Section 1. C++ fundamentals

Section materials curated by Neel Kishnani, drawing upon materials from previous quarters.

This week's section exercises explore the very fundamentals of programming in C++. We'll be exploring the material from Week 1 and the beginning of Week 2 (functions, parameters, return, decomposition, strings and basic data structures). Have fun!

Each week, we will also be releasing a Qt Creator project containing starter code and testing infrastructure for that week's section problems. When a problem name is followed by the name of a **.cpp** file, that means you can practice writing the code for that problem in the named file of the Qt Creator project. Here is the zip of the section starter code:

Note: we intentionally put in more problems on section handouts than you can cover in section. You can use the extra problems for practice and review, especially for exams.

Starter code

Note: `Maps` will be covered in lecture on Friday, Apr. 8. For those that have section Wednesday or Thursday, these problems can be useful practice to cement your understanding after lecture!

1) Returning and Printing

Topics: Function call and return, return types

Below is a series of four **`printLyrics_v#`** functions, each of which has a blank where the return type should be. For each function, determine

- what the return type of the function should be,
- what value, if any, is returned, and
- what output, if any, will be produced if that function is called.

Is it appropriate for each of these functions to be named **`printLyrics`**? Why or why not?

```
_____ printLyrics_v1() {
    cout << "Havana ooh na na" << endl;
}
_____ printLyrics_v2() {
    return "Havana ooh na na";
}
_____ printLyrics_v3() {
    return "H";
}
_____ printLyrics_v4() {
    return 'H';
}
```

Solution

```
void printLyrics_v1() {
    cout << "Havana ooh na na" << endl;
}

string printLyrics_v2() {
    return "Havana ooh na na";
}

string printLyrics_v3() {
    return "H";
}

char printLyrics_v4() {
    return 'H';
}
```

Of these four functions, only **`printLyrics_v1`** will print anything. Specifically, it prints out the string **`"Havana ooh na na."`**. The name "**`printLyrics`**" is inappropriate for the other functions, as those functions don't actually print anything. 😃

The function **`printLyrics_v1`** doesn't return anything – it just sends information to the console. As a result, its return type should be **`void`**. The functions **`printLyrics_v2`** and **`printLyrics_v3`** each return strings, since C++ treats anything in double-quotes as a string. Finally, **`printLyrics_v4`** returns a **`char`**, since C++ treats anything in single-quotes as a character.

2) References Available Upon Request

Topic: Reference parameters, range-based for loops

Reference parameters are an important part of C++ programming, but can take some getting used to if you're not familiar with them. Trace through the following code. What does it print?

```
void printVector(Vector<int>& values) {
    for (int elem: values) {
        cout << elem << " ";
    }
    cout << endl;
}

void maui(Vector<int> values) {
    for (int i = 0; i < values.size(); i++) {
        values[i] = 1258 * values[i] * (values[2] - values[0]);
    }
}

void moana(Vector<int>& values) {
    for (int elem: values) {
        elem *= 137;
    }
}

void heihei(Vector<int>& values) {
    for (int& elem: values) {
        elem++;
    }
}

Vector<int> teFiti(Vector<int>& values) {
    Vector<int> result;
    for (int elem: values) {
        result += (elem * 137);
    }
    return result;
}

int main() {
    Vector<int> values = { 1, 3, 7 };
    maui(values);
    printVector(values);
    moana(values);
    printVector(values);
    heihei(values);
    printVector(values);
    teFiti(values);
    printVector(values);
    return 0;
}
```

Solution

Here's the output from the program:

```
1 3 7
1 3 7
2 4 8
2 4 8
```

Here's a breakdown of where this comes from:

- The **`maui`** function takes its argument by value, so it's making changes to a copy of the original vector, not the vector itself. That means that the values are unchanged back in main.
- The **`moana`** function uses a range-based for loop to access the elements of the vector. This makes a copy of each element of the vector, so the changes made in the loop

only change the temporary copy and not the elements of the vector. That makes that the values are unchanged back in main.
- **heihei**, on the other hand, uses **int&** as its type for the range-based for loop, so in a sense it's really iterating over the elements of the underlying vector. Therefore, its changes stick.
- The **teFiti** function creates and returns a new vector with a bunch of updated values, but the return value isn't captured back in main.

3) SumNumbers (sum.cpp)

Topics: Vectors, strings, file reading

The function **sumNumbers** reads a text file and sums the numbers found within the text. Here are some library functions that will be useful for this task:

- **readEntireFile**, to read all lines from a file stream into a Vector
- **stringSplit**, to divide a string into tokens
- **isdigit**, to determine whether char is a digit
- **stringToInteger**, to convert a string of digits to integer value

In particular you will be asked to write the following function

int sumNumbers(string filename)

When given the following file, named **numbers.txt**, as input, your function should return 42.

```
42 is the Answer to the Ultimate Question of Life, the Universe, and
Everything
This is a negative number: -9
Welcome to CS106B!
I want to own 9 cats.
```

Solution

```
bool isNumber(string s)
{
    // strip negative sign off negative numbers
    if (s.length() > 0 && s[0] == '-') {
        s = s.substr(1);
    }
    for (char ch : s) {
        if (!isdigit(ch)) return false;
    }
    return s.length() > 0;
}

int sumNumbers(string filepath)
{
    ifstream in;
    Vector<string> lines;
    int sum = 0;

    if (!openFile(in, filepath)) {
        return 0;
    }

    readEntireFile(in, lines);
    for (string line : lines) {
        Vector<string> tokens = stringSplit(line, " ");
        for (string t : tokens) {
            if (isNumber(t)) {
                sum += stringToInteger(t);
            }
        }
    }
    return sum;
}
```

4) Debugging Deduplicating (**`deduplicate.cpp`**)

Topics: Vector, strings, debugging

Consider the following ***incorrect*** C++ function, which accepts as input a **`Vector<string>`** and tries to modify it by removing adjacent duplicate elements:

⚠️⚠️⚠️⚠️⚠️⚠️⚠️⚠️⚠️⚠️⚠️⚠️⚠️⚠️⚠️⚠️⚠️⚠️

```
void deduplicate(Vector<string> vec) {
    for (int i = 0; i < vec.size(); i++) {
        if (vec[i] == vec[i + 1]) {
            vec.remove(i);
        }
    }
}
```

⚠️⚠️⚠️⚠️⚠️⚠️⚠️⚠️⚠️⚠️⚠️⚠️⚠️⚠️⚠️⚠️⚠️⚠️

The intent behind this function is that we could do something like this:

```
Vector<string> hiddenFigures = {
    "Katherine Johnson",
    "Katherine Johnson",
    "Katherine Johnson",
    "Mary Jackson",
    "Dorothy Vaughan",
    "Dorothy Vaughan"
};

deduplicate(hiddenFigures);
// hiddenFigures = ["Katherine Johnson", "Mary Jackson", "Dorothy Vaughan"]
```

The problem is that the above implementation of **`deduplicate`** does not work correctly. In particular, it contains three bugs. First, find these bugs by writing test cases that pinpoint potentially erroneous situations in which the provided code might fail, then explain what the problems are, and finally fix those errors in code.

Solution

There are three errors here:

1. Calling **`.remove()`** on the **`Vector`** while iterating over it doesn't work particularly nicely. Specifically, if you remove the element at index **`i`** and then increment **`i`** in the for loop, you'll skip over the element that shifted into the position you were previously in.
2. There's an off-by-one error here: when **`i = vec.size() - 1`**, the indexing **`vec[i + 1]`** reads off the end of the **`Vector`**.
3. The **`Vector`** is passed in by value, not by reference, so none of the changes made to it will persist to the caller.

Here's a corrected version of the code:

```
void deduplicate(Vector<string>& vec) {
    for (int i = 0; i < vec.size() - 1; ) {
        if (vec[i] == vec[i + 1]) {
            vec.remove(i);
        } else {
            i++;
        }
    }
}
```

[Credit to Andrew Tierno for the alternate solution] Alternatively, you can also re-write the function to use a loop that traverses the vector from right-to-left, which is a common pattern when working with deleting items from linear collections. A solution that does so could look like this:

```
void deduplicate(Vector<string>& vec) {
    for (int i = vec.size() - 1; i > 0; i--) {
        if (vec[i] == vec[i - 1]) {
            vec.remove(i);
        }
    }
}
```

5) Pig-Latin (**`piglatin.cpp`**)

Topics: Strings, reference parameters, return types

Write two functions, **`pigLatinReturn`** and **`pigLatinReference`**, that accept a string and convert said string into its pig-Latin form. To convert a string into pig-Latin, you must follow these steps:

- Split the input string into 2 strings: a string of characters BEFORE the first vowel, and a string of characters AFTER (and including) the first vowel.
- Append the first string (letters before the first vowel) to the second string.
- Append the string "ay" to the resulting string.

Here are a few examples…

`nick -> icknay`

`chase -> asechay`

`chris -> ischray`

You will need to write this routine in two ways: once as a function that returns the pig-Latin string to the caller, and once as a function that modifies the supplied parameter string and uses it to store the resulting pig-Latin string. These will be done in **`pigLatinReturn`** and **`pigLatinReference`**, respectively. You may assume that your input is always a one-word, all lowercase string with at least one vowel.

Here's a code example of how these functions differ…

```
string name = "julie";
string str1 = pigLatinReturn(name);
cout << str1 << endl; // prints "uliejay"

pigLatinReference(name);
cout << name << endl; // prints "uliejay"
```

Once you've written these functions, **discuss with your section the benefits and drawbacks of these two approaches**. Which do you feel is easier to write? Which do you think is more convenient for the caller? Do you think one is better style than the other?

Solution

```
// Use const because VOWELS won't change -- no need to declare repeatedly
// in isVowel.
const string VOWELS = "aeiouy";

// Helper function, which I'd highly recommend writing!
bool isVowel(char ch) {
    // A little kludgy, but the handout guarantees that
    // ch will ALWAYS be lower case :)
    // NOTE: For an assignment, you probably want a more robust isVowel.
    return VOWELS.find(ch) != string::npos;
}

string pigLatinReturn(string input) {
    int strOneIndex = 0;
    for (int i = 0; i < input.length(); i++) {
        if (isVowel(input[i])) {
            strOneIndex = i;
            break;
        }
    }
    string strOne = input.substr(0, strOneIndex);
    string strTwo = input.substr(strOneIndex);
    return strTwo + strOne + "ay";
}

void pigLatinReference(string &input) {
    int strOneIndex = 0;
    for (int i = 0; i < input.length(); i++) {
        if (isVowel(input[i])) {
            strOneIndex = i;
            break;
        }
    }
    string strOne = input.substr(0, strOneIndex);
    string strTwo = input.substr(strOneIndex);
    input = strTwo + strOne + "ay";
}
```

Notice how similar these two approaches are – the only difference is how the result is handled at the very end. To address the discussion questions, although the **`pigLatinReference`** function is marginally more efficient because it doesn't need to make a copy of the input string, **`pigLatinReturn`** is probably more intuitive for both the caller and the writer: if the function's job is to somehow output some product, returning is the most explicit way to do so. In that way, a function that returns is also better style – it's makes the purpose of the function clearer to the reader.

If you wanted to combine the efficiency of **`pigLatinReference`** with the clarity of **`pigLatinReturn`**, I would recommend writing a function that takes in the input string by **`const reference`**, basically

```
string pigLatin(const string &input);
```

Although the const isn't explicitly necessary, it's nice to have because you never need to modify input. Moreover, you still get the efficiency gains from pass-by-reference while also writing very-understandable code.

6) Mirror (`mirror.cpp`)

Topic: Grids

Write a function **`mirror`** that accepts a reference to a **`Grid`** of integers as a parameter and flips the grid along its diagonal. You may assume the grid is square; in other words, that it has the same number of rows as columns. For example, the grid below that comes first would be altered to give it the new grid state shown afterwards:

```
Original state:
{ { 6, 1, 9, 4},
  {-2, 5, 8, 12},
  {14, 39, -6, 18},
  {21, 55, 73, -3} }

Mirrored state:
 { {6, -2, 14, 21},
   {1, 5, 39, 55},
   {9, 8, -6, 73},
   {4, 12, 18, -3} }
```

Bonus: How would you solve this problem if the grid were not square?

Solution

```
// solution
void mirror(Grid<int>& grid) {
    for (int r = 0;r < grid.numRows(); r++) {
        // start at r+1 rather than 0 to avoid double-swapping
        for (int c = r + 1; c < grid.numCols(); c++) {
            int temp = grid[r][c];
            grid[r][c] = grid[c][r];
            grid[c][r] = temp;
        }
    }
}
// bonus
void mirror(Grid<int>& grid) {
    Grid<int> result(grid.numCols(), grid.numRows());
    for (int r = 0; r < grid.numRows(); r++) {
        for (int c = 0; c < grid.numCols(); c++) {
            result[r][c] = grid[c][r];
        }
    }
    grid = result;
}
```

7) Check Balance (**`balance.cpp`**)

Topic: Stacks

Write a function named **`checkBalance`** that accepts a string of source code and uses a **`Stack`** to check whether the braces/parentheses are balanced. Every **`(`** or **`{`** must be closed by a **`}`** or **`)`** in the opposite order. Return the index at which an imbalance occurs, or `-1` if the string is balanced. If any **`(`** or **`{`** are never closed, return the string's length.

Here are some example calls:

```
//   index    0123456789012345678901234567
checkBalance("if (a(4) > 9) { foo(a(2)); }")
// returns -1 (balanced)

//   index    01234567890123456789012345678901
checkBalance("for (i=0;i<a;(3};i++) { foo{); )")
// returns 15 because } is out of order

//   index    0123456789012345678901234
checkBalance("while (true) foo(); }{ ()")
// returns 20 because } doesn't match any {

//   index    01234567
checkBalance("if (x) {")
// returns 8 because { is never closed
```

Solution

```
int checkBalance(string code) {
    Stack<char> parens;
    for (int i = 0; i < (int) code.length(); i++) {
        char c = code[i];
        if (c == '(' || c == '{') {
        parens.push(c);
        } else if (c == ')' || c == '}') {
            if (parens.isEmpty()) {
                return i;
            }
            char top = parens.pop();
            if ((top == '(' && c != ')') || (top == '{' && c != '}')) {
                return i;
            }
        }
    }

    if (parens.isEmpty()) {
        return -1; // balanced
    } else {
        return code.length();
    }
}
```

8) Collection Mystery

Topics: Stacks and Queues

```
void collectionMystery(Stack<int>& s)
{
    Queue<int> q;
    Stack<int> s2;

    while (!s.isEmpty()) {
       if (s.peek() % 2 == 0) {
            q.enqueue(s.pop());
        } else {
            s2.push(s.pop());
        }
    }
    while (!q.isEmpty()) {
        s.push(q.dequeue());
    }
    while(!s2.isEmpty()) {
        s.push(s2.pop());
    }
    cout<< s << endl;
}
```

Write the output produced by the above function when passed each of the following stacks. Note that stacks and queues are written in front to back order, with the oldest element on the left side of the queue/stack.

Stacks:

```
{1, 2, 3, 4, 5, 6}            ________________________________________
{42, 3, 12, 15, 9, 71, 88}    ________________________________________
{65, 30, 10, 20, 45, 55, 6, 1}  ________________________________________
```

Solution

```
{6, 4, 2, 1, 3, 5}
{88, 12, 42, 3, 15, 9, 7}
{6, 20, 10, 30, 65, 45, 55, 1}
```

9) Friend List (`friendlist.cpp`)

Topic: Maps

Write a function named **`friendList`** that takes in a file name and reads friend relationships from a file and writes them to a **`Map`**. **`friendList`** should return the populated **`Map`**. Friendships are bi-directional, so if Abby is friends with Barney, Barney is friends with Abby. The file contains one friend relationship per line, with names separated by a single space. You do not have to worry about malformed entries.

If an input file named **`buddies.txt`** looked like this:

```
Barney Abby
Abby Clyde
```

Then the call of **`friendList("buddies.txt")`** should return a resulting **`map`** that looks like this:

`{"Abby":{"Barney", "Clyde"}, "Barney":{"Abby"}, "Clyde":{"Abby"}}`

Here is the function prototype you should implement:

`Map<string, Vector<string> > friendList(String filename)`

Solution

```
Map<string, Vector<string> > friendList(string filename) {
    ifstream in;
    Vector<string> lines;

    if (openFile(in, filepath)) {
        readEntireFile(in, lines);
    }

    Map<string, Vector<string> > friends;
    for (string line: lines) {
        Vector<string> people = stringSplit(line, " ");
        string s1 = people[0];
        string s2 = people[1];
        friends[s1] += s2;
        friends[s2] += s1;
    }
    return friends;
}
```

10) Twice (`twice.cpp`)

Topic: Sets

Write a function named `twice` that takes a vector of integers and returns a set containing all the numbers in the vector that appear exactly twice.

Example: passing `{1, 3, 1, 4, 3, 7, -2, 0, 7, -2, -2, 1}` returns `{3, 7}`.

Bonus: do the same thing, but you are not allowed to declare any kind of data structure other than sets.

Solution

```
// solution
Set<int> twice(Vector<int>& v) {
    Map<int, int> counts;
    for (int i : v) {
        counts[i]++;
    }
    Set<int> twice;
    for (int i : counts) {
        if (counts[i] == 2) {
            twice += i;
        }
    }
    return twice;
}

// bonus
Set<int> twice(Vector<int>& v) {
    Set<int> once;
    Set<int> twice;
    Set<int> more;
    for (int i : v) {
        if (once.contains(i)) {
            once.remove(i);
            twice.add(i);
        } else if (twice.contains(i)) {
            twice.remove(i);
            more.add(i);
        } else if (!more.contains(i)) {
            once.add(i);
        }
    }
    return twice;
}
```