Autogenerated Questions Paper Draft

Contents

1 Abstract

2 Problem Description

maybe this is better as the abstract?

For any large introductory undergraduate course, testing becomes virtually impossible on a clerical level, especially for a small examination such as an in-class quiz. Creating, handing out, collecting, and grading tests can be very time consuming. However, technology allows instructors to develop and distribute exams in an online environment, easing the arduous task of passing out and collecting paper exams, yet extra problems are presented with online tests (especially those without a proctor), such as student access to publisher test banks, and – more importantly – cheating (Envisioning the use of online tests).

In an effort to mitigate cheating, it has been suggested that instructors randomize test items or use different questions for different students (Envisioning the use of online tests), but the task of creating exam questions (let alone multiple versions) has been noted to be very intensive and time-consuming (reference). Some solutions have been presented using NLP techniques and machine learning to generate questions from a corpus(A Proposed Framework for Generating Random Objective Exams using Paragraphs of Electronic Courses and Generating Diagnostic Multiple Choice Comprehension Cloze Questions), but these systems due not produce completely reliable questions or wrong answers. Another possible solution to this problem has been presented by (Creation of a dynamic question database for pharmacokinetics), which uses deterministic methods within Excel, but this implementation doesn't describe integration with other common online tools (such as Course Management Systems, or CMS), and it doesn't discuss important details such as the methods for generating distractors or the relative difficulty of the questions.

To address these points, we propose a system to be used for introductory programming classes at the undergraduate level, with a focus on syntax, control structures, and logic used in C++. This system offers a number of different categories, each with different levels of difficulty, that focus on fairly testing students. In addition, we will discuss distractor generation methods used within the system to more effectively test students.

3 Input/output Flow

Given the executable file and a templated file for input, it becomes extremely easy to generate any number of questions repeatedly. In most cases, we wish to generate a single category of questions at a time, and this is the basis for the overall structure of the main program. It simply checks the user's input from the predefined file to read basic inputs such as the main category and subcategory names (for identification in our CMS, Moodle), the filename for output, the number of total questions to generate, the question set – or category – to generate from (such as **Arrays**, **Functions**, etc.), the base difficulty, and the max difficulty of questions. By having this input file, it becomes easy to slightly modify any parameters and repeatedly run the program.

However, this method becomes tedious if one wishes to generate questions from every category. For this reason, the executable program also excepts 7 command-line arguments (that, in any other case, are cumbersome and virtually

unusable) which allows a small shell script to easily generate questions from every category in one command.

Once the program has finished, all output files generated in .txt format containing the GIFT-formatted questions, and they can be uploaded directly to the Moodle course with no extra effort. Each category is contained in its own file, and any difficulties selected within the category are automatically separated into their own question banks, allowing for exam questions to be selected by difficulty. This functionality is handled by Moodle.

4 Common Functions

Each question, regardless of category, follows the same format: the title (including unique question number), a prompt to explain the question, the right answer, and a number of wrong answers that can vary between categories. A standard question in GIFT is shown below.

```
What is the variable "size" equal to?
<pre> int item = 4;
int size = item / 3 - 3;</pre>
        {
        =-2
        ~-0.5
        ~-1.667
        ~1.5
        ~0
        }
```

Listing 1: A full question in GIFT format.

The title at the beginning (enclosed by two colons on each side) contains the subcategory name, "level" (i.e. difficulty) and a unique number (simply the current number of questions generated in this batch). This information is necessary in Moodle to differentiate between different question banks (e.g. "Dynamic Expressions Level 1") and different questions within the question bank (e.g. #005 vs. #006). For this reason, it is recommended that only one output file per category be used to avoid any confusion in the naming of question banks on Moodle.

The answer bank at the end (enclosed by the braces) contains all answer options for the question. The correct answer is preceded by the "=" and all wrong answers are denoted by the "∼". When either of these identifiers is used, a answer is taken to be any text until a line break.

Anything between the title and the answer bank is considered to be the prompt, the actual question being asked. GIFT format also allows HTML tags to be inserted, so usually `<pre>` tags are used to separate code excerpts for readability.

The reason we have chosen GIFT as the output markup language to these questions – as opposed to pure HTML or XML – is for a number of reasons:

this language was constructed specifically for use in our CMS, Moodle (footnote: `https://docs.moodle.org/23/en/GIFT_format`); GIFT has very minimal syntax, which allows it to be easily parsed and read by humans, which is useful when quickly evaluating the questions; and, similar to other courses (Generating Practice Questions as a Preparation Strategy for Introductory Programming Exams), the instructors of this paper require students to construct their own multiple choice questions that may be used in a real exam. The reason we chose Moodle is because of its support through our institution. (TODO: should I mention this either?)

Since we follow this basic structure for each question, we can abstract the implementation so each category only needs to generate a prompt and the corresponding answers. For this reason, the act of creating a new category of questions takes much less time to implement, and question-writers can be focused on the content of the question, rather than its final format. This also allows questions to have some certain specialized attributes (e.g. a different type of answer options, a true/false question, etc.), but most attributes are fixed or required, such as each question must have a title, a prompt, etc..

4.1 Difficulty

The most important of these fixed attributes are the static number of difficulties offered. Each question category is limited by four possible difficulties, however it is not necessary that all four difficulties are implemented by each category as long as it is made explicit to the user. Although this fixed number of difficulties appears to be a shortfall of this implementation (and it is a focus for improvement in the future), these difficulties offer a great deal of flexibility within a category of a question. The ability to randomize elements of a question, yet have students tested fairly and with the same relative difficulty, is one of the aims of this system.

A simple and informative example of difficulty is found in the **Expressions** category (see section 5.1.3) that simply serves to test a student's understanding of integer operations and variable assignment in programming. Each difficulty is the same essential type of question, but the number of operands increases with increasing difficulty. Some may consider this a naive approach to difficulty, but it is effective and showcases the fundamental relation between question categories and difficulties.

Other question categories – take **Switch Cases**, for example – don't exhibit such a straightforward notion of difficulty, but still offer interesting examples. Each difficulty tests students on the same essential concept of the control flow of a switch case statement by assigning a value to a variable (see section 5.3.1 for more detail). The first difficulty randomly generates break statements (for each case) and it may or may not include a default case, where the second difficulty always includes a break statement for each case and automatically includes a default case. Each difficulty of the **Switch Cases** category follows the same pattern by simply modifying random elements of the same core quesiton. While one may seem like more common – but not necessarily always correct – example

of a switch case block, it seems difficult to argue that one is inherently more "difficult" than another. However, it is clear that each difficulty of this category would be beneficial to test in its own right, while still being able to test students fairly.

In contrast to these categories that implement the same essential question with slight changes, some categories (such as **Functions** or **Arrays**), hold completely different questions in each difficulty. Here, a descriptor like "subcategory" seems more appropriate (since it becomes harder to evaluate question on difficulty in relation to a different question), but to avoid confusion, we will continue to reference these subcategories as difficulties. At first, it also seems like these questions shouldn't be included in the same category, as each may be warranted their own category. However, because each relates to the same question bank category (by testing the same fundamental concept) and any less or extra randomization in the question would not provide anything beneficial, but rather would shift focus from the objective of the question. We won't discuss in detail how a level of randomness affects a question, but there is a certain amount of randomness in a question that no longer serves the purpose of making a question unique, but instead it makes a question unnecessarily difficult to comprehend. In short, these different difficulties are not substantial enough to create their own category and are similar enough to each other to be included in the same category.

Although it now may seem that this concept of difficulty is unclear and unfocused, it is important to note that each of these in Moodle are formed as their own question bank, meaning students are each given questions from the same bank (therefore the same difficulty) and are not simply given questions of arbitrary difficulty within the same category, which suggests that different students receive different difficulties. The instructor then has the option of using whichever question bank (i.e. difficulty) is most appropriate, and it offers a fair test to each student. Overall, it gives the instructor more choice on how to exactly test his or her students.

4.2 Convincing Distractors

One of the most interesting aspect of this system, and one of its main goals, is to produce convincing distractors – that is, seemingly plausible, yet wrong, answers offerred as an option to the student. Distractors are essential to all multiple choice questions, for any options that students can easily identify as wrong take away from the difficulty of a question, and ultimately, does not add to the measure of knowledge of a student. Any blatant distractors make it easier for students to select (or guess) the correct answer.

This notion is also present in other systems (Generating Diagnostic Multiple Choice Comprehension Cloze Questions), but with varying success. This implementation is novel for similar systems based on deterministic methods (pharmacy reference). Between categories, and sometimes between difficulties, there are different methods for generating distractors, specific to each question. We discuss all of the methods below within each category. Here are examples

within a question using GIFT format.

```
What is the variable "bar" equal to?
int foo = 2;
int bar = foo - 4 / 3 + 4;
{
        =5
        ~4
        ~4.67
        ~2
        ~3.33
}
```

Listing 2: A question with distractors in GIFT format.

The first distractor method disregards the order of operations and simply evaluates the expression from left to right. The next method uses float operands and operations (a common mistake for beginners), and the third method evaluates the expression from right to left. The fourth and final method combines the first two, by disregarding the order of operations and using float operands and operations.

While our main goal is to have every distractor be, not only plausible, but convincing, it becomes difficult when very general methods are used, as it can be the case that the same answer occurs twice. We will now discuss how to prevent duplicates and guarantee that all answers are unique.

4.3 Unique Answer Guarantee

Because of the universal format for questions, extra error checks can be done across all categories. While there is no general way to ensure that each question provides the correct answer (other than unit testing), we can confirm that each answer is unique and, especially, that no distractor is the same as the right answer – a seemingly trivial, but essential, restraint. If there are repeating answers, then either the right answer is repeated as a wrong answer (in which case, a student may be penalized for picking the right answer), or a wrong answer is repeated twice (most students will disregard this answer as false). Both of these situations results in an unusable question, and this is unacceptable.

TODO: do a fancy list thing here Producing unique answers can become complicated is for two reasons: i) each category generates its own distractors using "clever" methods (e.g. evaluating an expression without order of operations, or evaluating an expression based on wrong datatypes) and ii) complex answers (in particular, answers containing vectors as in **Arrays**) become more difficult to compare and even more difficult to generate an arbitrary new distractor (for the sake of uniqueness) that is still a somewhat plausible answer. In the case of (i), it is fairly simple to compare all answers, and any distractors that are not unique are reassigned a random value close to the answer. This is usually done with computational questions where the answer is simply a numerical value. However, in the case of (ii), if we were to simply alter a value

in the vector, the distractor would become completely implausible because the answer only contains values that are originally in the array (see **Arrays Difficulty 3** in section 5.3.2). It would make no sense at all to change a value in the array. In order to generate distractors that are still plausible (but are guaranteed to be unique), we can simply shuffle the order of the vector. Yet, this has another precondition that the number of permutations for the correct answer must be greater than or equal to the number of available answers. Without this restraint, there would not be enough unique answers. If this is not met, then it suffices to revert to the original strategy and simply assign a random value to the vector. Although it may not be very convincing, unique distractors take precedence over plausible ones.

4.4 Prevention of Trivial Cases

While this is not a main focus of this paper, it should be noted that there are measures taken within each question to prevent trivial cases. For example, simple expressions such as `1 * x - x` or a loop body that never executes (in the context of counting the number of iterations) are far too simple to test student knowledge, and this takes away from the objective of the question. We are able to set parameters when generating the questions that will hopefully restrict their occurrence, yet, due to the random nature of the questions, trivial questions may still appear. As a response, this system checks for predetermined, common trivial cases (such as a loop that iterates not enough) are changed to try and provide a more difficult question.

5 Categories

There are basically three types/waves of categories: introductory, loops (intermediate), and advanced. The names of these categories reference not only how they are implemented, but how they are presented to students. As with any course, it is imperative to test student knowledge on material as it is presented to ensure that learning is occurring and prerequisite concepts are understood well enough to proceed. In this section, we'll give a more in-depth explanation of each category, their subcategories, and how some categories are related, based on their order of relative difficulty of content and implementation.

TODO: another fancy list thing here There are four essential components we wish to answer with each of these categories: a) what is the goal of the question; what is it asking? b) an example question (removed from GIFT format) for easy reference, c) what parts of the question are random? which of those parts are essential to the question (i.e. what parts affect the answer or structure of the question)? which of those parts are non-essential (i.e. what parts have no affect over the question whatsoever)? and are there any essential parts of the question that are not randomized (i.e. are there any constant factors of the question)? and lastly d) what methods are used to generate the distractors? It is important to ask (a) so each question can have a definitive objective, and it should be easy

to note if certain randomness adds/hinders that objective or if the randomness does not preserve a comparable difficulty for different students. This basically tries to prevent elements making a question "too random". It is also important to ask [TODO: should I include this part c? is it talked about enough later on?] (c) for a deep understanding of how the question is randomized – and how some parts are kept constant – to produce independent yet similarly difficult questions. This part answers the main methodology of all question generation. And as a consequence, (d) becomes relevant to ask to offer convincing distractors for a cohesive question.

The first wave we will examine is introductory questions.

5.1 Introductory Questions

The categories included in this wave are **Identifiers**, **Conditional Expressions**, and **Expressions**. Each of these are topics are presented fairly early, and their main focus includes – explicitly and implicitly – basic ideas such as syntax, variable access, integer operations, logical operations, and operator precedence. The first two categories are simply true/false questions, so there are no sophisticated methods of generating distractors, and this is part of the reason these categories are considered simple (from the point of view of implementation). However, these categories are still worth discussing for their testing methods and more of a concrete introduction. Let's take a detailed look at each category.

5.1.1 Identifiers

This first category is the simplest of all the questions here. The main objective of the question is to test students' knowledge on identifier syntax in C++. All difficulties are also the same, just with a differing length of string, which only contributes a small amount to the essence of the question, so we'll consider them all at once. Here is an example of the true/false question:

```
Is the following a valid C++ identifier?
Tb306wN
```

Listing 3: A question on identifiers.

The reason this question is so simple is because there are not a lot of different elements that attribute to the question; there is only the identifier. Furthermore, as noted before, the true/false nature of the question obviously limits the number of available answers and doesn't allow for convincing distractors. The entire string of the identifier is randomized. First, the length is selected (between 5 and 9 characters over all difficulties), and then each character is randomly selected from a group (either a numeric character, alpha character, or an invalid character). There is a 1 in 5 chance that the first character is a number (and therefore invalid) and a 1 in 10 chance for every character that it is an invalid character. Consequently, a string of 5 characters has a 45.9% chance of being valid, and a string of length 9 has a 30.1% chance of being valid.

NOTE: should i add something on how this adds to difficulty? I'm not sure how to close out this section

5.1.2 Conditional Expressions

This category is also fairly simple, mostly because it is another true/false question, but it does introduce some of the structure that we will reuse in other categories (mostly the expression structure). The aim for this category is to have students evaluate a conditional expression, which tests logical operations and their precedence. It also implicitly tests variable access, but this concept should be understood by this point, and it does not affect the question greatly. Listing 4 serves as an example.

```
What is the result of the following logical expression?
int numbers = 3;
int marks = 8;
(numbers <= 4) == (marks <= 10)
```

Listing 4: Conditional Expressions Question

The only important random elements are the operand values (each is between 1 and 10) and all operators. It should first be noted that the structure of the expression is always in the form of

```
(<sub-expression>) <logical operator> (<sub-expression>)
```

where each sub-expression is of the form

```
<variable> <comparison operator> <literal>
```

Each comparison operator is chosen randomly from `(>, >=, <, <=)` and each logical operator is chosen randomly from `(==, !=, &&, ||)`.

Each difficulty does not differ greatly from the other. Difficulties 1 and 3 always have 2 sub-expressions, and difficulties 2 and 4 always have 3 sub-expressions. The other difference between difficulties 1 and 2 from 3 and 4 is that the latter two difficulties have a random order of operands in the expression. See listing 5 as an example.

```
int documents = 8;
int bar = 10;
(bar <= documents) != (10 < 9)
```

Listing 5: Conditional Question with Random Order of Operands

This does not offer any significant difficulty change, but it does alter a format that students may be familiar with, and does a better job of testing the variable accesses.

5.1.3 Expressions

This category is arguably the most important of all, as it serves as a basis for all other, more complex categories, and it is even utilized in other other categories (namely **Functions**). It still follows the basic structure as **Identifiers** and **Conditional Expressions** (with simple prompts), but it introduces methods used to generate convincing distractors. The main aim is to test students' ability to evaluate simple arithmetic expressions in C++ using integer operands and operations. Similar to **Conditional Expressions**, it also tests minor variable accessing. There is an example at listing 6.

```
What is the variable "bar" equal to?
int foo = 2;
int bar = foo * 4 % 3 + 7;
```

Listing 6: Expressions Question

Again, each difficulty does not change significantly (only the number of operands are increased), but these simple questions are sufficient for one last category for introduction. The main sources of randomness obviously come from the value (between 2 and 10) and number of operands (between 3 and 7 depending on difficulty). It is also important to note that operator precedence is implicitly tested (no parentheses are generated), but there is room to grow and generate interesting parenthetical expressions in the future that may make these types of questions more challenging for students. However, this does add the need to check for more trivial cases.

The most interesting change in this category from the last is the greater possibility of answers (that is, it is not a simple true/false question) that allow for new distractor methods to be implemented. Compared to later categories, these methods are not very complex, but they focus on common mistakes made by students, which makes for a more difficult question. The main purpose of these distractors (and really of all distractors) is to try and generalize common mistakes for convincing answers, just as an instructor would try to create with a single question.

TODO: fancy list thing below

In this instance, the four distractor methods alter how the expression is evaluated by 1) disregarding order of operations, 2) using float values and operations, 3) evaluating the expression from right to left, and 4) disregarding the order of operations and using the float values (a combination of 1 and 2). These methods, while not particularly novel or sophisticated, aid the main objective of the question in their deliberate misunderstanding of the main concepts. For example, any student that is unfamiliar with integer operations is certain to select the distractor provided by method 2. Of course, in some cases, certain methods would not provide a different answer than the correct one (as integer addition and float addition are similar, etc.), however different answers are still guaranteed by the unique answer checking (see Unique Answer Guarantee in section 4.3).

5.2 Loops

This wave of questions is usually offered to students next, although – implementation wise – it doesn't have many similarities to the previous wave other than general format. Instead, **Loops** provides a general template for all of its own categories. Since we may wish to test all types of loops (we will only consider for loops, while loops, and do while loops), we can create a general base for all types of loops and test the same questions in each (as the main difference between each is mostly syntax). We separate each of these loops by category so instructors have the option of choosing exactly what kind of loop they wish to test. Instead of outlining each category as in the previous wave, we will outline two different types of questions (called **Loop Counting** and **Loop Printing**) that then create a category for each type of loops (6 categories total). Implementation between categories of the same question type (e.g. For Loops Counting vs. While Loops Counting) have virtually the same implementation (except for an edge case in do while loops as readers can imagine), so we will not distinguish between them and instead only talk of the different question types.

5.2.1 Loop Counting

This first question type of loops is probably similar to questions used in most introductory programming courses, as it is the fundamental concept behind loops (especially for loops). Simply, these questions ask a student how many times a statement within a for loop is executed, based on the common structure of a "counter" variable that is updated each loop, and then compared to a terminating condition: the same structure as a for loop. This is the essence of the question at each difficulty, however, due to the many dependent parts of the question, many different forms of randomization (mostly pertaining to the domain) occur between each difficulty, hence the distinction.

Difficulty 1

This difficulty is the most common form of for loops, and, even to some new students, can be trivial to evaluate. However, it still ensures that the main concepts of loop syntax, terminating conditions, and updating statements. Tere is an example from the **For Loops** category at listing 7.

```
How many times does the '*' print?
for (int b = 4; b < 7; b = b + 1)
{
        cout << '*';
}
```

Listing 7: Loop Counting Difficulty 1 Example

To ensure each question is relatively easy, the comparison operator used is always "<" and the increment operator is always "+=". This still leaves many essential details to be randomized. First, the increment is chosen (it has the option of being either 1, 5, or 10 for simplicity), and then based on the increment,

the starting value of the variable is chosen (between 2 and 10 and scaled based on the increment). Finally, the value used for the terminating condition is selected, which is between 3 and 6 greater than the starting value (again, also scaled based on the increment). This last value also has a random value added to it (from 0 to the incremental value minus 1) to make the terminating value seem "less nice", That is, it is not guaranteed that the variable will be equal to the terminating value when it exits the loop. This guarantees that the loop will execute between 3 and 6 times, without making the actual aspects of the loop seem too formulaic.

Due to the nature of this question, there are not many elaborate distractors, yet they are still effective. Obviously, the first two distractors are $\pm$ 1 of the correct answer (the common off-by-one error), and the last two distractors are a random amount away from the original question ($\pm$ 4). To eliminate any obvious negative answers, the absolute value is used for each of these distractors.

Difficulty 2

To increase the effort, more comparison operators (`>, >=, <, <=`) are used for the terminating condition, and the subtraction operator is used in the update statement. Listing 8 shows an example from the **Do While Loops** category.

```
How many times does the '*' print?
int j = 53;
do
{
        cout << '*';
        j = j - 5;
} while (j >= 30);
```

Listing 8: Loop Counting Difficulty 2 Example

It uses the same domain of randomized values, except it swaps the initial value and the terminating value when the operator used to update the variable is "-=". There is also a guarantee to match the correct comparison operators (e.g. `>, >=`) with the proper operator in the update assignment (e.g. +) unless otherwise specified by the user. If they do allow such operators to be paired with each other, "improper" loops can occur. These are loops that run infinitely (or more precisely, a couple billion times due to integer overflow), and this has a possibility of being the correct answer. When this situation does occur, the correct answer becomes "Around a couple billion", and in all other cases this is used as a distractor. This option aids in the essence of the question by making it more difficult, and it also introduces the option for non-homogeneous answer types (as all other available options are integer values). Other than this new option, all distractor methods are the same as Difficulty 1.

Difficulty 3

Using the same structure as the previous difficulty, the most significant difference is that the update statements use the multiplication and division operators as opposed to the addition and subtraction operators. An example from the **While Loops** category is found at listing 9.

```
How many times does the '*' print?
int i = 13;
while (i >= 3)
{
        cout << '*';
        i /= 2;
}
```

Listing 9: Loop Counting Difficulty 3 Example

TODO: maybe the list thing here

As a consequence of using different operators in the update statement, a slightly different domain of random values is used for all parts. The increment is either 2 or 10, and the starting value is chosen from 1 to 4 (and of course scaled by the increment as before). The value used in the terminating condition is calculated by taking the incremental value and raising it to a power between 1 and 3. That value is then multiplied by the starting value to guarantee that the loop executes at least between 1 and 3 times. However, similar to in Difficulty 1, we wish to make values seem less formulaic, so a random value from 0 to the increment less 1 is added to the stopping value to make it appear "less nice". This may also add another iteration in the loop, so there is a final check to ensure the number of iterations is within the a defined amount, and the members are altered if not.

As in Difficulty 2, there is an option for "improper" loops, and the distractor methods are the same. The most interesting new aspect that this difficulty brings is the division operator, which tests students' ability on integer operations when updating variables, as well as for comparison. Overall, this difficulty is comparable to Difficulty 2, except it adds extra difficulty using new operations.

Difficulty 4

While still offering the main objective of all of these difficulties, this difficulty can appear to be fundamentally different. It uses the same generation methods as Difficulty 2, except it uses a nested loop for execution instead of a single loop. There is a concrete example from the **For Loops** category at listing 10.

Each of these loops is created independently (yet always within the same category), and then evaluated together to create the correct answer. The nested loop structure allows for many more interesting distractors, and there may be some debate on choosing the optimal distractors (as there is a lot of room for student errors/misunderstandings). For this section we have used four basic distractors where 1) is the number of iterations of the outer loop, 2) is the number of iterations of the outer loop + 1, 3) is the number of iterations of the outer loop + 1 and then multiplied by the number of iterations for the inner

```
How many times does the '*' print?
for (int i = 43; i >= 20; i = i - 10)
{
        for (int j = 56; j > 45; j = j - 5)
        {
                cout << '*';
        }
}
```

Listing 10: Loop Counting Difficulty 4 Example

loop, and 4) is the number of iterations of the outer loop + 1 multiplied by the number of iterations in the inner loop + 1. It is important to note that we only used the number of iterations of the first two distractors and we did not introduce the inner loop, as some students may recognize that these two values can be multiplied together to easily find the answer, without actually testing any skill.

5.2.2 Loop Printing

This next type of question in the **Loops** categories contains similarities to **Loop Counting**. It aims at the same essence, but it tries to test a deeper understanding of loops by printing out the "counter" variable at each iteration. Implicitly, it still tests the number of iterations the loop executes, but the main objective is to understand, explicitly, when a variable is incremented, how integer operations affect the update, and the value that makes the loop terminate. For the sake of questions not being too difficult, each loop only executes between 3 and 5 iterations, and there is no fourth difficulty including nested loops. Listing 11 is an example from the **Do While Loops** category.

```
What is the following output of the code?
int b = 61;
do
{
        cout << b << '␣';
        b = b - 10;
} while (b > 20);
```

Listing 11: Loop Printing Difficulty 2 Example

Each difficulty is generated using the exact same methods as **Loop Counting** but obviously has different distractor methods, yet the same distractors are used for each difficulty. The first distractor uses all of the same values as the correct answer, except it omits the first value. The second distractor uses the same values as the correct answer, but it appends one more iteration from the loop. The third distractor is the same values as the correct one, except it omits the last value. Finally, the fourth distractor combines the first two methods by

omitting the first value and appends an extra iteration to the end. While there are many options for generating distractors, these all seem to provide convincing answers that some students may create on their own.

5.3 Advanced

While this wave is not particularly more difficult than the others, we will refer to them as advanced because they are usually introduced near the end of a course, and they some categories also utilize other earlier categories. Some of these difficulties also use very involved methods for question generation, evaluation, and distractor generation – as well as the unique answer guarantee for **Arrays**.

5.3.1 Switch Cases

The first category in this wave has the same essence for each difficulty; the only significant differences are the random elements, which then determine the difficulty. This is similar to the **Loop Counting** types described before. For the sake of brevitiy, here is a description of all shared aspects, and we will later discuss each difficulty's subtleties. First, a general example is shown at listing 12.

```
What is the value of "bar" after the switch case block?
int bar = 4;
switch (bar) {
        case 2:
                bar = 5;
                break;
        case 4:
                bar = 6;
        case 1:
                bar = 2;
                break;
        case 3:
                bar = 0;
        case 0:
                bar = 1;
                break;
}
```

Listing 12: **Switch Cases** Example

This category's main objective is to test students' knowledge of the control structure through a switch case block. This focuses on the syntax of a switch case statment, how the conditional cases are compared, and how break statements function as opposed to fall through. Occasionally, there may be a default case as well (depending on difficulty).

All distractor methods are also shared between each difficulty, which may be room for improvement, but for now, these methods offer plausible answers

for each difficulty. The first distractor is the original value of the variable. The second distractor is always the value that is *compared* to in the first case (e.g. the value 2 in the example above) and is a good distractor to identify students that are not very familiar with the block. The next two distractors (to be used more generally) are based on some conditions.

The third distractor is the value assigned to the variable in the first matching case. In the example above, this would be the value 6 (as the variable starts with value 4). This is a particularly good distractor when there is not a break statement and fall through occurs. However, if there is no matching case value for the original variable, the third distractor is then the last assigned value *before* the default case, if there is a default case. If there is no default case, the third distractor is simply a random value from those that are assigned within the block (in the case above, this would be one of 5, 6, 2, 0, or 1).

TODO: should i even talk about this fourth distractor? it's really only values that are in the case statements, which aren't very interesting to those with experience The fourth distractor follows the same conditions as the third, except with different methods at each. First, if there is a matching case to the original variable value, then the fourth distractor is the next compared case value that has a break statement. In the example above, this corresponds to the value 1 (since it is the next compared value with a break statement). Again, this value does not make much sense to one with experience with switch case statements, but it may outline errors for those that are unfamiliar with them. If there is no matching case to the original variable, on the other hand, then the fourth distractor is the compared value in the case statement before the default case (if there is one) or the first case value.

As it can be seen, these methods are somewhat lengthy and complicated, but they offer methods that can be used in each difficulty. However, it can be seen that there are many cases where these distractors are the same value, which we rely upon the unique answer guarntee to solve. It is also important to note that there is a possibility of confusion between the values compared in the case statements, and the values that are assigned after each case statement. This is because, while each assigned value (including the original) is guaranteed to be unique, there is no guarantee that any value compared in a case statement is not the same as one assigned to the variable. This may be improved upon for added clarity in further iterations.

As for the generation of the question, each difficulty shares the domain of nearly all random elements. Namely, the number of case statements (between 3 and 5), all assigned values (between 1 and the number of case statements multiplied by 1.5), and the values compared in the case statement (with the same domain as the assigned values). The only differing randomness occurs in the break statements and default statements.

Difficulty 1

The first difficulty is probably the most common, or most "uniform" switch case statement. There is guaranteed that all cases have a corresponding break

statement, and there is always a default case. This is a useful difficulty when just introducing the syntax of switch case statements.

Difficulty 2

Here, there are no break statements, but always a default statement. This is a good test on the concept of fall through in a switch case block.

Difficulty 3

Similar to the last difficulty, this difficulty guarantees that there are no break statements, but that there is no default statement either. This is also useful for testing fall through, but it is interesting because it does not guarantee that the original value of the variable is changed (in the instance that it matches no case statement), which makes this question slightly more difficulty.

Difficulty 4

This final difficulty is completely random in all aspects. Instead of any guarantees, there is a 50/50 chance for each break statement to be generated, as well as a 50/50 chance for a default case to be present. This is best used when testing all concepts of switch cases including fall through, break statements, and a block with or without a default case. For that reason, this is regarded as the most difficulty question in the category.

5.3.2 Arrays

This category (along with the **Functions** category) differs greatly from the other types of categories and poses an almost altogether different type of category. The main difference being how difficulties are defined within it. As mentioned above in the Difficulty section (4.1), it is hard to alter a single aspect of these questions to make it more or less difficult, and for that reason each single difficulty in this category is a different question altogether – yet still ordered by their relative difficulty. Of course, all of these questions still pertain to array concepts as a whole, and it is much easier to separate them and even use them in different settings. But here, the concept of difficulties is slightly different than what has been mentioned previously.

Another major difference that is specific to the **Arrays** category is its guarantee of unique answers. Although some answers have already contained a vector of values (such as **Loop Printing**), the answers have an overwhelming chance of being unique (as nearly all vectors are of different sizes). In **Arrays**, however, some vector answers are all of the same length (some as short as 3) and contain all of the same values from a small domain. Solutions to guarantee uniqueness (while still trying to maintain plausible distractors) are discussed below in Difficulty 1. First, we will look at the question as a whole.

Difficulty 1

This first type of question aims to test student's knowledge on array accesses dealing with fetching values and assigning values. There is an example at listing 13.

```
What are the contents of the array after the following
    statements?
int bar[6] = {6, 5, 3, 8, 1, 9};
int temp;
temp = bar[4];
bar[4] = bar[1];
bar[1] = temp;
```

Listing 13: **Arrays** Difficulty 1 Example

Simple randomization methods are used in each aspect of the array, which makes for a clear, focused question. The size of the array is random (between 4 and 6), as well as its contents (unique values between 0 and 10), and the indices used to access the array (two different valid indices of the array).

The distractor methods used here are i) the original array, ii) a new array, made from both indices being subtracted by 1, iii) both indices are incremented by 1, and iv) the first index is subtracted by 1 and the second index is added by 1. Of course, all of these methods use the modulus operation to ensure that they are still all valid indices. Still, they offer convincing answers to students confused by the common "off-by-one" error.

However, this poses another problem. Since all values are modulus the size of the array, there may be some cases where the same answers appear. Usually, when we identify identical answers, we simply generate a new random answer that – while maybe not very convincing – is guaranteed to be unique. Doing the same here would be useless, as a completely new array would not even be considered the correct answer given the question and the same is true if we even generate a single new element. Any array that contains an element that is not present in the original array is a definite outlier and is in no way plausible. However, while it is still not sophisticated, a last resort may be shuffling the order of the original array. This way, it still contains all values of the original array while being unique. Again, any student familiar with the material is likely to quickly identify this as the wrong answer, but it is more plausible than our other methods.

Difficulty 2

This type of question, now different from the first, has an objective to combine array access/updates with loops and loop variables. It uses a for loop to update each element in the array. This is useful for testing basic syntax and concepts of combining loops with array accessing. See listing 14 as an example.

The randomization in this question is also fairly minimal. The for loop variable is predefined to start at 0 and increment by 1 until the variable is equal to the size of the array (given by the condition). The only random parts here are the size of the array (between 3 and 6), the amount that each array element

```
What are the contents of the array after the following?
int documents[6] = {3, 3, 10, 1, 1, 2};
for (int i = 0; i < 6; i += 1)
{
        documents[i] += 5;
}
```

Listing 14: **Arrays** Difficulty 2 Example

is updated by (1, 5 or 10), and the contents of the array (any value between 0 and 10).

The distractor methods are also very simple, here they are i) the original array, ii) an off-by-one error of updating the array (i.e. update by the value of 6 instead of 5), iii) an array as if it was updated using the loop variable, and iv) the contents of the array are decremented instead of incremented. No index errors are used in these distractors as it is clear that the whole array is updated, but these may be altered in the future to use this strategy for better distractors. As for the unique answer guarantee, the same method as Difficulty 1 is used. While in this case, it doesn't really offer better distractors, the use of this method is very minimal as the current distractors will theoretically never give identical answers.

Difficulty 3

This question also has the objective of testing correct array access, but it also has the possibility of out-of-bounds access that the students must identify. Instead of making this a simple one time access, a for loop is used similar to Difficulty 2 to iterate through the loop, yet in a less deterministic way. Listing 15 shows an example.

```
What is printed after the following statements?
int dogs[6] = {0, 1, 7, 3, 0, 2};
for (int i = 1; i < 4; i = i + 1)
{
        cout << dogs[i] << "␣";
}
```

Listing 15: **Arrays** Difficulty 3 Example

The number of random aspects also increases relative to Difficulty 2. All random elements include the size of the array (from 3 to 6), the contents of the array (from 0 to 10), the beginning value of the loop variable (from 0 to half the size of the array), the loop comparison operator (either < or <=), the terminating loop value in the comparison (the size of the array plus or minus 1), and the amount the loop variable is incremented (between 1 and 2). In order to minimize some randomness, the same methods used in the Loops category are used to ensure that each loop iterates at least 2 times and no more than 4

times. This means that some loop components may be changed, but this is a negligible cost to avoid loops that only iterate once, or some loops that iterate 6 times.

Distractors in this question include i) the loop variable is incremented an extra time (to simulate the next loop iteration, including one that never takes place), ii) the loop variable is decremented by one increment (to simulate a previous loop iteration), iii) the loop variable is subtracted by 1 (to simulate an off-by-one error), and iv) literally a "Run-time error due to out of bound index". The last distractor is a possibility, because the correct answer may sometimes be "Compile-time error due to out of bound index" based on the loop parameters. Then this not only tests student's ability on out of bounds accesses, but what specific type of error it generates. If this is too complex, the last distractor may be substituted for another vector for a more straightforward question.

The unique answer guarantee used here is also the same as Difficulty 1, but more constraints must be put in place. Because the array elements are not guaranteed to be unique, then the elements of the correct answer vector are also not guaranteed to be unique. Then there may be a case where the number of the permutations of this vector is not great enough that we can simply shuffle the answer. Take U to be the set of unique values in the correct answer (a vector called C). We then construct another vector, say V, that, for each element in U, counts the number of occurences in C. Then the factorial of the length of C divided by the product of the factorial of each element in V (i.e. the number of permutations) must be greater than or equal to 5 (the total number of answers).

$$V = \text{Number of occurrences in } C, \forall u \in U$$

$$\text{Number of permutations} = \frac{length(C)!}{\prod_{v \in V} v!}$$

Here is a direct example:

```
C = {1, 1, 1, 2};
S = {1, 2};
V = {3, 1};
```

$$\text{Number of permutations} = \frac{length(C)!}{\prod_{v \in V} v!} = \frac{24}{6} = 4$$

In this case, there are not enough permutations of the answer to be used uniquely in each answer. If this case ever occurs, we finally give up and simply add a new random element to the vector.

5.3.3 Functions

This next wave has similar structure to that of **Arrays**, as each difficulty is its own question, yet each contains main concepts of functions as a whole. There

may be room to fine tune certain random elements to spread some questions over multiple difficulties to mirror the earlier categories, but for now, we will simply propose each difficulty on its own.

The main problem with implementing this category is finding a particular balance between randomness to fulfill our original goals, while still maintaining a simple, easily understandable question for students to answer. Some questions can have a large degree of randomness, yet – at a certain point – this starts to take away from the question as it becomes hard to read and answer quickly, essential to these questions when used in a timed format. This can be illustrated in the further difficulties in the category, where some can contain very syntax-heavy code snippets.

Difficulty 1

The objective of this first question is to simply test return types and any type coercion that takes place. It features a function with no parameters and a single return statement with a simple arithmetic statement. In the return expression, two float operands are used (any value between 1 and 10 in steps of 0.5) with a single operation (any of +, -, *, or /). In the future, there may be an opportunity here to randomize the amount of operands in the expression, but as it's not essential to the question, we have not yet considered it. There is an example at listing 16.

Given the following function definition:

```
int Pow()
{
        return 4.5 + 1.0;
}
```

What is printed from the following call?

```
cout << Pow() << endl;
```

Listing 16: **Functions** Difficulty 1 Example

In addition to the randomness described above, the essential random element is the return type of the function. Of course, a different return type will produce different answers for the same return expression, so this provides the main random element. The return type can be any of `int`, `float`, or `void`. In the case that the return type is `void`, the correct answer is actually a "Compile-time error", similar to some difficulties in Loop Printing.

As a consequence of this possibilty, one distractor is always "Run-time error" to test student knowledge on specific error types here. The other distractor methods include i) the float value of the expression evaluated from right to left and ii) if the return value is float, this uses the integer value (and vice versa for Int return types).

Overall, this difficulty can benefit from its relative simplicity to explicitly test student knowledge on function return types and type coercion.

Difficulty 2

This next difficulty is in the same vain as Difficulty 1, except it focuses on data types of parameters instead of the return type. This way, students are tested more on knowledge on parameter passing and type coercion, as well as the operations that result from them. Because of the similar objective of the two difficulties, a format that resembles Difficulty 1 is used here as well.

Given the following function definition:

```
float Switch(float a, int b)
{
    return a - b;
}
```

What is printed from the following call?

```
cout << Switch(3.5, 1.5) << endl;
```

Listing 17: **Functions** Difficulty 2 Example

Furthermore, similar randomization methods are used in this difficulty as well. That is, each literal ranges from 1 to 10 (ending in intervals of 0.5), the same operations are used, and a constant number of operands are used as well. The main difference in random elements here is obviously the parameters of the function (although the underlying implementation and evaluation is nearly equivalent) and the return type (which is a constant value of float to ensure the largest amount of precision, adding to the questions simplicity). Again, all of these static methods (such as number of operands and return type) can be randomized further to add extra difficulty. For the sake of simplicity (both in explanation and question difficulty), we will only consider this form.

TODO: decide about list thing

As for distractor methods, this question uses methods similar to the first difficulty (which are also similar to those in **Expressions**), but the most interesting methods here include i) evaluating the expression with all float operations (regardless of any previous type coercion), ii) evaluating the expression with all the original float values (essentially ignoring type coercion), and iii) using the floor of the correct answer (to provide an integer-like answer). Of course, as this question is furthered, these methods will have to be slightly altered to try and provide more convincing distractors in a general case (as in an arbitrary number of operands or an arbitrary return type).

Difficulty 3

The last difficulty of **Functions** is the most difficult, not only due to content, but simply because of the amount of syntax contained within the prompt. Of course, students should be farily competent in reading code at this point in the course, but it would be preferable to use a simpiler, yet still relatively challenging, strategy to test this question.

```
Given the following function definition:
void Calculate(int& a, int& b, int c)
{
        b = a / b + c;
}

What value is printed after the following statements?
int marks = 5;
int matrix = 4;
int item = 5;
Calculate(marks, matrix, item);
cout << matrix << endl;
```

Listing 18: **Functions** Difficulty 3 Example

The main objective here is to test students on reference parameters . This is possible to test with only one variable, but that limits the difficulty (as well as the overall randomness) that is the goal of these questions. Conversely, some may argue that three parameters is too complex in itself. In the current implementation, the number of variables (as well as the number of parameters) are constant, but can be easily changed and easily randomized if desired. Another option for added randomization is the datatype of all parameters and all variables (currently, only integers are used). The final static elements of this question are a) the order that the variables are defined in is the same order that the variables are passed to the function; b) the variable printed at the end of the prompt always corresponds to the parameter that is assigned in the function; c) the order that the parameters are defined in is the same order they appear in the expression; and d) the number of operands in the expression is the same as the number of parameters. All of these elements can be randomized, but again, they usually add difficulty without changing the essence of the question.

Despite all of these static elements, there are still many options for randomization. Most importantly, each parameter has a 50/50 chance of being pass-by-reference or pass-by-value and the parameter that is assigned in the function. In addition, the regular randomization methods for creating expressions (with random operators) and assigning operand values are used.

Because this question depends on the expression inside the function, some of the same distractor methods are used within this category, such as disregarding order of operations and using float values and float operations. The novel distractor methods are much more interesting. The first (and most obvious method) distractor being equal to the original value of the variable – if the parameter is pass-by-reference – or the value of the expression – if the parameter is pass-by-value. A fairly straightforward method, but effective in testing the objective of this question. The next distractor is simply the value of the variable declared after the printed variable (in the example above, this would be item).

This method is not as particularly sophisticated, but serves as an obvious wrong answer for knowledgeable students.

6 Future Work

TODO: list thing

The next research that should take place is in regards to actual use of the system that produces quantitative results. Any statistics would not only be helpful in evaluating the system as a whole, but also useful in identifying a) the efficacy of each category, b) a statistically-based distinction between difficulties within categories, and c) efficacy of certain distractor methods for a question. With these results, this system (and other systems) would be better able to produce and classify the questions within them, as well as produce more effective distractor methods. However, there are some ethical problems that will need to be addressed in some of this work. Namely, any test for difficulty between students may cause lower test scores than usual or what is considered "fair", which adversely and unintentionally affects students.

In addition to requiring data, simply adding to the system with more question categories, or improving each question method and each distractor method is always a potential focus.

7 Conclusion

The proposed system offers instructors a simple interface to easily generate and upload a multitude of multiple choice questions to be used in an online exam. With a unique question used for each student, this will hopefully reduce the presence of cheating in online exams. Along with the many different categories, this system distinguishes between relative difficulty of a question (to challenge and test all students fairly), and it also offers methods of generating convincing distractors to more effectively test student knowledge.