

1 Week 7

1.1 Object-oriented programming

We are now beginning the topic of object-oriented programming (OOP). In the past week, we've learned how to use different data types such as strings and vectors, but we haven't gotten into how those were created to have their specific purposes or how we can create our own data types to suit our own purposes. Structs and classes will provide a plan for a new data type to be created. The thing you create from the struct or class will be called the object. A struct or class can create many objects.

In this course we might create objects such as geometric shapes such as Rectangles, Triangles, Circles, Points, Accounts to represent real-world things. OOP in practice is often used to develop more complex data structures and applications, such as graphical user interfaces (GUIs), web applications, and video games.

1.2 Struct

Let's review some OOP terminology. We will start with **structs**. **Classes** won't be too different but they have more features to them.

Struct: A custom data type with a plan on how to create an object/instance.

Object: An instance of a struct. It contains member variables and performs based on instructions from the struct.

Member variable: A variable that's part of the object.

Member function: A function that is defined within a struct. An object will have access to this function.

Constructor: A special member function that is used to initialize member variables of an object.

Instead of using a vector of ints to store two ints. We can create our own custom data type to store two ints and print them out easily.

```
struct Point {  
    // Member variables  
    int x;  
    int y;  
  
    // Constructors  
    Point() : x(0), y(0) {} // Default constructor  
    Point(int _x, int _y) : x(_x), y(_y) {} // "obvious" constructor  
  
    // Member function  
    void print() {
```

```

        cout << "(" << x << ", " << y << ")\n";
    }
};

int main() {
    // Create a Point object using the default constructor
    Point pt00;

    // Create a Point object using the "obvious" constructor
    Point pt11(1,1);

    // Call the member function of the Point object
    pt00.print();
    pt11.print();

    return 0;
}

```

In this example we create a plan (or **struct**) for creating **Point** objects. We say a **Point** object will be composed of two member variables **x** and **y**. We can assign them two different ways. In the example **pt00** is created by writing the struct name **Point** and the variable name we wish to call it the new object, **pt00**. This called the default constructor which assigns the member variables to both be 0. Another way a **Point** object can be created here is to use the other construct. This is called the “obvious” or “natural” constructor because we assign to all the member variables the arguments we provide in the order they’re initialized in the struct. Note that the format of the constructor looks like a function. The main point of the constructors is to pass arguments to assign member variables but we do have the option to execute some code in the curly braces "{}" just like you would with an ordinary function. You might do this to ensure the member variables initialized make sense for the object you’re creating. Maybe the member variables of this **Point** struct are meant to track the pH of a solution over time. So you’d might check the first variable to be between 0 and 14. Then the second variable to be nonnegative. The last component of this struct is the **print** member function which allows us to conveniently print the points to the console in any format we fashion.

```

(0, 0)
(1, 1)

```

1.3 Exercises

1.3.1 Practice coding exercises

1. (a) Write a function **formFiller** that fills and prints out answers to a questionnaire. The function should take an object **Person** which has information like **name**, **age**, **hobby**. Write a **Person** struct designed to hold the information about a specific person as member variables. Calling the **formFiller** on an instance of a **Person** should look like

```

Person toddler = {"JP", 2, "biking"};
formFiller(toddler);

```

print out a message like:

JP is 2 years old and likes biking.

or

```
Name:           JP
Age:            2
Favorite hobby:  biking
```

Create a few instance of **Person** and execute **formFiller**.

- (b) If you're **Person** struct looks like

```
struct Person {
    string name;
    int age;
    string hobby;
};
```

Add the default constructor (choose some default values like "", 0, ""). Add an obvious constructor. The member variable **age** should be nonnegative since age is nonnegative. In the curly braces of the obvious constructor, write some control flow that make negative ages take on the value 0. Be careful to keep track of what is the argument and what is the member variable. Also, be mindful of const/ref correction –constructors are just special function!

- (c) Rewrite the ordinary **formFiller** function as a member function of the **Person** struct. You will no longer need to take in arguments explicitly.

2. Write a struct **DataPoint** that's meant to hold the data points of pH and time. Have the construct check if pH is in the correct range of values. If it's not, a runtime error will be thrown. Have the struct contain a print member function. Test the constructors. You should use the function **assert** from the header file **cassert**. Here's an example:

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

int main() {
    int x = 15;
    int y = 10;

    assert(x < y);

    cout << "x is less than y." << std::endl;

    return 0;
}
```

Results in the following...

Assertion failed: (x < y), function main, file main.cpp, line 10.
(lldb)

3. In this activity, you will learn how to create a C++ struct called **Line** that represents a line segment using two **Point** objects. Follow the steps below:
 - (a) Define the **Point** struct (or use the one above) with the following member variables and a constructor:
 - **int x**; (integer for the x-coordinate)
 - **int y**; (integer for the y-coordinate)
 - A constructor with two parameters to initialize **x** and **y**.
 - (b) Define the **Line** struct with the following member variables and a constructor:
 - **Point start**; (a **Point** object representing the start point)
 - **Point end**; (a **Point** object representing the end point)
 - A constructor with two parameters to initialize **start** and **end**.
 - (c) Implement a member function called **print** for the **Line** struct that displays information about the line, including the start and end points.
 - (d) In the **main** function, create instances of the **Point** and **Line** structs. Use the **print** function to display information about the line.
4. Create a shopping cart system to manage items, their quantities, and calculate the total price.
 - (a) Define an **Item** struct with the following member variables:
 - **string name** to store the item's name.
 - **int quantity** to store the quantity of the item.
 - **double price** to store the price of the item.
 - (b) Implement a default constructor for the **Item** struct and the constructor that initializes all the member variables.
 - (c) Define a member function named **print** in the **Item** struct. This function should print the item's details in the following format:

```
Item Name: [name]
Quantity: [quantity]
Price: $[price]
```

We want things printed nicely so that the prices look like dollar amounts. Include the header file **iomanip** so you can use **setprecision** and **fixed**.
 - (d) Create another struct. Call it **ShoppingCart**. Give it the member variable:
 - **vector<Item> items** to store a collection of items in the shopping cart.

- (e) Add a member function in the `ShoppingCart` struct named `add` that adds an `Item` to the shopping cart. How can you add things to a shopping cart object? What does it consist of? `items`? It's a vector, so recalling the vector functions could help.
- (f) Add a member function in the `ShoppingCart` struct named `remove` that removes an `Item` from the shopping cart. The removal should be based on the item's name. If the item is not found in the cart, do nothing. When you remove the item, you should do it in a special way to keep the vector in a valid state (don't worry too much about what that means). To do this, `swap` the item you wish to remove with the last item in `items`. Then remove the last item in the normal way you would remove the last item from a vector.
- (g) Create a member function in the `ShoppingCart` struct named `getTotalPrice` that calculates and returns the total price of all items in the shopping cart. The total price is the sum of the quantity times the price for each item.
- (h) Implement a member function named `print` in the `ShoppingCart` struct. This function should print the contents of the shopping cart by calling the `print` function for each item. Additionally, it should print the total price of all items in the cart.
- (i) Test your code with the following in `main`:

```
int main() {
    ShoppingCart cart;

    cart.add(Item("Apples", 2, 1.0));
    cart.add(Item("Oranges", 3, 0.5));
    cart.add(Item("Bananas", 4, 0.25));
    cart.print();

    cart.remove(Item("Oranges", 1, 0.5));
    cart.remove(Item("Bananas", 2, 0.25));
    cart.print();

    return 0;
}
```

Executing this code will print to the console:

```
Shopping cart contents:
Apples: 2 @ $1.00
Oranges: 3 @ $0.50
Bananas: 4 @ $0.25
Total price: $4.50
Shopping cart contents:
Apples: 2 @ $1.00
Total price: $2.00
```

5. In this activity, you will practice working with vectors and implementing custom functions to perform specific tasks.

- (a) In `main`, initialize a vector named `v` with the integers 1, 2, and 3. Use a `for` loop to output the elements of the vector `v` to the console.
- (b) Build a function named `count_odds` that takes a vector of integers as a parameter. Inside the function, count the number of odd integers in the vector (integers that have a remainder of 1 when divided by 2). Return the count of odd integers.
- (c) Build a function named `in` that takes a vector of integers and an integer `x` as parameters. Inside the function, check if the integer `x` exists in the vector. Return `true` if `x` is found in the vector, otherwise return `false`.
- (d) Test the Functions. Use the `count_odds` function to count the number of odd integers in the vector `v`. Use the `in` function to check if a specific integer exists in the vector `v`.

6. Consider the following C++ code that defines a `Vector` class with Carrays:

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

struct Vector {
    int* array; // array = {}
    int size = 0;
    int capacity = 0;

    // Add constructors and member function declarations...
};

void Vector::push_back(int value) {
    // Implementation...
}

void Vector::pop_back() {
    // Implementation...
}

int Vector::get(int index) {
    // Implementation...
}

void Vector::set(int value, int index) {
    // Implementation...
}

int Vector::get_size() {
```

```

        // Implementation...
    }

    int Vector::get_capacity() {
        // Implementation...
    }

    void Vector::print() {
        // Implementation...
    }

    int main() {
        // Vector usage...
    }

```

Your task is to complete the implementation of the **Vector** class and test its functionality. Follow these steps:

- (a) Start by examining the provided C++ code. It defines a **Vector** class with some member functions. Your goal is to complete the implementation of this class. Add the declarations for the member functions. Have all functions which are non-void return something of the appropriate type.
- (b) Create the following constructors of the **Vector** class:
 - i. **Vector()**: Create a default constructor that initializes the vector with an initial size of 0 and a capacity of 1. Ensure proper memory allocation and initialization of the dynamic array.
 - ii. **Vector(int _size)**: Create a constructor that takes an integer parameter for the initial size. Initialize the vector with the specified size and set all elements to 0. Handle memory allocation and array initialization.
 - iii. **Vector(int _size, int _value)**: Implement a constructor that takes both size and initial value as parameters. Initialize the vector with the specified size, and set all elements to the provided value. Properly manage memory allocation.
- (c) Implement the member functions of the **Vector** class:
 - i. **push_back(int value)**: Add an element to the vector.
 - ii. **pop_back()**: Remove the last element from the vector.
 - iii. **get(int index)**: Retrieve the element at a specific index.
 - iv. **set(int value, int index)**: Set the value of an element at a specific index.
 - v. **get_size()**: Get the current size (number of elements) of the vector.
 - vi. **get_capacity()**: Get the current capacity of the vector (the maximum number of elements it can hold).
 - vii. **print()**: Print the elements of the vector.
- (d) In the **main** function, create instances of the **Vector** class and test its operations:

- i. Create a `Vector` instance.
- ii. Use `push_back` to add elements to the vector.
- iii. Use `get` to retrieve elements by index.
- iv. Use `set` to modify elements by index.
- v. Use `pop_back` to remove elements from the end.
- vi. Use `get_size` and `get_capacity` to check the size and capacity.
- vii. Use `print` to display the elements in the vector.

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.

7. Review the code in the database folder.

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.

```
8.  1  #include <iostream>
    2  using namespace std;
    3
    4  /* This code creates a struct with
    5  member variables width, length,
    6  and a member function area. */
    7  struct Rectangle {
    8      int width;
    9      int length;
   10      int area() {
   11          return width * length;
   12      }
   13 };
```

```

9.  1  #include <iostream>
    2  using namespace std;
    3
    4  /* This code create a struct with member variables and a
    5  constructor to assign the arguments the member variables. */
    6  struct Employee {
    7      string name;
    8      int age;
    9      double salary;
   10
   11      Employee(string _name, int _age, double _salary)
   12          : name(_name), age(_age), salary(_salary) {}
   13  };

10. 1  #include <iostream>
    2  using namespace std;
    3
    4  /* This code is meant to create a Car struct with a
    5  parametrized constructor assigning the member variables. */
    6  struct Car {
    7      string make;
    8      string model;
    9      int year;
   10      bool automatic;
   11
   12      Car(const string& m, const string& mod, int y, bool auto) {
   13          make = m;
   14          model = mod;
   15          year = y;
   16          automatic = auto;
   17      }
   18  };

11. 1  #include <iostream>
    2  #include <cassert>
    3  using namespace std;
    4
    5  /* This code create a struct called DataPoint which tracks temperatures
    6  only in the range [-2, 40] and will raise an error if we try and
    7  create a data point out of that range. */
    8  struct DataPoint {
    9      double temp;

```

```

10     double time;
11
12     DataPoint(double _temp, double _time) : temp(_temp), time(_time) {
13         assert(_temp >= -20.0 && _temp <= 40.0);
14     }
15
16     void print() {
17         cout << "Temperature: " << temp << "°C, Time: " << time << " seconds" << endl;
18     }
19 };
20
21 int main() {
22     DataPoint point1(25.5, 10.0);
23     DataPoint point2(-30.0, 5.0);
24
25     point1.print();
26     point2.print();
27
28     return 0;
29 }

```