Midterm Review Worksheet Answer Key

CPSC121 Fall 2022

JC Dy and Paul Salvador Inventado

Introduction

This document is meant to provide you supplementary practice questions for the upcoming midterm. It reflects on material that you may have already seen in labs and lectures. Do not use this as a “be all end all” guide! It is still highly recommended that you review previous course material.

First, I would recommend you read through and maybe take notes on the lecture slides (all in the format of tinyurl.com/cpsc121-f22-lectureX where X is the lecture number), YouTube videos, lecture worksheets, and all the quizzes we’ve done up until now. Remember that the tests we write will take from material that you’ve already seen!

Finally, make sure you don’t stress! This class is hard and will only be made harder if you stress yourself out. But you smart, you loyal #blessup. Ask lots of questions and let me and Paul help you (believe me when I say we want you all to succeed).

If you find yourself beginning to panic, simply pause, breathe, and believe. On the off chance you don’t believe in yourself, then believe in me who believes in you.

Variables & Syntax

Initialization & Assignment

Below each line, write **one** line of code in perfect C++ syntax to:

1. Initialize a variable named `num` holding an `int`, with value 121.
   ```
   int num = 121;
   ```

2. Initialize a variable named `hi` holding a `std::string`, initialized to `"hello"`.
   ```
   std::string hi = "hello";
   ```

3. Append the string `" world!"` to the `hi` variable.
   ```
   hi = hi + " world!";
   ```
   OR
   ```
   hi += " world!";
   ```

4. Accept user input using `std::cin` and assign it to `num`.
   ```
   std::cin >> num;
   ```

5. Initialize a variable named `area_codes` holding a `std::vector` containing `int`s, containing elements 626, 925, 510, and 818.
   ```
   std::vector<int> area_codes {626, 925, 510, 818};
   ```

6. Initialize a constant named `PI` set to the value 3.1415.
   ```
   const double PI = 3.1415;
   ```

7. Given a `std::vector<double>` stored in variable `grades`, initialize a new variable named `grade` set to the value of the 5th element in the vector.
   ```
   double grade = grades.at(4);
   ```

8. Reassign the first element in `grades` to the value 98.6.
   ```
   grades.at(0) = 98.6;
   ```

9. Given a 10 digit `int` stored in variable `phone`, initialize `area_code` to the **first 3 digits** of the value stored in `phone`. (example: 1018044000 becomes 101)
   ```
   int area_code = phone / 10000000;
   ```

10. Given a 9 digit `int` stored in variable `ssn`, reassign `ssn` to just the **last 4 digits** of what it contained. (example: 909772022 would become 2022)
    ```
    ssn = ssn % 10000;
    ```

Functions

Functions Declarations

A function declaration is composed of a function's **name**, **return type**, and **parameters**. For the following function declarations, answer each prompt.

1. Underline the return type in the following function declaration:

```
bool IsMusicVideo(int video_id);
```

2. Underline the parameter(s) in the following function declaration:

```
int CountCharFrequency(std::string text, char c);
```

3. Does the following function have a return value? ◯ Yes ◯ **No**
   The void keyword specifies that the function doesn't return a value.

```
void PrintItemsToDisplay(std::vector<Product> products);
```

Function Definitions

A function definition is composed of a function's **name**, **return type**, **parameters**, AND **the function's body**. Below each line, define a function to accomplish the task:

4. Define a function called **`IsEven`** that accepts an **`int`** named **`n`** and returns true if the number is even, false if it's odd.

```
bool IsEven(int n) {
  return n % 2 == 0;
}
```

5. Define a function called **`IsTeen`** that accepts an **`int`** named **`age`** and returns true if the number is between 13 and 19, inclusive.

```
bool IsTeen(int age) {
  return age >= 13 && age <= 19;
}
```

6. Define a function called **`IsEvenTeen`** that accepts an **`int`** named **`age`** and calls the two functions above, and returns true if **`age`** is BOTH even and a teenager.

```
bool IsEvenTeen(int age) {
  return IsEven(age) && IsTeen(age);
}
```

Function Calls

A function call is composed of a function name followed by the function call operator, (). If the function accepts input, the arguments passed in are listed inside the parentheses.

Listed below are several function declarations:

```
// Returns the given string reversed.
std::string Reverse(std::string text);

// Returns true if `num` can be found in `text`.
bool IsNumberInText(std::string text, int num);

// Prints out a random number from [1, n]
void DiceRoll(int n);

// Computes and returns the factorial of num.
int Factorial(int num);
```

Using the function declarations above, determine if the following function calls are valid. If it is syntactically correct and compiles, bubble in “Valid”, otherwise, bubble in “Invalid”. *Challenge: if the function call is invalid, describe what the error is or how you’d fix it.*

1. `std::string text = Reverse(“racecar”);`

   ◯ **Valid** ◯ Invalid

2. `text = Reverse(‘c’);`

   ‘c’ is a character (char) data type. Reverse only accepts a std::string.

   ◯ Valid ◯ **Invalid**

3. `IsNumberInText(“123”, 2);`

   While the return value is not stored anywhere, this is still syntactically correct.

   ◯ **Valid** ◯ Invalid

4. `bool exists = IsNumberInText("123", 2);`

   ◯ **Valid** ◯ Invalid

5. ```
   int n = 6;
   DiceRoll(int n);
   ```
   This is a common mistake made when calling a function. You may pass in the parameter to the function e.g. DiceRoll(n), but do not specify its data type in the function call.

   ◯ Valid ◯ **Invalid**

6. ```
   int roll = DiceRoll(6);
   ```
   DiceRoll is a void function.

   ◯ Valid ◯ **Invalid**

7. `int x = Factorial(5.0);`

   ◯ **Valid** ◯ Invalid

Let's say you have a function **SumDigits** that sums up all the digits in a given integer. It should take in as input one `int` parameter `**digits**`, and return an `int`.
For example, **SumDigits(31415)** should return 3+1+4+1+5 = 14.

Write the function **declaration** of **SumDigits** below:

```
// Returns the total sum of the digits in the given integer.

int SumDigits(int digits);
```

Complete the print statement and call your **SumDigits** function such that it outputs 22.
Note: there are many acceptable answers to this question.
This is one:

```
std::cout << SumDigits(25582);
```

Challenge: Sum Digits

Write a function that sums up all the digits in an integer iteratively (i.e. using a for or while loop). For example, **SumDigits(31415)** should return 3+1+4+1+5 = 14.

```
int SumDigits(int digits) {
  int total = 0;
  for (int num = digits; num > 0; num /= 10) {
    total += num % 10;
  }
  return total;
}
```

Challenge: What Would C++ Display?

For each of the statements in the table below, write the output displayed by C++ when the statement is executed. The output may have multiple lines. If an error occurs, write "Error", but include all output displayed before the error. For an extra challenge, indicate if an error is a compiler error or a runtime error.

Note: ***Each row is compiled and run independently of one other.***

The first two rows have been provided as examples. **Visualizer**

```
void Tik() {
   std::cout << "Tok";
}

bool Winter(int snow) {
   std::cout << "is coming\n";
   return snow > 1;
}

int Print4(std::string msg) {
   std::cout << msg[4];
   return msg.size();
}

int Foo(int num) {
   if (Winter(num)) {
      Tik();
      std::cout << "!";
   }
   return num / 0;
}
```

Statement	Output
`Tik()`	`Tok`
`std::cout << Winter(1);`	`is coming` `false`
`Winter(2022);`	**`is coming`**
`Print4("Disney")`	**`e`**
`Print4(Tik() + Tik())`	**Compiler error**
`Foo(3.14)`	**`is coming`** **`Tok!`** **Runtime error**
`Foo(3)`	**`is coming`** **`Tok!`** **Runtime error**
`Winter(Print4("titan"));`	**`nis coming`**

Classes & Objects

Class Syntax **Visualizer**

OOP allows us to treat data as objects, allowing us to represent real world objects in our code. For example, consider the class TV:

```
// TV represents a television set.
class TV {
    public:
        TV(int chnl, int vol) : channel_(chnl), volume_(vol) {}
        int GetChannel() { return channel_; }
        int GetVolume() { return volume_; }

    private:
      // Represents the current channel.
      int channel_;
      // Represents the current volume out of 100.
      int volume_;
};
```

1. We currently have **accessor functions**, or getters, `GetChannel` and `GetVolume`, but we have no **mutator functions**, or setters, to let us change the channel or set the volume. Select the missing mutator function for the channel.

   ◯ `void SetChannel(int channel) { channel = channel_; }`

   ◯ **`void SetChannel(int channel) { channel_ = channel; }`**

2. Is a default constructor provided in the TV class?

   ◯ Yes, the compiler always implicitly provides a default constructor.

   ◯ **No, a non-default constructor was provided, so a default constructor is not provided by the compiler.**

3. Details that all TVs have, such as its current channel, are called **member variables**. Select all the member variables of the TV class below:

   ◯ `volume` ◯ **`volume_`** ◯ `vol` ◯ `GetVolume` ◯ `public` ◯ `class`

   ◯ `channel` ◯ **`channel_`** ◯ `chnl` ◯ `GetChannel` ◯ `private`

4. All TVs can flip to a channel, or change its volume. These are behaviors, or functions, that belong to a specific TV, so we call them **member functions**. How many member functions are defined in the TV class?

   `The TV constructor, GetChannel, and GetVolume are defined.`

   ◯ 1 ◯ 2 ◯ **3** ◯ 4 ◯ 5

Instantiating Objects

Below we have defined the classes `Professor` and `Student`. **Visualizer**

```
class Student {
    public:
        Student(std::string name)
            : name_(name), iq_(0) {
          std::cout << name << " joined!\n";
        }
        std::string GetName() { return name_; }
        int GetIq() { return iq_; }
        void SetIq(int iq) { iq_ = iq; }

    private:
      std::string name_;
      int iq_;
};
```

```
class Professor {
    public:
        void HostOfficeHours(Student &student) {
          student.SetIq(student.GetIq() + 1);
          std::cout << student.GetName() << " leveled up!\n";
        }
};
```

What will the following lines output? Write the output, if any, to the right of each line.

```
Professor jc;                    // Output:
Student harry("Harry");          // Output: Harry joined!
Student ron("Ron");              // Output: Ron joined!

jc.HostOfficeHours(harry);       // Output: Harry leveled up!
std::cout << harry.GetIq();      // Output: 1
std::cout << ron.GetIq();        // Output: 0

Professor paul;                  // Output:
paul.HostOfficeHours(harry);     // Output: Harry leveled up!
std::cout << harry.GetIq();      // Output: 2
ron.SetIq(harry.GetIq());        // Output:
std::cout << ron.GetIq();        // Output: 2
```

Challenge: Functions as Objects

In some programming languages (e.g. Python), functions can be treated like first-class objects. This means, we can assign a function to a variable, pass a function in as a parameter, or even return a function as output. Unfortunately, in C++ this isn't possible… unless?

Let's use what we know about object-oriented programming to get around this.

Consider `SquareFunction`, a class that represents a function that squares an input argument:

```
// SquareFunction represents a function that squares ints.
class SquareFunction {
    public:
        void setArg(int arg) { arg_ = arg;}
        int apply() { return arg_ * arg_; }
    private:
      // Represents an incoming argument.
      int arg_;
};
```

We can now apply it in a functional manner!

```
int main() {
    std::vector<int> inputs {1, 2, 3, 4, 5};
    SquareFunction sf;
    for (int i = 0; i < inputs.size(); i += 1) {
        sf.setArg(inputs.at(i));
        inputs.at(i) = sf.apply();
    }
    // now inputs = {1, 4, 9, 16, 25};
}
```

Challenge: define a class representing the `isMultipleOf` function. That is, your object should be able to take in an integer and N, returning the true if it is a multiple of N else returning false.

```
class IsMultipleOfFunction {
    public:
        void setArg(int arg) { arg_ = arg;}
        void setMultiple(int mul) { multiple_ = mul;}
        bool apply() { return arg_ % multiple_ == 0; }

    private:
        int arg_;
        int multiple_;
};

Visualizer
```

Vectors

Vector Constructors & Functions

Below each line, draw the state of the vector after executing the statement. The first two have been done for you. **Visualizer**

1. `std::vector<string> countries;`

countries:	{ }

2.
```
countries.push_back("Mexico");
countries.push_back("Philippines");
countries.push_back("Canada");
```

countries:	{"Mexico", "Philippines", "Canada"}

3. `countries.at(1) = "Spain";`

countries:	{"Mexico", "Spain", "Canada"}

4. `countries.clear();`

countries:	{ }

5. `std::vector<int> zip_codes(4);`

zip_codes:	{0, 0, 0, 0}

6.
```
zip_codes.at(1) = 94583;
zip_codes.at(2) = 92831;
```

zip_codes:	{0, 94583, 92831, 0}

7. `zip_codes.push_back(92620);`

zip_codes:	{0, 94583, 92831, 0, 92620}

8. `std::vector<int> zip_codes2(zip_codes);`

zip_codes:	{0, 94583, 92831, 0, 92620}
zip_codes2:	{0, 94583, 92831, 0, 92620}

9. `zip_codes.clear();`

zip_codes:	{ }
zip_codes2:	{0, 94583, 92831, 0, 92620}

Looping Through Vectors

Using the vector constructor that accepts an initializer list, we can initialize a vector containing all specified elements within curly braces:

```
std::vector<float> water_levels {1.23, 2.56, 2.43, 1.91, 2.07};
```

Now, we can access each of these `float` elements using a range based for loop:

```
for (float level : water_levels) {
      std::cout << level << std::endl;
}
```

1. How many water level decimal values are printed out after this for loop?
   ◯ 0 ◯ 1 ◯ 2 ◯ 3 ◯ 4 ◯ **5**
2. At each iteration of this range based for loop, the variable `level` gets assigned to one of the elements in the water_levels vector.
   At the first iteration, `level` is 1.23. How many iterations are run in total?
   ◯ 1 ◯ 2 ◯ 3 ◯ 4 ◯ **5**

Given a `vector` of ints, return its max element. Use a **range-based for loop**. You can assume that `list` always has at least one element. **Visualizer**

```
int MaxElement(std::vector<int> list) {
   int max = list.at(0);
   for (int num : list) {
      if (max < num) {
        max = num;
      }
   }
   return max;
}
```

Challenge: Merge Two Sorted Vectors

Given two vectors whose elements are sorted in ascending order, both containing `int`s, return a new vector containing all of the elements of the two lists, in ascending order

Visualizer

```
std::vector<int> Merge(std::vector<int> a, std::vector<int> b) {
   // Hint: first construct a new, empty vector
   std::vector<int> merged;

   int index_a = 0;
   int index_b = 0;
   while (index_a < a.size() && index_b < b.size()) {
      if (a.at(index_a) < b.at(index_b)) {
         merged.push_back(a.at(index_a));
         index_a += 1;
      } else {
         merged.push_back(b.at(index_b));
         index_b += 1;
      }
   }
   while (index_a < a.size()) {
      merged.push_back(a.at(index_a));
      index_a += 1;
   }
   while (index_b < b.size()) {
      merged.push_back(b.at(index_b));
      index_b += 1;
   }

   return merged;
}
```

Challenge: 2D Vectors

A 2D vector is a vector whose elements are each vectors. 2D vectors are often used to represent a matrix. Given a 2D vector of `int`s, print all the elements in any order.

Visualizer

```
void PrintMatrix(std::vector<std::vector<int>> matrix) {
   for (std::vector<int> row : matrix) {
      for (int elem : row) {
         std::cout << elem << “ ”;
      }
   }
}
```

Maps

Map Constructors & Functions

Below each line, draw the state of the map after executing the statement. We can visualize it with the initializer list syntax for maps. The first two have been done for you.

1. `std::map<char, int> char_count;` **Visualizer**

`char_count:`	`{ }`

2.
```
char_count.insert({'e', 5});
char_count.insert({'s', 3});
char_count.insert({'t', 2});
```

`char_count:`	`{{'e', 5}, {'s', 3}, {'t', 2}}`

3. `char_count.at('e') = 8;`

`char_count:`	**`{{'e', 8}, {'s', 3}, {'t', 2}}`**

4. `char_count.at('s') += 1;`

`char_count:`	**`{{'e', 8}, {'s', 4}, {'t', 2}}`**

5. `char_count.insert({'n', 9});`

`char_count:`	**`{{'e', 8}, {'n', 9}, {'s', 4}, {'t', 2}}`**

6. `std::cout << char_count.size();`

`Output:`	**`4`**

7. `std::map<std::string, int> country_codes {{"US", 1}, {"MX", 52}};`

`country_codes:`	**`{{"MX", 52}, {"US", 1}}`**

8. `country_codes.insert({"PH", 63});` **Visualizer**

`country_codes:`	**`{{"MX", 52}, {"PH", 63}, {"US", 1}}`**

9. `std::cout << country_codes.empty();`

`Output:`	**`false (may be printed out as 0)`**

10. `std::map<std::string, int> my_copy(country_codes);`

`country_codes:`	**`{{"MX", 52}, {"PH", 63}, {"US", 1}}`**
`my_copy:`	**`{{"MX", 52}, {"PH", 63}, {"US", 1}}`**

Looping Through Maps

Using the map constructor that accepts an initializer list, we can initialize a map containing all specified elements within curly braces: **Visualizer**

```
std::map<std::string, int> products_sold {
   {“Fitbit”, 20},
   {“Pixel 6”, 56},
   {“Chromecast”, 12}
};
```

1. What is the data type of the **key** in the `products_sold` map?

   ◯ `std::map` ◯ **`std::string`** ◯ `std::pair` ◯ `int`
2. What is the data type of the **value** in the `products_sold` map?

   ◯ `std::map` ◯ `std::string` ◯ `std::pair` ◯ **`int`**
3. How many elements are in this map (i.e. the value of `products_sold.size()`)?

   ◯ `0` ◯ `1` ◯ `2` ◯ **`3`** ◯ `4` ◯ `5`

Remember, each element in a map is a (key, value) pair, represented with the `std::pair` in C++.
A `std::pair` has two member variables:

- `first`, which holds the key, and
- `second`, which holds the value it maps to.

Now, let’s access each of the map’s (key, value) pairs using a range based for loop:

```
for (std::pair<std::string, int> pair : products_sold) {
   std::cout << pair.first + “: sold ” << pair.second << std::endl;
}
```

4. How many lines are printed out to the console after this for loop?

   ◯ `0` ◯ `1` ◯ `2` ◯ **`3`** ◯ `4` ◯ `5`
5. At each iteration of this range based for loop, the variable `pair` gets assigned to one of the (key, value) pairs in the `products_sold` map. Recall that by default, `std::map` is sorted in increasing order **by the key** - much like real dictionaries! At the first iteration of the loop, what is printed out?

◯ `“Fitbit: sold 20”` ◯ `“Pixel 6: sold 56”` ◯ **`“Chromecast: sold 12”`**

Challenge: One to One

Implement a function `OneToOne`, which takes a `std::map<char, int> my_map` and returns true if every value in `my_map` only has one corresponding key.

For example:

```
std::map<char, int> map1 {{'a', 4}, {'b', 5}, {'c', 3}};
OneToOne(map1) // Returns true.
```

In contrast:

```
std::map<char, int> map2 {{'a', 2}, {'b', 4}, {'c', 2}};
OneToOne(map2) // Returns false.
```

```
bool OneToOne(std::map<char, int> my_map) {
   // keep track of the values we've seen so far
   std::vector<int> seen;

   for (std::pair<char, int> pair : my_map) {
      for (int num : seen) {
         if (num == pair.second) {
            // We've seen this value already, so the map isn't One-to-One.
            return false;
         }
      }
      seen.push_back(pair.second);
   }
   return true;
}
```

Note: there is a more efficient way of implementing OneToOne, using a data structure we haven't learned yet, called the Set (sadly you may need to wait til CPSC 131 to learn it).

- **But if you're interested in the most algorithmically efficient solution, let me (JC) know and I'll walk you through it!**

Loops

Counting Iterations

A standard `for` loop is executed in the following order:

1. **Initialization** is performed. This is executed only once, at the very beginning.
2. If the **condition** is false, exit the loop and execute the code that comes after.
3. If the **condition** is true, run the loop body. Perform the **update**, repeat step 2.

In the following `for` loops, determine how many iterations of the loop are executed. Assume `i` is not modified within the loop body.

1. `for (int i = 0; i < 3; i++) { ... }`

   ◯ 0 ◯ 1 ◯ 2 ◯ **3** ◯ 4 ◯ Infinite

2. `for (int i = 0; i <= 0; i += 1) { ... }`

   ◯ 0 ◯ **1** ◯ 2 ◯ 3 ◯ 4 ◯ Infinite

3. `for (int i = 3; i >= 0; i--) { ... }`

   ◯ 0 ◯ 1 ◯ 2 ◯ 3 ◯ **4** ◯ Infinite

4. ```
   std::vector<int> numbers {10, 20, 30, 40};
   for (int i = 1; i < numbers.size(); i++) { ... }
   ```

   ◯ 0 ◯ 1 ◯ 2 ◯ **3** ◯ 4 ◯ Infinite

It can be easy to make an error (e.g. off-by-one errors like in #4 above) using standard for loops, especially when indexing into vectors. Instead, if we use the **range based** `for` loop, we can rest safe knowing that we loop through each element in a container, without dealing with indices!

In the following `for` loops, determine how many iterations of the loop are executed.

5. ```
   std::vector<int> numbers {10, 20, 30, 40};
   for (int num : numbers) { ... }
   ```

   ◯ 0 ◯ 1 ◯ 2 ◯ 3 ◯ **4** ◯ Infinite

6. ```
   std::vector<double> temperatures;
   temperatures.push_back(74.7);
   temperatures.push_back(68.2);
   for (double temp : temperatures) { ... }
   ```

   ◯ 0 ◯ 1 ◯ **2** ◯ 3 ◯ 4 ◯ Infinite

7. ```
   std::map<char, int> tx {{'c', 2}, {'s', 0}, {'u', 2}, {'f', 2}};
   for (std::pair<char, int> t : tx) { ... }
   ```

   ◯ 0 ◯ 1 ◯ 2 ◯ 3 ◯ **4** ◯ 8

Notice that in the following **for** loop, we're updating k in two places.

```
for (int k = 1; k < 10; k = k + 1) {
     k = k + 1;
     std::cout << k << " ";
 }
```

What output is produced by the above for loop? We recommend you use a table to keep track of the value of k.

2 4 6 8 10

Challenge: Deciphering Code

In the real world, often you will have to read code you did not write. Sometimes you will be provided documentation, other times you will not. Being able to read code and recognize its purpose is an important skill.

Provide a good (i.e. descriptive) name for each of the following functions. Assume that `**values**` contains at least one element.

1.
```
int GetMaximumElement(std::vector<int> values) { Visualization
   int rtn = values.at(0);
   for (int k = 1; k < values.size(); k++) {
        if (rtn < values.at(k)) {
           rtn = values.at(k);
        }
   }
   return rtn;
}
```

2.
```
void Reverse(std::vector<int> &values) { Visualization
   for (int k = 0; k < values.size() / 2; k++) {
        int temp = values.at(k);
        values.at(k) = values.at(values.size() - 1 - k);
        values.at(values.size() - 1 - k) = temp;
   }
}
```

References

Pass by Value vs Pass by Reference

Pass by value is when we make a copy of the value inside a variable, and pass that **copy** to a function. Pass by reference is when we pass a reference to the original variable, so the function can modify the original variable's value.

Take these two functions, which both aim to swap the values of the two `int`s passed in.

```
void SwapByVal(int a, int b) {
  int temp = a;
  a = b;
  b = temp;
}
```

```
void SwapByRef(int &a, int &b) {
  int temp = a;
  a = b;
  b = temp;
}
```

1. What symbol is used to indicate that a variable is being passed by reference?

   ◯ ! ◯ = ◯ **&** ◯ ? ◯ %

2. Write down the output to the console after running the following lines of code:

```
int my_money = 0;
int your_money = 2500;
SwapByVal(my_money, your_money);
```

`std::cout << my_money;`	**0**
`std::cout << your_money;`	**2500**

3. Write down the output to the console after running the following lines of code:

```
int my_money = 0;
int your_money = 2500;
SwapByRef(my_money, your_money);
```

Visualization

`std::cout << my_money;`	**2500**
`std::cout << your_money;`	**0**

4. When would it be useful to pass a variable by reference? Mark all that apply.

   ▢ When you want a function to modify the value of the original variable.

   ▢ To save on the additional memory space used when copying large objects being passed into a function.

   ▢ To save on the time it takes to copy large objects when passing them to functions.

For the function below, select the effect(s) of the & keyword on the function.

5. `int ComputeAverage(std::vector<int> &nums) { ... }`
   - ▫ When `ComputeAverage` is called, a copy of the vector is stored in `nums`.
   - ▫ When `ComputeAverage` is called, a reference to the original vector is passed to `nums`, so the elements of `nums` can be modified within `ComputeAverage`.
   - ▫ `ComputeAverage` accepts a reference to the original vector passed in, avoiding the inefficiency of needing to copy a potentially large vector.

What if we don't want to modify the original data, but still want to avoid making a potentially expensive copy? This is where `const` comes in.

Select the effect of adding the `const` keyword on the function below.

6. `int ComputeAverage(const std::vector<int> &nums) { ... }`
   - ◯ `ComputeAverage` may not return a value in `nums`.
   - ◯ **`ComputeAverage` may not modify the contents of `nums`.**
   - ◯ `ComputeAverage` may not access the original contents of `nums`.

Overloading

Overloading is when you have more than one function of the same name. For example:

```
void Display(int num) { ... }
void Display(double decimal) { ... }
void Display(std::string text) { ... }
```

1. If we write `Display("Hello world!")`, how does the compiler know which function overload should be called?
   - ◯ **The compiler knows which overload to call based on the types and number of arguments passed in to the function call.**
   - ◯ The compiler knows because the parameter names are all different.
2. Would it be valid to add the following function as another overload:
   `void Display(int x, int y) { ... }`
   - ◯ **Valid** ◯ Invalid
3. Would it be valid to add the following function as another overload:
   `std::string Display(int x) { ... }`
   `Functions can not be overloaded if they differ only in the return type.`
   - ◯ Valid ◯ **Invalid**