result of my division function, but the output is 1 not 4.

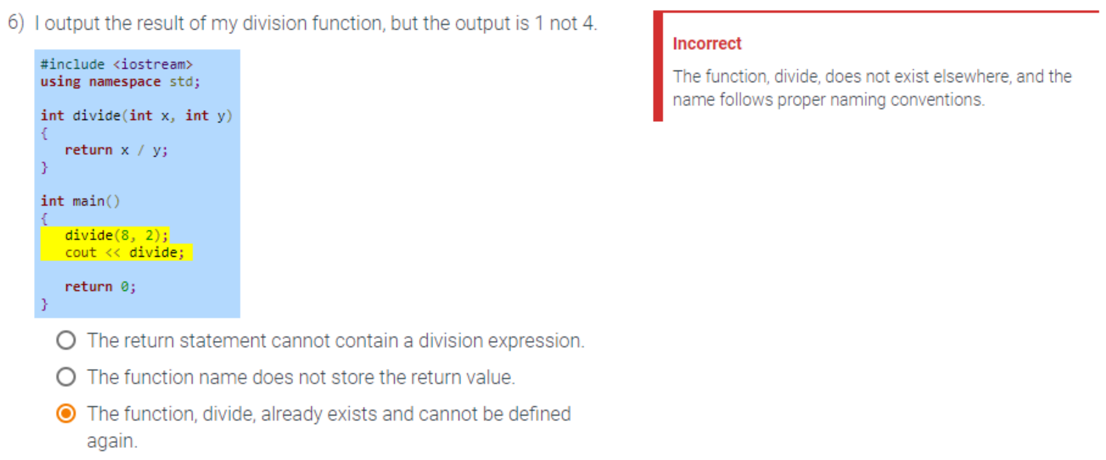
```
#include <iostream>
using namespace std;

int divide(int x, int y)
{
   return x / y;
}

int main()
{
   divide(8, 2);
   cout << divide;

   return 0;
}
```

- ○ The return statement cannot contain a division expression.
- [✓ The function name does not store the return value.]
- ○ The function, divide, already exists and cannot be defined again.

Correct

The return value must be assigned into a variable, and the variable can be used in a cout statement.

Figure 154: Function Invocation, Question #6, Answer #2



6) I output the result of my division function, but the output is 1 not 4.

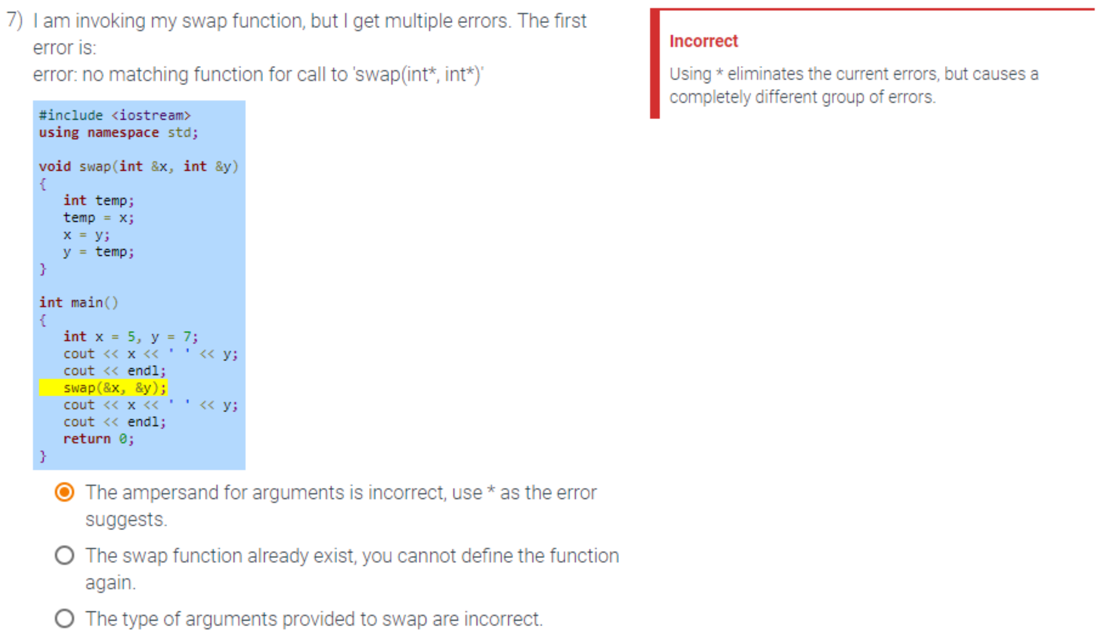
```
#include <iostream>
using namespace std;

int divide(int x, int y)
{
   return x / y;
}

int main()
{
   divide(8, 2);
   cout << divide;

   return 0;
}
```

- ○ The return statement cannot contain a division expression.
- ○ The function name does not store the return value.
- [✓ The function, divide, already exists and cannot be defined again.]

Incorrect

The function, divide, does not exist elsewhere, and the name follows proper naming conventions.

Figure 155: Function Invocation, Question #6, Answer #3


7) I am invoking my swap function, but I get multiple errors. The first error is:
error: no matching function for call to 'swap(int*, int*)'

```
#include <iostream>
using namespace std;

void swap(int &x, int &y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}

int main()
{
    int x = 5, y = 7;
    cout << x << ' ' << y;
    cout << endl;
    swap(&x, &y);
    cout << x << ' ' << y;
    cout << endl;
    return 0;
}
```

[✓ The ampersand for arguments is incorrect, use * as the error suggests.]

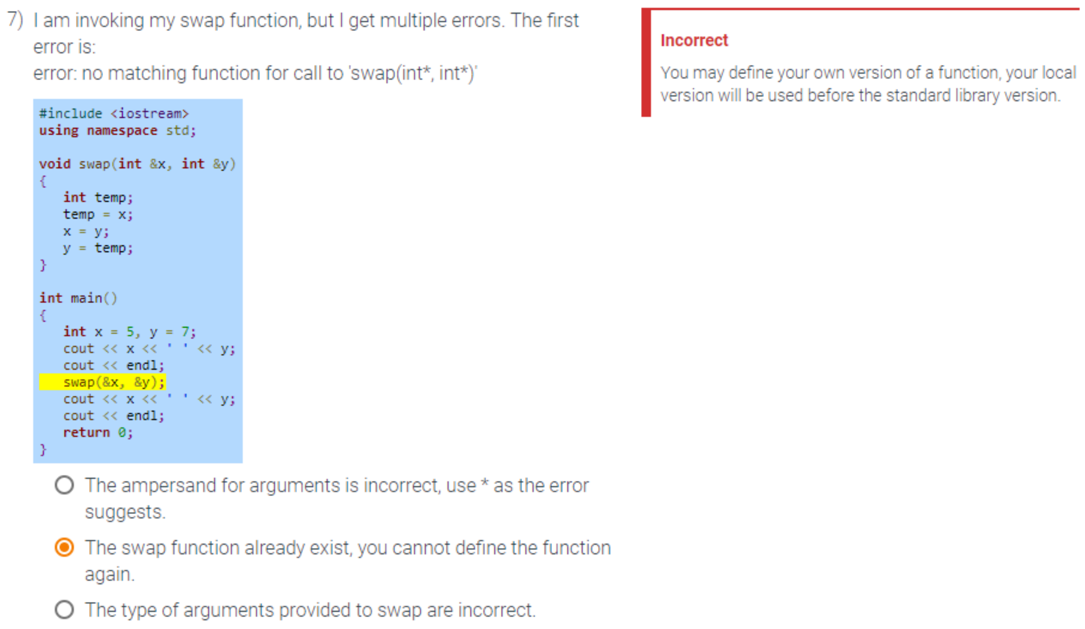
The swap function already exist, you cannot define the function again.

The type of arguments provided to swap are incorrect.

Incorrect

Using * eliminates the current errors, but causes a completely different group of errors.

Figure 156: Function Invocation, Question #7, Answer #1


7) I am invoking my swap function, but I get multiple errors. The first error is:
error: no matching function for call to 'swap(int*, int*)'

```
#include <iostream>
using namespace std;

void swap(int &x, int &y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}

int main()
{
    int x = 5, y = 7;
    cout << x << ' ' << y;
    cout << endl;
    swap(&x, &y);
    cout << x << ' ' << y;
    cout << endl;
    return 0;
}
```

The ampersand for arguments is incorrect, use * as the error suggests.

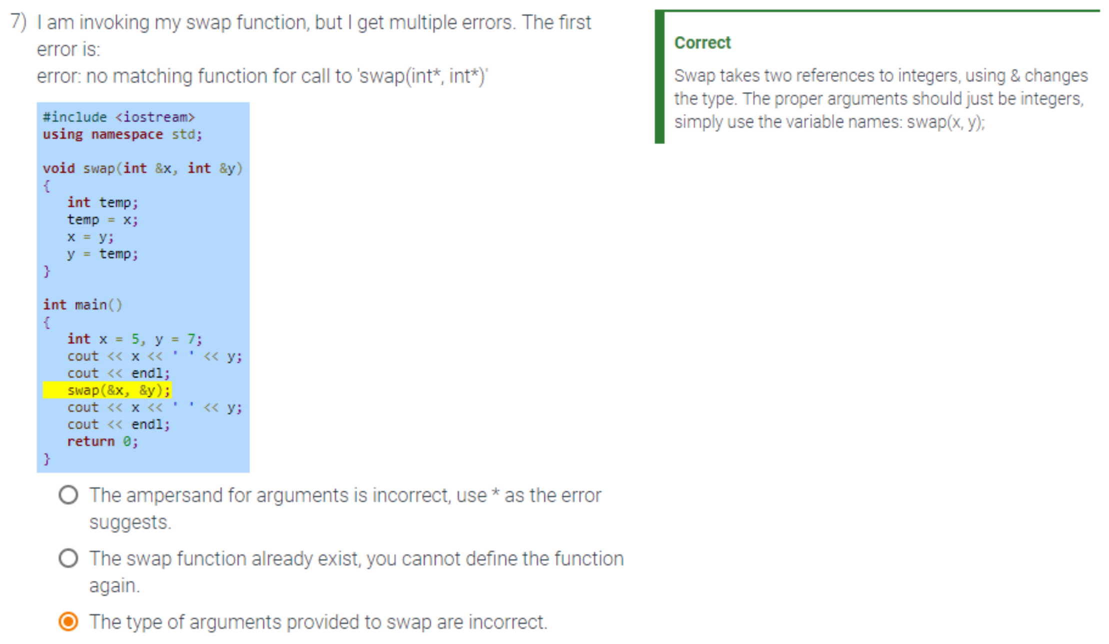
[✓ The swap function already exist, you cannot define the function again.]

The type of arguments provided to swap are incorrect.

Incorrect

You may define your own version of a function, your local version will be used before the standard library version.

Figure 157: Function Invocation, Question #7, Answer #2



7) I am invoking my swap function, but I get multiple errors. The first error is:
error: no matching function for call to 'swap(int*, int*)'

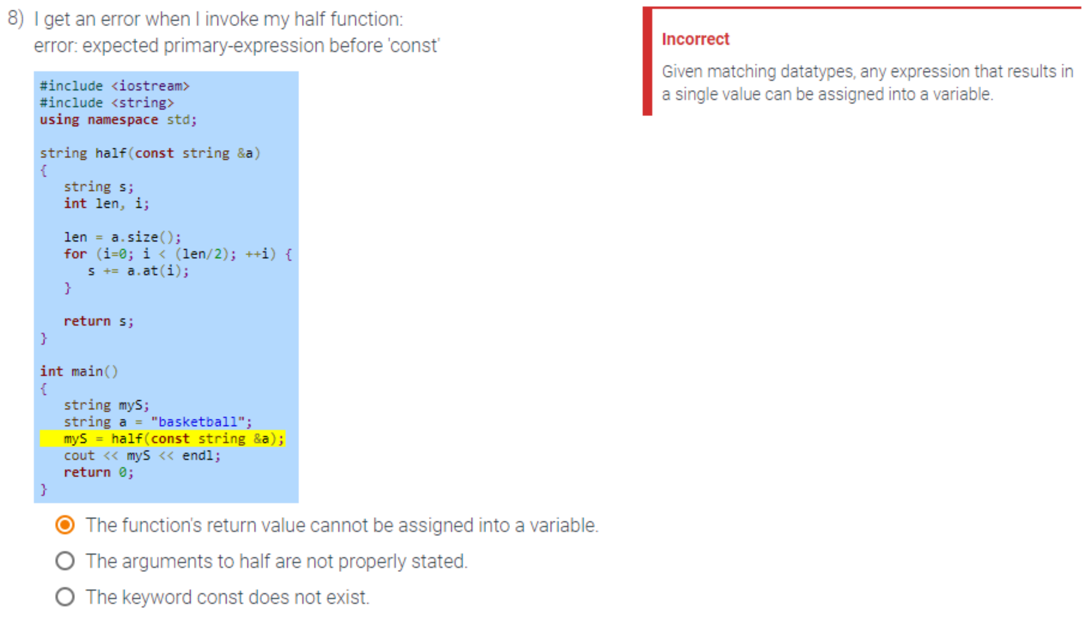
```
#include <iostream>
using namespace std;

void swap(int &x, int &y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}

int main()
{
    int x = 5, y = 7;
    cout << x << ' ' << y;
    cout << endl;
    swap(&x, &y);
    cout << x << ' ' << y;
    cout << endl;
    return 0;
}
```

- The ampersand for arguments is incorrect, use * as the error suggests.
- The swap function already exist, you cannot define the function again.
- [✓ The type of arguments provided to swap are incorrect.]

Correct

Swap takes two references to integers, using & changes the type. The proper arguments should just be integers, simply use the variable names: swap(x, y);

Figure 158: Function Invocation, Question #7, Answer #3



8) I get an error when I invoke my half function:
error: expected primary-expression before 'const'

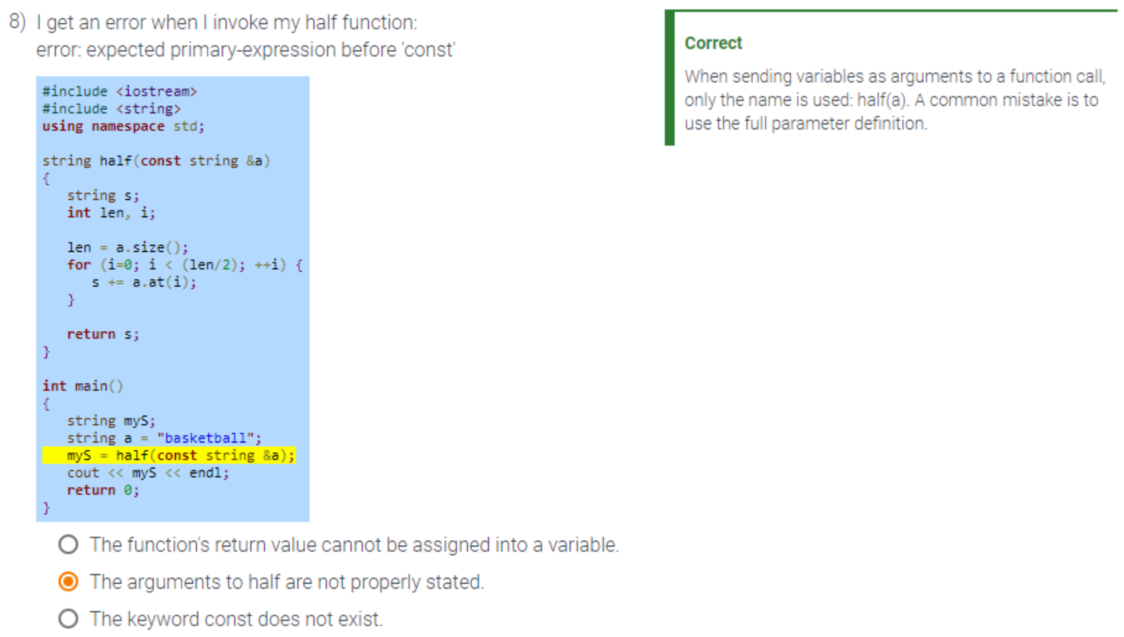
```
#include <iostream>
#include <string>
using namespace std;

string half(const string &a)
{
    string s;
    int len, i;

    len = a.size();
    for (i=0; i < (len/2); ++i) {
        s += a.at(i);
    }

    return s;
}

int main()
{
    string myS;
    string a = "basketball";
    myS = half(const string &a);
    cout << myS << endl;
    return 0;
}
```

- [✓ The function's return value cannot be assigned into a variable.]
- The arguments to half are not properly stated.
- The keyword const does not exist.

Incorrect

Given matching datatypes, any expression that results in a single value can be assigned into a variable.

Figure 159: Function Invocation, Question #8, Answer #1



8) I get an error when I invoke my half function:
error: expected primary-expression before 'const'

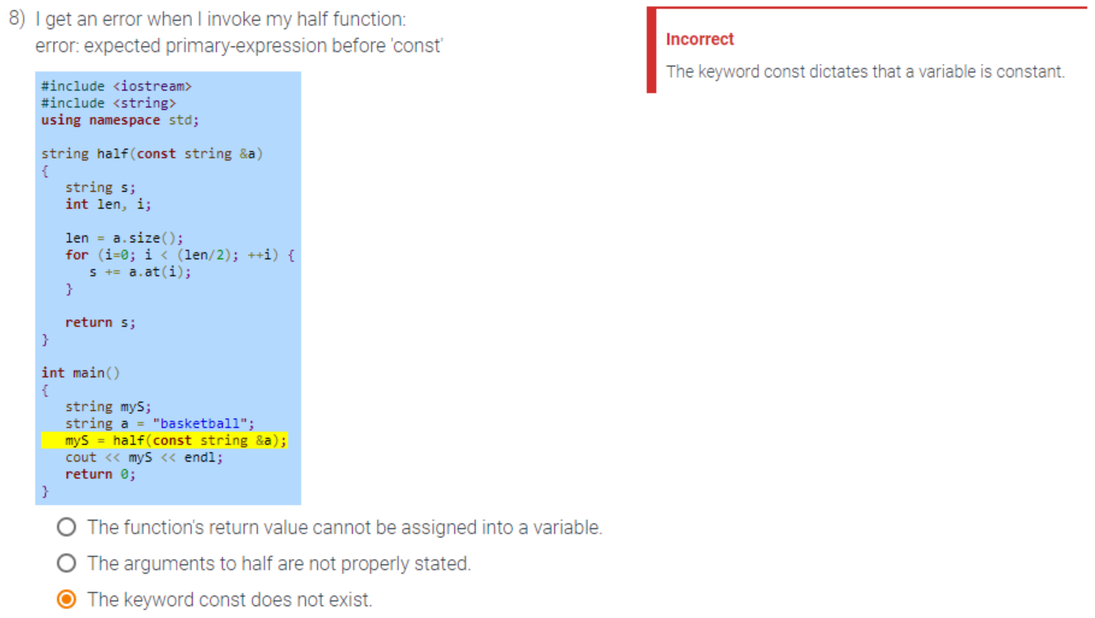
```
#include <iostream>
#include <string>
using namespace std;

string half(const string &a)
{
   string s;
   int len, i;

   len = a.size();
   for (i=0; i < (len/2); ++i) {
      s += a.at(i);
   }

   return s;
}

int main()
{
   string myS;
   string a = "basketball";
   myS = half(const string &a);
   cout << myS << endl;
   return 0;
}
```

- ○ The function's return value cannot be assigned into a variable.
- ◉ The arguments to half are not properly stated.
- ○ The keyword const does not exist.

Correct

When sending variables as arguments to a function call, only the name is used: half(a). A common mistake is to use the full parameter definition.

Figure 160: Function Invocation, Question #8, Answer #2



8) I get an error when I invoke my half function:
error: expected primary-expression before 'const'

```
#include <iostream>
#include <string>
using namespace std;

string half(const string &a)
{
   string s;
   int len, i;

   len = a.size();
   for (i=0; i < (len/2); ++i) {
      s += a.at(i);
   }

   return s;
}

int main()
{
   string myS;
   string a = "basketball";
   myS = half(const string &a);
   cout << myS << endl;
   return 0;
}
```

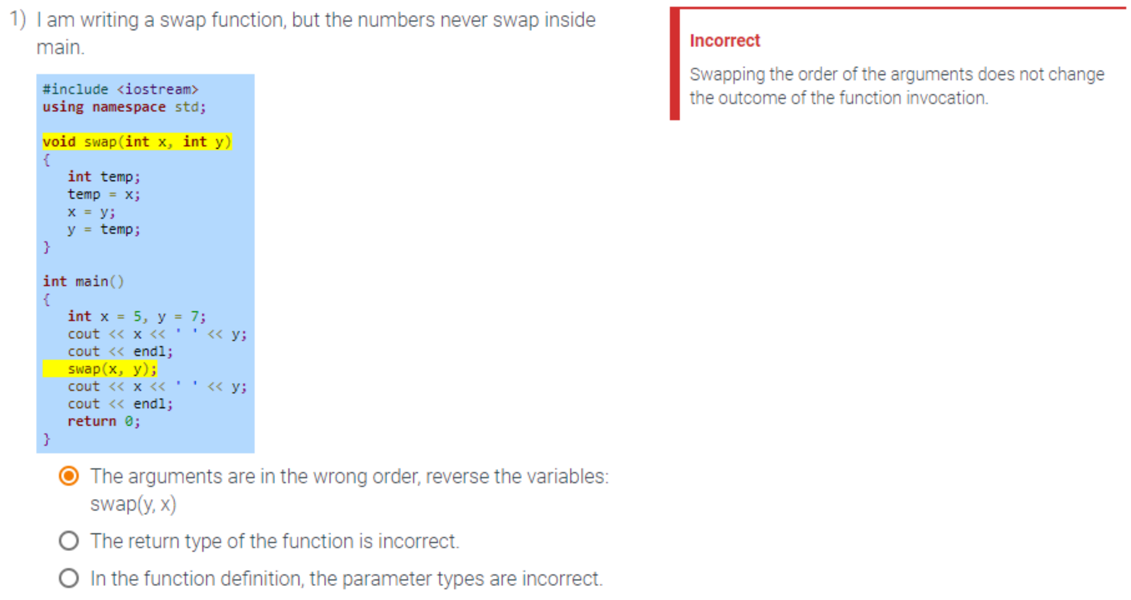
- ○ The function's return value cannot be assigned into a variable.
- ○ The arguments to half are not properly stated.
- ◉ The keyword const does not exist.

Incorrect

The keyword const dictates that a variable is constant.

Figure 161: Function Invocation, Question #8, Answer #3

Function Writing Exercises



1) I am writing a swap function, but the numbers never swap inside main.

```
#include <iostream>
using namespace std;

void swap(int x, int y)
{
   int temp;
   temp = x;
   x = y;
   y = temp;
}

int main()
{
   int x = 5, y = 7;
   cout << x << ' ' << y;
   cout << endl;
   swap(x, y);
   cout << x << ' ' << y;
   cout << endl;
   return 0;
}
```

[✓ The arguments are in the wrong order, reverse the variables: swap(y, x)]

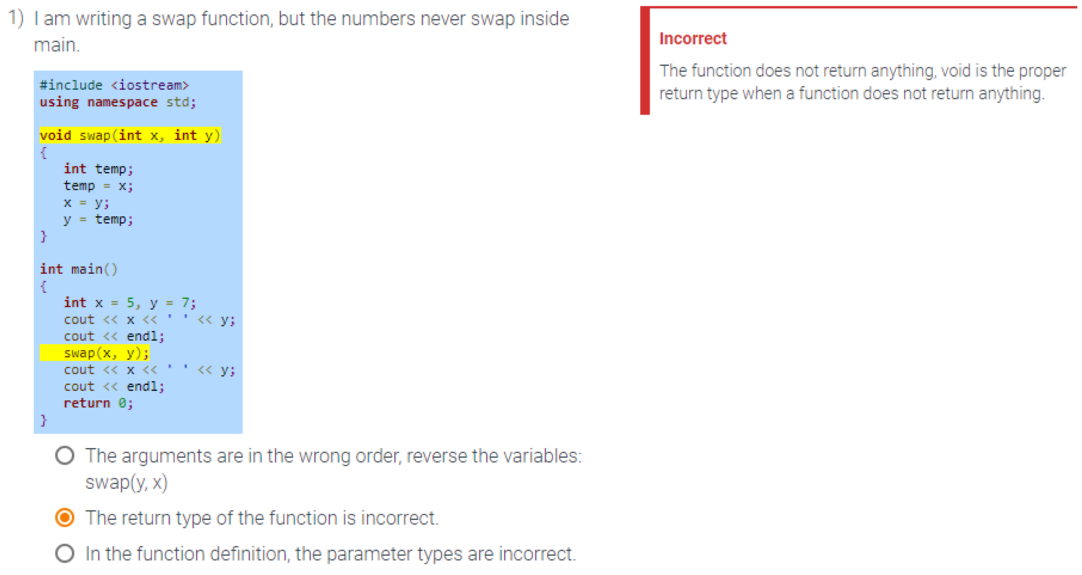
The return type of the function is incorrect.

In the function definition, the parameter types are incorrect.

Incorrect

Swapping the order of the arguments does not change the outcome of the function invocation.

Figure 162: Function Writing, Question #1, Answer #1



1) I am writing a swap function, but the numbers never swap inside main.

```
#include <iostream>
using namespace std;

void swap(int x, int y)
{
   int temp;
   temp = x;
   x = y;
   y = temp;
}

int main()
{
   int x = 5, y = 7;
   cout << x << ' ' << y;
   cout << endl;
   swap(x, y);
   cout << x << ' ' << y;
   cout << endl;
   return 0;
}
```

The arguments are in the wrong order, reverse the variables: swap(y, x)

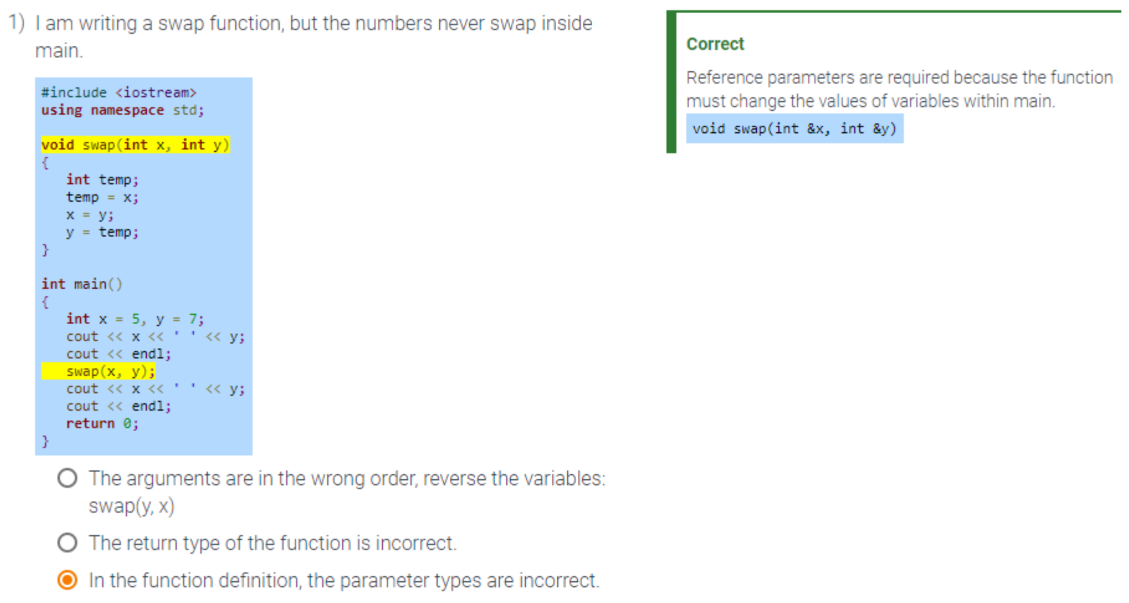
[✓ The return type of the function is incorrect.]

In the function definition, the parameter types are incorrect.

Incorrect

The function does not return anything, void is the proper return type when a function does not return anything.

Figure 163: Function Writing, Question #1, Answer #2



1) I am writing a swap function, but the numbers never swap inside main.

```
#include <iostream>
using namespace std;

void swap(int x, int y)
{
   int temp;
   temp = x;
   x = y;
   y = temp;
}

int main()
{
   int x = 5, y = 7;
   cout << x << ' ' << y;
   cout << endl;
   swap(x, y);
   cout << x << ' ' << y;
   cout << endl;
   return 0;
}
```

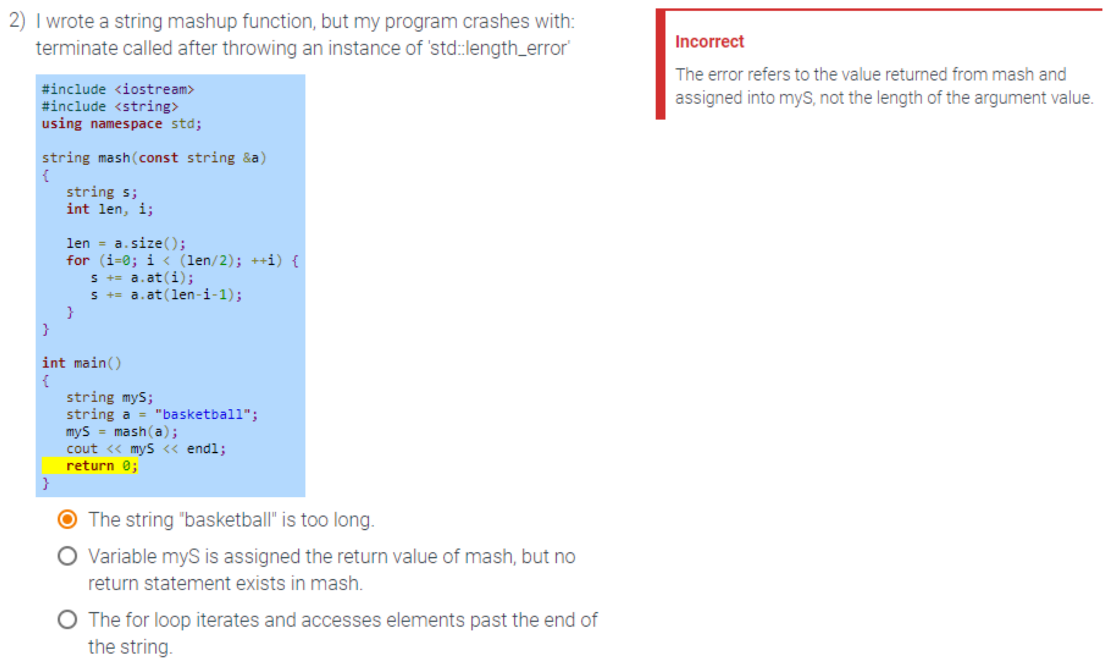
- The arguments are in the wrong order, reverse the variables: swap(y, x)
- The return type of the function is incorrect.
- [✓ In the function definition, the parameter types are incorrect.]

Correct

Reference parameters are required because the function must change the values of variables within main.

```
void swap(int &x, int &y)
```

Figure 164: Function Writing, Question #1, Answer #1



2) I wrote a string mashup function, but my program crashes with: terminate called after throwing an instance of 'std::length_error'

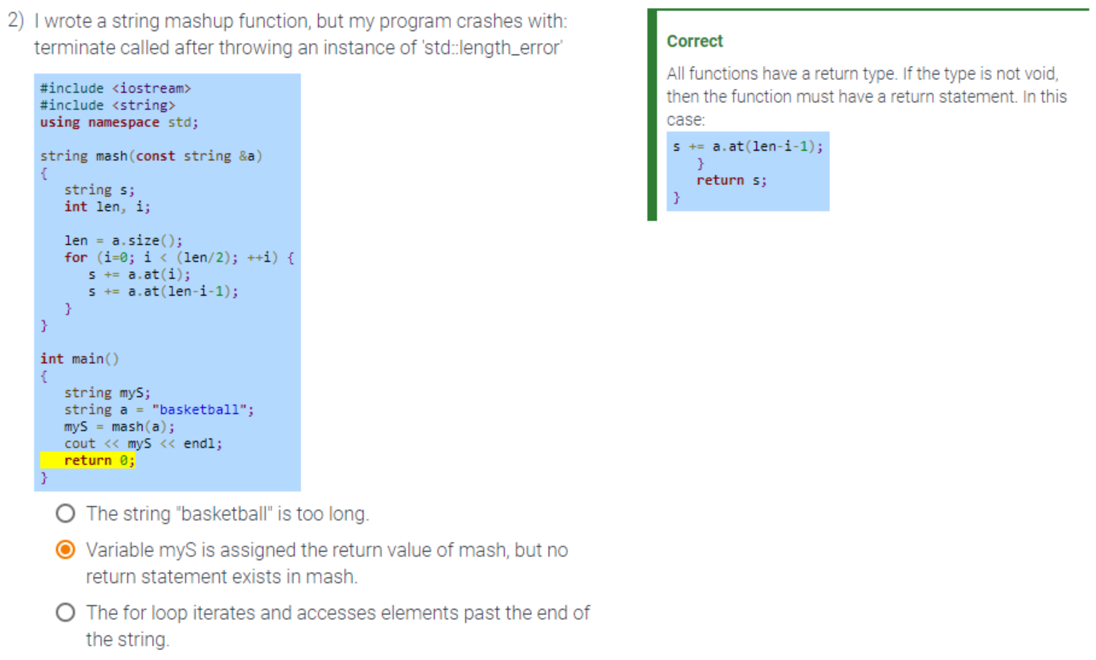
```
#include <iostream>
#include <string>
using namespace std;

string mash(const string &a)
{
   string s;
   int len, i;

   len = a.size();
   for (i=0; i < (len/2); ++i) {
      s += a.at(i);
      s += a.at(len-i-1);
   }
}

int main()
{
   string myS;
   string a = "basketball";
   myS = mash(a);
   cout << myS << endl;
   return 0;
}
```

- [✓ The string "basketball" is too long.]
- Variable myS is assigned the return value of mash, but no return statement exists in mash.
- The for loop iterates and accesses elements past the end of the string.

Incorrect

The error refers to the value returned from mash and assigned into myS, not the length of the argument value.

Figure 165: Function Writing, Question #2, Answer #1



2) I wrote a string mashup function, but my program crashes with: terminate called after throwing an instance of 'std::length_error'

```
#include <iostream>
#include <string>
using namespace std;

string mash(const string &a)
{
   string s;
   int len, i;

   len = a.size();
   for (i=0; i < (len/2); ++i) {
      s += a.at(i);
      s += a.at(len-i-1);
   }
}

int main()
{
   string myS;
   string a = "basketball";
   myS = mash(a);
   cout << myS << endl;
   return 0;
}
```

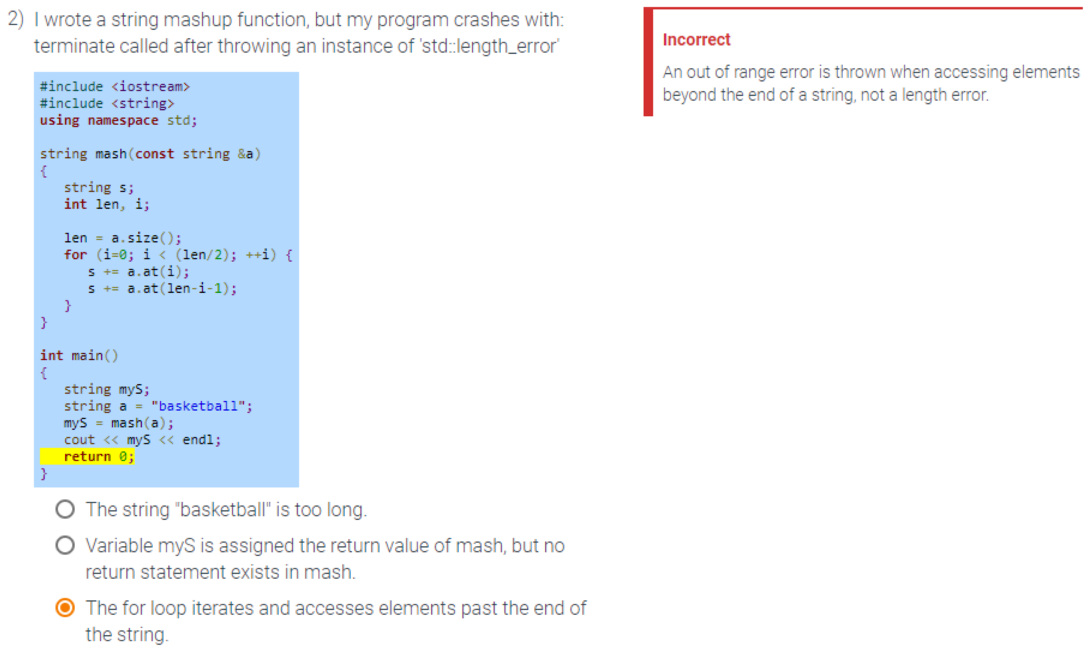
- The string "basketball" is too long.
- [✓ Variable myS is assigned the return value of mash, but no return statement exists in mash.]
- The for loop iterates and accesses elements past the end of the string.

Correct

All functions have a return type. If the type is not void, then the function must have a return statement. In this case:

```
s += a.at(len-i-1);
   }
   return s;
}
```

Figure 166: Function Writing, Question #2, Answer #2



2) I wrote a string mashup function, but my program crashes with: terminate called after throwing an instance of 'std::length_error'

```
#include <iostream>
#include <string>
using namespace std;

string mash(const string &a)
{
   string s;
   int len, i;

   len = a.size();
   for (i=0; i < (len/2); ++i) {
      s += a.at(i);
      s += a.at(len-i-1);
   }
}

int main()
{
   string myS;
   string a = "basketball";
   myS = mash(a);
   cout << myS << endl;
   return 0;
}
```

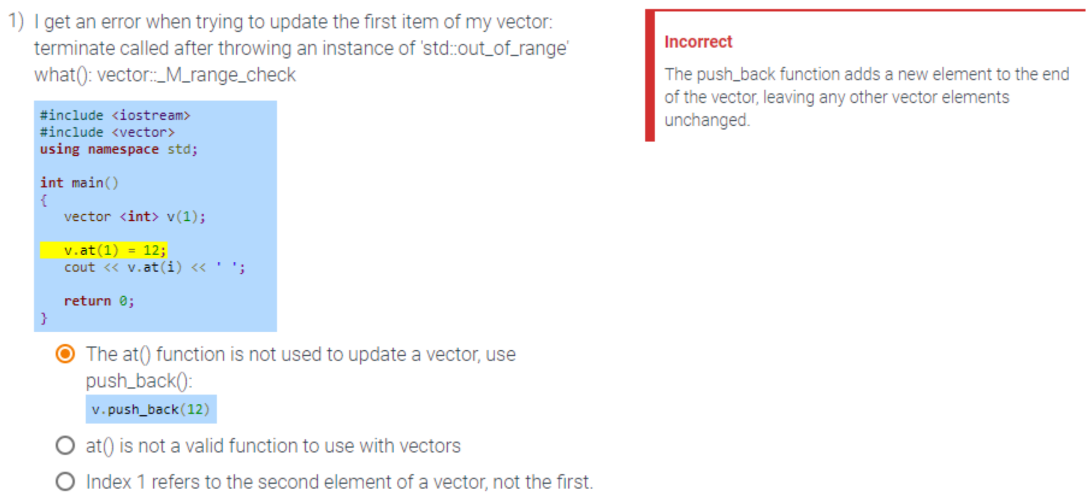
- The string "basketball" is too long.
- Variable myS is assigned the return value of mash, but no return statement exists in mash.
- [✓ The for loop iterates and accesses elements past the end of the string.]

Incorrect

An out of range error is thrown when accessing elements beyond the end of a string, not a length error.

Figure 167: Function Writing, Question #2, Answer #3

Vector Exercises



1) I get an error when trying to update the first item of my vector:
terminate called after throwing an instance of 'std::out_of_range'
what(): vector::_M_range_check

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
   vector <int> v(1);

   v.at(1) = 12;
   cout << v.at(i) << ' ';

   return 0;
}
```

[✓ The at() function is not used to update a vector, use push_back():]

```
v.push_back(12)
```

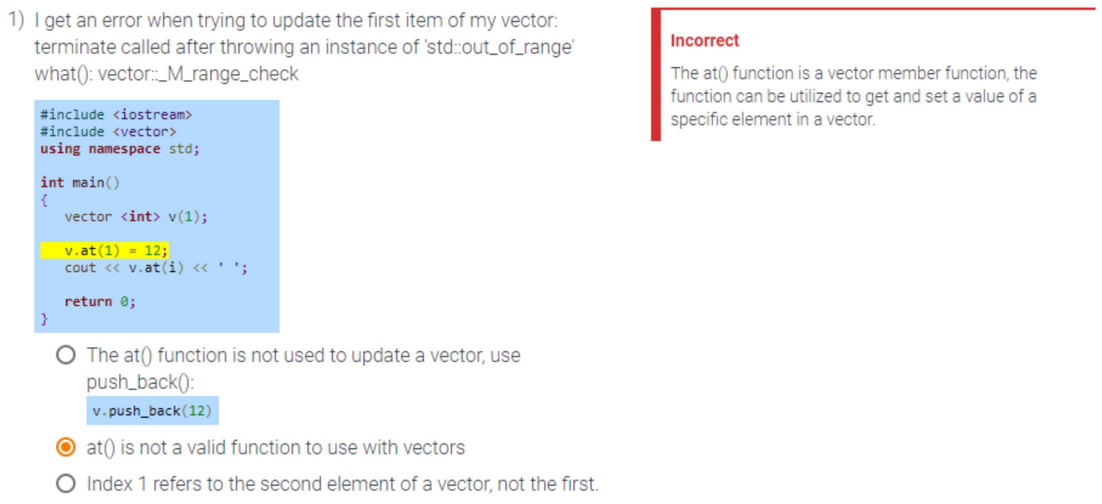
at() is not a valid function to use with vectors

Index 1 refers to the second element of a vector, not the first.

Incorrect

The push_back function adds a new element to the end of the vector, leaving any other vector elements unchanged.

Figure 168: Vectors, Question #1, Answer #1



1) I get an error when trying to update the first item of my vector:
terminate called after throwing an instance of 'std::out_of_range'
what(): vector::_M_range_check

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
   vector <int> v(1);

   v.at(1) = 12;
   cout << v.at(i) << ' ';

   return 0;
}
```

The at() function is not used to update a vector, use push_back():

```
v.push_back(12)
```

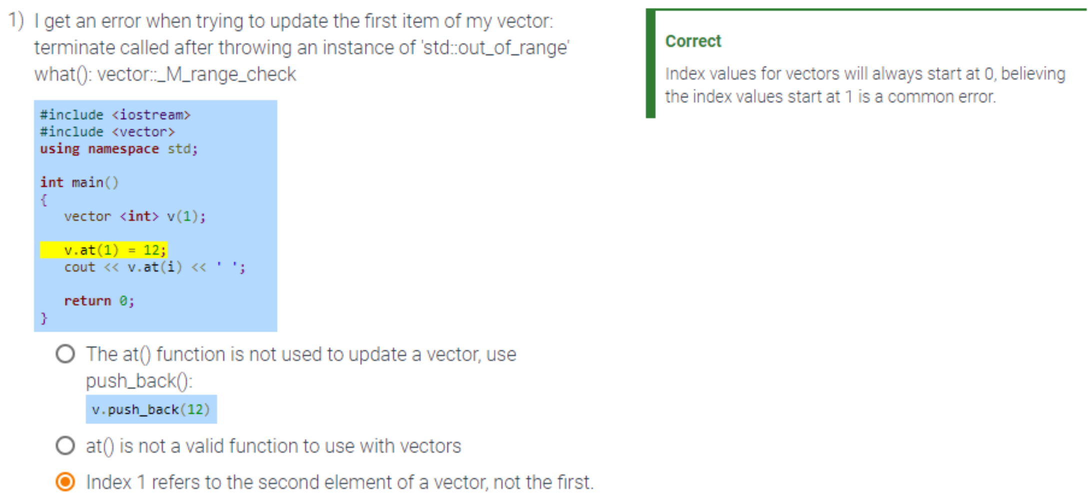
[✓ at() is not a valid function to use with vectors]

Index 1 refers to the second element of a vector, not the first.

Incorrect

The at() function is a vector member function, the function can be utilized to get and set a value of a specific element in a vector.

Figure 169: Vectors, Question #1, Answer #2


1) I get an error when trying to update the first item of my vector:
terminate called after throwing an instance of 'std::out_of_range'
what(): vector::_M_range_check

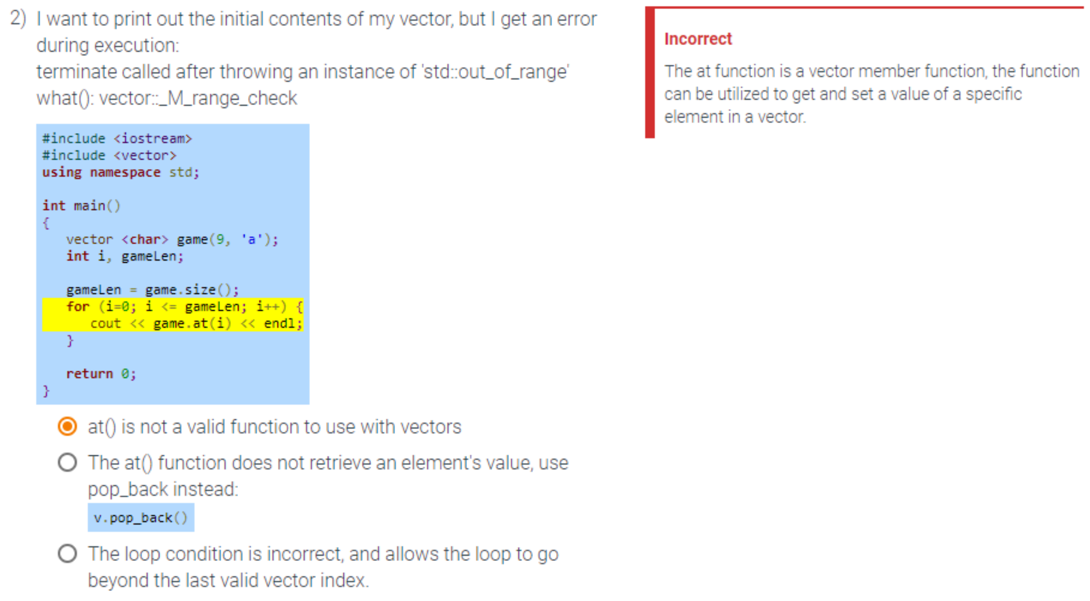
```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
   vector <int> v(1);

   v.at(1) = 12;
   cout << v.at(i) << ' ';

   return 0;
}
```

- The at() function is not used to update a vector, use push_back():
  `v.push_back(12)`
- at() is not a valid function to use with vectors
- [✓ Index 1 refers to the second element of a vector, not the first.]

Correct

Index values for vectors will always start at 0, believing the index values start at 1 is a common error.

Figure 170: Vectors, Question #1, Answer #3


2) I want to print out the initial contents of my vector, but I get an error during execution:
terminate called after throwing an instance of 'std::out_of_range'
what(): vector::_M_range_check

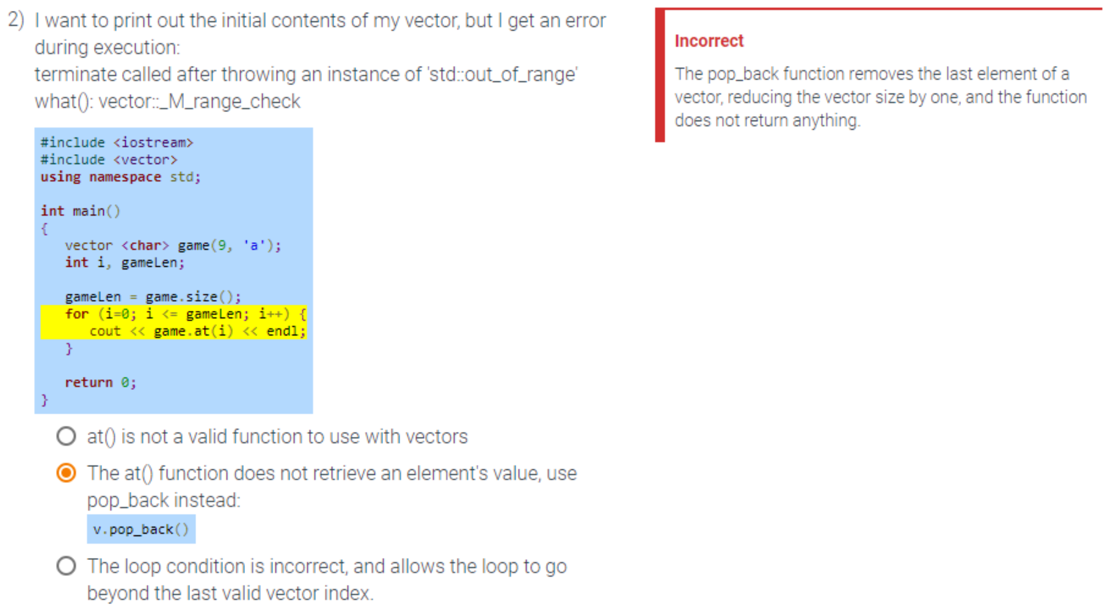
```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
   vector <char> game(9, 'a');
   int i, gameLen;

   gameLen = game.size();
   for (i=0; i <= gameLen; i++) {
      cout << game.at(i) << endl;
   }

   return 0;
}
```

- [✓ at() is not a valid function to use with vectors]
- The at() function does not retrieve an element's value, use pop_back instead:
  `v.pop_back()`
- The loop condition is incorrect, and allows the loop to go beyond the last valid vector index.

Incorrect

The at function is a vector member function, the function can be utilized to get and set a value of a specific element in a vector.

Figure 171: Vectors, Question #2, Answer #1


2) I want to print out the initial contents of my vector, but I get an error during execution:
terminate called after throwing an instance of 'std::out_of_range'
what(): vector::_M_range_check

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
   vector <char> game(9, 'a');
   int i, gameLen;

   gameLen = game.size();
   for (i=0; i <= gameLen; i++) {
      cout << game.at(i) << endl;
   }

   return 0;
}
```

○ at() is not a valid function to use with vectors

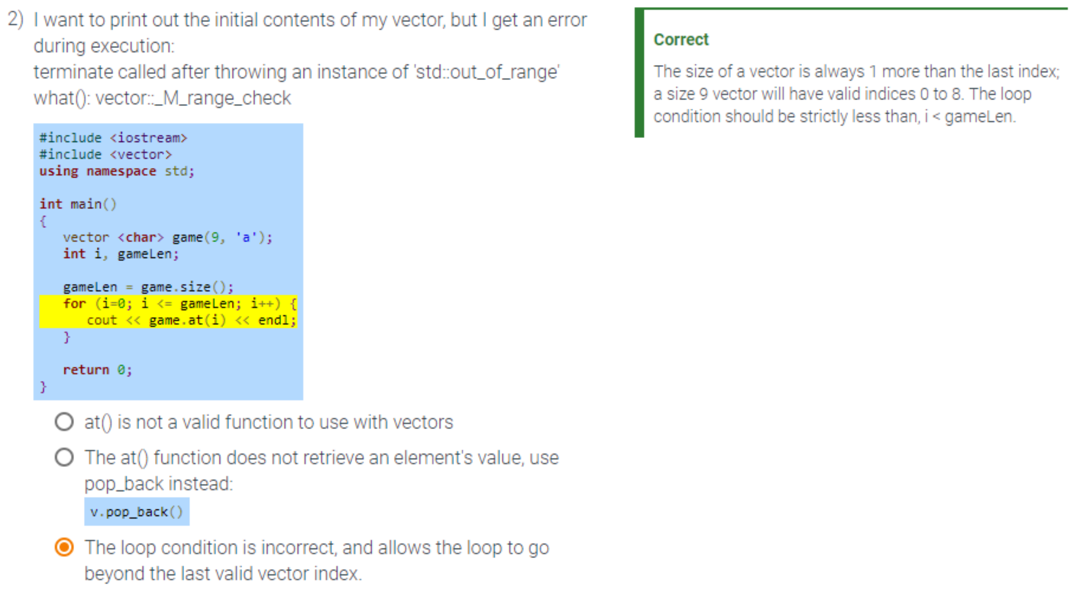
● The at() function does not retrieve an element's value, use pop_back instead:
`v.pop_back()`

○ The loop condition is incorrect, and allows the loop to go beyond the last valid vector index.

Incorrect

The pop_back function removes the last element of a vector, reducing the vector size by one, and the function does not return anything.

Figure 172: Vectors, Question #2, Answer #2


2) I want to print out the initial contents of my vector, but I get an error during execution:
terminate called after throwing an instance of 'std::out_of_range'
what(): vector::_M_range_check

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
   vector <char> game(9, 'a');
   int i, gameLen;

   gameLen = game.size();
   for (i=0; i <= gameLen; i++) {
      cout << game.at(i) << endl;
   }

   return 0;
}
```

○ at() is not a valid function to use with vectors

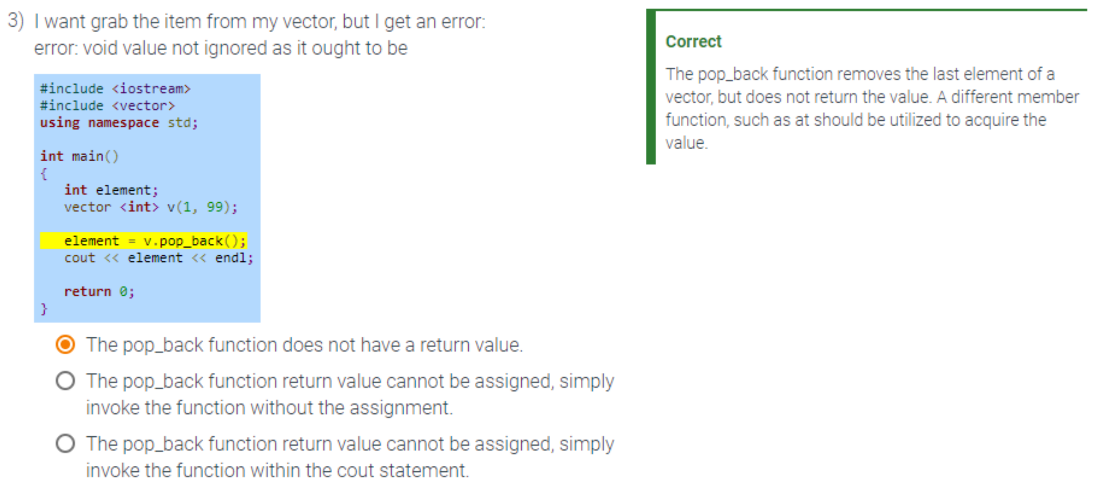
○ The at() function does not retrieve an element's value, use pop_back instead:
`v.pop_back()`

● The loop condition is incorrect, and allows the loop to go beyond the last valid vector index.

Correct

The size of a vector is always 1 more than the last index; a size 9 vector will have valid indices 0 to 8. The loop condition should be strictly less than, i < gameLen.

Figure 173: Vectors, Question #2, Answer #3


3) I want grab the item from my vector, but I get an error:
error: void value not ignored as it ought to be

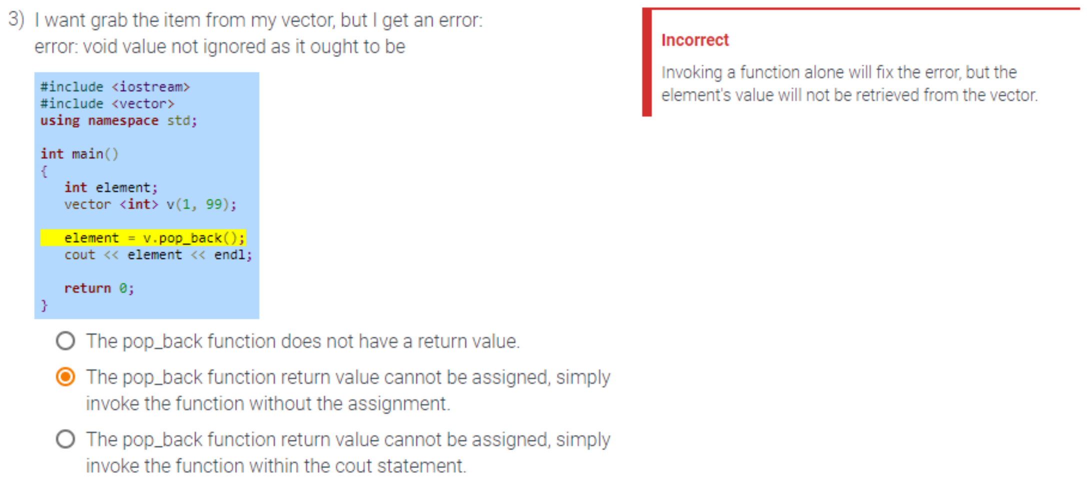
```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    int element;
    vector <int> v(1, 99);

    element = v.pop_back();
    cout << element << endl;

    return 0;
}
```

- ● The pop_back function does not have a return value.
- ○ The pop_back function return value cannot be assigned, simply invoke the function without the assignment.
- ○ The pop_back function return value cannot be assigned, simply invoke the function within the cout statement.

Correct

The pop_back function removes the last element of a vector, but does not return the value. A different member function, such as at should be utilized to acquire the value.

Figure 174: Vectors, Question #3, Answer #1


3) I want grab the item from my vector, but I get an error:
error: void value not ignored as it ought to be

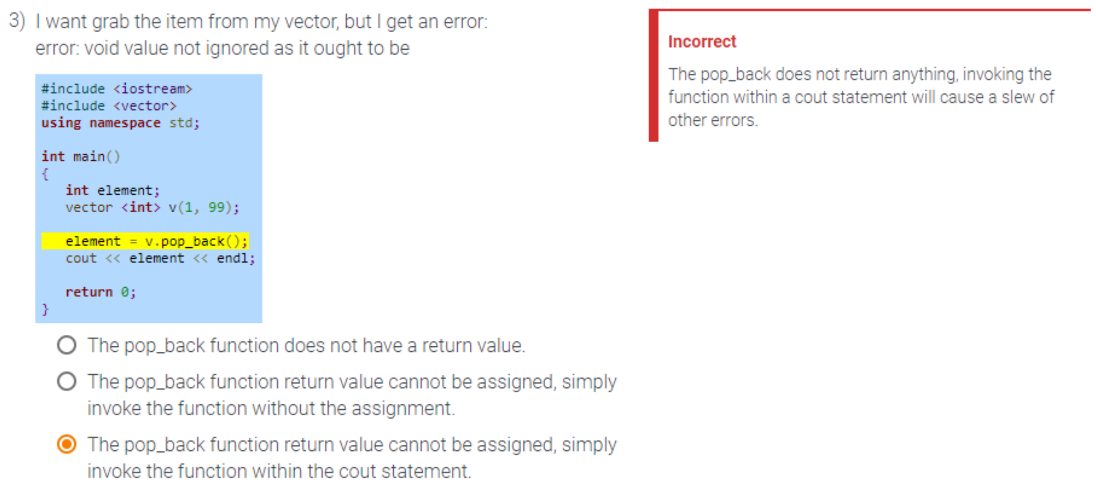
```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    int element;
    vector <int> v(1, 99);

    element = v.pop_back();
    cout << element << endl;

    return 0;
}
```

- ○ The pop_back function does not have a return value.
- ● The pop_back function return value cannot be assigned, simply invoke the function without the assignment.
- ○ The pop_back function return value cannot be assigned, simply invoke the function within the cout statement.

Incorrect

Invoking a function alone will fix the error, but the element's value will not be retrieved from the vector.

Figure 175: Vectors, Question #3, Answer #2



3) I want grab the item from my vector, but I get an error:
error: void value not ignored as it ought to be

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
   int element;
   vector <int> v(1, 99);

   element = v.pop_back();
   cout << element << endl;

   return 0;
}
```

○ The pop_back function does not have a return value.

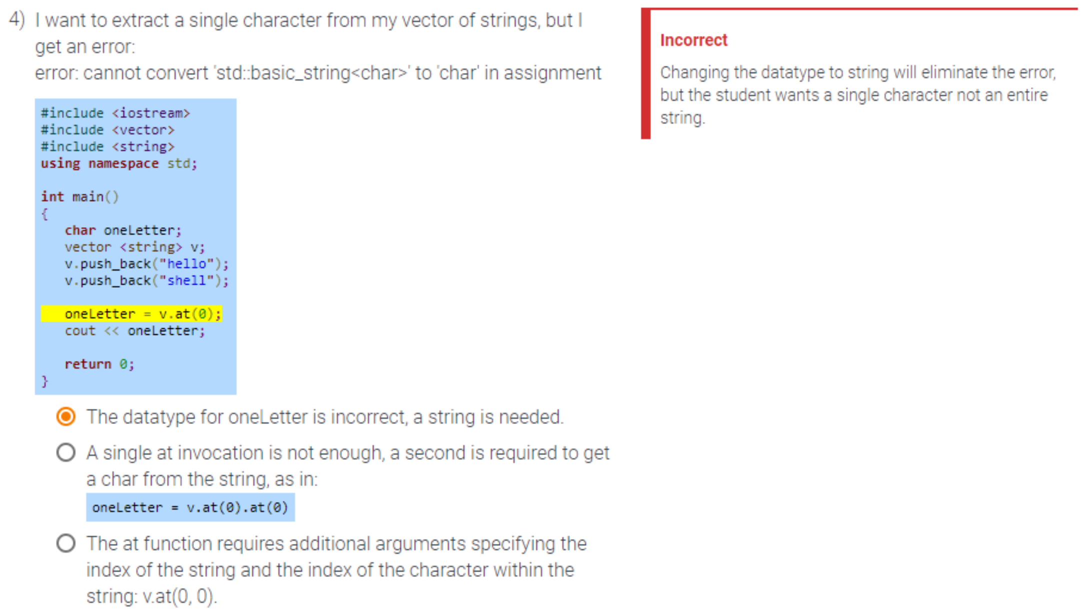
○ The pop_back function return value cannot be assigned, simply invoke the function without the assignment.

● The pop_back function return value cannot be assigned, simply invoke the function within the cout statement.

Incorrect

The pop_back does not return anything, invoking the function within a cout statement will cause a slew of other errors.

Figure 176: Vectors, Question #3, Answer #3



4) I want to extract a single character from my vector of strings, but I get an error:
error: cannot convert 'std::basic_string<char>' to 'char' in assignment

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

int main()
{
   char oneLetter;
   vector <string> v;
   v.push_back("hello");
   v.push_back("shell");

   oneLetter = v.at(0);
   cout << oneLetter;

   return 0;
}
```

● The datatype for oneLetter is incorrect, a string is needed.

○ A single at invocation is not enough, a second is required to get a char from the string, as in:

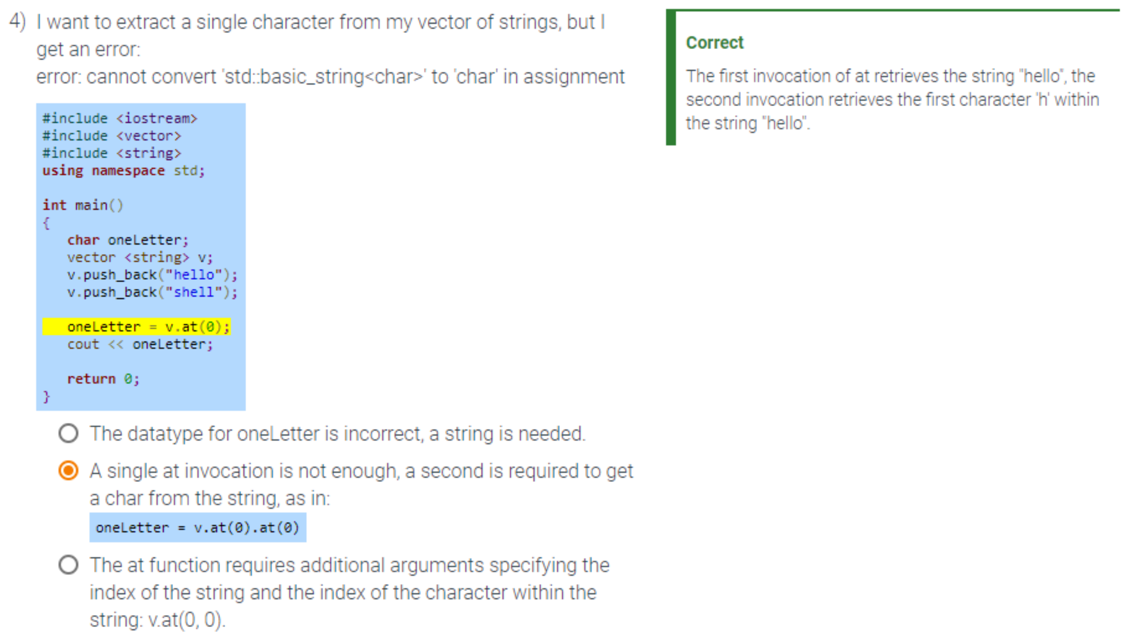
```
oneLetter = v.at(0).at(0)
```

○ The at function requires additional arguments specifying the index of the string and the index of the character within the string: v.at(0, 0).

Incorrect

Changing the datatype to string will eliminate the error, but the student wants a single character not an entire string.

Figure 177: Vectors, Question #4, Answer #1



4) I want to extract a single character from my vector of strings, but I get an error:
error: cannot convert 'std::basic_string<char>' to 'char' in assignment

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

int main()
{
    char oneLetter;
    vector <string> v;
    v.push_back("hello");
    v.push_back("shell");

    oneLetter = v.at(0);
    cout << oneLetter;

    return 0;
}
```

○ The datatype for oneLetter is incorrect, a string is needed.

● A single at invocation is not enough, a second is required to get a char from the string, as in:

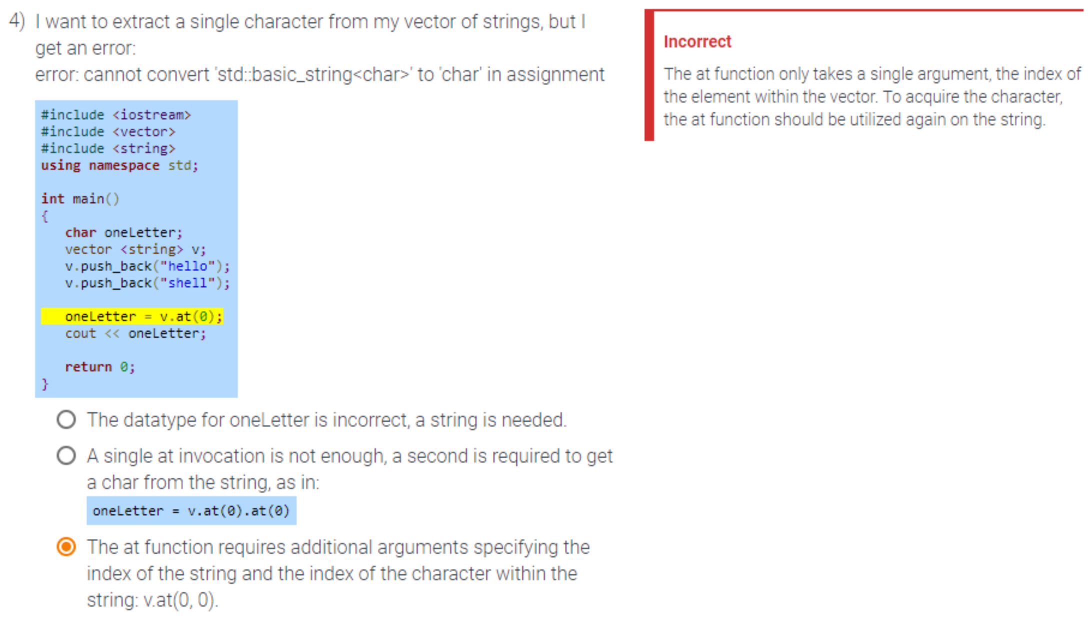
```
oneLetter = v.at(0).at(0)
```

○ The at function requires additional arguments specifying the index of the string and the index of the character within the string: v.at(0, 0).

Correct

The first invocation of at retrieves the string "hello", the second invocation retrieves the first character 'h' within the string "hello".

Figure 178: Vectors, Question #4, Answer #2



4) I want to extract a single character from my vector of strings, but I get an error:
error: cannot convert 'std::basic_string<char>' to 'char' in assignment

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

int main()
{
    char oneLetter;
    vector <string> v;
    v.push_back("hello");
    v.push_back("shell");

    oneLetter = v.at(0);
    cout << oneLetter;

    return 0;
}
```

○ The datatype for oneLetter is incorrect, a string is needed.

○ A single at invocation is not enough, a second is required to get a char from the string, as in:

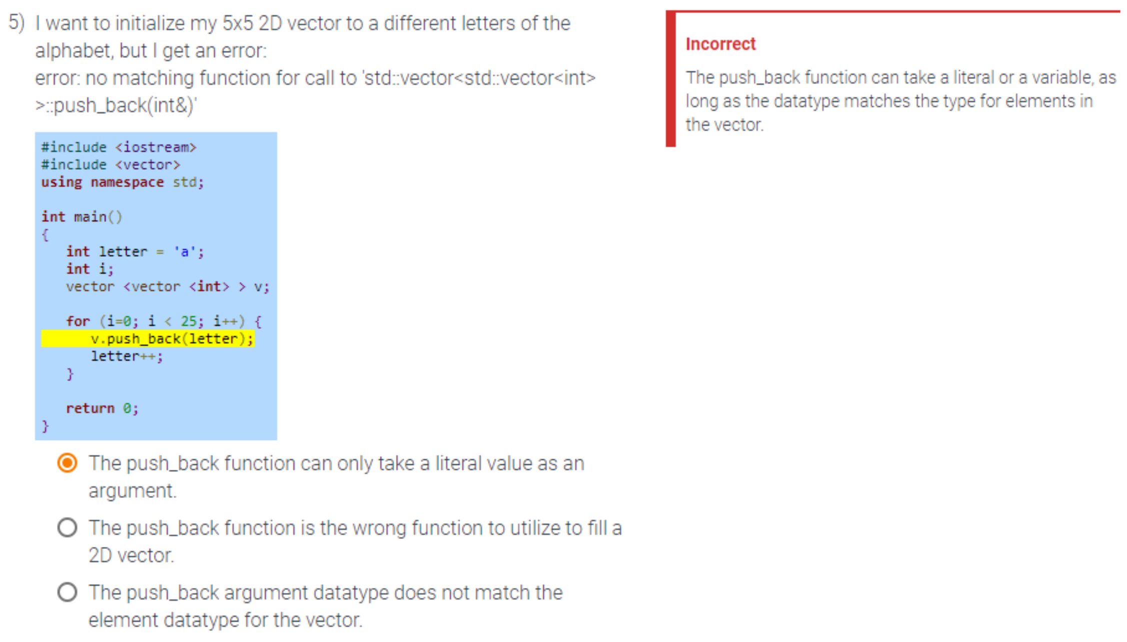
```
oneLetter = v.at(0).at(0)
```

● The at function requires additional arguments specifying the index of the string and the index of the character within the string: v.at(0, 0).

Incorrect

The at function only takes a single argument, the index of the element within the vector. To acquire the character, the at function should be utilized again on the string.

Figure 179: Vectors, Question #4, Answer #3


5) I want to initialize my 5x5 2D vector to a different letters of the alphabet, but I get an error:
error: no matching function for call to 'std::vector<std::vector<int> >::push_back(int&)'

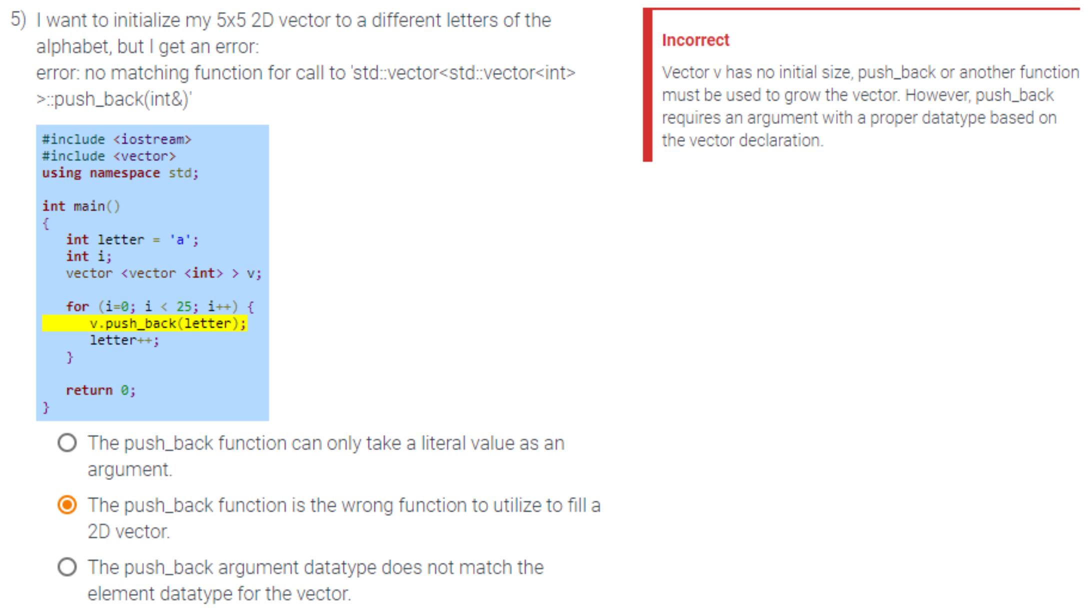
```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
   int letter = 'a';
   int i;
   vector <vector <int> > v;

   for (i=0; i < 25; i++) {
      v.push_back(letter);
      letter++;
   }

   return 0;
}
```

- [✓ The push_back function can only take a literal value as an argument.]
- The push_back function is the wrong function to utilize to fill a 2D vector.
- The push_back argument datatype does not match the element datatype for the vector.

Incorrect

The push_back function can take a literal or a variable, as long as the datatype matches the type for elements in the vector.

Figure 180: Vectors, Question #5, Answer #1


5) I want to initialize my 5x5 2D vector to a different letters of the alphabet, but I get an error:
error: no matching function for call to 'std::vector<std::vector<int> >::push_back(int&)'

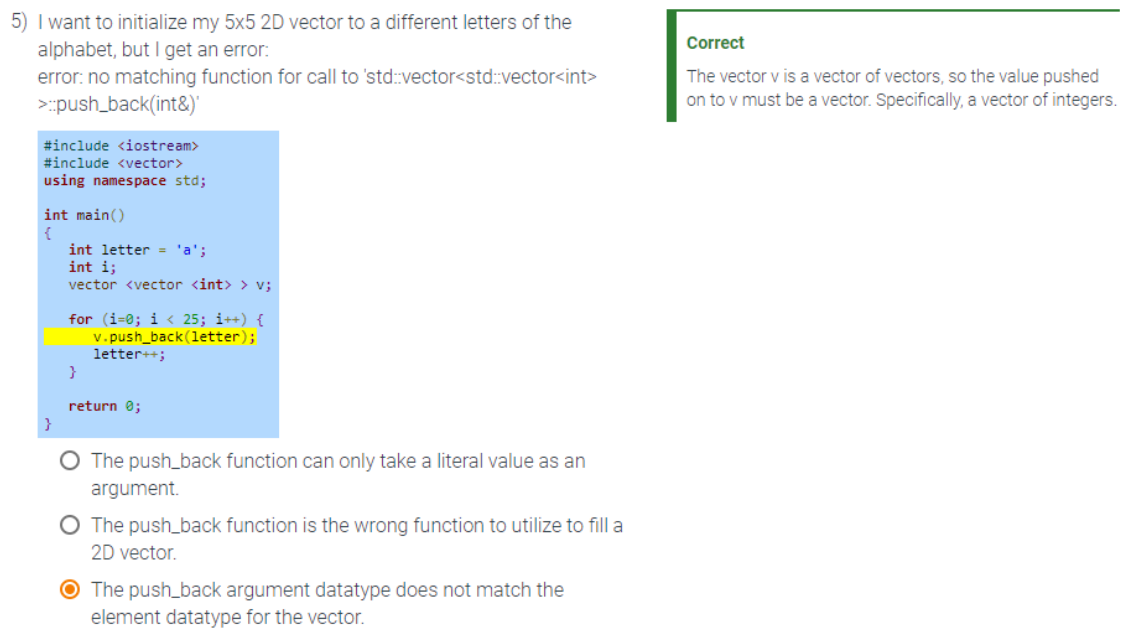
```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
   int letter = 'a';
   int i;
   vector <vector <int> > v;

   for (i=0; i < 25; i++) {
      v.push_back(letter);
      letter++;
   }

   return 0;
}
```

- The push_back function can only take a literal value as an argument.
- [✓ The push_back function is the wrong function to utilize to fill a 2D vector.]
- The push_back argument datatype does not match the element datatype for the vector.

Incorrect

Vector v has no initial size, push_back or another function must be used to grow the vector. However, push_back requires an argument with a proper datatype based on the vector declaration.

Figure 181: Vectors, Question #5, Answer #2


5) I want to initialize my 5x5 2D vector to a different letters of the alphabet, but I get an error:
error: no matching function for call to 'std::vector<std::vector<int> >::push_back(int&)'

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
   int letter = 'a';
   int i;
   vector <vector <int> > v;

   for (i=0; i < 25; i++) {
      v.push_back(letter);
      letter++;
   }

   return 0;
}
```

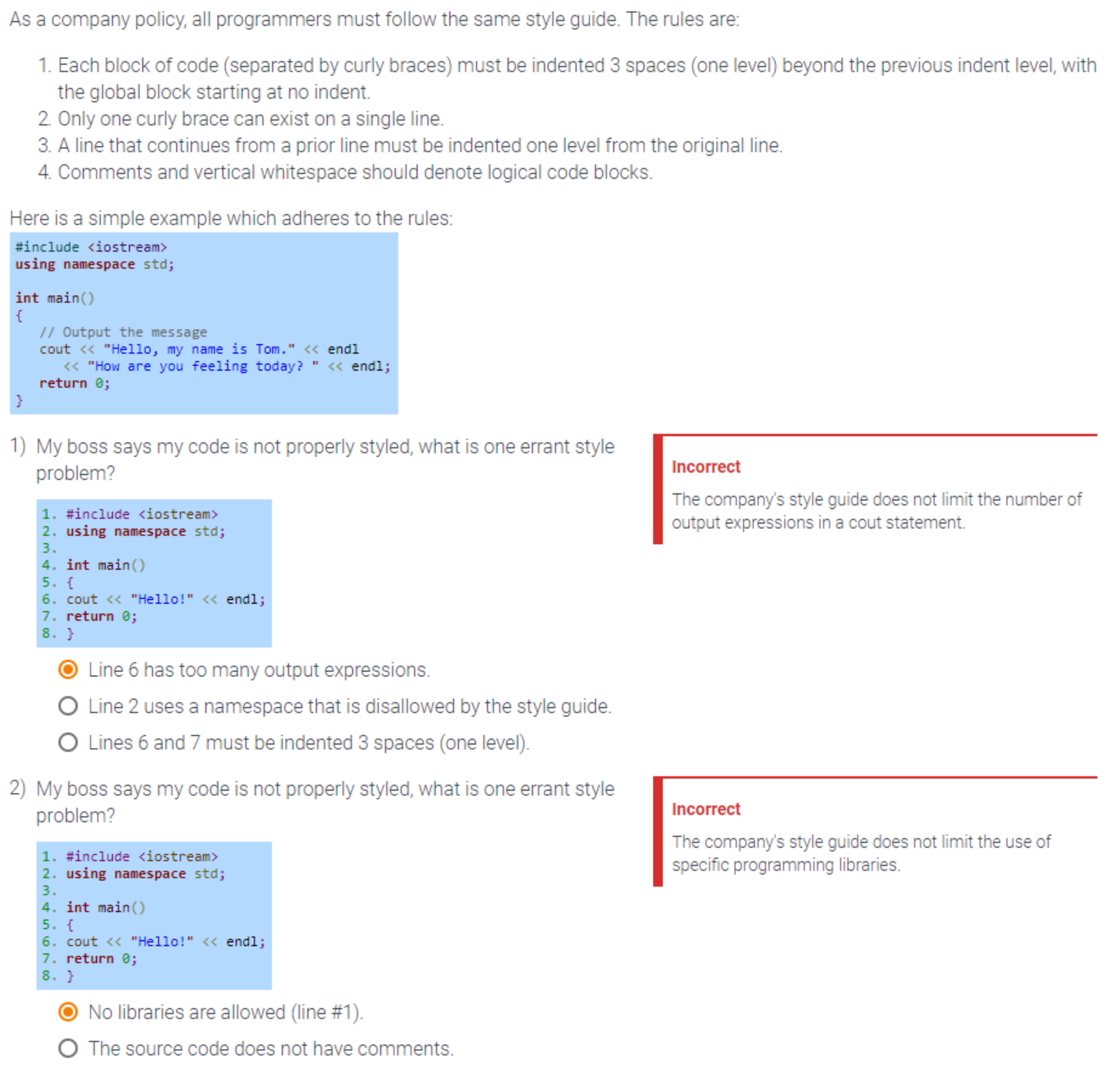
- The push_back function can only take a literal value as an argument.
- The push_back function is the wrong function to utilize to fill a 2D vector.
- [✓ The push_back argument datatype does not match the element datatype for the vector.]

Correct

The vector v is a vector of vectors, so the value pushed on to v must be a vector. Specifically, a vector of integers.

Figure 182: Vectors, Question #5, Answer #3

Appendix E: What's Wrong With My Style Exercises



As a company policy, all programmers must follow the same style guide. The rules are:

1. Each block of code (separated by curly braces) must be indented 3 spaces (one level) beyond the previous indent level, with the global block starting at no indent.
2. Only one curly brace can exist on a single line.
3. A line that continues from a prior line must be indented one level from the original line.
4. Comments and vertical whitespace should denote logical code blocks.

Here is a simple example which adheres to the rules:

```
#include <iostream>
using namespace std;

int main()
{
   // Output the message
   cout << "Hello, my name is Tom." << endl
      << "How are you feeling today? " << endl;
   return 0;
}
```

1) My boss says my code is not properly styled, what is one errant style problem?

```
1. #include <iostream>
2. using namespace std;
3.
4. int main()
5. {
6. cout << "Hello!" << endl;
7. return 0;
8. }
```

- [✓ Line 6 has too many output expressions.]
- Line 2 uses a namespace that is disallowed by the style guide.
- Lines 6 and 7 must be indented 3 spaces (one level).

Incorrect

The company's style guide does not limit the number of output expressions in a cout statement.

2) My boss says my code is not properly styled, what is one errant style problem?

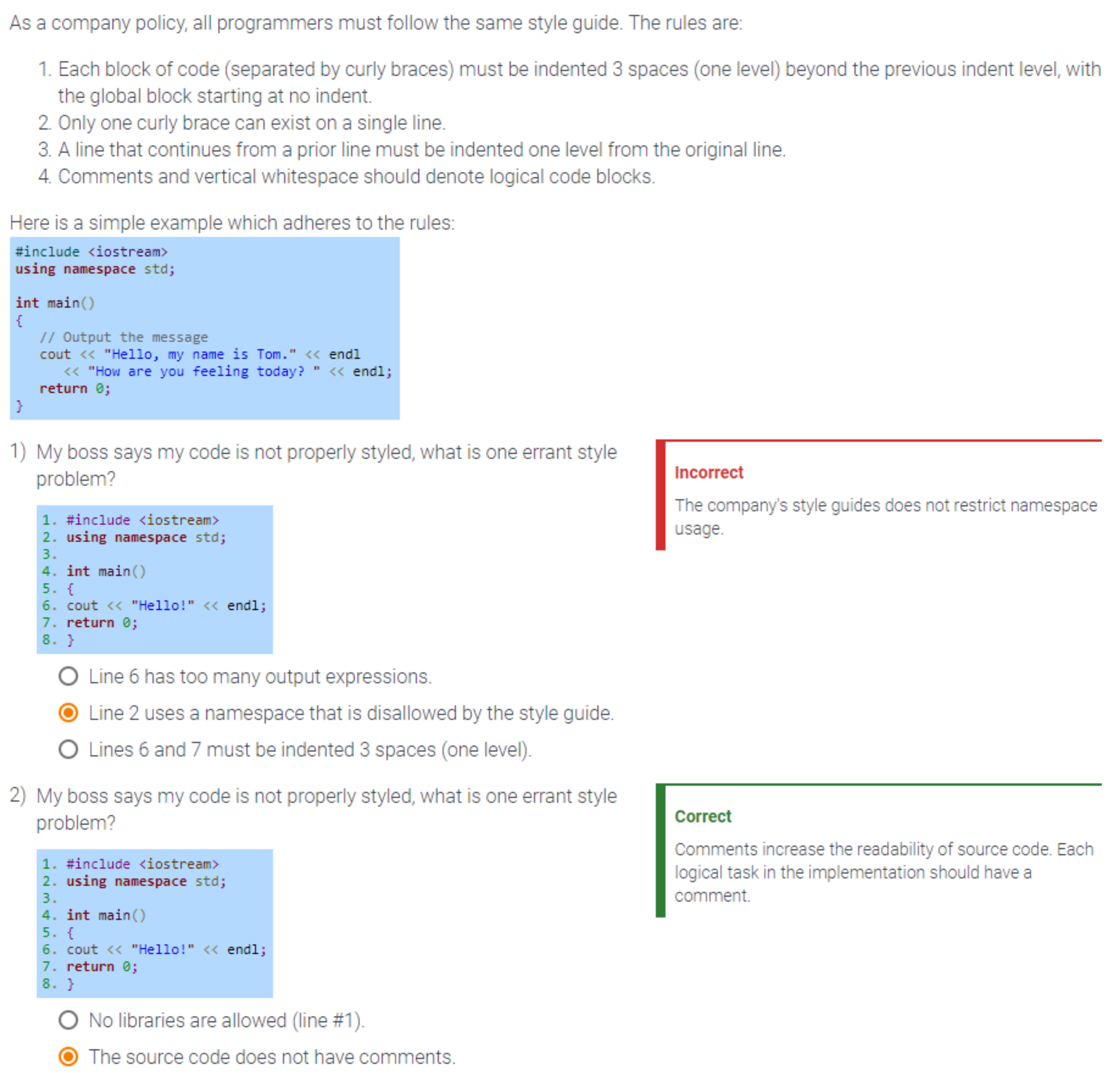
```
1. #include <iostream>
2. using namespace std;
3.
4. int main()
5. {
6. cout << "Hello!" << endl;
7. return 0;
8. }
```

- [✓ No libraries are allowed (line #1).]
- The source code does not have comments.

Incorrect

The company's style guide does not limit the use of specific programming libraries.

Figure 183: Style Exercise #1 and #2 - Basic Style, Answers #1 and #1


As a company policy, all programmers must follow the same style guide. The rules are:

1. Each block of code (separated by curly braces) must be indented 3 spaces (one level) beyond the previous indent level, with the global block starting at no indent.
2. Only one curly brace can exist on a single line.
3. A line that continues from a prior line must be indented one level from the original line.
4. Comments and vertical whitespace should denote logical code blocks.

Here is a simple example which adheres to the rules:

```
#include <iostream>
using namespace std;

int main()
{
   // Output the message
   cout << "Hello, my name is Tom." << endl
      << "How are you feeling today? " << endl;
   return 0;
}
```

1) My boss says my code is not properly styled, what is one errant style problem?

```
1. #include <iostream>
2. using namespace std;
3.
4. int main()
5. {
6. cout << "Hello!" << endl;
7. return 0;
8. }
```

- Line 6 has too many output expressions.
- [✓ Line 2 uses a namespace that is disallowed by the style guide.]
- Lines 6 and 7 must be indented 3 spaces (one level).

Incorrect

The company's style guides does not restrict namespace usage.

2) My boss says my code is not properly styled, what is one errant style problem?

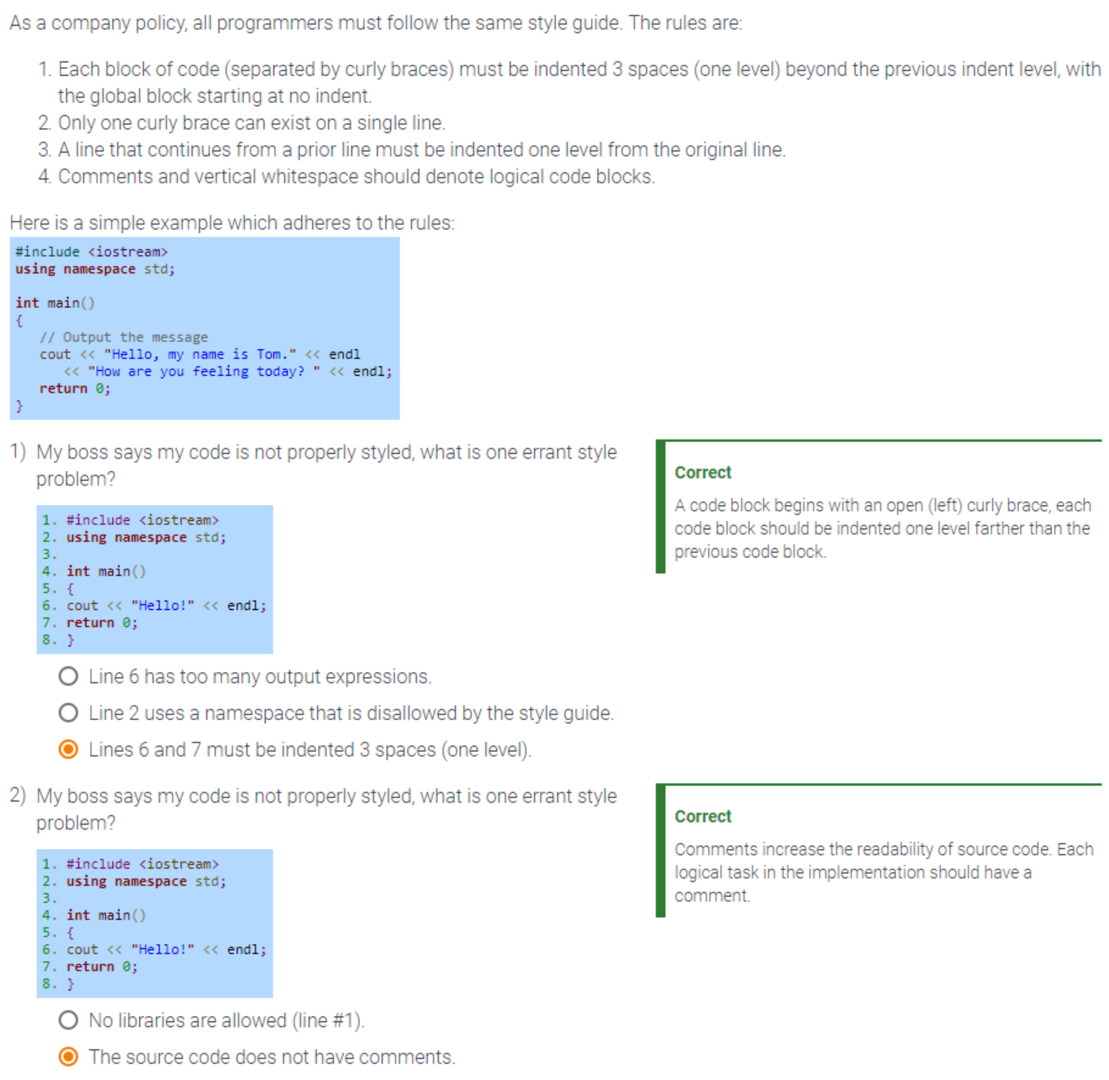
```
1. #include <iostream>
2. using namespace std;
3.
4. int main()
5. {
6. cout << "Hello!" << endl;
7. return 0;
8. }
```

- No libraries are allowed (line #1).
- [✓ The source code does not have comments.]

Correct

Comments increase the readability of source code. Each logical task in the implementation should have a comment.

Figure 184: Style Exercise #1 and #2 - Basic Style, Answers #2 and #2



As a company policy, all programmers must follow the same style guide. The rules are:

1. Each block of code (separated by curly braces) must be indented 3 spaces (one level) beyond the previous indent level, with the global block starting at no indent.
2. Only one curly brace can exist on a single line.
3. A line that continues from a prior line must be indented one level from the original line.
4. Comments and vertical whitespace should denote logical code blocks.

Here is a simple example which adheres to the rules:

```cpp
#include <iostream>
using namespace std;

int main()
{
   // Output the message
   cout << "Hello, my name is Tom." << endl
      << "How are you feeling today? " << endl;
   return 0;
}
```

1) My boss says my code is not properly styled, what is one errant style problem?

```cpp
1. #include <iostream>
2. using namespace std;
3.
4. int main()
5. {
6. cout << "Hello!" << endl;
7. return 0;
8. }
```

- Line 6 has too many output expressions.
- Line 2 uses a namespace that is disallowed by the style guide.
- [✓ Lines 6 and 7 must be indented 3 spaces (one level).]

Correct

A code block begins with an open (left) curly brace, each code block should be indented one level farther than the previous code block.

2) My boss says my code is not properly styled, what is one errant style problem?

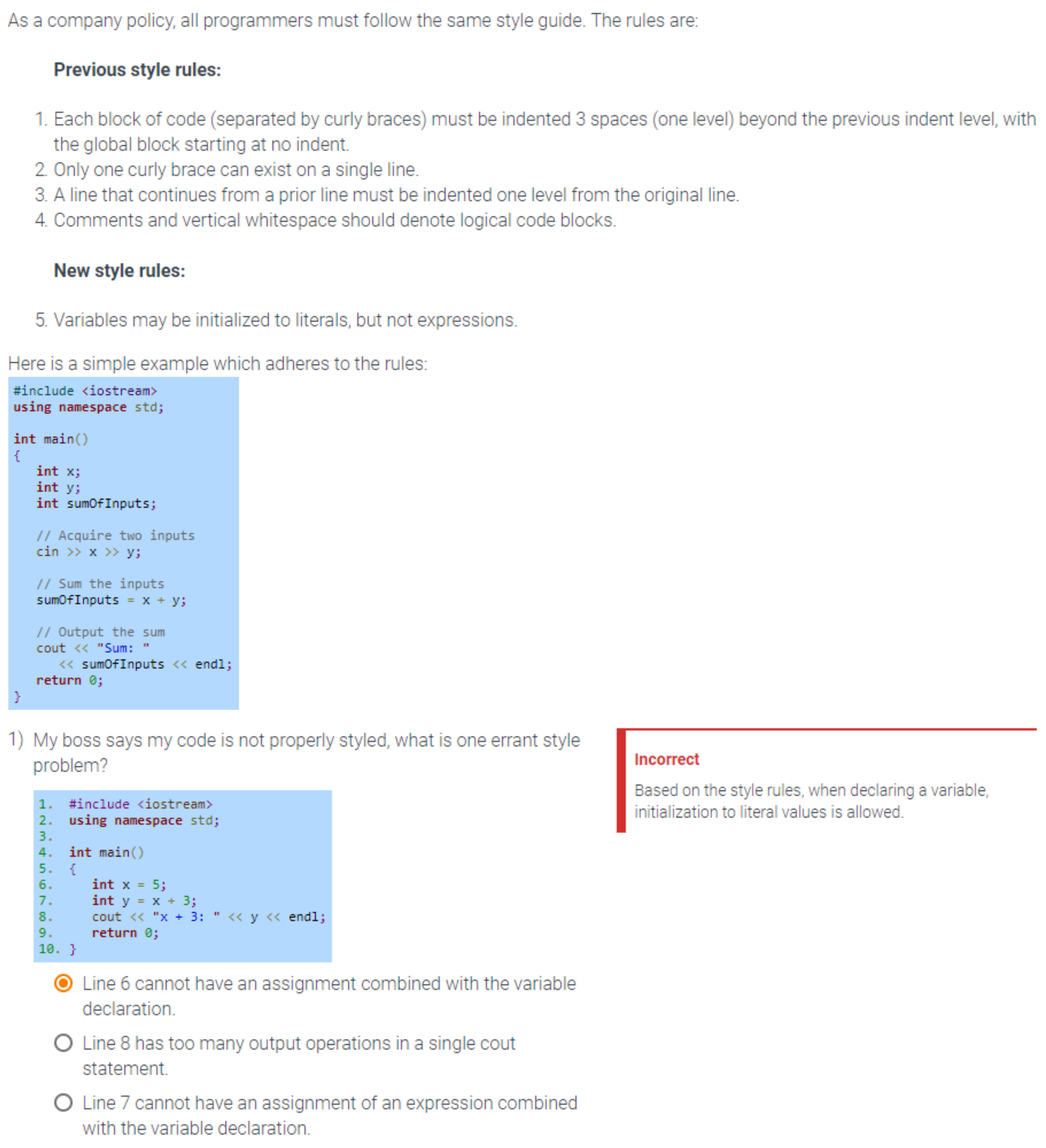
```cpp
1. #include <iostream>
2. using namespace std;
3.
4. int main()
5. {
6. cout << "Hello!" << endl;
7. return 0;
8. }
```

- No libraries are allowed (line #1).
- [✓ The source code does not have comments.]

Correct

Comments increase the readability of source code. Each logical task in the implementation should have a comment.

Figure 185: Style Exercise #1 and #2 - Basic Style, Answers #3 and #2


As a company policy, all programmers must follow the same style guide. The rules are:

Previous style rules:

1. Each block of code (separated by curly braces) must be indented 3 spaces (one level) beyond the previous indent level, with the global block starting at no indent.
2. Only one curly brace can exist on a single line.
3. A line that continues from a prior line must be indented one level from the original line.
4. Comments and vertical whitespace should denote logical code blocks.

New style rules:

5. Variables may be initialized to literals, but not expressions.

Here is a simple example which adheres to the rules:

```cpp
#include <iostream>
using namespace std;

int main()
{
   int x;
   int y;
   int sumOfInputs;

   // Acquire two inputs
   cin >> x >> y;

   // Sum the inputs
   sumOfInputs = x + y;

   // Output the sum
   cout << "Sum: "
      << sumOfInputs << endl;
   return 0;
}
```

1) My boss says my code is not properly styled, what is one errant style problem?

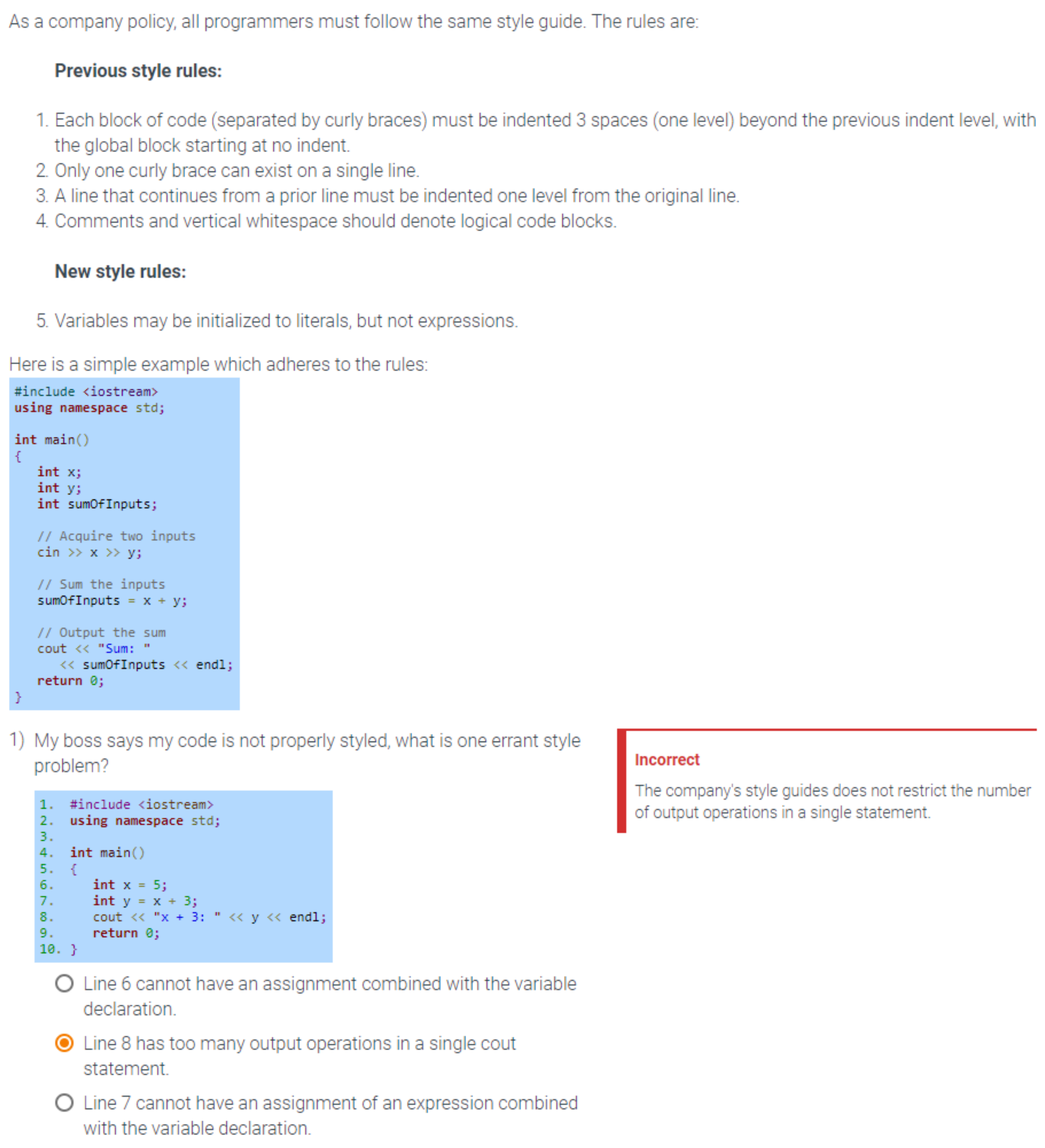
```cpp
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main()
5.  {
6.     int x = 5;
7.     int y = x + 3;
8.     cout << "x + 3: " << y << endl;
9.     return 0;
10. }
```

- [✓ Line 6 cannot have an assignment combined with the variable declaration.]
- Line 8 has too many output operations in a single cout statement.
- Line 7 cannot have an assignment of an expression combined with the variable declaration.

Incorrect

Based on the style rules, when declaring a variable, initialization to literal values is allowed.

Figure 186: Style Exercise #3 - Style with Variables, Answer #1



As a company policy, all programmers must follow the same style guide. The rules are:

Previous style rules:

1. Each block of code (separated by curly braces) must be indented 3 spaces (one level) beyond the previous indent level, with the global block starting at no indent.
2. Only one curly brace can exist on a single line.
3. A line that continues from a prior line must be indented one level from the original line.
4. Comments and vertical whitespace should denote logical code blocks.

New style rules:

5. Variables may be initialized to literals, but not expressions.

Here is a simple example which adheres to the rules:

```
#include <iostream>
using namespace std;

int main()
{
   int x;
   int y;
   int sumOfInputs;

   // Acquire two inputs
   cin >> x >> y;

   // Sum the inputs
   sumOfInputs = x + y;

   // Output the sum
   cout << "Sum: "
      << sumOfInputs << endl;
   return 0;
}
```

1) My boss says my code is not properly styled, what is one errant style problem?

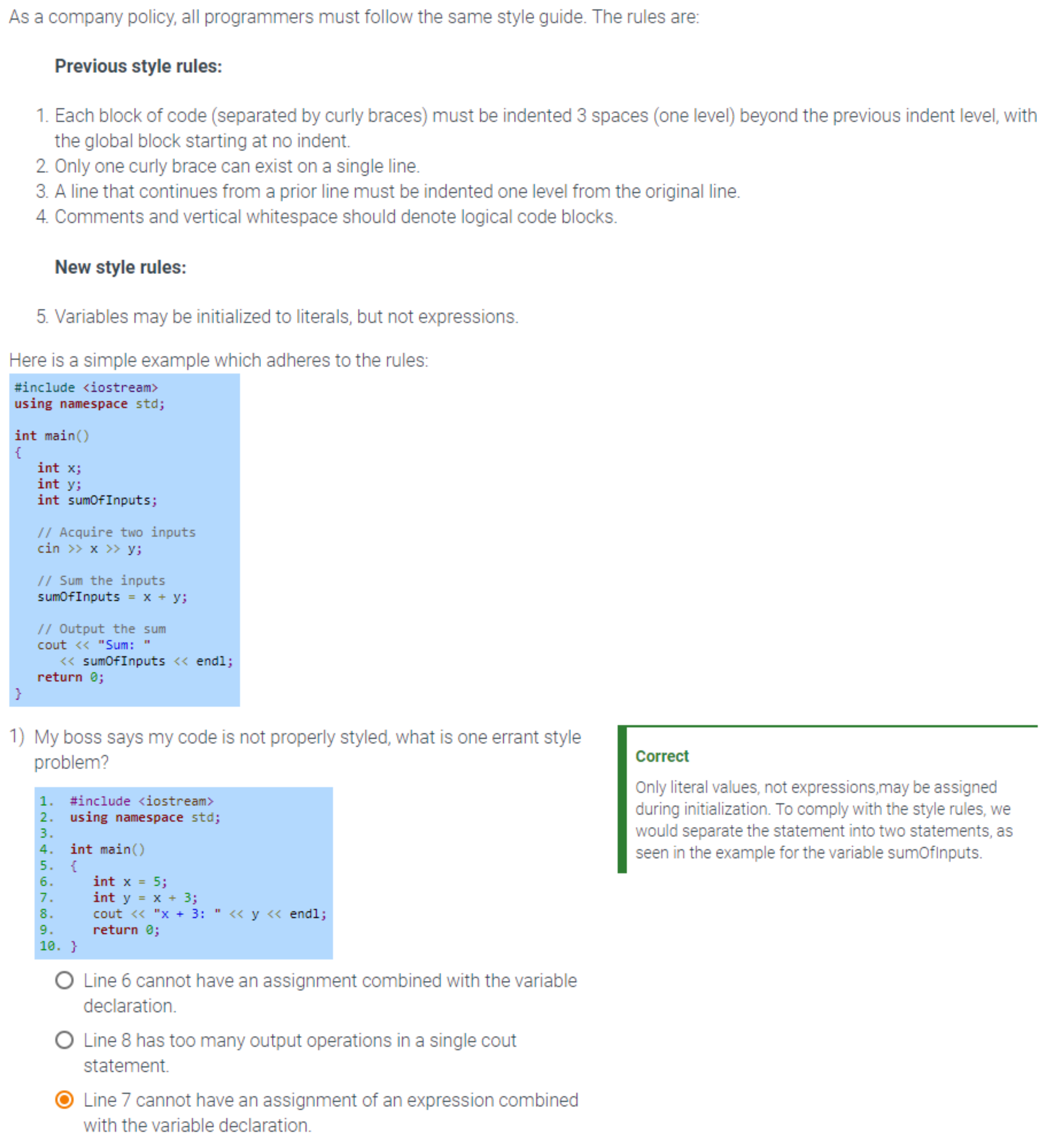
```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main()
5.  {
6.     int x = 5;
7.     int y = x + 3;
8.     cout << "x + 3: " << y << endl;
9.     return 0;
10. }
```

- ○ Line 6 cannot have an assignment combined with the variable declaration.
- [✓ Line 8 has too many output operations in a single cout statement.]
- ○ Line 7 cannot have an assignment of an expression combined with the variable declaration.

Incorrect

The company's style guides does not restrict the number of output operations in a single statement.

Figure 187: Style Exercise #3 - Style with Variables, Answer #2



As a company policy, all programmers must follow the same style guide. The rules are:

Previous style rules:

1. Each block of code (separated by curly braces) must be indented 3 spaces (one level) beyond the previous indent level, with the global block starting at no indent.
2. Only one curly brace can exist on a single line.
3. A line that continues from a prior line must be indented one level from the original line.
4. Comments and vertical whitespace should denote logical code blocks.

New style rules:

5. Variables may be initialized to literals, but not expressions.

Here is a simple example which adheres to the rules:

```cpp
#include <iostream>
using namespace std;

int main()
{
   int x;
   int y;
   int sumOfInputs;

   // Acquire two inputs
   cin >> x >> y;

   // Sum the inputs
   sumOfInputs = x + y;

   // Output the sum
   cout << "Sum: "
      << sumOfInputs << endl;
   return 0;
}
```

1) My boss says my code is not properly styled, what is one errant style problem?

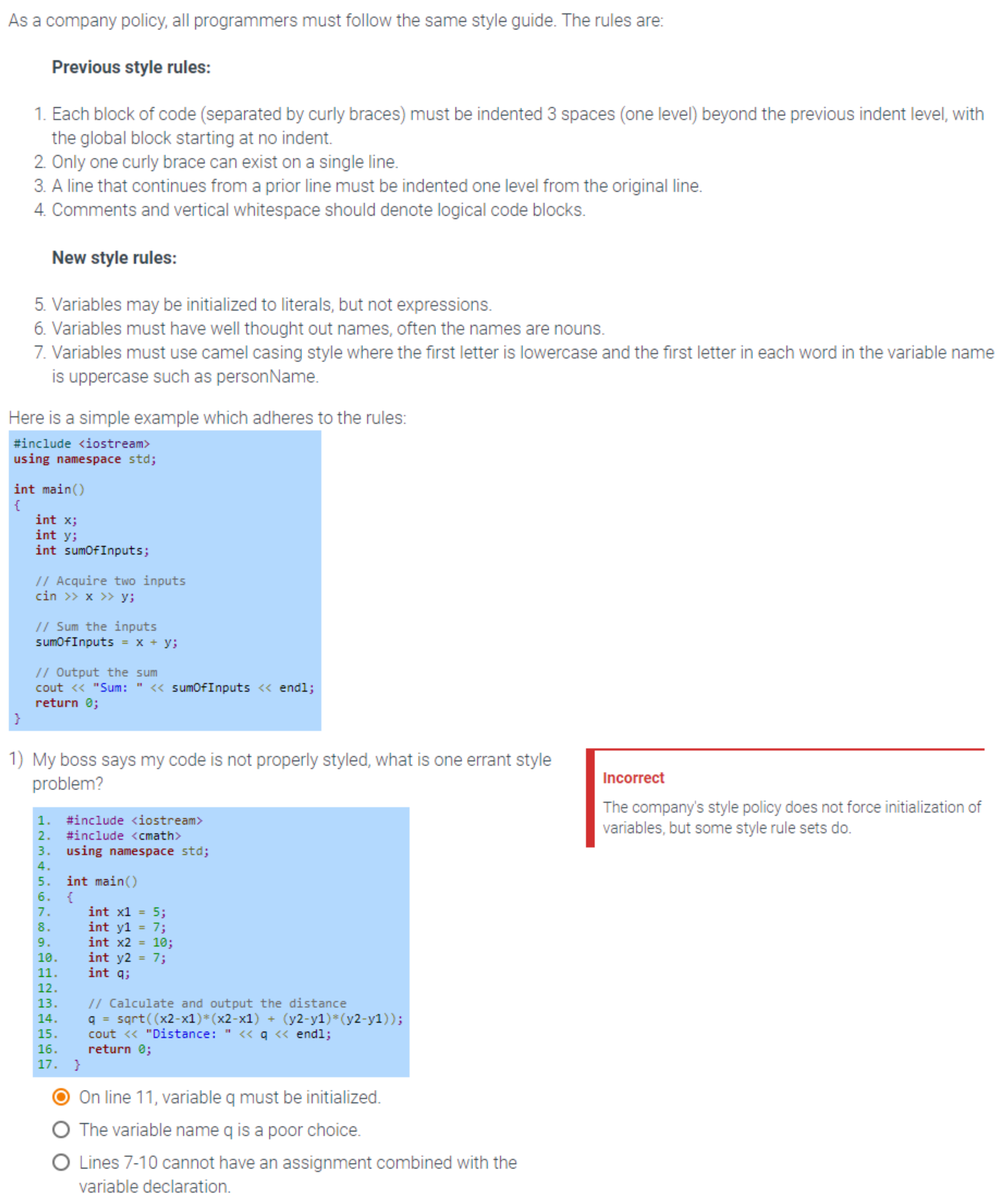
```cpp
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main()
5.  {
6.     int x = 5;
7.     int y = x + 3;
8.     cout << "x + 3: " << y << endl;
9.     return 0;
10. }
```

- Line 6 cannot have an assignment combined with the variable declaration.
- Line 8 has too many output operations in a single cout statement.
- [✓ Line 7 cannot have an assignment of an expression combined with the variable declaration.]

Correct

Only literal values, not expressions,may be assigned during initialization. To comply with the style rules, we would separate the statement into two statements, as seen in the example for the variable sumOfInputs.

Figure 188: Style Exercise #3 - Style with Variables, Answer #3



As a company policy, all programmers must follow the same style guide. The rules are:

Previous style rules:

1. Each block of code (separated by curly braces) must be indented 3 spaces (one level) beyond the previous indent level, with the global block starting at no indent.
2. Only one curly brace can exist on a single line.
3. A line that continues from a prior line must be indented one level from the original line.
4. Comments and vertical whitespace should denote logical code blocks.

New style rules:

5. Variables may be initialized to literals, but not expressions.
6. Variables must have well thought out names, often the names are nouns.
7. Variables must use camel casing style where the first letter is lowercase and the first letter in each word in the variable name is uppercase such as personName.

Here is a simple example which adheres to the rules:

```
#include <iostream>
using namespace std;

int main()
{
   int x;
   int y;
   int sumOfInputs;

   // Acquire two inputs
   cin >> x >> y;

   // Sum the inputs
   sumOfInputs = x + y;

   // Output the sum
   cout << "Sum: " << sumOfInputs << endl;
   return 0;
}
```

1) My boss says my code is not properly styled, what is one errant style problem?

```
1.  #include <iostream>
2.  #include <cmath>
3.  using namespace std;
4.
5.  int main()
6.  {
7.     int x1 = 5;
8.     int y1 = 7;
9.     int x2 = 10;
10.    int y2 = 7;
11.    int q;
12.
13.    // Calculate and output the distance
14.    q = sqrt((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1));
15.    cout << "Distance: " << q << endl;
16.    return 0;
17. }
```

[✓ On line 11, variable q must be initialized.]

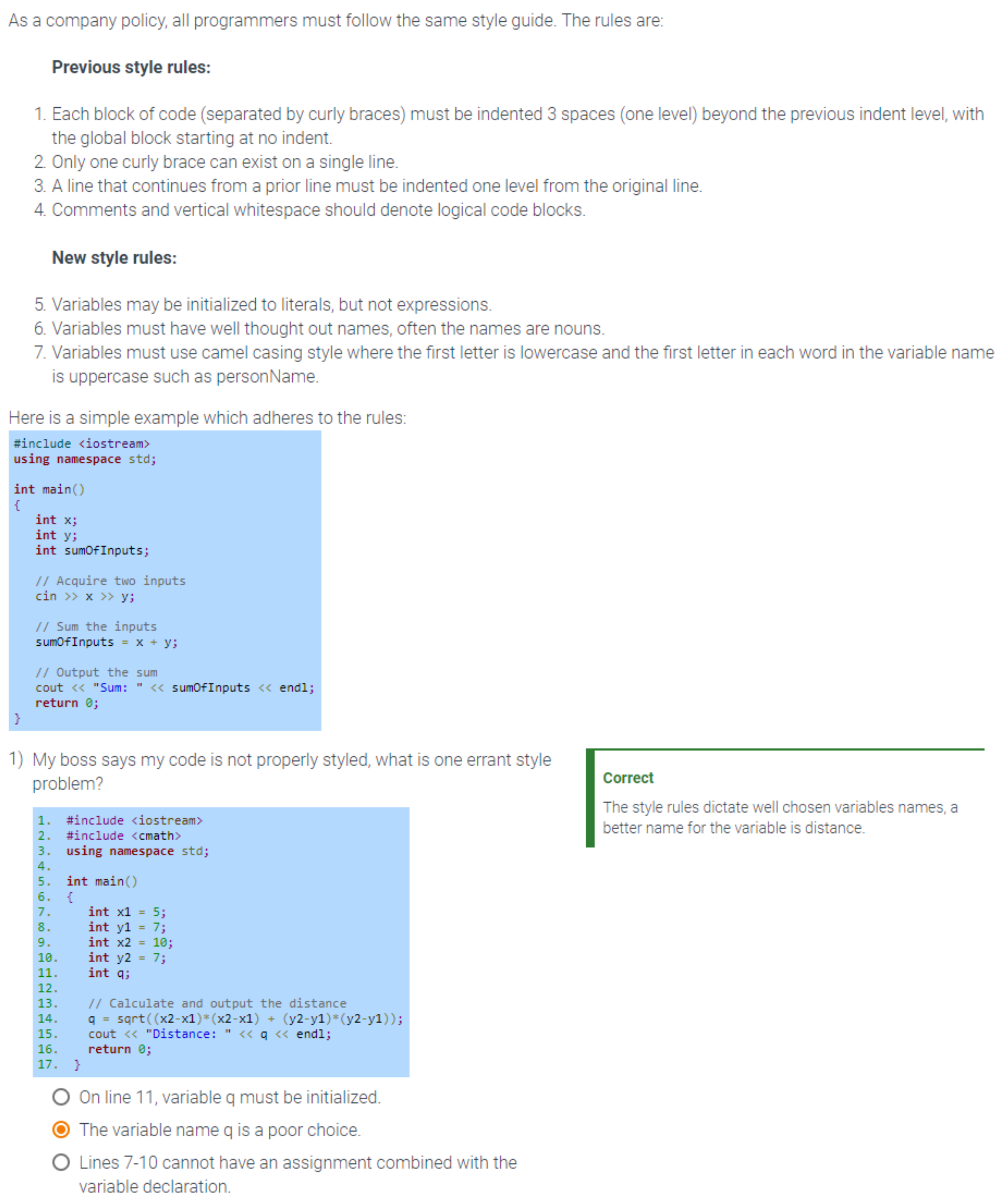
The variable name q is a poor choice.

Lines 7-10 cannot have an assignment combined with the variable declaration.

Incorrect

The company's style policy does not force initialization of variables, but some style rule sets do.

Figure 189: Style Exercise #4 - Style with Variables, Answer #1



As a company policy, all programmers must follow the same style guide. The rules are:

Previous style rules:

1. Each block of code (separated by curly braces) must be indented 3 spaces (one level) beyond the previous indent level, with the global block starting at no indent.
2. Only one curly brace can exist on a single line.
3. A line that continues from a prior line must be indented one level from the original line.
4. Comments and vertical whitespace should denote logical code blocks.

New style rules:

5. Variables may be initialized to literals, but not expressions.
6. Variables must have well thought out names, often the names are nouns.
7. Variables must use camel casing style where the first letter is lowercase and the first letter in each word in the variable name is uppercase such as personName.

Here is a simple example which adheres to the rules:

```
#include <iostream>
using namespace std;

int main()
{
   int x;
   int y;
   int sumOfInputs;

   // Acquire two inputs
   cin >> x >> y;

   // Sum the inputs
   sumOfInputs = x + y;

   // Output the sum
   cout << "Sum: " << sumOfInputs << endl;
   return 0;
}
```

1) My boss says my code is not properly styled, what is one errant style problem?

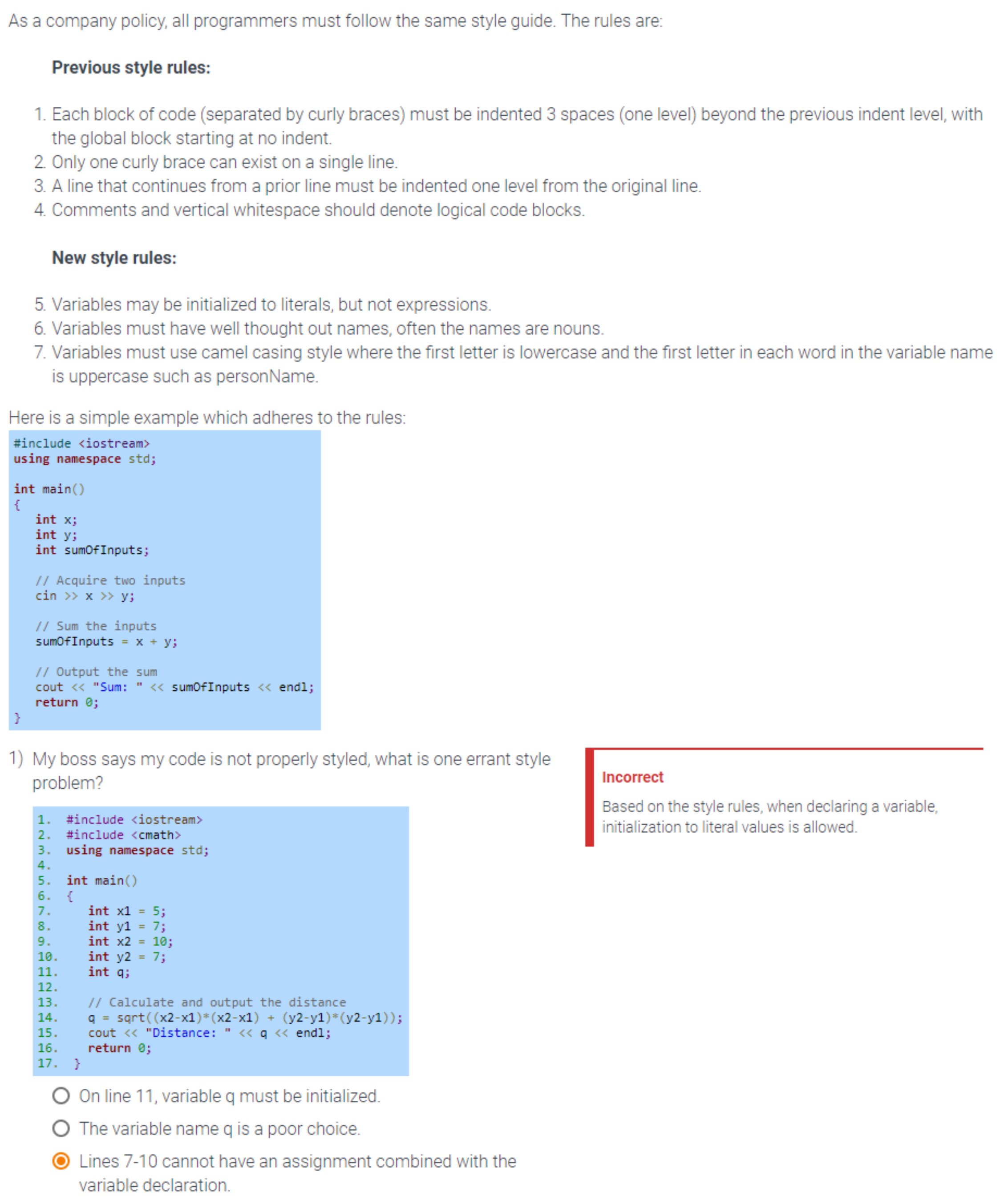
```
1.  #include <iostream>
2.  #include <cmath>
3.  using namespace std;
4.
5.  int main()
6.  {
7.     int x1 = 5;
8.     int y1 = 7;
9.     int x2 = 10;
10.    int y2 = 7;
11.    int q;
12.
13.    // Calculate and output the distance
14.    q = sqrt((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1));
15.    cout << "Distance: " << q << endl;
16.    return 0;
17. }
```

- On line 11, variable q must be initialized.
- [✓ The variable name q is a poor choice.]
- Lines 7-10 cannot have an assignment combined with the variable declaration.

Correct

The style rules dictate well chosen variables names, a better name for the variable is distance.

Figure 190: Style Exercise #4 - Style with Variables, Answer #2



As a company policy, all programmers must follow the same style guide. The rules are:

Previous style rules:

1. Each block of code (separated by curly braces) must be indented 3 spaces (one level) beyond the previous indent level, with the global block starting at no indent.
2. Only one curly brace can exist on a single line.
3. A line that continues from a prior line must be indented one level from the original line.
4. Comments and vertical whitespace should denote logical code blocks.

New style rules:

5. Variables may be initialized to literals, but not expressions.
6. Variables must have well thought out names, often the names are nouns.
7. Variables must use camel casing style where the first letter is lowercase and the first letter in each word in the variable name is uppercase such as personName.

Here is a simple example which adheres to the rules:

```
#include <iostream>
using namespace std;

int main()
{
   int x;
   int y;
   int sumOfInputs;

   // Acquire two inputs
   cin >> x >> y;

   // Sum the inputs
   sumOfInputs = x + y;

   // Output the sum
   cout << "Sum: " << sumOfInputs << endl;
   return 0;
}
```

1) My boss says my code is not properly styled, what is one errant style problem?

```
1.  #include <iostream>
2.  #include <cmath>
3.  using namespace std;
4.
5.  int main()
6.  {
7.     int x1 = 5;
8.     int y1 = 7;
9.     int x2 = 10;
10.    int y2 = 7;
11.    int q;
12.
13.    // Calculate and output the distance
14.    q = sqrt((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1));
15.    cout << "Distance: " << q << endl;
16.    return 0;
17. }
```

○ On line 11, variable q must be initialized.

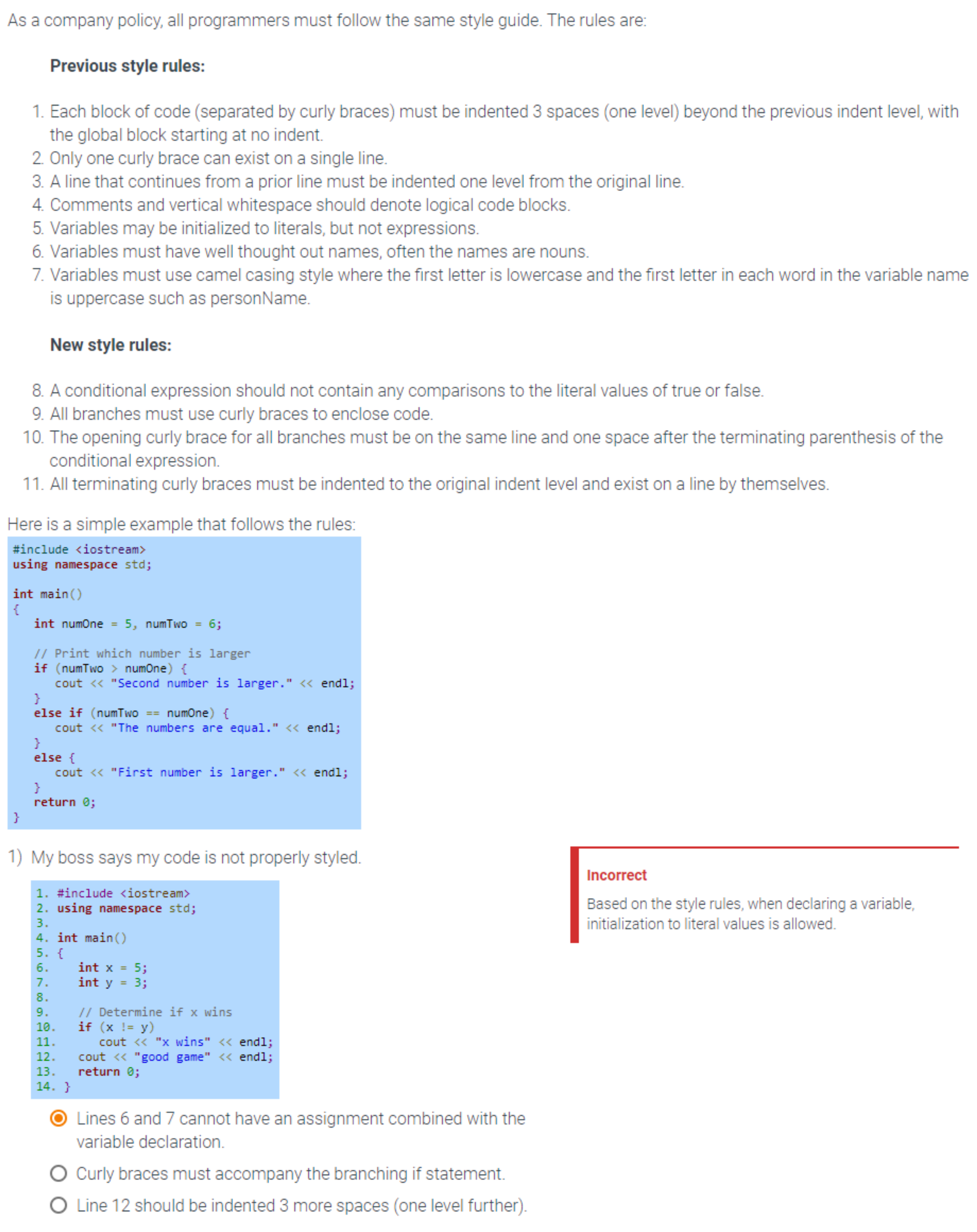
○ The variable name q is a poor choice.

[✓ Lines 7-10 cannot have an assignment combined with the variable declaration.]

Incorrect

Based on the style rules, when declaring a variable, initialization to literal values is allowed.

Figure 191: Style Exercise #4 - Style with Variables, Answer #3


As a company policy, all programmers must follow the same style guide. The rules are:

Previous style rules:

1. Each block of code (separated by curly braces) must be indented 3 spaces (one level) beyond the previous indent level, with the global block starting at no indent.
2. Only one curly brace can exist on a single line.
3. A line that continues from a prior line must be indented one level from the original line.
4. Comments and vertical whitespace should denote logical code blocks.
5. Variables may be initialized to literals, but not expressions.
6. Variables must have well thought out names, often the names are nouns.
7. Variables must use camel casing style where the first letter is lowercase and the first letter in each word in the variable name is uppercase such as personName.

New style rules:

8. A conditional expression should not contain any comparisons to the literal values of true or false.
9. All branches must use curly braces to enclose code.
10. The opening curly brace for all branches must be on the same line and one space after the terminating parenthesis of the conditional expression.
11. All terminating curly braces must be indented to the original indent level and exist on a line by themselves.

Here is a simple example that follows the rules:

```
#include <iostream>
using namespace std;

int main()
{
   int numOne = 5, numTwo = 6;

   // Print which number is larger
   if (numTwo > numOne) {
      cout << "Second number is larger." << endl;
   }
   else if (numTwo == numOne) {
      cout << "The numbers are equal." << endl;
   }
   else {
      cout << "First number is larger." << endl;
   }
   return 0;
}
```

1) My boss says my code is not properly styled.

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main()
5.  {
6.     int x = 5;
7.     int y = 3;
8.
9.     // Determine if x wins
10.    if (x != y)
11.       cout << "x wins" << endl;
12.    cout << "good game" << endl;
13.    return 0;
14. }
```

[✓ Lines 6 and 7 cannot have an assignment combined with the variable declaration.]

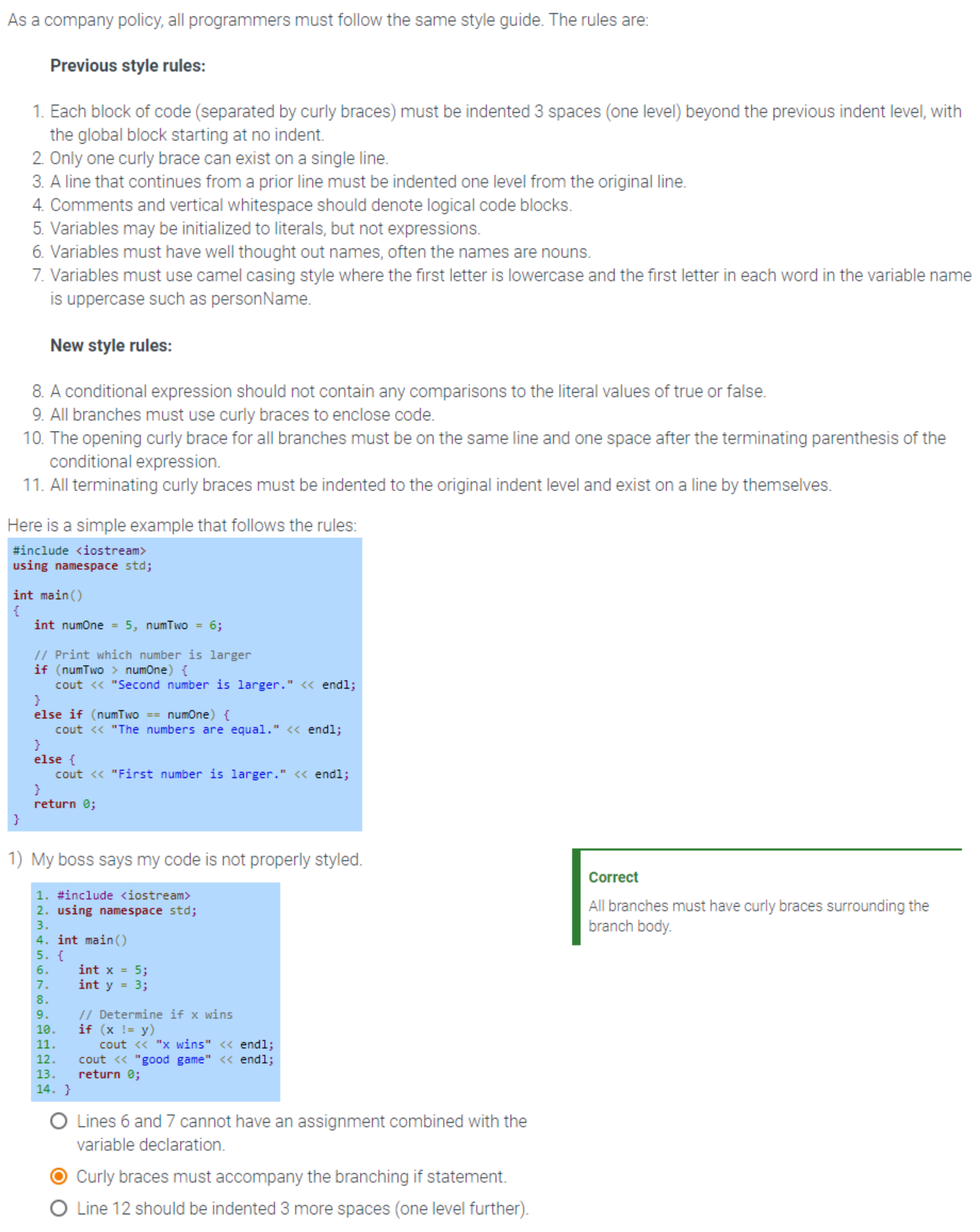
Curly braces must accompany the branching if statement.

Line 12 should be indented 3 more spaces (one level further).

Incorrect

Based on the style rules, when declaring a variable, initialization to literal values is allowed.

Figure 192: Style Exercise #5 - Style with Branches, Answer #1



As a company policy, all programmers must follow the same style guide. The rules are:

Previous style rules:

1. Each block of code (separated by curly braces) must be indented 3 spaces (one level) beyond the previous indent level, with the global block starting at no indent.
2. Only one curly brace can exist on a single line.
3. A line that continues from a prior line must be indented one level from the original line.
4. Comments and vertical whitespace should denote logical code blocks.
5. Variables may be initialized to literals, but not expressions.
6. Variables must have well thought out names, often the names are nouns.
7. Variables must use camel casing style where the first letter is lowercase and the first letter in each word in the variable name is uppercase such as personName.

New style rules:

8. A conditional expression should not contain any comparisons to the literal values of true or false.
9. All branches must use curly braces to enclose code.
10. The opening curly brace for all branches must be on the same line and one space after the terminating parenthesis of the conditional expression.
11. All terminating curly braces must be indented to the original indent level and exist on a line by themselves.

Here is a simple example that follows the rules:

```
#include <iostream>
using namespace std;

int main()
{
   int numOne = 5, numTwo = 6;

   // Print which number is larger
   if (numTwo > numOne) {
      cout << "Second number is larger." << endl;
   }
   else if (numTwo == numOne) {
      cout << "The numbers are equal." << endl;
   }
   else {
      cout << "First number is larger." << endl;
   }
   return 0;
}
```

1) My boss says my code is not properly styled.

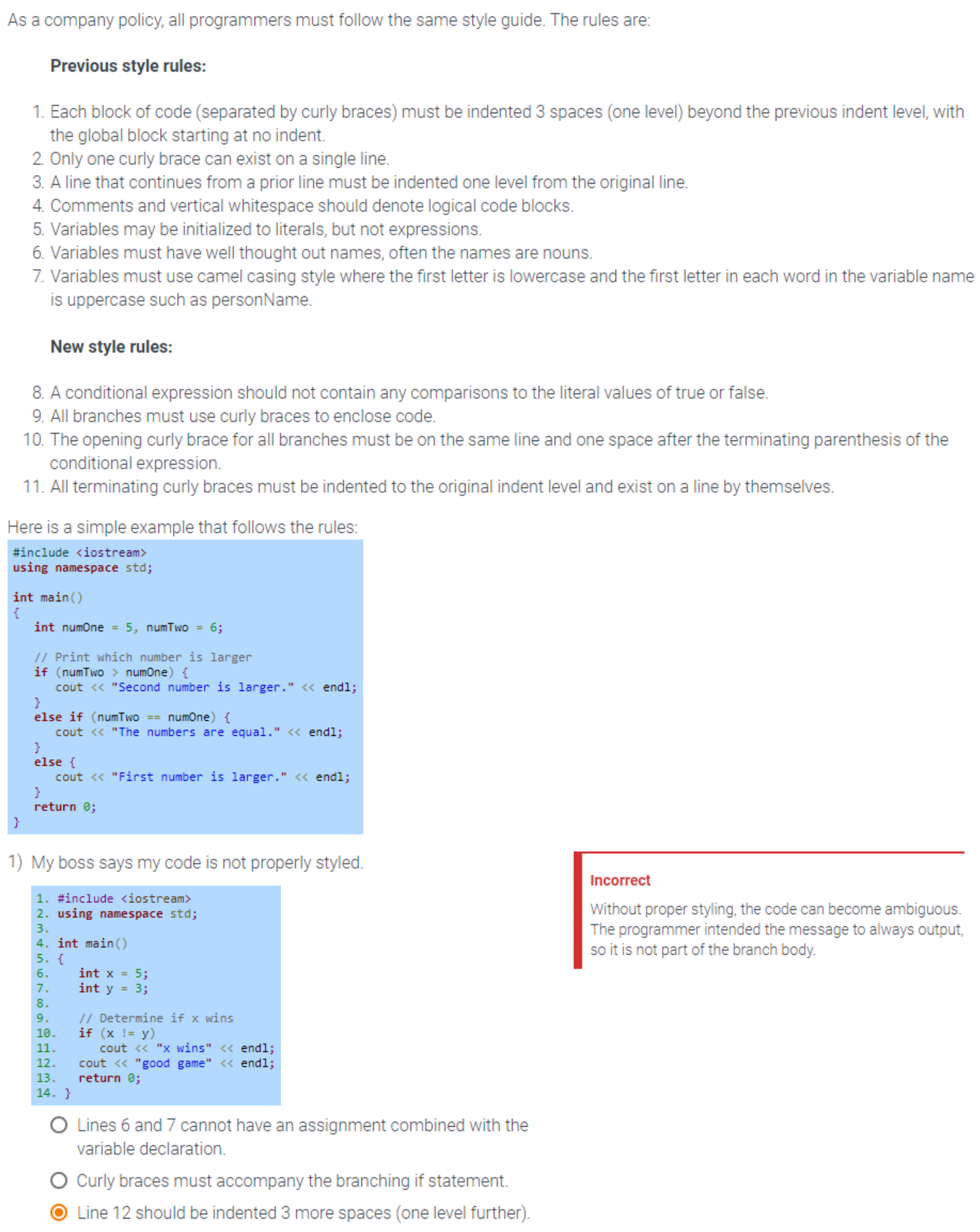
```
1. #include <iostream>
2. using namespace std;
3.
4. int main()
5. {
6.    int x = 5;
7.    int y = 3;
8.
9.    // Determine if x wins
10.   if (x != y)
11.      cout << "x wins" << endl;
12.   cout << "good game" << endl;
13.   return 0;
14. }
```

- Lines 6 and 7 cannot have an assignment combined with the variable declaration.
- [✓ Curly braces must accompany the branching if statement.]
- Line 12 should be indented 3 more spaces (one level further).

Correct

All branches must have curly braces surrounding the branch body.

Figure 193: Style Exercise #5 - Style with Branches, Answer #2



As a company policy, all programmers must follow the same style guide. The rules are:

Previous style rules:

1. Each block of code (separated by curly braces) must be indented 3 spaces (one level) beyond the previous indent level, with the global block starting at no indent.
2. Only one curly brace can exist on a single line.
3. A line that continues from a prior line must be indented one level from the original line.
4. Comments and vertical whitespace should denote logical code blocks.
5. Variables may be initialized to literals, but not expressions.
6. Variables must have well thought out names, often the names are nouns.
7. Variables must use camel casing style where the first letter is lowercase and the first letter in each word in the variable name is uppercase such as personName.

New style rules:

8. A conditional expression should not contain any comparisons to the literal values of true or false.
9. All branches must use curly braces to enclose code.
10. The opening curly brace for all branches must be on the same line and one space after the terminating parenthesis of the conditional expression.
11. All terminating curly braces must be indented to the original indent level and exist on a line by themselves.

Here is a simple example that follows the rules:

```
#include <iostream>
using namespace std;

int main()
{
   int numOne = 5, numTwo = 6;

   // Print which number is larger
   if (numTwo > numOne) {
      cout << "Second number is larger." << endl;
   }
   else if (numTwo == numOne) {
      cout << "The numbers are equal." << endl;
   }
   else {
      cout << "First number is larger." << endl;
   }
   return 0;
}
```

1) My boss says my code is not properly styled.

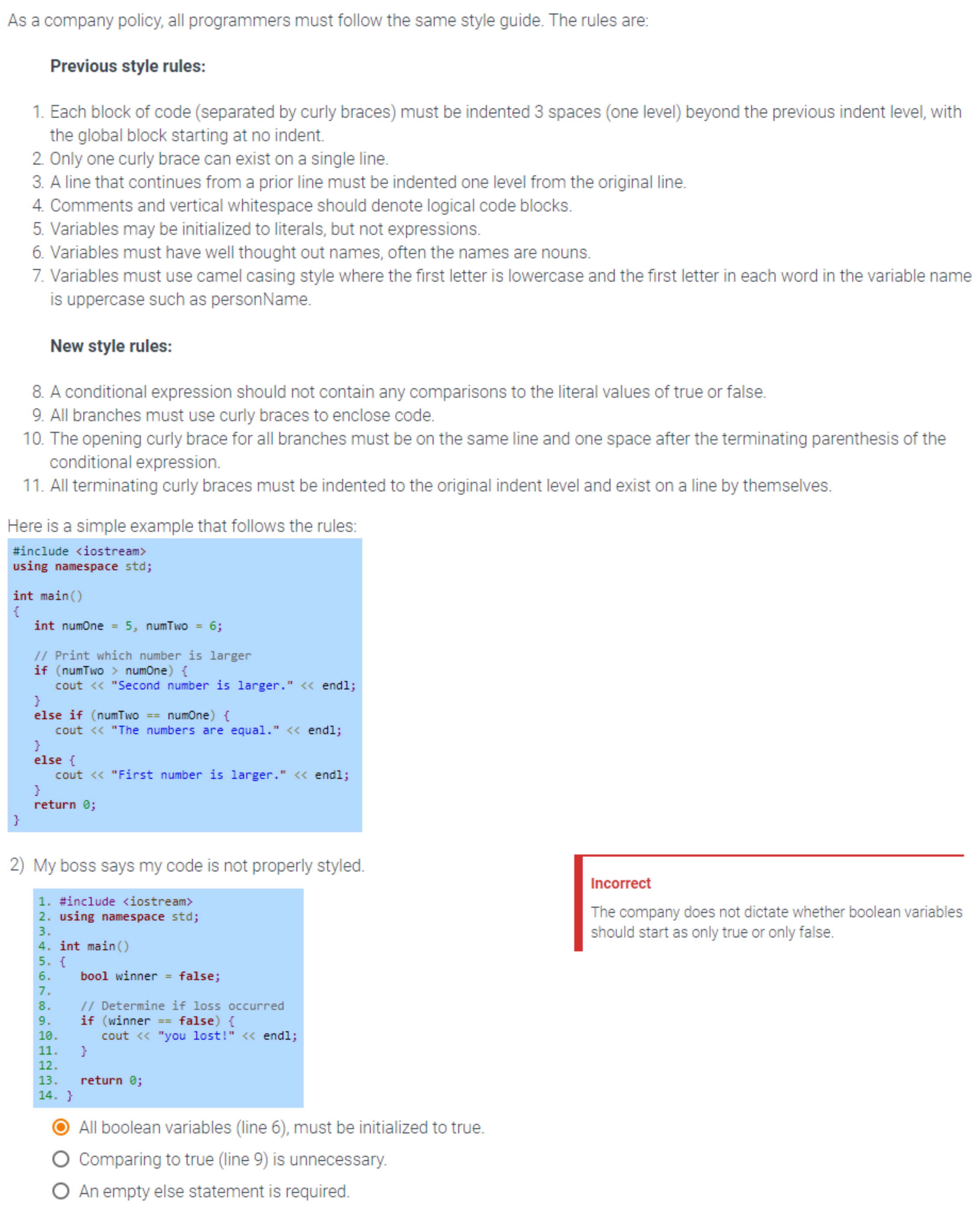
```
1. #include <iostream>
2. using namespace std;
3.
4. int main()
5. {
6.    int x = 5;
7.    int y = 3;
8.
9.    // Determine if x wins
10.   if (x != y)
11.      cout << "x wins" << endl;
12.   cout << "good game" << endl;
13.   return 0;
14. }
```

- ○ Lines 6 and 7 cannot have an assignment combined with the variable declaration.
- ○ Curly braces must accompany the branching if statement.
- [✓ Line 12 should be indented 3 more spaces (one level further).]

Incorrect

Without proper styling, the code can become ambiguous. The programmer intended the message to always output, so it is not part of the branch body.

Figure 194: Style Exercise #5 - Style with Branches, Answer #3



As a company policy, all programmers must follow the same style guide. The rules are:

Previous style rules:

1. Each block of code (separated by curly braces) must be indented 3 spaces (one level) beyond the previous indent level, with the global block starting at no indent.
2. Only one curly brace can exist on a single line.
3. A line that continues from a prior line must be indented one level from the original line.
4. Comments and vertical whitespace should denote logical code blocks.
5. Variables may be initialized to literals, but not expressions.
6. Variables must have well thought out names, often the names are nouns.
7. Variables must use camel casing style where the first letter is lowercase and the first letter in each word in the variable name is uppercase such as personName.

New style rules:

8. A conditional expression should not contain any comparisons to the literal values of true or false.
9. All branches must use curly braces to enclose code.
10. The opening curly brace for all branches must be on the same line and one space after the terminating parenthesis of the conditional expression.
11. All terminating curly braces must be indented to the original indent level and exist on a line by themselves.

Here is a simple example that follows the rules:

```
#include <iostream>
using namespace std;

int main()
{
   int numOne = 5, numTwo = 6;

   // Print which number is larger
   if (numTwo > numOne) {
      cout << "Second number is larger." << endl;
   }
   else if (numTwo == numOne) {
      cout << "The numbers are equal." << endl;
   }
   else {
      cout << "First number is larger." << endl;
   }
   return 0;
}
```

2) My boss says my code is not properly styled.

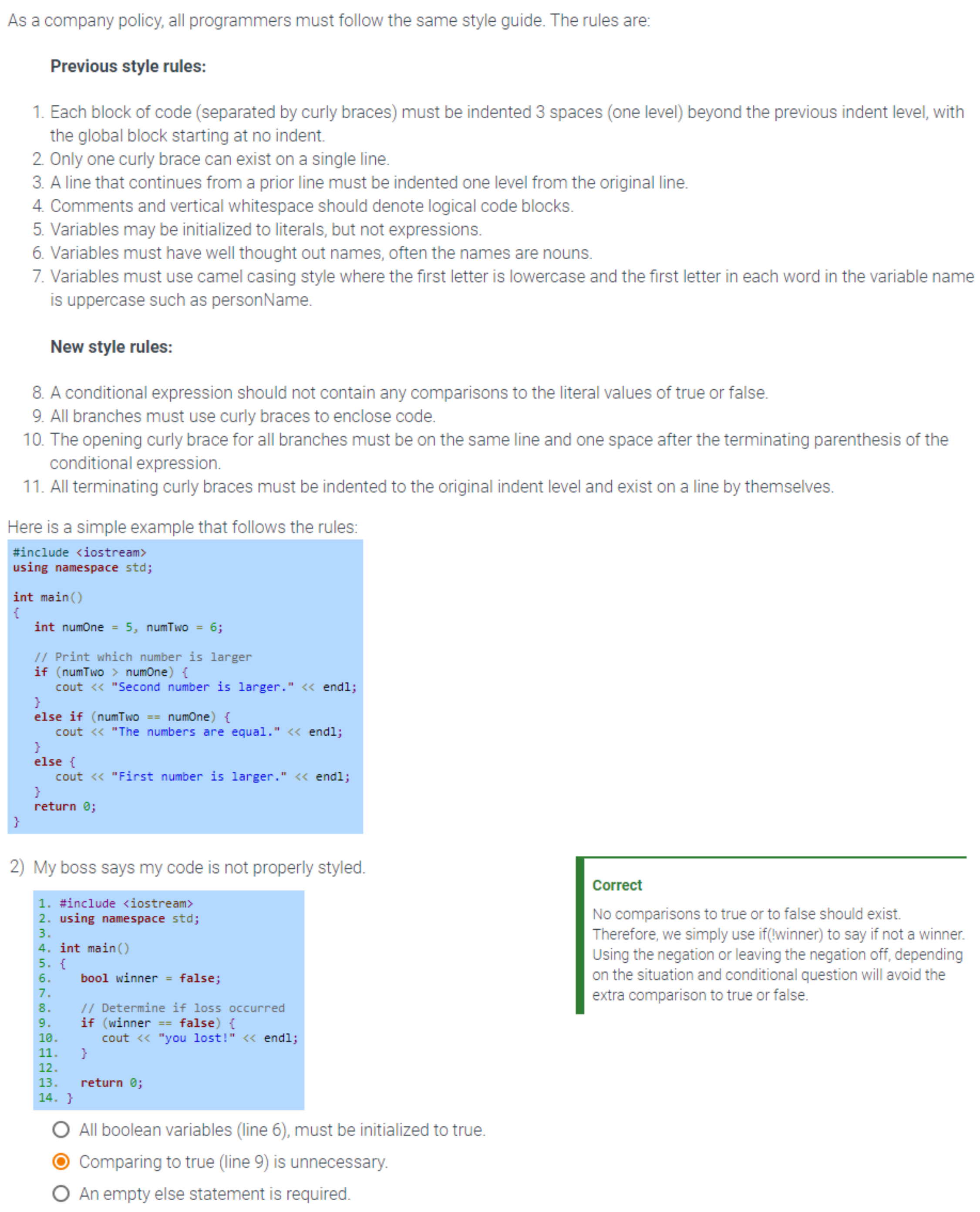
```
1. #include <iostream>
2. using namespace std;
3.
4. int main()
5. {
6.    bool winner = false;
7.
8.    // Determine if loss occurred
9.    if (winner == false) {
10.      cout << "you lost!" << endl;
11.   }
12.
13.   return 0;
14. }
```

- [✓ All boolean variables (line 6), must be initialized to true.]
- Comparing to true (line 9) is unnecessary.
- An empty else statement is required.

Incorrect

The company does not dictate whether boolean variables should start as only true or only false.

Figure 195: Style Exercise #6 - Style with Branches, Answer #1



As a company policy, all programmers must follow the same style guide. The rules are:

Previous style rules:

1. Each block of code (separated by curly braces) must be indented 3 spaces (one level) beyond the previous indent level, with the global block starting at no indent.
2. Only one curly brace can exist on a single line.
3. A line that continues from a prior line must be indented one level from the original line.
4. Comments and vertical whitespace should denote logical code blocks.
5. Variables may be initialized to literals, but not expressions.
6. Variables must have well thought out names, often the names are nouns.
7. Variables must use camel casing style where the first letter is lowercase and the first letter in each word in the variable name is uppercase such as personName.

New style rules:

8. A conditional expression should not contain any comparisons to the literal values of true or false.
9. All branches must use curly braces to enclose code.
10. The opening curly brace for all branches must be on the same line and one space after the terminating parenthesis of the conditional expression.
11. All terminating curly braces must be indented to the original indent level and exist on a line by themselves.

Here is a simple example that follows the rules:

```
#include <iostream>
using namespace std;

int main()
{
   int numOne = 5, numTwo = 6;

   // Print which number is larger
   if (numTwo > numOne) {
      cout << "Second number is larger." << endl;
   }
   else if (numTwo == numOne) {
      cout << "The numbers are equal." << endl;
   }
   else {
      cout << "First number is larger." << endl;
   }
   return 0;
}
```

2) My boss says my code is not properly styled.

```
1. #include <iostream>
2. using namespace std;
3.
4. int main()
5. {
6.    bool winner = false;
7.
8.    // Determine if loss occurred
9.    if (winner == false) {
10.      cout << "you lost!" << endl;
11.   }
12.
13.   return 0;
14. }
```

○ All boolean variables (line 6), must be initialized to true.

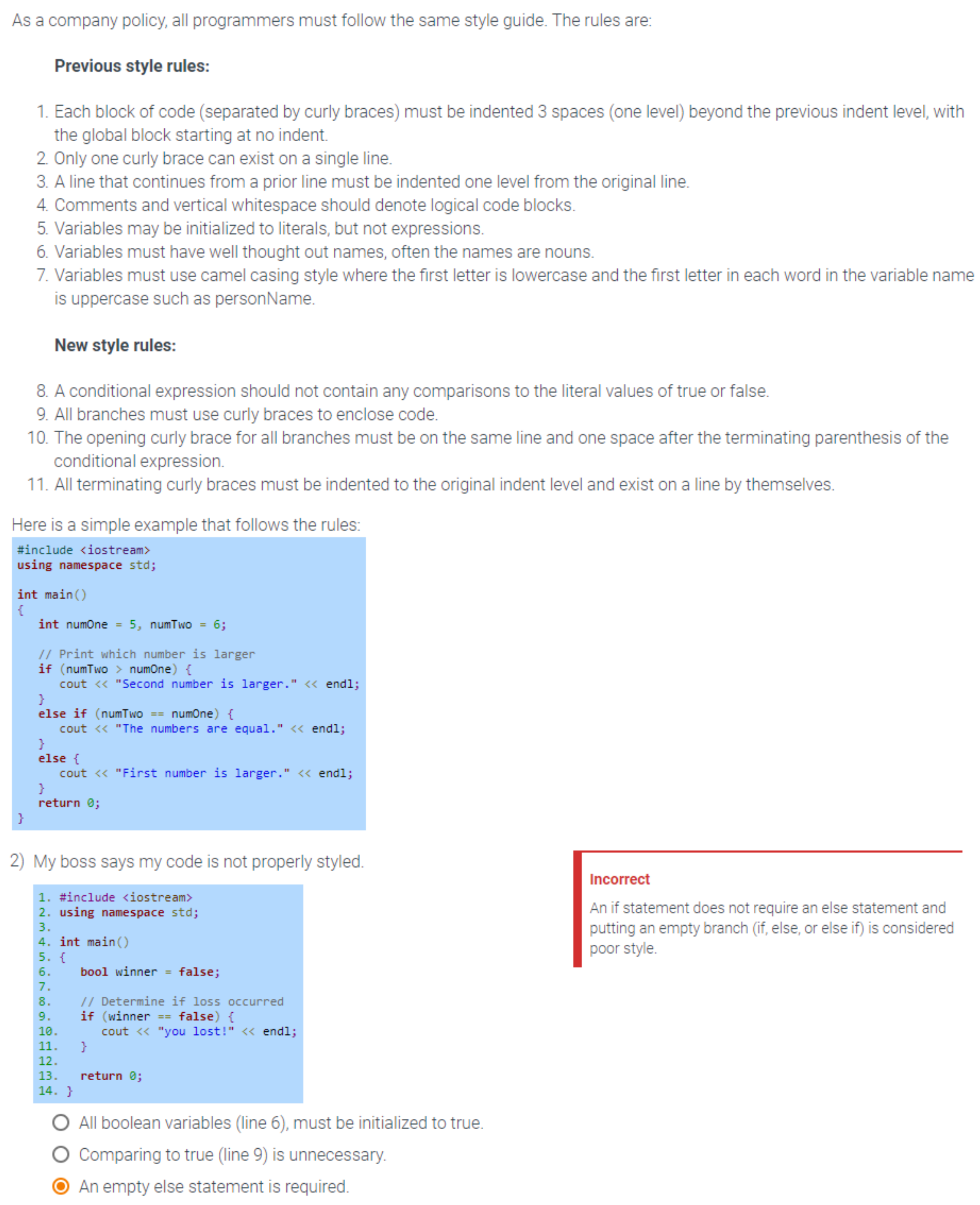
[✓ Comparing to true (line 9) is unnecessary.]

○ An empty else statement is required.

Correct

No comparisons to true or to false should exist. Therefore, we simply use if(!winner) to say if not a winner. Using the negation or leaving the negation off, depending on the situation and conditional question will avoid the extra comparison to true or false.

Figure 196: Style Exercise #6 - Style with Branches, Answer #2



As a company policy, all programmers must follow the same style guide. The rules are:

Previous style rules:

1. Each block of code (separated by curly braces) must be indented 3 spaces (one level) beyond the previous indent level, with the global block starting at no indent.
2. Only one curly brace can exist on a single line.
3. A line that continues from a prior line must be indented one level from the original line.
4. Comments and vertical whitespace should denote logical code blocks.
5. Variables may be initialized to literals, but not expressions.
6. Variables must have well thought out names, often the names are nouns.
7. Variables must use camel casing style where the first letter is lowercase and the first letter in each word in the variable name is uppercase such as personName.

New style rules:

8. A conditional expression should not contain any comparisons to the literal values of true or false.
9. All branches must use curly braces to enclose code.
10. The opening curly brace for all branches must be on the same line and one space after the terminating parenthesis of the conditional expression.
11. All terminating curly braces must be indented to the original indent level and exist on a line by themselves.

Here is a simple example that follows the rules:

```
#include <iostream>
using namespace std;

int main()
{
   int numOne = 5, numTwo = 6;

   // Print which number is larger
   if (numTwo > numOne) {
      cout << "Second number is larger." << endl;
   }
   else if (numTwo == numOne) {
      cout << "The numbers are equal." << endl;
   }
   else {
      cout << "First number is larger." << endl;
   }
   return 0;
}
```

2) My boss says my code is not properly styled.

```
1. #include <iostream>
2. using namespace std;
3.
4. int main()
5. {
6.    bool winner = false;
7.
8.    // Determine if loss occurred
9.    if (winner == false) {
10.      cout << "you lost!" << endl;
11.   }
12.
13.   return 0;
14. }
```

○ All boolean variables (line 6), must be initialized to true.

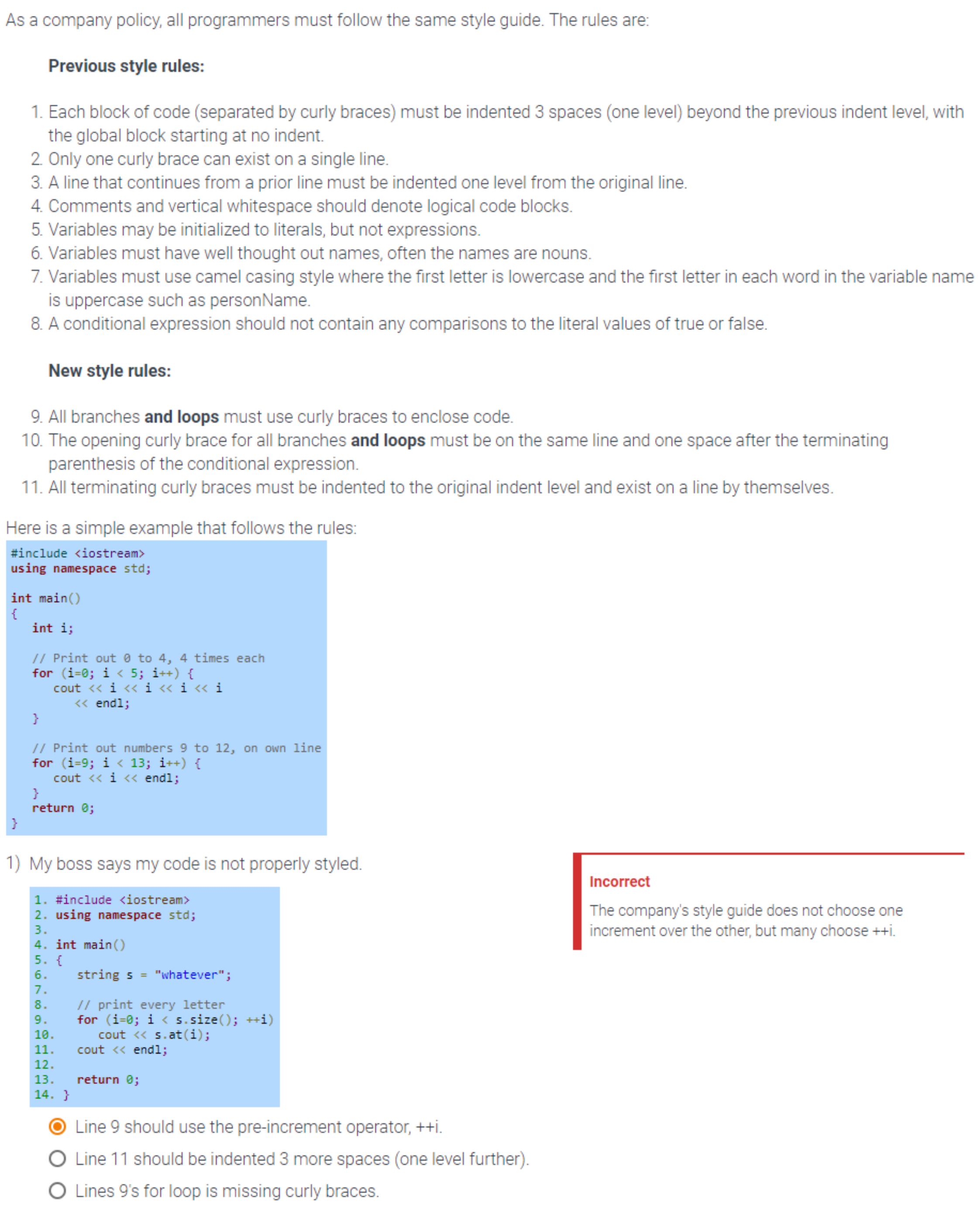
○ Comparing to true (line 9) is unnecessary.

[✓ An empty else statement is required.]

Incorrect

An if statement does not require an else statement and putting an empty branch (if, else, or else if) is considered poor style.

Figure 197: Style Exercise #6 - Style with Branches, Answer #3



As a company policy, all programmers must follow the same style guide. The rules are:

Previous style rules:

1. Each block of code (separated by curly braces) must be indented 3 spaces (one level) beyond the previous indent level, with the global block starting at no indent.
2. Only one curly brace can exist on a single line.
3. A line that continues from a prior line must be indented one level from the original line.
4. Comments and vertical whitespace should denote logical code blocks.
5. Variables may be initialized to literals, but not expressions.
6. Variables must have well thought out names, often the names are nouns.
7. Variables must use camel casing style where the first letter is lowercase and the first letter in each word in the variable name is uppercase such as personName.
8. A conditional expression should not contain any comparisons to the literal values of true or false.

New style rules:

9. All branches **and loops** must use curly braces to enclose code.
10. The opening curly brace for all branches **and loops** must be on the same line and one space after the terminating parenthesis of the conditional expression.
11. All terminating curly braces must be indented to the original indent level and exist on a line by themselves.

Here is a simple example that follows the rules:

```
#include <iostream>
using namespace std;

int main()
{
   int i;

   // Print out 0 to 4, 4 times each
   for (i=0; i < 5; i++) {
      cout << i << i << i << i
         << endl;
   }

   // Print out numbers 9 to 12, on own line
   for (i=9; i < 13; i++) {
      cout << i << endl;
   }
   return 0;
}
```

1) My boss says my code is not properly styled.

```
1. #include <iostream>
2. using namespace std;
3.
4. int main()
5. {
6.    string s = "whatever";
7.
8.    // print every letter
9.    for (i=0; i < s.size(); ++i)
10.      cout << s.at(i);
11.   cout << endl;
12.
13.   return 0;
14. }
```

[✓ Line 9 should use the pre-increment operator, ++i.]

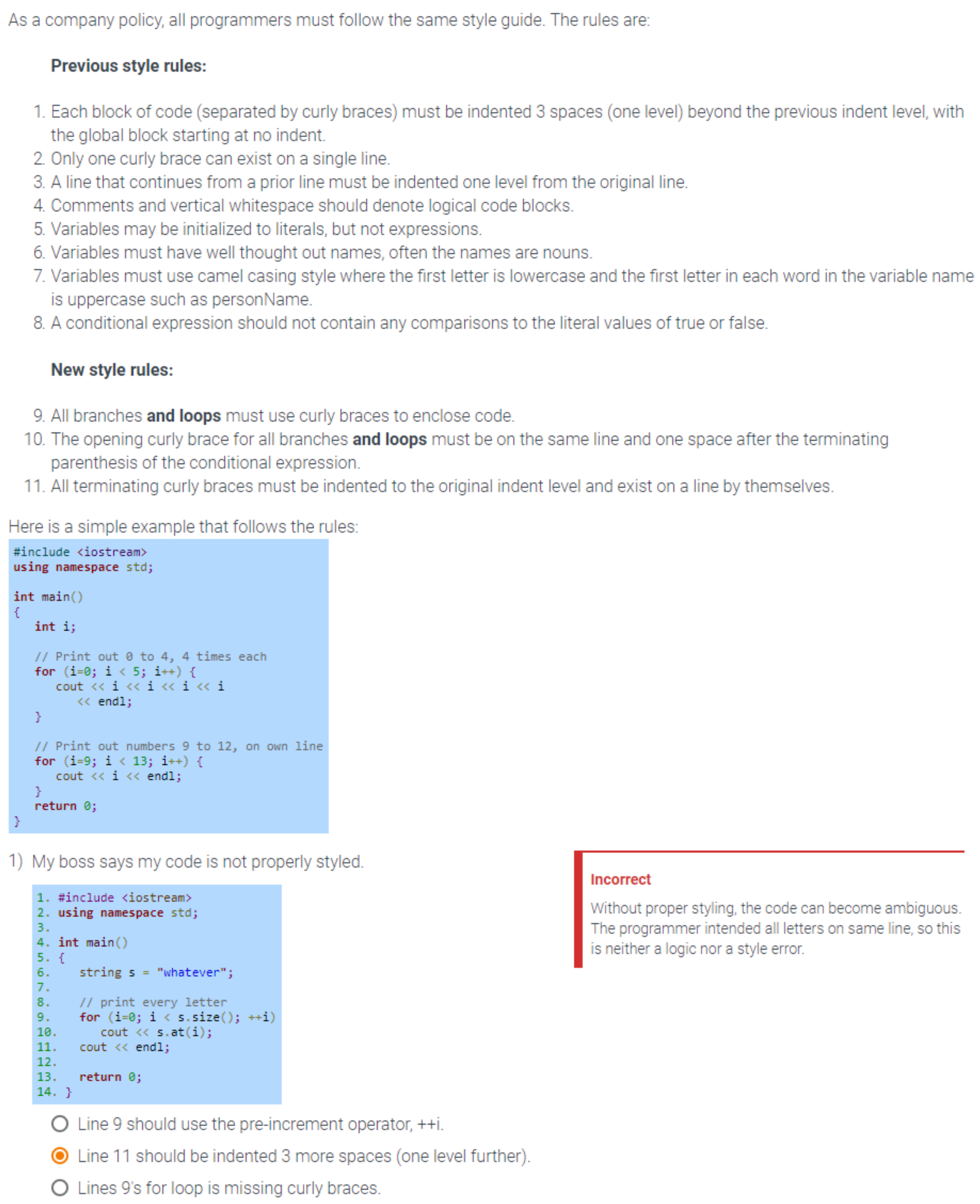
Line 11 should be indented 3 more spaces (one level further).

Lines 9's for loop is missing curly braces.

Incorrect

The company's style guide does not choose one increment over the other, but many choose ++i.

Figure 198: Style Exercise #7 - Style with Loops, Answer #1



As a company policy, all programmers must follow the same style guide. The rules are:

Previous style rules:

1. Each block of code (separated by curly braces) must be indented 3 spaces (one level) beyond the previous indent level, with the global block starting at no indent.
2. Only one curly brace can exist on a single line.
3. A line that continues from a prior line must be indented one level from the original line.
4. Comments and vertical whitespace should denote logical code blocks.
5. Variables may be initialized to literals, but not expressions.
6. Variables must have well thought out names, often the names are nouns.
7. Variables must use camel casing style where the first letter is lowercase and the first letter in each word in the variable name is uppercase such as personName.
8. A conditional expression should not contain any comparisons to the literal values of true or false.

New style rules:

9. All branches **and loops** must use curly braces to enclose code.
10. The opening curly brace for all branches **and loops** must be on the same line and one space after the terminating parenthesis of the conditional expression.
11. All terminating curly braces must be indented to the original indent level and exist on a line by themselves.

Here is a simple example that follows the rules:

```
#include <iostream>
using namespace std;

int main()
{
   int i;

   // Print out 0 to 4, 4 times each
   for (i=0; i < 5; i++) {
      cout << i << i << i << i
         << endl;
   }

   // Print out numbers 9 to 12, on own line
   for (i=9; i < 13; i++) {
      cout << i << endl;
   }
   return 0;
}
```

1) My boss says my code is not properly styled.

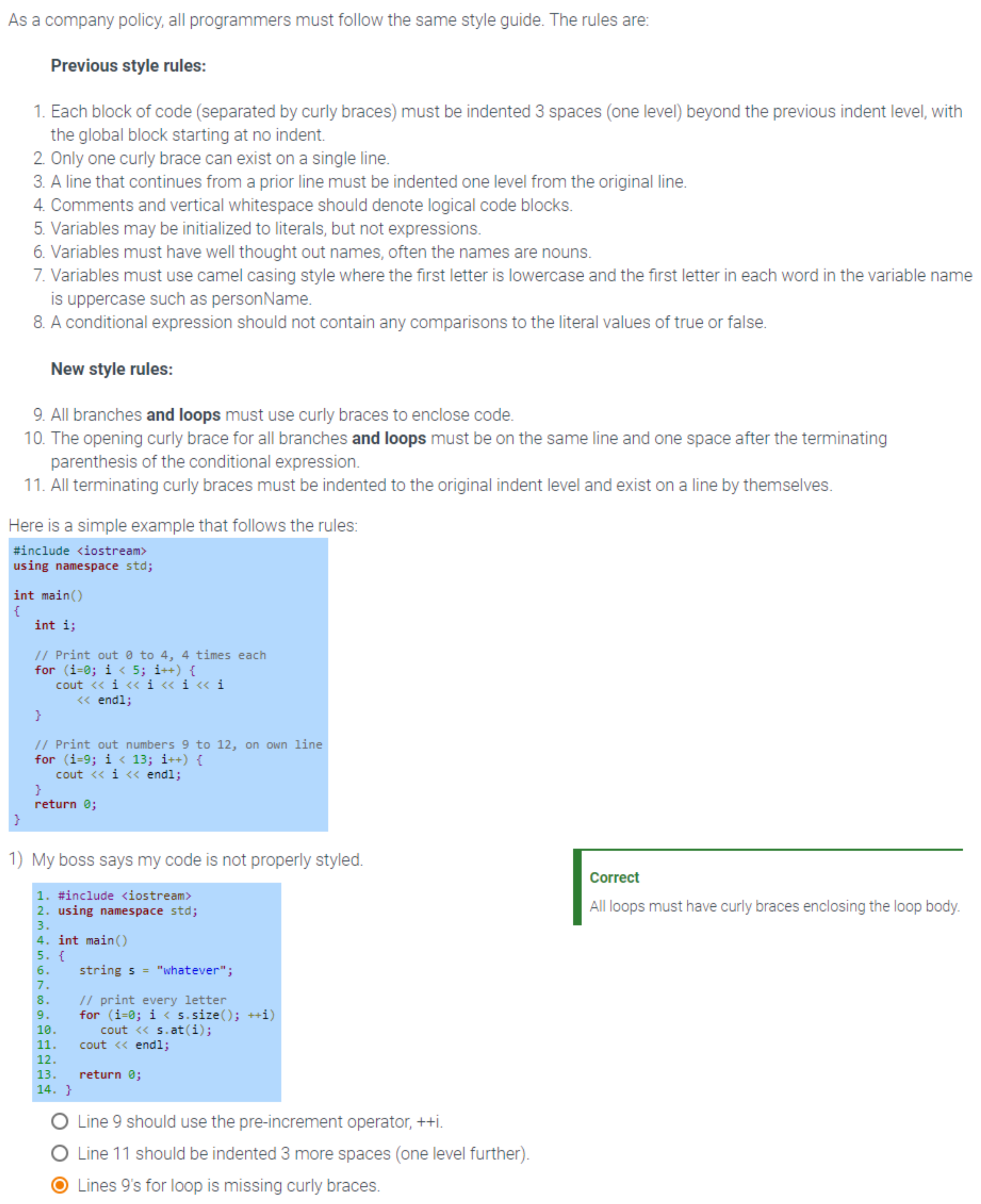
```
1. #include <iostream>
2. using namespace std;
3.
4. int main()
5. {
6.    string s = "whatever";
7.
8.    // print every letter
9.    for (i=0; i < s.size(); ++i)
10.      cout << s.at(i);
11.   cout << endl;
12.
13.   return 0;
14. }
```

- Line 9 should use the pre-increment operator, ++i.
- [✓ Line 11 should be indented 3 more spaces (one level further).]
- Lines 9's for loop is missing curly braces.

Incorrect

Without proper styling, the code can become ambiguous. The programmer intended all letters on same line, so this is neither a logic nor a style error.

Figure 199: Style Exercise #7 - Style with Loops, Answer #2



As a company policy, all programmers must follow the same style guide. The rules are:

Previous style rules:

1. Each block of code (separated by curly braces) must be indented 3 spaces (one level) beyond the previous indent level, with the global block starting at no indent.
2. Only one curly brace can exist on a single line.
3. A line that continues from a prior line must be indented one level from the original line.
4. Comments and vertical whitespace should denote logical code blocks.
5. Variables may be initialized to literals, but not expressions.
6. Variables must have well thought out names, often the names are nouns.
7. Variables must use camel casing style where the first letter is lowercase and the first letter in each word in the variable name is uppercase such as personName.
8. A conditional expression should not contain any comparisons to the literal values of true or false.

New style rules:

9. All branches **and loops** must use curly braces to enclose code.
10. The opening curly brace for all branches **and loops** must be on the same line and one space after the terminating parenthesis of the conditional expression.
11. All terminating curly braces must be indented to the original indent level and exist on a line by themselves.

Here is a simple example that follows the rules:

```
#include <iostream>
using namespace std;

int main()
{
   int i;

   // Print out 0 to 4, 4 times each
   for (i=0; i < 5; i++) {
      cout << i << i << i << i
         << endl;
   }

   // Print out numbers 9 to 12, on own line
   for (i=9; i < 13; i++) {
      cout << i << endl;
   }
   return 0;
}
```

1) My boss says my code is not properly styled.

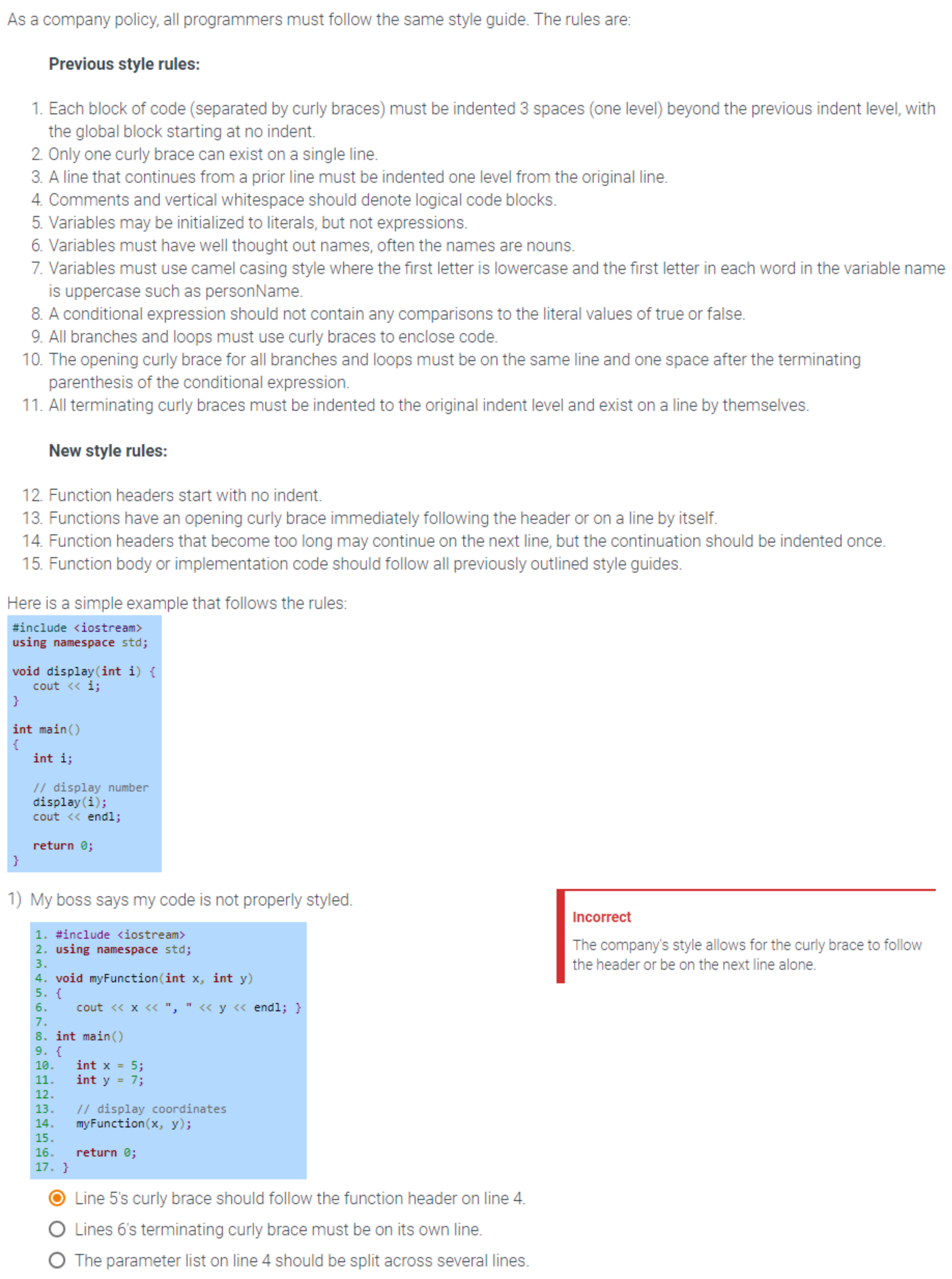
```
1. #include <iostream>
2. using namespace std;
3.
4. int main()
5. {
6.    string s = "whatever";
7.
8.    // print every letter
9.    for (i=0; i < s.size(); ++i)
10.      cout << s.at(i);
11.   cout << endl;
12.
13.   return 0;
14. }
```

- Line 9 should use the pre-increment operator, ++i.
- Line 11 should be indented 3 more spaces (one level further).
- [✓ Lines 9's for loop is missing curly braces.]

Correct

All loops must have curly braces enclosing the loop body.

Figure 200: Style Exercise #7 - Style with Loops, Answer #3



As a company policy, all programmers must follow the same style guide. The rules are:

Previous style rules:

1. Each block of code (separated by curly braces) must be indented 3 spaces (one level) beyond the previous indent level, with the global block starting at no indent.
2. Only one curly brace can exist on a single line.
3. A line that continues from a prior line must be indented one level from the original line.
4. Comments and vertical whitespace should denote logical code blocks.
5. Variables may be initialized to literals, but not expressions.
6. Variables must have well thought out names, often the names are nouns.
7. Variables must use camel casing style where the first letter is lowercase and the first letter in each word in the variable name is uppercase such as personName.
8. A conditional expression should not contain any comparisons to the literal values of true or false.
9. All branches and loops must use curly braces to enclose code.
10. The opening curly brace for all branches and loops must be on the same line and one space after the terminating parenthesis of the conditional expression.
11. All terminating curly braces must be indented to the original indent level and exist on a line by themselves.

New style rules:

12. Function headers start with no indent.
13. Functions have an opening curly brace immediately following the header or on a line by itself.
14. Function headers that become too long may continue on the next line, but the continuation should be indented once.
15. Function body or implementation code should follow all previously outlined style guides.

Here is a simple example that follows the rules:

```
#include <iostream>
using namespace std;

void display(int i) {
   cout << i;
}

int main()
{
   int i;

   // display number
   display(i);
   cout << endl;

   return 0;
}
```

1) My boss says my code is not properly styled.

```
1. #include <iostream>
2. using namespace std;
3.
4. void myFunction(int x, int y)
5. {
6.    cout << x << ", " << y << endl; }
7.
8. int main()
9. {
10.    int x = 5;
11.    int y = 7;
12.
13.    // display coordinates
14.    myFunction(x, y);
15.
16.    return 0;
17. }
```

[✓ Line 5's curly brace should follow the function header on line 4.]

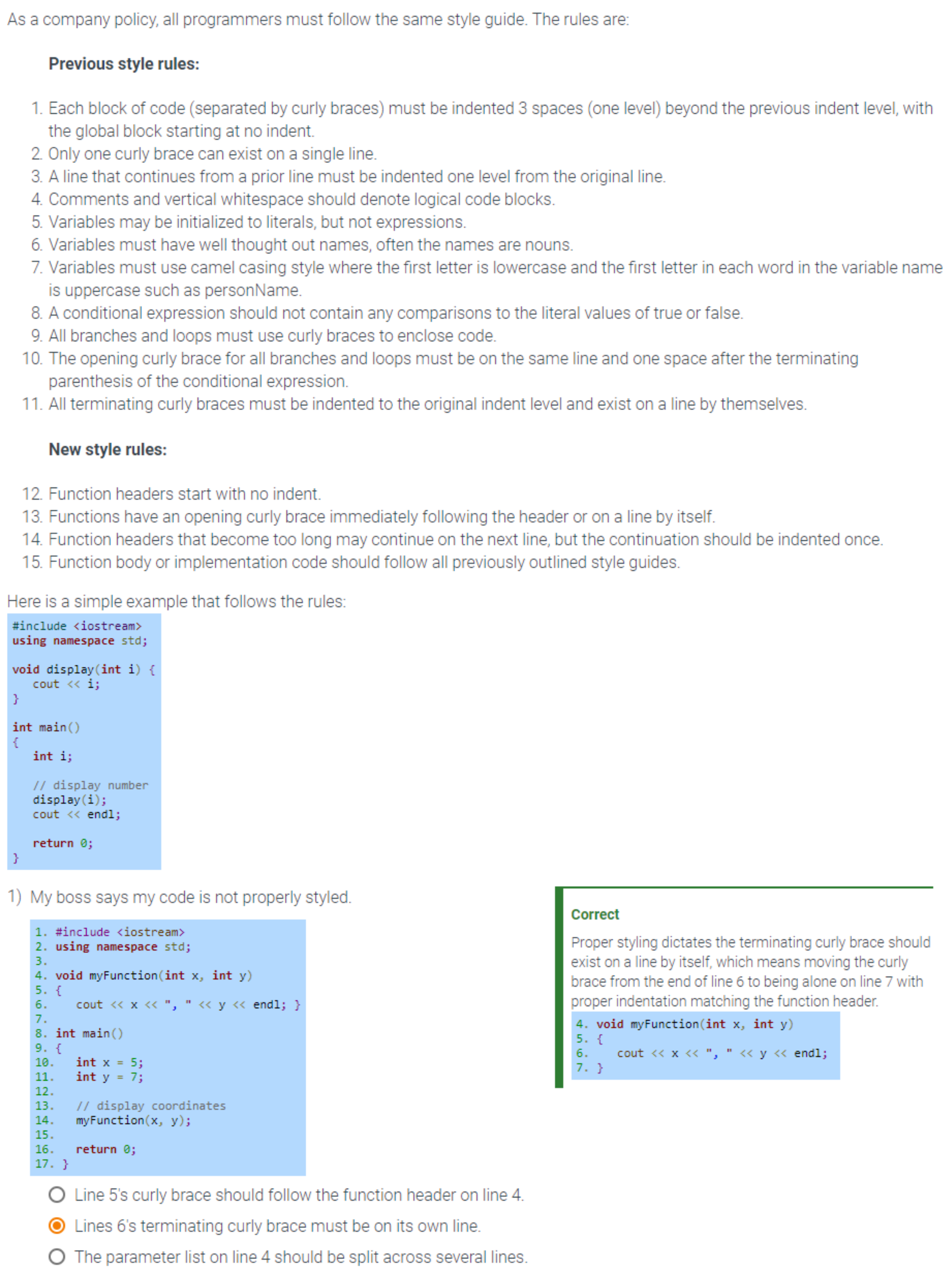
Lines 6's terminating curly brace must be on its own line.

The parameter list on line 4 should be split across several lines.

Incorrect

The company's style allows for the curly brace to follow the header or be on the next line alone.

Figure 201: Style Exercise #8 - Style with Functions, Answer #1


As a company policy, all programmers must follow the same style guide. The rules are:

Previous style rules:

1. Each block of code (separated by curly braces) must be indented 3 spaces (one level) beyond the previous indent level, with the global block starting at no indent.
2. Only one curly brace can exist on a single line.
3. A line that continues from a prior line must be indented one level from the original line.
4. Comments and vertical whitespace should denote logical code blocks.
5. Variables may be initialized to literals, but not expressions.
6. Variables must have well thought out names, often the names are nouns.
7. Variables must use camel casing style where the first letter is lowercase and the first letter in each word in the variable name is uppercase such as personName.
8. A conditional expression should not contain any comparisons to the literal values of true or false.
9. All branches and loops must use curly braces to enclose code.
10. The opening curly brace for all branches and loops must be on the same line and one space after the terminating parenthesis of the conditional expression.
11. All terminating curly braces must be indented to the original indent level and exist on a line by themselves.

New style rules:

12. Function headers start with no indent.
13. Functions have an opening curly brace immediately following the header or on a line by itself.
14. Function headers that become too long may continue on the next line, but the continuation should be indented once.
15. Function body or implementation code should follow all previously outlined style guides.

Here is a simple example that follows the rules:

```
#include <iostream>
using namespace std;

void display(int i) {
   cout << i;
}

int main()
{
   int i;

   // display number
   display(i);
   cout << endl;

   return 0;
}
```

1) My boss says my code is not properly styled.

```
1. #include <iostream>
2. using namespace std;
3.
4. void myFunction(int x, int y)
5. {
6.    cout << x << ", " << y << endl; }
7.
8. int main()
9. {
10.   int x = 5;
11.   int y = 7;
12.
13.   // display coordinates
14.   myFunction(x, y);
15.
16.   return 0;
17. }
```

Line 5's curly brace should follow the function header on line 4.

[✓ Lines 6's terminating curly brace must be on its own line.]

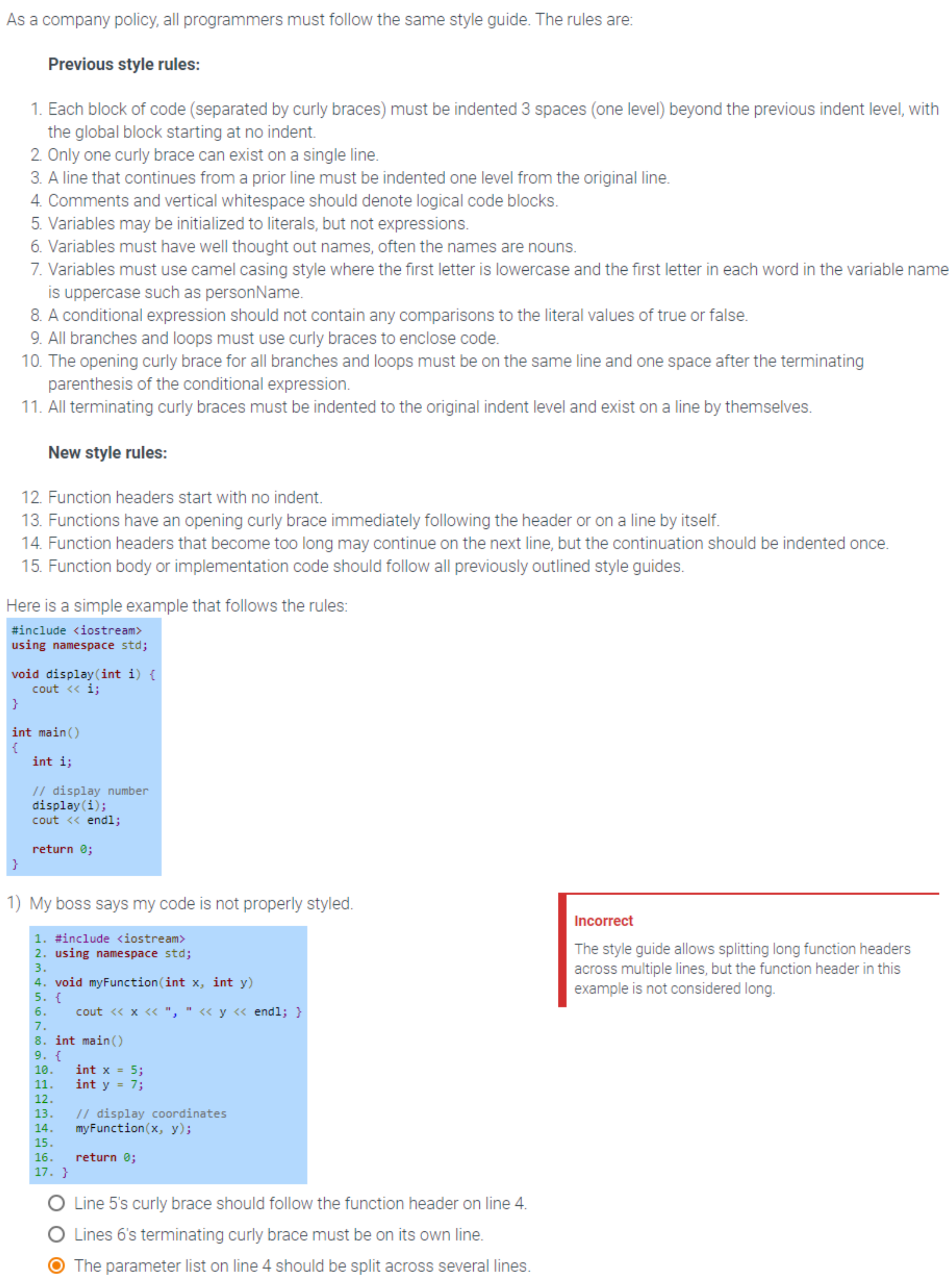
The parameter list on line 4 should be split across several lines.

Correct

Proper styling dictates the terminating curly brace should exist on a line by itself, which means moving the curly brace from the end of line 6 to being alone on line 7 with proper indentation matching the function header.

```
4. void myFunction(int x, int y)
5. {
6.    cout << x << ", " << y << endl;
7. }
```

Figure 202: Style Exercise #8 - Style with Functions, Answer #2



As a company policy, all programmers must follow the same style guide. The rules are:

Previous style rules:

1. Each block of code (separated by curly braces) must be indented 3 spaces (one level) beyond the previous indent level, with the global block starting at no indent.
2. Only one curly brace can exist on a single line.
3. A line that continues from a prior line must be indented one level from the original line.
4. Comments and vertical whitespace should denote logical code blocks.
5. Variables may be initialized to literals, but not expressions.
6. Variables must have well thought out names, often the names are nouns.
7. Variables must use camel casing style where the first letter is lowercase and the first letter in each word in the variable name is uppercase such as personName.
8. A conditional expression should not contain any comparisons to the literal values of true or false.
9. All branches and loops must use curly braces to enclose code.
10. The opening curly brace for all branches and loops must be on the same line and one space after the terminating parenthesis of the conditional expression.
11. All terminating curly braces must be indented to the original indent level and exist on a line by themselves.

New style rules:

12. Function headers start with no indent.
13. Functions have an opening curly brace immediately following the header or on a line by itself.
14. Function headers that become too long may continue on the next line, but the continuation should be indented once.
15. Function body or implementation code should follow all previously outlined style guides.

Here is a simple example that follows the rules:

```
#include <iostream>
using namespace std;

void display(int i) {
   cout << i;
}

int main()
{
   int i;

   // display number
   display(i);
   cout << endl;

   return 0;
}
```

1) My boss says my code is not properly styled.

```
1. #include <iostream>
2. using namespace std;
3.
4. void myFunction(int x, int y)
5. {
6.    cout << x << ", " << y << endl; }
7.
8. int main()
9. {
10.   int x = 5;
11.   int y = 7;
12.
13.   // display coordinates
14.   myFunction(x, y);
15.
16.   return 0;
17. }
```

○ Line 5's curly brace should follow the function header on line 4.

○ Lines 6's terminating curly brace must be on its own line.

[✓ The parameter list on line 4 should be split across several lines.]

Incorrect

The style guide allows splitting long function headers across multiple lines, but the function header in this example is not considered long.

Figure 203: Style Exercise #8 - Style with Functions, Answer #3