

1 Week 5/6

1.1 hpp / cpp arrangement

This is a demonstration going from a single `main.cpp` to `main.cpp` plus two other files with the extensions `.cpp` and `.hpp`.

Let's start with our typical code where everything is in `main.cpp`

1.1.1 start

`main.cpp`:

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  struct Person {
6      string name;
7      int age;
8
9      Person(const string& name, int age) : name(name), age(age) {}
10
11     /**
12      * The person introduces themselves by printing their name and age.
13      */
14     void introduceSelf() const {
15         cout << "Hello, I'm " << name << " and I'm " << age << " years old." << endl;
16     }
17 };
18
19
20 /**
21  * Greets a person using their name.
22  * @param person : a Person to greet.
23  */
24 void greet(const Person& person) {
25     cout << "Hello, " << person.name << "!" << endl;
26 }
27
28
29 int main() {
30     Person person1("Sarah", 30);
31     Person person2("JP", 2);
32
33     person1.introduceSelf();
34     greet(person1);
35     person2.introduceSelf();
```

```

36     greet(person2);
37
38     return 0;
39 }

```

In this code, lines 4-17 is a **Person** struct. It contains member variables **name** and **age**. An instance of **Person** can be created by using the initializer in line 9. This assigns arguments to member variables, similar to how a function assigns arguments to parameters. Lines 14-17 is a member function which prints to the consoles information about the particular **Person** created. Lines 24-26 is an ordinary function definition that takes in an instance of **Person** as an argument. In **main**, we test out our initializer and functions. In lines 30 & 31, we create two instance of a **Person**: **person1** and **person2**. Lines 33-36 show various calls to functions. Lines 33 & 35 are calls to member functions. Lines 34 & 36 are calls to an ordinary function.

1.1.2 finish

functions.hpp:

```

1  #ifndef functions_hpp // Don't worry about this, just include it.
2  #define functions_hpp
3
4  // Relevant includes are put in the hpp file.
5  // You should NOT use namespaces in hpp files.
6  #include <iostream>
7  #include <string>
8
9  // Class / struct definitions should be in the hpp file.
10 struct Person {
11     std::string name;
12     int age;
13
14     Person(const std::string& name, int age) : name(name), age(age) {}
15
16     /**
17         The person introduces themselves by printing their name and age.
18     */
19     void introduceSelf() const;
20 };
21 // Class definition should be in the cpp file.
22 // Member function comments and declarations should be in an hpp file.
23 // Member function definitions should be in the corresponding cpp file.
24 // Use the scoping operator ::
25
26
27 // Ordinary comments and declarations should be in an hpp file.
28 // Ordinary definitions should be in the corresponding cpp file.
29 /**

```

```

30  Greets a person using their name.
31  @param person : a Person to greet.
32  */
33  void greet(const Person& person);
34
35  #endif /* functions.hpp */

```

functions.cpp:

```

1  // Include the corresponding header file.
2  #include "functions.hpp"
3
4  // It is okay to use namespaces in cpp files.
5
6  // Function definitions should be in a cpp file.
7  void greet(const Person& person) {
8      std::cout << "Hello, " << person.name << "!" << std::endl;
9      // You could imagine this being a very long function definition.
10     // With these comments representing many lines of code.
11     // Maybe instead of greeting a person, it does a entire background check
12     // on any person with the same name and age.
13     // ... Many lines of code ... that no one really wants to read.
14     // But they could want to use this useful function.
15     // They'd want to know the information to use it.
16     // But maybe they don't want to read through these many lines.
17     // So they look for the "need-to-know" information in the.hpp.
18     // The function comment has a concise description of the function's use.
19     // The function signature let's you know how you're allowed to use it.
20     // You won't have to scroll past many lines of code.
21     // From other long function definitions.
22     // To see that information.
23     // ...
24     // ..
25
26
27     // ...
28
29
30     // ..
31 }
32 void Person::introduceSelf() const {
33     std::cout << "Hello, I'm " << name << " and I'm " << age << " years old." << std::endl;
34     // This could also be a long definition.
35     // It might take awhile to see what you need to use it.
36     // Scroll past the function definition above.
37     //.hpp should have what we need to know to use the function.
38 }

```

main.cpp:

```
1  #include "functions.hpp"
2  using namespace std;
3
4  int main() {
5      // This is still where you test your code
6      // Here I create some people
7      Person person1("Sarah", 30);
8      Person person2("JP", 2);
9
10     // Use the member function from Person
11     person1.introduceSelf();
12     // Call the function with an argument that is an instance of Person
13     greet(person1);
14     person2.introduceSelf();
15     greet(person2);
16
17     return 0;
18 }
```

1.2 References and const conventions

References and const are new ways we facilitate and communicate variable usage. They allow us to write more efficient and predictable code. **References** are aliases for other variables. When you create a reference to a variable, you are saying that the reference and the variable are essentially the same thing. This means that any changes you make to the reference will also happen to the original variable. **Const conventions** are a set of rules that govern how you can use const objects and references. Const variables cannot be modified. This can help to prevent accidentally changing variables and improve the readability of your code.

Let's go over the practical implementation.

1.2.1 Functions

We call the example below **passing a variable by value to a function**. This will seem most familiar to you. Based on the code in the example below, you might expect the value of `x` to change in `main`, but passing by value makes a copy of the value of `x` to the local variable `y` inside the function call. Then `y` is changed to 20 and lastly is destroyed when the function ends. After the function call, the value of `x` remains unchanged in `main`.

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

void f(int y) {
    y = 20;
}
```

```

int main() {
    int x = 10;
    f(x);
    cout << x << endl;

    return 0;
}

```

We call the example below **passing a variable by reference to a function**. You might use a reference if you intend `x` to change in `main`. By passing `x` to the local reference `y`, we see any changes made to `y` reflect on `x`. After the function call the value of `x` has changed in `main`.

```

#include <iostream>
using namespace std;

void f(int& y) {
    y = 20;
}

int main() {
    int x = 10;
    f(x);
    cout << x << endl;

    return 0;
}

```

Other than wanting to make a change to a variable in `main`, another reason to pass by reference is that it is efficient. Potentially large containers like `strings` or `vectors` should always be passed by reference, whether or not you intend to change the argument passed to the variable. Often copying nonfundamental types takes more memory than creating a reference.

```

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

void f(string& y) {
    y += ", World";
}

int main() {
    string x = "Hello";
    f(x);
    cout << x << endl;

    return 0;
}

```

We call the following example code **passing a variable by reference to const to a function**. Unlike the last two examples, we do not intend for `x` to be changed by the function call. In this code, we protect the variable `x` from being changed by using the `const` keyword.

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

void f(const string& y) {
    cout << y << ", World" << endl;
}

int main() {
    string x = "Hello";
    f(x);
    cout << x << endl;

    return 0;
}
```

In conclusion, when writing a function signature there are two things we want to regard: Is the parameter we're passing a fundamental type? Do we mean for the argument we pass to the parameter to be changed in `main` because of us changing the local parameter in the function call? If the answer to the latter question is true, then no matter what type the variable is, we should pass it by reference. If you don't want to change the argument in `main` and it's not a fundamental type, then pass by reference to const. If you don't want to change the argument in `main` and it's a fundamental type, then pass by value.

1.2.2 Structs / Classes

References in structs and classes are treated similarly. Consider the following code of a struct `Point` which stores a 2-dimensional point by holding two integers: `x` and `y`. Here there is a function `move` that's designed to take in a `Point` and move that `Point` in `x` by `dx` and in `y` by `dy`. The function signature of `move` has the argument passed by reference to the parameter `point` because it is not a fundamental type and not by const because the `move` function modifies the member variables `x` and `y` of `point`. The parameters `dx` and `dy` are passed by value. They are fundamental types (integers) and we do not intend to change them in `main`.

In the example below, a `Point` (or `Point` object) `pt` is created with the member variables `x` and `y` assigned the values 10 and 20, respectively. This is shown in the first console output. Then the `move` function is called. This takes in the arguments `pt`, 1, & 2 and assigns them to the function parameters `point`, `dx`, & `dy`. Since the value `pt` is assigned to `point` by reference, the changes made to `point` in the function reflect on `pt`. So when the member variables `x` and `y` are accessed and modified in the function, we see that they are changed when we console out.

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

```

struct Point {
    int x;
    int y;
};

void move(Point& point, int dx, int dy) {
    point.x += dx;
    point.y += dy;
}

void query(const Point& point) {
    cout << point.x << " " << point.y << endl;
}

int main() {
    Point pt = {10, 20};
    cout << pt.x << " " << pt.y << endl;
    move(pt, 1, 2);
    cout << pt.x << " " << pt.y << endl;

    return 0;
}

```

Console output:

```

10 20
11 22

```

There's another function in the example called `query` that is unused. Calling that function with a `Point` as an argument acts like the lines used to print out the `Points`. Calling that function isn't meant to change the `Point` so the argument is passed by reference to `const`.

```

// same struct and function definitions from above ...
int main() {
    Point pt = {10, 20};
    query(pt);
    move(pt, 1, 2);
    query(pt);

    return 0;
}

```

Console output:

```

10 20
11 22

```

In this next example, there are now two member functions added to the `struct Point` definition. It's more common to see a function like `move`, where the function acts on the object itself, defined as

a member function rather than a ordinary function. In this example below, the `Point` isn't passed explicitly but rather implicitly when we make the function call using the member access operator. Therefore, in the member function definition, we can refer `x` and `y` directly. Those are member variables of the `Point` that made the function call. No need for member access operators in this member function definition. So we see in the example below after the `move` function is called on the `Point`, the `Point` has changed.

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

struct Point {
    int x;
    int y;

    void move(int dx, int dy) {
        x += dx;
        y += dy;
    }

    void query() const {
        cout << x << " " << y << endl;
    }
};

int main() {
    Point pt = {10, 20};
    pt.query();
    pt.move(1, 2);
    pt.query();

    return 0;
}
```

Console output:

```
10 20
11 22
```

The other member function `query` is used to print the points out. It is not meant to change anything about the point calling the `query`. To enforce that, we place a `const` keyword before the curly braces of the function definition.

1.3 Exercises

1.3.1 Practice coding exercises

1. Write a function `formFiller` that fills and prints out answers to a questionnaire. The function should take in things like `name`, `age`, `hobby`. What would you use for the input parameters? Calling the function with specified input parameters will output a message like:

[name] is [age] years old and likes [hobby]

or

Name: [name]
Age: [age]
Favorite hobby: [hobby]

Be mindful of the reference and const correctness rules.

2. A magic square is a $n \times n$ grid of distinct numbers in which the sums of each row, column, and diagonal are all the same. In this problem, you are given a 3×3 grid represented as a vector of vector of ints. Write a C++ program to check whether the given grid is a magic square or not. So that `result` would be `true` in the following code:

```
int main() {  
    vector<vector<int>> grid1 = {{2, 7, 6}, {9, 5, 1}, {4, 3, 8}};  
    bool result1 = isMagicSquare(grid1);  
  
    return 0;  
}
```

and false for this code:

```
int main() {  
    vector<vector<int>> grid2 = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};  
    bool result2 = isMagicSquare(grid2);  
  
    return 0;  
}
```

- (a) Begin by creating the function signature. If we are intending for `isMagicSquare` to only perform checks on the grid, how should we pass the arguments to parameters?
- (b) We want to check that the sum of the rows, columns, and diagonals is the same. Inside the function definition, compute and store the expected sum by taking the sum of the first row.
- (c) Now we will check the row sums are the expected sum. We will do this in a nested for loop. The outer loop should iterate over the 3 rows, computing and storing the sum of the current row. So inside the outer loop, you should store the sum of the row as a variable, then once you add the integers across the columns, check if the row sum is the same as the expected sum. If they are not the same, return `false` and leave the function.
- (d) Do the same for the column sums.

- (e) Compute the diagonal sums. There are two. You don't need to use a for loop.
 - (f) If they reach this point in the code, then all sums matched the expected sum, and we can return `true`.
3. (a) Write a function called `area` that takes in the radius of a circle and returns the area of the circle. You might use `cmath` to get `pi`.
- (b) Keeping the function you wrote in (a), write a separate function called `area` that takes in the length and width of a rectangle and returns the area of the rectangle.
- (c) Write a function called `area` that takes in the base and height of a triangle and returns the area of the triangle. What issue arises here? Why didn't it show up in (b)?
4. In the next exercise, we can look ahead to use overloading functions to define the `area` functions for parameters of a specific shape class. So then we can use `area` for a `Circle`, `area` for a `Rectangle`, or `area` for a `Triangle`. The function will overload based on the type of the argument. That argument would also hold information specific to type such as a `Circle` having a radius.
- (a) Define 3 structs. One for `Circle` with member variable `radius`. One for `Rectangle` with member variables `length` and `width`. One for `Triangle` with member variables `base` and `height`. Here is what the struct for `Rectangle` looks like:
- ```
struct Rectangle {
 double length; // This is called a "member variable".
 double width;
};
```
- Place this above your `main` function.
- (b) Create a `Circle`, a `Rectangle`, and `Triangle`. Down in `main`, initialize the new instance. For example,
- ```
int main() {
    Rectangle rec = {2, 4}; // create a rectangle with length 2 and width 4.
    cout << rec.length << endl;
    cout << rec.width << endl;
    return 0;
}
```
- (c) Define 3 overloaded functions, each called `area`. One should take in a `Circle` and return the area of a circle. One should take in a `Rectangle` and return the area of a rectangle. One should take in a `Triangle` and return the area of a triangle.

- (d) Review your code to check for correct use of `const` and references.
- (e) Test your code out by calling `area` on each of the shapes you created in (b) and printing out what it returns each time.

1.3.2 Predict the output exercises

Write the console outputs (not including the Program ended with exit code: 0 line). Assume the code runs error-free, includes the appropriate headers, and is using `std` namespace.

```
5. #include <iostream>
#include <vector>
using namespace std;

int main() {
    vector<int> v1(5, 1);
    vector<int> v2;
    vector<int> v3(5);

    cout << v1.size() << ' ' << v1[0] << endl;
    cout << v2.size() << endl; // what if we did cout << v2[0];?
    cout << v3.size() << ' ' << v3[0] << endl;

    return 0;
}
```

```
6. #include <iostream>
#include <vector>
using namespace std;

int main() {
    vector<int> numbers;
    for (int idx = 1; idx <= 5; ++idx) {
        numbers.push_back(idx * idx);
    }

    for (int idx = 0; idx < numbers.size(); ++idx) {
        cout << numbers[idx] << " ";
    }

    return 0;
}
```

```

7. #include <iostream>
   #include <vector>

   using namespace std;

   int main() {
       vector<char> letters = {'a', 'b', 'c'};

       cout << letters[0] << letters[1] << letters[2] << endl;

       int diff = 'A' - 'a';
       letters[0] += diff;
       letters[1] += diff;
       letters[2] += diff;
       cout << letters[0] << letters[1] << letters[2] << endl;

       return 0;
   }

8. #include <iostream>
   using namespace std;

   struct ThreeLetters {
       char ch1, ch2, ch3;

       void query() const {
           cout << ch1 << ch2 << ch3 << endl;
       }

       void loud_query() {
           int diff = 'A' - 'a';
           cout << (ch1+=diff) << (ch2+=diff) << (ch3+=diff) << endl;
       }
   };

   int main() {
       ThreeLetters letters = {'a', 'b', 'c'};
       letters.query();
       letters.loud_query();

       return 0;
   }

```

Try to create a similar struct `Letters`. Given it the member variable `chars` that's a vector of chars. The following code

```
int main() {
    Letters letters;
    letters.chars = {'a', 'b', 'c', 'd'};
    letters.query();
    letters.loud_query();

    return 0;
}
```

should output to the console:

```
abcd
ABCD
```

1.3.3 Error spotting exercises

Error spotting will be on the exam. Look at the follow code below and state:

- What errors occurred (build, run time, or logic) or if there are none.
- In the case of a logic error, what does the code currently accomplish?
- How the code can be fixed (use the line numbers to help with the description).

A description of how the code should perform will be put in a block comment. After you make your initial guess, type the code in your IDE and run it. Copying very carefully may help you find some of the errors. Play around with the code to fix it.

```
9.  1  #include <iostream>
    2  using namespace std;
    3
    4  /**
    5   This function takes in an integer and turns it to 42.
    6   @param val : integer to turn to 42.
    7   */
    8  void modify(int& val) {
    9      val = 42;
   10  }
   11
   12  /* This code is meant to print 42. */
   13  int main() {
   14      int num = 10;
   15      modify(num);
   16      cout << num << endl;
```

```

17     return 0;
18 }

```

10. (The all_caps function is fine.)

```

1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  /**
6   Capitalizes all letters in a string that are lowercase.
7   @param str string to capitalize.
8   @return All caps string.
9   */
10 string all_caps(string& str) {
11     for (int idx = 0; idx < str.length(); idx++) {
12         if (str[idx] >= 'a' && str[idx] <= 'z') {
13             str[idx] -= 'a' - 'A';
14         }
15     }
16     return str;
17 }
18
19 /**
20  Yells a name.
21  @param first The first name.
22  @param last The last name.
23  */
24 void yell_name(const string& first, const string& last) {
25     cout << all_caps(first) << '!' << ' ' << all_caps(last) << '!' << endl;
26 }
27
28 /* This code is meant to print:
29    BART! SIMPSON!
30    */
31 int main() {
32     string first = "Bart";
33     string last = "Simpson";
34     yell_name(first, last);
35     return 0;
36 }

```

11.

```

1  #include <iostream>
2  using namespace std;
3

```

```
4  struct Point {
5      int x;
6      int y;
7  }
8
9  int main() {
10     Point pt = {10, 20};
11     cout << pt.x << " " << pt.y << endl;
12
13     return 0;
14 }
```